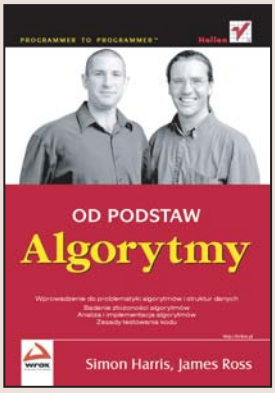
Algorytmy. Od podstaw

Autorzy: Simon Harris, James Ross
Tłumaczenie: Andrzej Grażyński
ISBN: 83-246-0372-7
Tytuł oryginału: Beginning Algorithms
Format: B5, stron: 610



Wprowadzenie do problematyki algorytmów i struktur danych

- Badanie złożoności algorytmów
- Analiza i implementacja algorytmów
- Zasady testowania kodu

Algorytmy leżą u podstaw programowania. Zasady rozwiązywania typowych problemów programistycznych, opisane w postaci blokowej lub za pomocą uniwersalnego „pseudokodu", są wykorzystywane codziennie przez tysiące informatyków na całym świecie. Właściwe zrozumienie zarówno samych algorytmów, jak i zasad ich stosowania w praktyce, jest kluczem do tworzenia wydajnych aplikacji. Umiejętność oceny efektywności i złożoności algorytmów przyda się również przy wyborze najlepszego rozwiązania określonego problemu.

Książka „Algorytmy. Od podstaw" przedstawia podstawowe zagadnienia związane z algorytmami. Dzięki niej nauczysz się wyznaczać złożoność obliczeniową algorytmów i implementować algorytmy w programach. Poznasz algorytmy sortowania, przeszukiwania i przetwarzania danych. Dowiesz się, czym są testy jednostkowe i dlaczego ich stosowanie jest tak ważne podczas tworzenia oprogramowania.

W książce omówiono m.in. następujące zagadnienia:

- Testy jednostkowe i biblioteka JUnit
- Iteracja i rekurencja
- Kolejki FIFO
- Listy i stosy
- Algorytmy sortowania
- Binarne wyszukiwanie i zastępowanie
- Zbiory, mapy i drzewa wyszukiwawcze
- Wyszukiwanie tekstu

Poznaj sprawdzone i powszechnie używane algorytmy i zastosuj je w swoich aplikacjach

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

6

Sortowanie — proste algorytmy

Opisywane w poprzednich rozdziałach struktury danych pełnią fundamentalną rolę w tworzonych aplikacjach jako środki organizujące przetwarzanie ogromnych ilości danych. W szczególności sortowanie danych według pewnego kryterium stanowi nieodłączny element wielu algorytmów, w tym algorytmów opisywanych w dalszych rozdziałach niniejszej książki. Jednocześnie staje się ono często wąskim gardłem wydajności aplikacji, nic więc dziwnego, że sortowanie danych rozmaitych typów było przedmiotem intensywnych badań w ostatnich dziesięcioleciach i nadal stanowi jeden z kluczowych punktów zainteresowań informatyki. W niniejszym rozdziale omawiamy trzy proste algorytmy sortowania, łatwe w implementacji, lecz przydatne raczej dla niewielkich ilości danych, a to ze względu na złożoność kwadratową $O(N^2)$ (N jest liczbą elementów w sortowanym zestawie). Bardziej wydajnym — i jednocześnie bardziej skomplikowanym — algorytmom sortowania poświęcony będzie rozdział 7.

W niniejszym rozdziale omawiamy:

- znaczenie sortowania — w tworzonych aplikacjach i nie tylko,
- rolę komparatorów w konstrukcji i przetwarzaniu struktur danych,
- algorytm sortowania bąbelkowego,
- algorytm sortowania przez wybieranie,
- algorytm sortowania przez wstawianie,
- własność stabilności sortowania,
- zalety i wady prostych metod sortowania.

Znaczenie sortowania

Przeglądając na co dzień książkę telefoniczną, spis teleadresowy itp., najczęściej nie uświadamiamy sobie, iż wykorzystujemy fakt ich posortowania. Szukając określonego nazwiska czy firmy, po prostu próbujemy zgadnąć, w którym miejscu spisu możemy się go spodziewać,

i już po kilku takich próbach trafiamy na żądaną stronę, na której w ciągu kilku sekund odnajdujemy to, czego szukamy (bądź stwierdzamy, że taki to a taki abonent w spisie teleadresowym nie figuruje). Wyobraźmy sobie teraz, że taki spis teleadresowy nie jest posortowany — abonenci występują w nim w kolejności przypadkowej[1]. Trzeba mieć dużo dobrej woli i determinacji, by w ogóle podjąć się próby znalezienia w nim czegokolwiek lub kogokolwiek — próby raczej z góry skazanej na niepowodzenie. Wiele zbiorów danych byłoby zupełnie bezużytecznych, gdyby nie zostały posortowane według pewnego użytecznego kryterium — dotyczy to nie tylko nazwisk czy nazw w spisie teleadresowym, lecz także np. książek na półkach bibliotecznych. Jako że często zbiory te posortowane są a priori, uważamy to za coś naturalnego i w ogóle o sortowaniu nie myślimy. W przypadku komputerowego przetwarzania danych jest zupełnie inaczej: trudno oczekiwać, że użytkownik aplikacji dostarczać będzie dane w kolejności posortowanej, a w każdym razie byłoby czymś kuriozalnym wymagać od niego czegokolwiek, co znacznie efektywniej może za niego wykonać komputer. Sortowanie rozmaitych danych staje się więc nieodłączną czynnością wielu aplikacji, a dobra znajomość różnych metod sortowania jest warunkiem wykonywania tej czynności w sposób efektywny.

Podstawy sortowania

Warunkiem wstępnym możliwości posortowania danych według pewnego kryterium jest istnienie struktury zdolnej przechowywać elementy tych danych w określonej kolejności. Jak widzieliśmy w rozdziale 3., to właśnie listy są strukturami zachowującymi (względną) kolejność wstawianych elementów — interfejs `List` nie zawiera metod zmieniających tę kolejność w sposób bezpośredni, zmiana pozycji elementu w liście nie jest możliwa bez jego usunięcia i ponownego wstawienia.

Każdy algorytm sortowania listy elementów opiera się na dwóch fundamentalnych operacjach:

- porównywaniu elementów w celu stwierdzenia, czy ich względna kolejność w liście zgodna jest z kryterium sortowania,
- przesuwaniu elementów na pozycje wyznaczane przez kryterium sortowania.

Zalety i wady danego algorytmu sortowania wynikają przede wszystkim z tego, ile wymienionych wyżej operacji należy wykonać w celu posortowania określonego zbioru danych i jak efektywna jest każda z tych operacji. Porównywanie elementów jest czynnością znacznie mniej oczywistą, niż mogłoby się to w pierwszej chwili wydawać; w kolejnym podrozdziale omawiamy wynikającą z niego koncepcję komparatora. Dokładny opis wykorzystywanych operacji listowych — `get()`, `set()`, `insert()` i `delete()` — znajdzie Czytelnik w rozdziale 3.

[1] Albo w kolejności wyznaczonej przez kryterium nieznane użytkownikowi, na przykład w kolejności zgłoszenia swych danych do wydawcy — *przyp. tłum.*

Komparatory

W języku Java i w większości innych języków programowania porównywanie wartości dwóch zmiennych całkowitoliczbowych jest czynnością niewymagającą komentarza:

```
int x, y;
  ...
if (x < y)  {
  ...
}
```

Podobnie ma się rzecz w przypadku podstawowych (*primitive*) typów danych, lecz w miarę postępującej komplikacji struktur danych porównywanie ich elementów (obiektów) szybko traci swą oczywistość. Wyobraźmy sobie na przykład listę plików znajdujących się w pewnym katalogu: listę tę można (stosownie do różnych potrzeb) sortować według rozmaitych kryteriów — nazwy, rozszerzenia, rozmiaru, daty utworzenia, daty ostatniej modyfikacji itp.

Ważne jest więc oddzielenie kryterium sortowania elementów od samej czynności sortowania. Mechanizm narzucający na listę obiektów pewne kryterium porządkujące nosi nazwę *komparatora*; dla danej listy (na przykład wspomnianej listy plików) określić można kilka różnych komparatorów wyrażających rozmaite kryteria uporządkowania. Dzięki temu sortowanie listy według różnych kryteriów — nazwy pliku, jego rozszerzenia, rozmiaru itp. — da się zrealizować w sposób jednolity, za pomocą tego samego algorytmu sortowania.

Wspomniane oddzielenie kryterium sortowania od samego sortowania jest przykładem bardziej ogólnej koncepcji projektowej, zwanej rozdzielaniem zagadnień (*separation of concerns*). Umożliwia ono rozszerzanie użyteczności samego algorytmu dzięki zastosowaniu rozmaitych „wtyczek” (*plugins*) — nawet takich, które trudno byłoby sobie wyobrazić w momencie implementowania samego algorytmu. Właśnie komparatory są przykładem takich „wtyczek” dla algorytmów sortowania. Można także spojrzeć na całą sprawę z drugiej strony: zastosowanie tego samego komparatora do różnych algorytmów sortowania umożliwia miarodajne porównywanie wydajności tych algorytmów.

Operacje komparatora

Komparator wykonuje tylko jedną operację — jest nią określenie względnej kolejności dwóch porównywanych obiektów. Zależnie od tego, czy pierwszy z wymienionych obiektów jest mniejszy, równy lub większy od drugiego (w sensie przyjętego kryterium porównywania), wynikiem tej operacji jest (odpowiednio) wartość ujemna, zero lub wartość dodatnia. Jeśli typ któregokolwiek z wymienionych obiektów wyklucza możliwość porównywania go z innymi obiektami, próba wykonania porównania powoduje wystąpienie wyjątku `ClassCastException`.

Interfejs komparatora

Jedyna operacja komparatora przekłada się na jedyną metodę interfejsu `Comparator`, określającą względną relację (porządek) między obiektami określonymi przez jej argumenty:

```
public interface Comparator {
    public int compare(Object left, Object right);
}
```

Argumenty metody nieprzypadkowo określone zostały jako „lewy" (*left*) i „prawy" (*right*), mogą być bowiem utożsamiane z (odpowiednio) lewym i prawym argumentem operatora porównania — metoda `compare()` w istocie stanowi uogólnienie operatora porównania dla typów podstawowych języka. Zależnie od tego, czy obiekt `left` jest (w sensie przyjętego kryterium porównywania) mniejszy od obiektu `right`, równy mu lub od niego większy, metoda zwraca (odpowiednio) wartość ujemną (zwykle –1, choć niekoniecznie), zero (koniecznie) lub wartość dodatnią (zwykle 1, choć niekoniecznie).

Niektóre komparatory standardowe

Oprócz nieograniczonych wręcz możliwości definiowania komparatorów przez programistę istnieje w języku Java kilka komparatorów standardowych w dużym stopniu upraszczających tworzenie kodu aplikacji. Każdy z nich jest prosty pod względem koncepcyjnym i wielce użyteczny w konstrukcji wielu złożonych algorytmów prezentowanych w niniejszej książce.

Komparator naturalny

W wielu typach danych, szczególnie typach podstawowych, jak łańcuchy czy liczby całkowite, zdefiniowane jest a priori uporządkowanie naturalne: 1 poprzedza 2, `A` poprzedza `B`, `B` poprzedza `C` itp. Komparator narzucający taki właśnie naturalny porządek nazywamy (jakżeby inaczej) *komparatorem naturalnym*. Jak pokażemy za chwilę, dla danych określonego typu zdefiniować można ich naturalne uporządkowanie, bazując na konwencjach obowiązujących w języku Java — umożliwiającym to środkiem jest interfejs `Comparable`.

Interfejs Comparable

Interfejs `Comparable` posiada tylko jedną metodę:

```
public interface Comparable {
    public int compareTo(Object other);
}
```

zwracającą — podobnie jak metoda `compare()` interfejsu `Comparator` — wartość ujemną, zero lub wartość dodatnią zależnie od tego, czy obiekt stanowiący podmiot wywołania metody jest (odpowiednio) mniejszy, równy lub większy od obiektu reprezentowanego przez parametr `other`. Zasadnicza różnica między metodą `compare()` a metodą `compareTo()` polega na tym, iż ta pierwsza porównuje ze sobą dwa wskazane obiekty, podczas gdy druga porównuje dany obiekt z innym obiektem.

Jest więc jasne, że aby zdefiniować naturalny porządek w stosunku do wartości danej klasy, należy zaimplementować w tej klasie interfejs `Comparable`. Przykładowo dla rekordów zawierające dane pracowników za uporządkowanie naturalne można przyjąć uporządkowanie według nazwiska i imienia. Koncepcja ta stanowi uogólnienie operatorów <, = i > na złożone typy danych — i faktycznie wiele powszechnie używanych klas z pakietu `java.lang` implementuje interfejs `Comparable`.

Gdy wyobrazimy sobie funkcjonowanie komparatora naturalnego (którego klasę nazwiemy `NaturalComparator`), szybko stanie się jasne, iż należy go przetestować w kontekście trzech przypadków: porównania obiektu „mniejszego z większym", „większego z mniejszym" oraz dwóch „równych" obiektów. Aby ułatwić sobie zadanie, wykorzystamy fakt, że interfejs `Comparable` zdefiniowany jest standardowo m.in. dla łańcuchów języka Java.

spróbuj sam Testowanie komparatora naturalnego

Rozpoczniemy od konfiguracji, w której metoda `compare()` komparatora naturalnego powinna zwrócić wartość ujemną:

```
public void testLessThan() {
    assertTrue(NaturalComparator.INSTANCE.compare("A", "B") < 0);
}
```

Zamieniając miejscami porównywane obiekty, otrzymamy konfigurację, w której metoda `compare()` powinna zwrócić wartość dodatnią:

```
public void testGreaterThan() {
    assertTrue(NaturalComparator.INSTANCE.compare("B", "A") > 0);
}
```

Przy porównywaniu dwóch identycznych wartości metoda `compare()` powinna zwrócić 0:

```
public void testEqualTo () {
    assertTrue(NaturalComparator.INSTANCE.compare("A", "A") = 0);
}
```

Jak to działa?

Dla każdego z trzech możliwych przypadków relacji między porównywanymi obiektami (mniejszy, równy, większy) zdefiniowano osobny przypadek testowy, wykonujący oczywiste porównanie łańcuchów jednoznakowych. Wszystkie przypadki testowe bazują na założeniu, że istnieje tylko jedna statyczna instancja klasy `NaturalComparator` i nie należy tworzyć innych jej instancji.

spróbuj sam Implementowanie komparatora naturalnego

Ponieważ klasa `NaturalComparator` nie przechowuje żadnych informacji o stanie, wystarczające jest istnienie tylko jednej, publicznie dostępnej jej instancji:

```
public final class NaturalComparator implements Comparator {
    /** jedyna, publicznie dostępna instancja komparatora */
    public static final NaturalComparator INSTANCE = new NaturalComparator();

    /**
     * konstruktor prywatny, nie jest możliwe samodzielne tworzenie instancji
     */
    private NaturalComparator() {
    }
      ...
}
```

Aby uniemożliwić samodzielne tworzenie kolejnych instancji, uczyniono konstruktor klasy prywatnym, a więc niewidocznym na zewnątrz niej. Ponadto sama klasa oznaczona została jako finalna (`final`) w celu zapobieżenia jej (być może błędnemu) rozszerzaniu.

Ponieważ metoda `compare()` komparatora naturalnego implementowana jest na bazie interfejsu `Comparable`, zasadnicza jej czynność scedowana została na jej argumenty, co czyni jej implementację niemal banalną:

```
public int compare(Object left, Object right) throws ClassCastException {
    assert left != null : "nie określono lewego obiektu ";
    return ((Comparable) left).compareTo(right);
}
```

Po upewnieniu się, że lewy argument nie jest argumentem pustym, następuje jego rzutowanie na instancję interfejsu `Comparable` i wywołanie metody `compareTo()` tego interfejsu z prawym obiektem jako argumentem.

Rzutując obiekt `left` na instancję interfejsu `Comparable` nie sprawdzamy, czy rzutowanie to jest wykonalne, tzn. czy typ tego obiektu nie wyklucza wykonywania jego porównań z innymi obiektami. Sprawdzenie takie jest niepotrzebne, bowiem interfejs `Comparator` dopuszcza występowanie wyjątku `ClassCastException` w sytuacji, gdy wymieniony wyżej warunek nie jest spełniony.

Jak to działa?

Klasa `NaturalComparator` skonstruowana została w celu porównywania dwóch obiektów implementujących interfejs `Comparable`. Interfejs ten implementowany jest standardowo przez wiele klas Javy i oczywiście można go implementować ad hoc w klasach definiowanych samodzielnie. Implementacja taka polega każdorazowo na zrzutowaniu lewego argumentu na instancję interfejsu `Comparable` i wywołaniu na jego rzecz metody `compareTo()` z prawym argumentem jako parametrem. Wszelka „logika porównywania" nie jest w tej implementacji widoczna, skrywa się bowiem całkowicie w implementacji metody `compareTo()`.

Komparator odwrotny

Zdarza się, że mając zdefiniowane pewne uporządkowanie wartości jakiegoś typu, chcielibyśmy posortować te wartości w kolejności dokładnie odwrotnej niż wynikająca z tego uporządkowania, na przykład wypisać nazwy plików pewnego katalogu w kolejności od-

wrotnej do kolejności alfabetycznej. Trywialnym sposobem wykonania tego zadania jest zamiana znaczeń obiektów stanowiących argumenty wywołania metody `compare()` komparatora `NaturalComparator`:

```
public int compare(Object left, Object right) throws ClassCastException {
     assert right != null : "nie określono prawego obiektu ";
     return ((Comparable) right).compareTo(left);
 }
```

Po upewnieniu się, że lewy argument nie jest argumentem pustym, następuje wywołanie jego metody `compareTo()` z prawym argumentem jako parametrem wywołania.

Mimo iż to doraźne rozwiązanie sprawdza się nieźle w tym szczególnym przypadku, jest mało uniwersalne, bowiem dla bardziej złożonych struktur danych, jak lista plików czy lista danych pracowniczych, wymaga definiowania *dwóch* komparatorów, po jednym dla każdego „kierunku" uporządkowania.

Znacznie bardziej eleganckie rozwiązanie, które za chwilę zaprezentujemy, polega na *odwróceniu* kierunku wskazanego komparatora poprzez jego „udekorowanie" (otoczenie) innym komparatorem, w wyniku czego otrzymuje się komparator zwany *komparatorem odwrotnym*. Dla każdego typu danych wystarczy wówczas zdefiniować tylko jeden komparator i w razie potrzeby „odwracać" wyznaczane przez niego uporządkowanie w sposób uniwersalny, za pomocą opisanego odwracania.

spróbuj sam Testowanie komparatora odwrotnego

Podobnie jak w przypadku komparatora `NaturalComparator`, tak i w przypadku komparatora odwrotnego — który nazwiemy `ReverseComparator` — przetestować musimy trzy możliwe przypadki relacji między porównywanymi obiektami. „Dekorowanym" komparatorem, którego kierunek będziemy odwracać, będzie przy tym sam komparator naturalny `NaturalComparator`.

Jeśli lewy argument metody `compare()` komparatora `NaturalComparator` jest mniejszy od jej prawego argumentu, metoda ta powinna zwrócić wartość ujemną. Jeżeli jednak na bazie komparatora `NaturalComparator` stworzymy komparator odwrotny, to jego metoda `compare()` zwrócić musi w takiej sytuacji wartość dodatnią:

```
public void testLessThanBecomesGreaterThan() {
    ReverseComparator comparator =
        new ReverseComparator(NaturalComparator.INSTANCE);

    assertTrue(comparator.compare("A", "B") > 0);
}
```

Analogicznie, jeśli lewy argument komparatora odwrotnego jest większy niż prawy, metoda `compare()` tego komparatora powinna zwrócić wartość ujemną:

```
public void testGreaterThanBecomesLessThan() {
    ReverseComparator comparator =
        new ReverseComparator(NaturalComparator.INSTANCE);

    assertTrue(comparator.compare("B", "A") < 0);
}
```

W przypadku porównywania identycznych argumentów nic się nie zmienia, zarówno dla komparatora oryginalnego, jak i odwrotnego metoda `compare()` powinna zwrócić 0:

```
    public void testEqualsRemainsUnchanged() {
        ReverseComparator comparator =
            new ReverseComparator(NaturalComparator.INSTANCE);

        assertTrue(comparator.compare("A", "A") == 0);
    }
```

Jak to działa?

Na bazie komparatora naturalnego `NaturalComparator` tworzony jest komparator odwrotny `ReverseComparator`, którego działanie (czyli wyniki porównań) powinno być dokładnie odwrotne w stosunku do pierwowzoru. W szczególności łańcuch „A” powinien być uznany za „większy” od łańcucha „B” i vice versa — łańcuch „B” powinien zostać uznany za „mniejszy” od łańcucha „A”. W przypadku porównywania dwóch identycznych obiektów komparator odwrotny także powinien uznać je za identyczne.

spróbuj sam Implementowanie komparatora odwrotnego

Implementacja komparatora `ReverseComparator` składa się z niewielu linii kodu:

```
package com.wrox.algorithms.sorting;

public class ReverseComparator implements Comparator {
    private final Comparator _comparator;

    public ReverseComparator(Comparator comparator) {
        assert comparator != null : "nie określono oryginalnego komparatora";
        _comparator = comparator;
    }
     ...
}
```

W konstruktorze klasy przekazywany jest oryginalny komparator, którego działanie ulec ma odwróceniu; jest on zapamiętywany w prywatnej zmiennej `_comparator`.

Pozostaje tylko zaimplementowanie metody `compare()` — jedynej metody implementowanego interfejsu `Comparator`:

```
    public int compare(Object left, Object right) throws ClassCastException {
        return _comparator.compare(right, left);
    }
```

Jak to działa?

Na pierwszy rzut oka wygląda to ma zwykłe delegowanie wywołania do metody `compare()` komparatora oryginalnego; jeśli jednak spojrzeć uważnie po raz drugi, łatwo można zauważyć, że delegowaniu temu towarzyszy *zmiana kolejności* argumentów: przykładowo wywołanie

`compare("A", "B")` w interfejsie odwrotnym (`ReverseComparator`) delegowane jest do interfejsu oryginalnego jako wywołanie `compare("B", "A")`. Zamiana kolejności argumentów wywołania metody daje w konsekwencji odwrotny wynik samej metody.

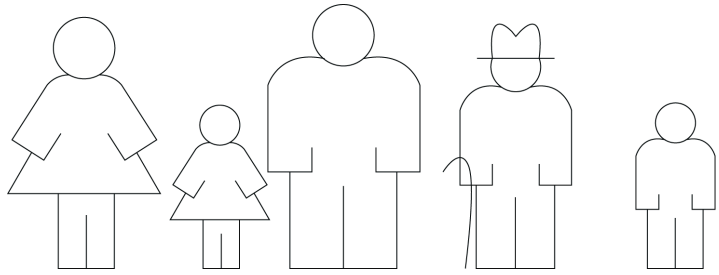
Ponieważ nie interesują nas żadne atrybuty porównywanych obiektów, a jedynie wynik ich porównania, opisane rozwiązanie jest w pełni uniwersalne: implementacja komparatora `ReverseComparator` jest całkowicie niezależna od implementacji komparatora oryginalnego. Skoro opisaliśmy już porównywanie elementów i jego implikacje w postaci komparatorów, zajmijmy się teraz trzema różnymi algorytmami sortowania.

Sortowanie bąbelkowe

Zanim przejdziemy do *sortowania bąbelkowego* (*bubblesort*), musimy zdefiniować kilka przypadków testowych dla różnych implementacji sortowania. Ponieważ każdy algorytm sortowania testowany będzie pod kątem spełnienia tego samego kryterium — poprawnego porządkowania sortowanych obiektów — zwyczajowo rozpoczniemy od zdefiniowania klasy bazowej definiującej te aspekty testowania, które są wspólne dla wszystkich algorytmów. Specyfikę konkretnych algorytmów powierzymy natomiast poszczególnym klasom pochodnym. W ten sposób otrzymamy zestaw testowy, który łatwo będzie można przystosowywać do dowolnych algorytmów sortowania — nawet takich, których być może jeszcze dziś nie znamy.

spróbuj sam Przeprowadzanie sortowania bąbelkowego

Wyobraźmy sobie rodzinę udającą się do fotografa. Na wspólnej fotografii członkowie rodziny powinni ustawić się według starszeństwa, od najmłodszego do najstarszego, gdy tymczasem ustawieni są w sposób przypadkowy, jak na rysunku 6.1.

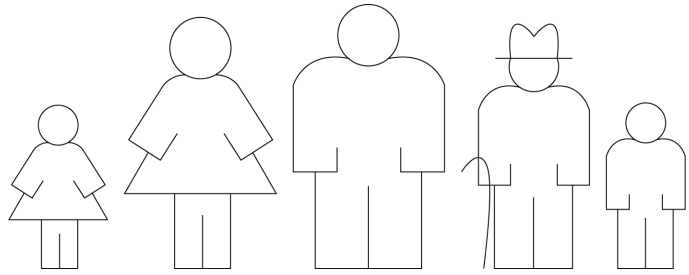


Rysunek 6.1. *Rodzina w przypadkowym szyku*

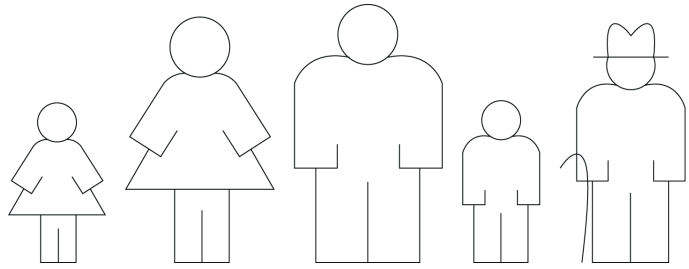
Aby dokonać przestawienia członków rodziny według algorytmu sortowania bąbelkowego, porównamy dwie skrajne osoby z lewej strony; nie są one ustawione zgodnie z wymaganiami — pierwsza z nich jest starsza od drugiej — poprosimy je więc, by zamieniły się miejscami, co da efekt widoczny na rysunku 6.2.

Porównując osobę drugą i trzecią, stwierdzamy, że ich względna kolejność jest prawidłowa. Nie można tego powiedzieć o osobie czwartej i piątej, które muszą zamienić się miejscami, doprowadzając do konfiguracji przedstawionej na rysunku 6.3.

Rysunek 6.2.
Po pierwszej zamianie miejsc



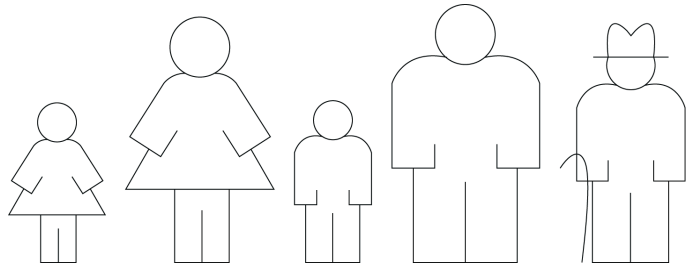
Rysunek 6.3.
Po wykonaniu pierwszego kroku — najstarsza osoba znajduje się już na swoim miejscu, czyli na skrajnej prawej pozycji



Choć senior rodu zajmuje już właściwą pozycję, kolejność, w jakiej ustawione są pozostałe osoby, nadal pozostawia wiele do życzenia, mimo że wykonaliśmy już kilka porównań i przestawień. Na razie musimy się pogodzić z tak nieefektywnym sortowaniem, w następnym rozdziale poznamy jego efektywniejsze algorytmy.

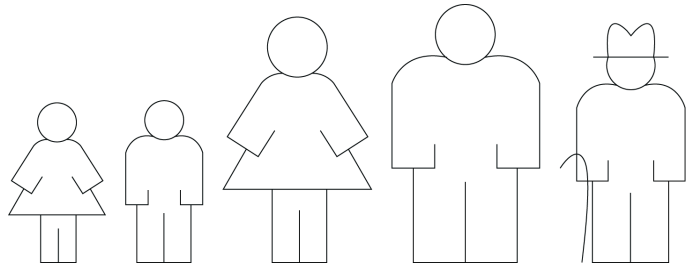
Kolejny krok sortowania bąbelkowego przebiega identycznie jak pierwszy z tą jednak różnicą, że skrajna prawa pozycja jest już „właściwie obsadzona" i możemy ją pominąć w porównaniach. Ostatecznie krok ten doprowadza do tego, że druga co do starszeństwa osoba trafia na przeznaczoną dla niej pozycję, jak na rysunku 6.4.

Rysunek 6.4.
Po wykonaniu drugiego kroku sortowania dwie najstarsze osoby stoją już na swoich miejscach



Wykonując jeszcze dwa kroki sortowania, z udziałem najpierw trzech, a potem dwóch osób, otrzymamy ostatecznie pożądany układ widoczny na rysunku 6.5.

Rysunek 6.5.
Rodzina prawidłowo ustawiona według starszeństwa



Interfejs ListSorter

Jak wiele interfejsów interfejs `ListSorter` jest skrajnie prosty, zawiera bowiem tylko jedną metodę, odpowiedzialną za posortowanie listy.

Metoda `sort()` otrzymuje listę jako argument wejściowy i zwraca jako wynik jej posortowaną wersję. Zależnie od implementacji lista wynikowa może być listą oryginalną, w której poprzestawiano elementy (sortowanie „w miejscu") lub listą nowo utworzoną, zawierającą kopie elementów pierwszej listy.

```
public interface ListSorter {
    public List sort(List list);
}
```

Abstrakcyjna klasa testowa dla sortowania list

Mimo iż nie napisaliśmy jeszcze ani jednej linijki algorytmu sortującego, rozpoczniemy od stworzenia zestawu testowego weryfikującego poprawność dowolnej implementacji interfejsu `ListSorter`. Zgodnie z wcześniejszymi uwagami za podstawę konstrukcyjną wszystkich testów posłuży nam abstrakcyjna klasa `AbstractListSorterTest`, obejmująca wszelkie aspekty testowe niezależne od konkretnego algorytmu sortowania, w szczególności:

- utworzenie nieposortowanej listy łańcuchów,
- utworzenie posortowanej listy tych samych łańcuchów służącej jako wzorzec oczekiwanego rezultatu sortowania,
- utworzenie instancji klasy implementującej interfejs `ListSorter` — ta operacja wykonywana jest w ramach metody abstrakcyjnej wymagającej zdefiniowania w klasie pochodnej,
- posortowanie oryginalnej listy za pomocą metody `sort()` instancji utworzonej w poprzednim punkcie,
- porównanie wyników sortowania z wzorcem utworzonym w punkcie drugim.

spróbuj sam Tworzenie abstrakcyjnej klasy testowej

Dwie pierwsze z wymienionych przed chwilą czynności — utworzenie listy wejściowej i jej posortowanego odpowiednika — dokonywane są przez metodę `setUp()` klasy testowej.

```
package com.wrox.algorithms.sorting;

import com.wrox.algorithms.lists.LinkedList;
import com.wrox.algorithms.lists.List;
import junit.framework.TestCase;

public abstract class AbstractListSorterTest extends TestCase {
    private List _unsortedList;
    private List _sortedList;

    protected void setUp() throws Exception {
```

```
        _unsortedList = new LinkedList();
        _unsortedList.add("programowanie");
        _unsortedList.add("sterowane");
        _unsortedList.add("testami");
        _unsortedList.add("to");
        _unsortedList.add("mały");
        _unsortedList.add("krok");
        _unsortedList.add("dla");
        _unsortedList.add("programisty");
        _unsortedList.add("lecz");
        _unsortedList.add("olbrzymi");
        _unsortedList.add("skok");
        _unsortedList.add("w");
        _unsortedList.add("dziejach");
        _unsortedList.add("programowania");

        _sortedList = new LinkedList();

        _sortedList.add("dla");
        _sortedList.add("dziejach");
        _sortedList.add("krok");
        _sortedList.add("lecz");
        _sortedList.add("mały");
        _sortedList.add("olbrzymi");
        _sortedList.add("programisty");
        _sortedList.add("programowania");
        _sortedList.add("programowanie");
        _sortedList.add("skok");
        _sortedList.add("sterowane");
        _sortedList.add("testami");
        _sortedList.add("to");
        _sortedList.add("w");
    }
```

Obydwie listy zostają zwolnione przez metodę `tearDown()`:

```
    protected void tearDown() throws Exception {
        _sortedList = null;
        _unsortedList = null;
    }
```

Musimy jeszcze zadeklarować abstrakcyjną metodę tworzącą instancję implementującą interfejs `ListSorter`:

```
protected abstract ListSorter createListSorter(Comparator comparator);
```

i zdefiniować metodę wykonującą właściwy test:

```
public void testListSorterCanSortSampleList() {
    ListSorter sorter = createListSorter(naturalComparator.INSTANCE);
    List result = sorter.sort(_unsortedList);

    assertEquals(result.size(), _sortedList.size());

    Iterator actual = result.iterator();
    actual.first();
```

```
        Iterator expected = _sortedList.iterator();
        expected.first();

        while (!expected.isDone()) {
            assertEquals(expected.current(), actual.current());

            expected.next();
            actual.next();
        }
    }
```

Jak to działa?

W pierwszym wierszu tworzona jest instancja klasy realizującej określony algorytm sortowania; sortowanie odbywa się w naturalnej kolejności alfabetycznej łańcuchów — specyfikowanym komparatorem jest bowiem komparator naturalny. W drugim wierszu wspomniany algorytm jest fizycznie realizowany w testowej liście `_unsortedList`. Po zakończeniu sortowania jego wynik porównywany jest ze wzorcem: w stosunku do obydwu list — wynikowej i wzorcowej — najpierw porównywane są ich rozmiary, a następnie przy użyciu iteratorów porównywane są kolejne pary odpowiadających sobie elementów. Identyczność obydwu list jest warunkiem, który spełniać musi dowolna implementacja algorytmu sortowania, jeżeli w ogóle zamierzamy jej użyć do posortowania czegokolwiek!

Przechodząc od ogółu do szczegółów, zajmijmy się testowaniem sortowania bąbelkowego.

spróbuj sam Testowanie klasy BubbleListSorter

Testową klasę dla sortowania bąbelkowego — `BubbleListSorterTest` — wyprowadzimy z klasy abstrakcyjnej `AbstractListSorterTest`, implementując odpowiednio jej metodę `createListSorter()`.

```
package com.wrox.algorithms.sorting;

public class BubblesortListSorterTest extends AbstractListSorterTest {
    protected ListSorter createListSorter(Comparator comparator) {
        return new BubblesortListSorter(comparator);
    }
}
```

Z kompilacją powyższego kodu musimy jednak poczekać, aż zdefiniujemy klasę `BubblesortListSorter` — uczynimy to niebawem.

Jak to działa?

Klasa `BubbleListSorterTest`, mimo iż jej zdefiniowanie sprowadzało się do zdefiniowania jednej metody, dziedziczy po klasie bazowej `AbstractListSorterTest` zestaw danych testowych oraz metodę `testListSorterCanSortSampleList()` zawierającą całą „logikę testową". Konkretyzuje ona jedyny abstrakcyjny element tej logiki — metodę `createListSorter()` tworzącą instancję klasy reprezentującej algorytm sortujący.

spróbuj sam Implementowanie algorytmu sortowania bąbelkowego — klasa BubbleListSorter

Implementacja klasy realizującej algorytm sortowania bąbelkowego musi spełniać trzy następujące kryteria:

- musi implementować interfejs `ListSorter`,
- musi dopuszczać dowolny komparator określający uporządkowanie elementów,
- musi przejść pozytywnie testy opisane przed chwilą.

Mając na uwadze powyższe wymogi, rozpocznijmy od zdefiniowania konstruktora:

```
package com.wrox.algorithms.sorting;

import com.wrox.algorithms.lists.List;

public class BubblesortListSorter implements ListSorter {
    private final Comparator _comparator;

    /**
     * Konstruktor
     * parametr: komparator określający uporządkowanie elementów
     */
    public BubblesortListSorter(Comparator comparator) {
        assert comparator != null : "nie określono komparatora";
        _comparator = comparator;
    }
      ...
}
```

Teraz przed nami najważniejsze — implementacja samego algorytmu sortowania bąbelkowego. Jak pamiętamy, algorytm ten wymaga wielu przejść przez sortowaną listę; w wyniku każdego przejścia kolejny element w pobliżu końca listy ustawiany jest na swej właściwej pozycji. Wynika stąd, że dla *N*-elementowej listy po wykonaniu *N*–1 kroków na swych docelowych pozycjach znajdzie się *N*–1 końcowych elementów, a więc także i element początkowy, ergo — liczba kroków potrzebnych do posortowania dowolnej listy jest o jeden mniejsza od liczby elementów zawartych w tej liście. Kod odpowiedzialny za powtarzanie wspomnianych kroków nazwiemy *pętlą zewnętrzną* (*outer loop*).

W każdym kroku porównywane są pary sąsiadujących elementów; jeżeli względna kolejność elementów pary nie jest zgodna z kryterium określonym przez komparator, elementy zamieniane są miejscami — ten cykl nazwiemy *pętlą wewnętrzną* (*inner loop*). Ponieważ w każdym kroku kolejny element końcowy „ląduje" na swej pozycji docelowej, liczba elementów porównywanych w kolejnych krokach systematycznie się zmniejsza: w pierwszym kroku musimy wykonać *N*–1 porównań, w drugim *N*–2 itd. Wyjaśnia to warunek kontynuowania pętli wewnętrznej `left < (size - pass)`.

```
    public List sort(List list) {
        assert list != null : "nie określono listy wejściowej";

        int size = list.size();

        for (int pass = 1; pass < size; ++pass) {                // pętla zewnętrzna
```

```
            for (int left = 0; left < (size - pass); ++left) {   // pętla wewnętrzna
                int right = left + 1;
                if (_comparator.compare(list.get(left), list.get(right)) > 0) {
                    swap(list, left, right);
                }
            }
        }

        return list;
    }
```

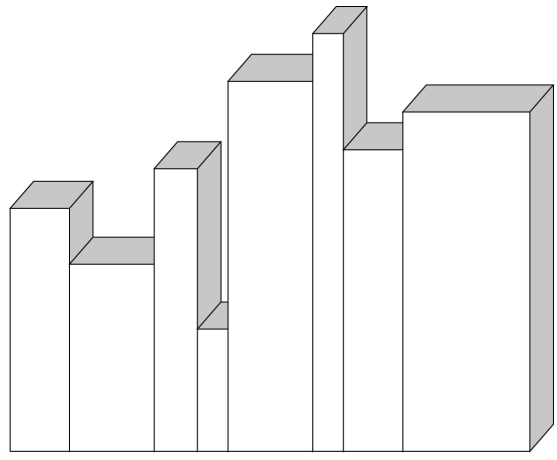
Jak przed chwilą wspomnieliśmy, jeśli kolejność sąsiadujących elementów nie jest zgodna z kryterium określonym przez komparator, elementy te zamieniane są miejscami. Musimy więc dysponować metodą zamieniającą miejscami wartości elementów o wskazanych indeksach.

```
    private void swap(List list, int left, int right) {
        Object temp = list.get(left);
        list.set(left, list.get(right));
        list.set(right, temp);
    }
```

Po zaimplementowaniu i (pomyślnym) przetestowaniu klasy `BubblesortListSorter` można celowo sprowokować załamanie testu, na przykład zmieniając wzorcową listę w taki sposób, by nie spełniała kryterium sortowania. Prędzej czy później trzeba jednak zająć się kolejnym algorytmem sortowania.

Sortowanie przez wybieranie

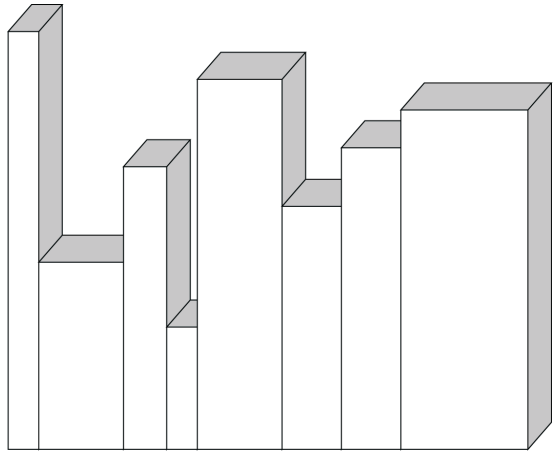
Wyobraź sobie książki o różnej wysokości, przypadkowo ułożono na półce, jak przedstawia to rysunek 6.6. Właśnie spodziewasz się odwiedzin mamy i chcesz jej zaimponować swoich zamiłowaniem do porządku domowego, postanawiasz więc poukładać książki według malejącej wysokości od lewej do prawej.



Rysunek 6.6. *Półka z losowo ustawionymi książkami*

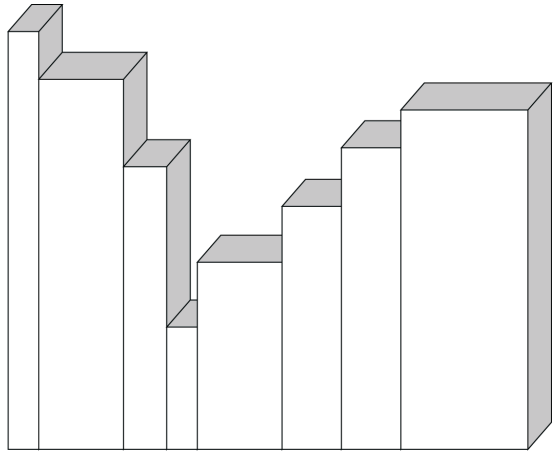
Sortowanie bąbelkowe raczej się do tego nie nada, bo przestawianie sąsiednich par byłoby stratą czasu — zamiana miejscami dwóch książek trwa bowiem znacznie dłużej niż porównanie ich wysokości. Zdecydowanie lepszą metodą na uzyskanie żądanego ułożenia książek będzie sortowanie przez wybieranie, zwane także sortowaniem przez selekcję (*selectionsort*).

Znajdź na półce najwyższą książkę i zdejmij ją z półki. Powinieneś ją ustawić jako pierwszą od lewej; zamiast przesuwać w prawo być może dużą liczbę innych książek, po prostu zamień ją z tą, która aktualnie znajduje się najbardziej na lewo (nie unikniesz całkowicie przesuwania książek, bowiem zapewne różnią się one od siebie grubością, ten szczegół nie ma jednak znaczenia w sytuacji, gdy zamiast książek sortowane są elementy listy). Opisana zamiana książek, zamiast przesuwania całej ich grupy, pozbawia sortowanie pewnej własności zwanej *stabilnością*; zajmiemy się nią w rozdziale 7., na razie jest ona bez znaczenia. Układ książek po pierwszej zamianie przedstawiony jest na rysunku 6.7.



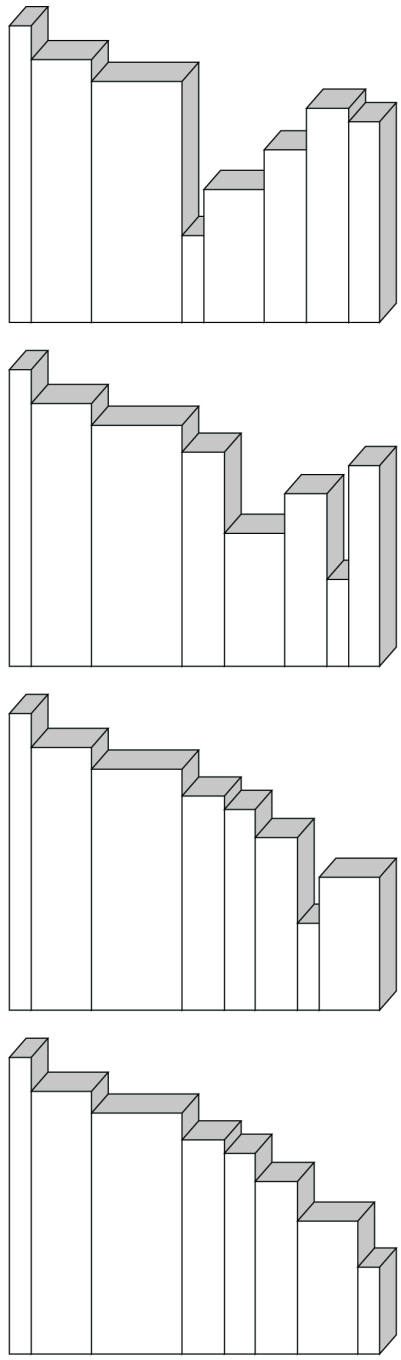
Rysunek 6.7. *Najwyższa książka znajduje się już na skrajnej lewej pozycji*

Jak łatwo się domyślić, w kolejnym kroku należy odszukać najwyższą z pozostałych książek i zamienić ją miejscami z tą, która aktualnie zajmuje pozycję drugą od lewej. Efekt tej zamiany przedstawiony jest na rysunku 6.8.



Rysunek 6.8. *Druga co do wysokości książka znajduje się na właściwej pozycji*

Kontynuując konsekwentnie to postępowanie, *wybieramy* z nieposortowanej jeszcze grupy książek najwyższą i wstawiamy ją na kolejne miejsce od lewej — dlatego właśnie opisana metoda nazywa się sortowaniem przez wybieranie. Kolejne stadia sortowania z użyciem tej metody przedstawione są na rysunku 6.9.



Rysunek 6.9. *Kolejne pozycje od lewej strony zapełniane są właściwymi książkami*

Oczywiście może się tak zdarzyć, że w którymś stadium sortowania książka będzie już znajdować się na swej pozycji docelowej i żadne przestawianie nie będzie wówczas konieczne. Tak czy inaczej nie zmienia to podstawowej własności sortowania przez wybór — tej mianowicie, że grupa elementów jeszcze nieposortowanych, początkowo obejmująca wszystkie elementy, zmniejsza się systematyczne, rozrasta się natomiast grupa elementów już posortowanych, początkowo pusta, a w końcu obejmująca wszystkie elementy. Co więcej, wybierana książka *od razu* trafia na swą docelową pozycję, w przeciwieństwie do sortowania bąbelkowego, gdzie elementy stopniowo przesuwane są małymi krokami.

Znaczna część kodu testowego stworzonego przy okazji sortowania bąbelkowego może być wykorzystana przy okazji sortowania przez wybieranie. Rozpoczniemy od stworzenia zestawu testowego, po czym zajmiemy się samym algorytmem sortowania.

spróbuj sam Testowanie klasy SelectionSortListSorter

Klasę testującą sortowanie przez wybieranie — `SelectionSortListSorterTest` — skonstruujemy w taki sam sposób jak klasę testową dla sortowania bąbelkowego — zaimplementujemy odpowiednio metodę abstrakcyjną `createListSorter()` tak, by zwracała instancję klasy `SelectionSortListSorter`.

```
package com.wrox.algorithms.sorting;

/**
 */
public class SelectionSortListSorterTest extends AbstractListSorterTest {
    protected ListSorter createListSorter(Comparator comparator) {
        return new SelectionSortListSorter(comparator);
    }
}
```

Jak to działa?

Klasa testowa `SelectionSortListSorterTest` dziedziczy po swej klasie bazowej `AbstractListSorterTest` wszystkie dane testowe i całą logikę testową. Jedynym elementem specyficznym dla sortowania przez wybieranie jest zaimplementowana metoda `createListSorter()`, dostarczająca instancji klasy realizującej algorytm sortowania.

spróbuj sam Implementowanie klasy SelectionSortListSorter

Klasa `SelectionSortListSorter` jest pod wieloma względami podobna do klasy `BubbleSortListSorter`: implementuje interfejs `ListSorter`, działa w oparciu o komparator wyznaczający kryterium sortowania i oczywiście musi pomyślnie „zaliczyć" testy przeprowadzane w oparciu o odpowiednią klasę testową. Rozpoczniemy od konstruktora klasy:

```
public class SelectionSortListSorter implements ListSorter {
    private final Comparator _comparator;

    /**
     * Konstruktor
     * parametr: komparator określający uporządkowanie elementów
     */
```

```
    public SelectionSortListSorter(Comparator comparator) {
        assert comparator != null : "nie określono komparatora";
        _comparator = comparator;
    }
      ...
}
```

Jak to działa?

Implementacja sortowania przez wybieranie ma postać dwóch zagnieżdżonych pętli — zewnętrznej i wewnętrznej — podobnie jak w przypadku sortowania bąbelkowego. Jest jednak kilka istotnych różnic, nie od razu zauważalnych. Po pierwsze, pętla zewnętrzna przebiega indeksy od 0 do *N*–2, a nie od 1 do *N*–1. Liczba kroków pozostaje ta sama, lecz zmienna sterująca pętli równa jest pozycji docelowej, na której umieszczany jest kolejny element — w pierwszym kroku jest to pozycja 0, w drugim — pozycja 1 itd. Po wykonaniu *N*–1 kroków ostatni, *N*-ty element samoczynnie znajduje się już na właściwej pozycji.

Po drugie, w pętli wewnętrznej nie dokonuje się żadnych przestawień, a jedynie wyszukuje (w grupie nieposortowanych jeszcze elementów) element o najmniejszej wartości. Co prawda jest to sytuacja odwrotna do przykładu z książkami, gdzie sortowanie następowało według *malejącej* wysokości, lecz dla algorytmu jako takiego nie ma to większego znaczenia — w razie potrzeby zawsze można użyć komparatora odwrotnego.

```
    public List sort(List list) {
        assert list != null : "nie określono listy";

        int size = list.size();

        for (int slot = 0; slot < size - 1; ++slot) {
            int smallest = slot;
            for (int check = slot + 1; check < size; ++check) {
                if (_comparator.compare(list.get(check), list.get(smallest)) < 0) {
                    smallest = check;
                }
            }
            swap(list, smallest, slot);
        }

        return list;
    }
```

Po trzecie, istnieje pewna drobna, lecz istotna różnica w procedurze przestawiającej elementy. Może się otóż zdarzyć, że kolejny element będzie się już znajdował na swoim miejscu i przestawianie go (z samym sobą) będzie niepotrzebne (w sortowaniu bąbelkowym sytuacja taka nie mogła się zdarzyć, bowiem przestawianie dotyczyło zawsze *sąsiadujących* elementów). Metoda `swap()` sprawdza więc każdorazowo, czy elementy specyfikowane do przestawienia są istotnie różne:

```
    private void swap(List list, int left, int right) {
        if (left == right) {     // czy istotnie chodzi o różne elementy?
            return;              // nie, nic nie rób.
        }
```

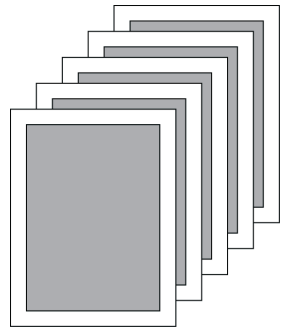
```
            Object temp = list.get(left);
            list.set(left, list.get(right));
            list.set(right, temp);
        }
```

Sortowanie przez wstawianie

Sortowanie przez wstawianie (*insertionsort*) charakterystyczne jest dla układania trzymanych w ręku kart w kolejności wzrastającej ważności. Załóżmy, że leży przed Tobą pięć odwróconych kart (rys. 6.10), które chciałbyś posortować według następującego kryterium:

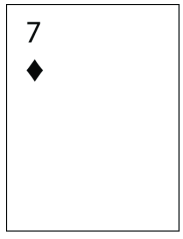
- najpierw piki (♠), potem trefle (♣), potem kara (♦), a na końcu kiery (♥),
- w ramach danego koloru as (**A**), 2, 3, …, 10, walet (**J**), dama (**Q**) i król (**K**).

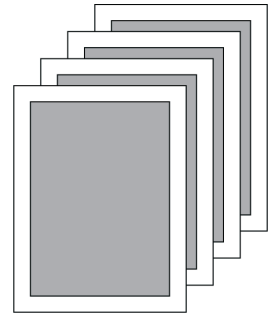
Rysunek 6.10. *„Ręka karciana” — pięć nieznanych jeszcze kart*



Odkrywamy pierwszą kartę; nie ma nic prostszego jak „posortowanie” jednego elementu, więc po prostu odkładamy kartę do grupy elementów posortowanych. W sytuacji na rysunku 6.11 odkrytą kartą jest siódemka karo.

Rysunek 6.11. *Pojedyncza karta jest zawsze „posortowana”*





Niech druga odkryta karta będzie waletem pik (rysunek 6.12). Według przyjętego kryterium poprzedza ona siódemkę karo, wstawiamy ją więc na pierwszą pozycję.

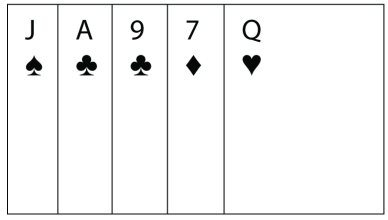
Trzecia karta okazuje się być asem trefl i według przyjętej kolejności plasuje się między dwiema już odkrytymi (rysunek 6.13).

Rysunek 6.12. *Druga karta zostaje wstawiona przed pierwszą*

Rysunek 6.13. *Trzecia karta zostaje wstawiona między dwie pozostałe*

Jak więc widzimy, sortowanie przez wstawianie polega na podziale sortowanych elementów na dwie grupy: posortowaną (początkowo pustą) i nieposortowaną (obejmującą początkowo wszystkie elementy). W każdym z kolejnych kroków z grupy nieposortowanej brany jest kolejny element i wstawiany na odpowiednie miejsce do grupy posortowanej — tak by pozostała ona nadal posortowana. W ten sposób grupa nieposortowana stopniowo się zmniejsza, a grupa posortowana powiększa się, by w końcu objąć wszystkie elementy — jak na rysunku 6.14, po odkryciu wszystkich pięciu kart.

Rysunek 6.14. *Odkrycie przedostatniej i ostatniej karty*



spróbuj sam Testowanie klasy InsertionSortListSorter

Podobnie jak w przypadku dwóch poprzednich algorytmów sortowania klasę testową wyprowadzimy z abstrakcyjnej klasy `AbstractListSorterTest`, konkretyzując jej metodę `createListSorter()`.

```
package com.wrox.algorithms.sorting;

public class InsertionSortListSorterTest extends AbstractListSorterTest {
    protected ListSorter createListSorter(Comparator comparator) {
        return new InsertionSortListSorter(comparator);
    }
}
```

Jak to działa?

Tak jak poprzednio klasa testowa (`InsertionSortListSorterTest`) dziedziczy po swej klasie bazowej `AbstractListSorterTest` wszystkie dane testowe i całą logikę testową. Jedynym elementem specyficznym dla sortowania przez wstawianie jest zaimplementowana metoda `createListSorter()`, dostarczająca instancji klasy realizującej algorytm sortowania.

spróbuj sam Implementowanie klasy InsertionSortListSorter

Podobnie jak dwie poprzednie klasy implementujące algorytmy sortowania klasa `InsertionSortListSorter` implementuje interfejs `ListSorter`, jej działanie opiera się na porządku wyznaczanym przez komparator i może być weryfikowane za pomocą odpowiedniej klasy testowej.

```
package com.wrox.algorithms.sorting;

import com.wrox.algorithms.lists.List;
import com.wrox.algorithms.lists.LinkedList;
import com.wrox.algorithms.iteration.Iterator;

public class InsertionSortListSorter implements ListSorter {
    private final Comparator _comparator;

   /**
     * Konstruktor
     * parametr: komparator określający uporządkowanie elementów
     */

    public InsertionSortListSorter(Comparator comparator) {
        assert comparator != null : "nie określono komparatora";
        _comparator = comparator;
    }
      ...
}
```

Metoda `sort()` klasy `InsertionSortListSorter` różni się zasadniczo od tej implementowanej w klasach `BubbleSortListSorter` i `SelectionSortListSorter` pod jednym względem: zamiast sortowania zawartości listy „w miejscu” tworzymy nową, pustą listę wynikową i sukcesywnie wstawiamy do niej (na właściwą pozycję) elementy pobierane kolejno z listy wejściowej.

```
    public List sort(List list) {
        assert list != null : "nie określono listy wejściowej";

        final List result = new LinkedList();
```

```
        Iterator it = list.iterator();
for (it.first(); !it.isDone(); it.next()) {          // pętla zewnętrzna
            int slot = result.size();
            while (slot > 0) {                                  // pętla wewnętrzna
                if (_comparator.compare(it.current(), result.get(slot - 1)) >= 0) {
                    break;
                }
                --slot;
            }
            result.insert(slot, it.current());
        }
        return result;
    }
```

Jak to działa?

W zewnętrznej pętli `for` za pomocą iteratora pobierane są kolejne elementy listy wejściowej; użycie iteratora jest rozwiązaniem bardziej uniwersalnym niż bezpośredni dostęp do elementów na podstawie ich indeksów. W pętli wewnętrznej — która nie jest pętlą `for`, lecz pętlą `while` — w (stopniowo zapełnianej) liście wynikowej poszukiwana jest pozycja, na którą należy wstawić element pobrany z listy wejściowej. W przeciwieństwie do listy wejściowej, której implementacja jest bez znaczenia, lista wynikowa jest listą wiązaną `LinkedList`, a dostęp do jej elementów odbywa się w sposób bezpośredni. Wybraliśmy listę wiązaną ze względu na efektywność, z jaką można wstawiać do niej elementy. Lista wynikowa pozostaje cały czas posortowana, a po wstawieniu do niej ostatniego elementu sortowanie się kończy.

Zwróćmy ponadto uwagę, że poszukiwanie (w pętli wewnętrznej) właściwej pozycji w liście wynikowej rozpoczyna się od jej końca. Mimo iż nie wpływa to na wydajność sortowania *przeciętnej* listy, to jednak drastycznie poprawia tę wydajność w przypadku, gdy lista wejściowa jest *już posortowana* (lub prawie posortowana) — wstawienie elementu (a właściwie jego dołączenie) odbywa się już po wykonaniu *jednego* porównania. Powrócimy do tej kwestii przy okazji porównywania prostych algorytmów sortowania w dalszej części niniejszego rozdziału. Kierunek przeglądania posortowanej listy wynikowej *nie* jest natomiast obojętny z punktu widzenia *stabilności* sortowania.

Stabilność sortowania

Niektóre algorytmy sortowania cechują się interesującą własnością zwaną *stabilnością*. Aby zrozumieć jej istotę, rozpatrzmy listę pracowników posortowaną według imion (tabela 6.1).

Załóżmy teraz, że chcemy posortować powyższą listę według nazwisk. Ponieważ niektóre nazwiska się powtarzają (Smith i Barnes), można to zrobić na kilka sposobów i ostateczna kolejność może być różna dla różnych algorytmów sortowania. Ponieważ pozycje o jednakowych nazwiskach występować mogą w dowolnej kolejności względem siebie, więc w ramach tego samego nazwiska posortowanie według imion może zostać zachowane lub nie. Innymi słowy, algorytm sortowania może, lecz nie musi zachowywać istniejącą *względną kolejność pozycji osób o tym samym nazwisku*. Te algorytmy, które kolejność tę zachowują, nazywamy algorytmami *stabilnymi*. Efekt posortowania listy z tabeli 6.1 w sposób stabilny przedstawiony jest w tabeli 6.2.

Tabela 6.1. *Lista posortowana według imion*

Imię	Nazwisko
Albert	Smith
Brian	Jackson
David	Barnes
John	Smith
John	Wilson
Mary	Smith
Tom	Barnes
Vince	De Marco
Walter	Clarke

Tabela 6.2. *Lista z tabeli 6.1 stabilnie posortowana według nazwisk*

Imię	Nazwisko
David	Barnes
Tom	Barnes
Walter	Clarke
Vince	De Marco
Brian	Jackson
Albert	Smith
John	Smith
Mary	Smith
John	Wilson

Przykład niestabilnego posortowania wspomnianej listy według nazwisk przedstawiony jest w tabeli 6.3 — w ramach nazwiska `Smith` nie została zachowana oryginalna kolejność imion.

Tabela 6.3. *Lista z tabeli 6.1 posortowana według nazwisk w sposób niestabilny*

Imię	Nazwisko
David	Barnes
Tom	Barnes
Walter	Clarke
Vince	De Marco
Brian	Jackson
Albert	Smith
Mary	Smith
John	Smith
John	Wilson

Spośród trzech opisanych dotąd algorytmów sortowania algorytmem stabilnym jest sortowanie bąbelkowe. To, czy stabilne jest sortowanie przez wstawianie, zależne jest od kolejności pobierania elementów z listy wejściowej i sposobu ich wstawiania do listy wynikowej; prezentowana przez nas implementacja *jest* implementacją stabilną. Podobnie stabilność sortowania przez wybieranie zależy od szczegółów jego implementacji. Omawiane w następnym rozdziale bardziej zaawansowane algorytmy sortowania, choć cechują się znacząco lepszą efektywnością, *nie* są algorytmami stabilnymi i jest to jedna z ich wad w porównaniu z prostymi algorytmami sortowania, o czym trzeba pamiętać przy tworzeniu aplikacji o konkretnych wymaganiach.

Porównanie prostych algorytmów sortowania

Po zapoznaniu się z trzema prostymi algorytmami sortowania — bąbelkowego, przez wybieranie i przez wstawianie — nie sposób nie zastanawiać się, który z nich okaże się najlepszy w danym zastosowaniu, a dokładniej — jakimi kryteriami należy się kierować przy dokonywaniu jego wyboru. W niniejszym podrozdziale dokonamy porównania wymienionych algorytmów; nie będzie to formalne porównanie matematyczne, lecz porównanie praktyczne oparte na obserwacji sortowania rzeczywistych danych. Nie jest naszym zadaniem definitywne sformułowanie kryteriów wyboru konkretnego algorytmu, lecz raczej pokazanie, jak wspomniana analiza porównawcza może dokonanie takiego wyboru ułatwić.

Na początku tego rozdziału informowaliśmy, ze istotą każdego sortowania jest intensywne wykonywanie dwóch operacji: porównywania elementów i ich przestawiania. Nasza analiza porównawcza koncentrować się będzie na pierwszej z tych operacji, a używane na jej potrzeby zestawy danych będą znacznie większe niż w zestawach testowych weryfikujących poprawność implementacji algorytmów; jest to konieczne z tego względu, że prawdziwy charakter każdego algorytmu, odzwierciedlany głównie przez jego zachowanie asymptotyczne wyrażone w notacji dużego *O*, uwidacznia się dopiero przy rozwiązywaniu problemów o dużych rozmiarach. Ponadto, ponieważ konkretne dane wejściowe algorytmu mają zwykle wpływ na jego efektywność, analizę naszą przeprowadzimy w oparciu o trzy szczególne rodzaje danych wejściowych:

- listę już posortowaną,
- listę posortowaną w kolejności odwrotnej do żądanej,
- listę o przypadkowej kolejności elementów.

Obserwując zachowanie się — czyli zliczając wykonywane porównania — wszystkich trzech algorytmów dla każdego z wymienionych przypadków, będzie można w przybliżeniu ocenić, który algorytm nadaje się najlepiej dla danego przypadku napotkanego w rzeczywistej aplikacji.

CallCountingListComparator

Ponieważ za wszystkie porównania, jakie wykonywane są w ramach algorytmu sortowania, odpowiedzialny jest komparator, a dokładniej — jego metoda `compare()`, najprostszym sposobem zliczania porównań wydaje się przechwycenie wywołania tej metody, czyli wzbogacenie jej o fragment kodu dokonujący zliczania wszystkich wywołań. Można by też posunąć się jeszcze dalej i wyposażyć w taki mechanizm zliczania w jakąś klasę bazową, z której wyprowadzane byłby wszystkie „zliczające" komparatory. Wymagałoby to jednak ponownego zaimplementowania od podstaw tych komparatorów, które chcemy uczynić zliczającymi. Chcąc wykorzystać w jak największym stopniu istniejący kod, postąpimy więc inaczej i funkcję zliczającą komparatora zrealizujemy za pomocą jego otoczki („dekoratora"), podobnie jak czyniliśmy to w przypadku odwracania uporządkowania za pomocą klasy `ReverseComparator`.

```
public final class CallCountingComparator implements Comparator {
    /** komparator oryginalny, który wyposażamy w funkcję zliczania */
    private final Comparator _comparator;

    /** zmienna przechowująca liczbę zarejestrowanych wywołań komparatora */
    private int _callCount;

    /**
     * Konstruktor.
     * Parametr: oryginalny komparator
     */
    public CallCountingComparator(Comparator comparator) {
        assert comparator != null : "nie określono komparatora";

        _comparator = comparator;
        _callCount = 0;
    }

    public int compare(Object left, Object right) throws ClassCastException {
        ++_callCount;
        return _comparator.compare(left, right);
    }

    public int getCallCount() {
        return _callCount;
    }
}
```

Podobnie jak komparator odwrotny `ReverseComparator`, tak i komparator zliczający `CallCountingComparator` definiowany jest na bazie dowolnego komparatora przekazywanego jako parametr wywołania konstruktora. Wywołanie metody `compare()` komparatora zliczającego jest rejestrowane poprzez zwiększenie wartości zmiennej `_callCount`, po czym delegowane jest do metody `compare()` komparatora oryginalnego. Wartość zmiennej `_callCount`, równa liczbie dokonanych wywołań, dostępna jest za pośrednictwem metody `getCallCount()`.

Mając do dyspozycji komparator zliczający, możemy tworzyć zestawy testowe badające zachowanie się poszczególnych algorytmów sortowania w odniesieniu do danych o różnym charakterze.

ListSorterCallCountingTest

Mimo iż tym razem nie zamierzamy testować poprawności zachowania się kodu, lecz mierzyć liczbę porównań wykonywanych przez algorytmy sortowania, skorzystamy z biblioteki `JUnit`, bowiem podobnie jak w przypadku testów modułów będziemy musieli wykonać kilka dyskretnych scenariuszy dla każdego algorytmu poprzedzonych pewnymi czynnościami przygotowawczymi (*setup*). Zdefiniujemy więc klasę testową, a w ramach niej stałą określającą rozmiar sortowanej listy, trzy listy o charakterystykach wcześniej wymienionych (posortowaną, posortowaną odwrotnie i nieposortowaną) oraz instancję komparatora zliczającego.

```
package com.wrox.algorithms.sorting;

import com.wrox.algorithms.lists.ArrayList;
import com.wrox.algorithms.lists.List;
import junit.framework.TestCase;

public class ListSorterCallCountingTest extends TestCase {
    private static final int TEST_SIZE = 1000;

    // lista posortowana
    private final List _sortedArrayList = new ArrayList(TEST_SIZE);

    // lista posortowana odwrotnie
    private final List _reverseArrayList = new ArrayList(TEST_SIZE);

    // lista o przypadkowej kolejności elementów
    private final List _randomArrayList = new ArrayList(TEST_SIZE);

    private CallCountingComparator _comparator;
      ...
}
```

Samo zdefiniowanie list `_sortedArrayList`, `_reverseArrayList` i `_randomArrayList` to dopiero początek, musimy bowiem wypełnić te listy wartościami w sposób odpowiadający ich założonej charakterystyce. Zakładamy, że elementami tymi będą liczby całkowite, czyli obiekty typu `integer`. W przypadku dwóch pierwszych list będą to kolejne liczby naturalne od 1 do 1 000 uszeregowane w kolejności (odpowiednio) rosnącej i malejącej; w przypadku trzeciej listy będą to losowe liczby całkowite z tego zakresu. Musimy także zdefiniować komparator zliczający, który oprzemy na komparatorze naturalnym (`NaturalComparator`). Jest to dopuszczalne, bowiem typ `java.lang.integer` implementuje interfejs `Comparable`, podobnie jak implementują go łańcuchy wykorzystywane we wcześniejszych przykładach.

```
    protected void setUp() throws Exception {
        super.setUp();
        _comparator = new CallCountingComparator(NaturalComparator.INSTANCE);

        for (int i = 1; i < TEST_SIZE; ++i) {
            _sortedArrayList.add(new Integer(i));
        }

        for (int i = TEST_SIZE; i > 0; --i) {
            _reverseArrayList.add(new Integer(i));
```

```
        }

        for (int i = 1; i < TEST_SIZE; ++i) {
            _randomArrayList.add(new Integer((int)(TEST_SIZE * Math.random())));
        }
    }
```

By zaobserwować działanie każdego algorytmów dla listy posortowanej odwrotnie, należy utworzyć kolejno trzy odpowiednie implementacje interfejsu `ListSorter` i użyć każdej z nich do posortowania listy `_reverseArrayList` utworzonej w ramach metody `setUp()`. Wnikliwy Czytelnik mógłby w tym miejscu stwierdzić, że po pierwszym posortowaniu listy `_reverseArrayList` dalsze sortowania nie mają sensu, bo lista ta *przestanie być lista posortowaną odwrotnie*. Otóż jest zupełnie inaczej: lista `_reverseArrayList` *tworzona jest na nowo* przed każdym z sortowań — przed wywołaniem każdej z metod testowych wywoływana jest metoda `setUp()` i to jest główny powód, dla którego użyliśmy biblioteki `JUnit` w zastosowaniu niemającym nic wspólnego z weryfikacją poprawności kodu. Dzięki temu wszystkie trzy metody testowe działają niezależnie od siebie.

```
    public void testReverseCaseBubblesort () {
        new BubblesortListSorter(_comparator).sort(_reverseArrayList);
        reportCalls();
    }

    public void testReverseCaseSelectionSort () {
        new SelectionSortListSorter(_comparator).sort(_reverseArrayList);
        reportCalls();
    }

    public void testReverseCaseInsertionSort () {
        new InsertionSortListSorter(_comparator).sort(_reverseArrayList);
        reportCalls();
    }
```

Wyniki obserwacji każdego z sortowań, czyli informacja o liczbie wywołań metody `compare()` odnośnego komparatora, wyświetlane są za pomocą metody `reportCalls()`, którą opiszemy za chwilę. W podobny sposób przeprowadzimy obserwację dla listy posortowanej w żądanej kolejności…

```
    public void testDirectCaseBubblesort () {
        new BubblesortListSorter(_comparator).sort(_sortedArrayList);
        reportCalls();
    }

    public void testDirectCaseSelectionSort () {
        new SelectionSortListSorter(_comparator).sort(_sortedArrayList);
        reportCalls();
    }

    public void testDirectCaseInsertionSort () {
        new InsertionSortListSorter(_comparator).sort(_sortedArrayList);
        reportCalls();
    }
```

… i dla listy o losowym układzie elementów:

```
    public void testRandomCaseBubblesort () {
        new BubblesortListSorter(_comparator).sort(_randomArrayList);
        reportCalls();
    }

    public void testRandomCaseSelectionSort () {
        new SelectionSortListSorter(_comparator).sort(_randomArrayList);
        reportCalls();
    }

    public void testRandomCaseInsertionSort () {
        new InsertionSortListSorter(_comparator).sort(_randomArrayList);
        reportCalls();
    }
```

Wspomniana wcześniej metoda `reportCalls()` odczytuje — za pomocą metody `callCount()` — licznik dokonanych porównań i wyprowadza jego wartość poprzedzoną nazwą klasy testowej:

```
    private void reportCalls() {
        System.out.println(getName() + ": " + _comparator.getCallCount() + " wywołań");
    }
```

Nazwa klasy testowej — jak łatwo się zorientować — udostępniana jest przez metodę `getName()`, która jest metodą klasy bazowej `TestCase` biblioteki `JUnit`. Oto przykładowy raport dla listy posortowanej odwrotnie:

```
testReverseCaseBubblesort: 499500 wywołań
testReverseCaseSelectionSort: 499500 wywołań
testReverseCaseInsertionSort: 499500 wywołań
```

Jak widać, wszystkie trzy algorytmy sortowania wykonały taką samą liczbę porównań dla listy — wygląda na to, że jest ona „jednakowo trudnym" przypadkiem dla każdego z nich. Nie należy jednak przyjmować tego jako reguły, a w przypadku danych o charakterze wyłącznie empirycznym (jak tutaj) należy wystrzegać się formułowania pochopnych, być może z gruntu fałszywych wniosków, choć oczywiście nie można nie zastanawiać się nad przyczynami obserwowanych faktów.

Wyniki analogicznej analizy dla listy już posortowanej wyglądają zgoła odmiennie:

```
testDirectCaseBubblesort: 498501 wywołań
testDirectCaseSelectionSort: 498501 wywołań
testDirectCaseInsertionSort: 998 wywołań
```

Tak duża wrażliwość sortowania przez wstawianie na fakt posortowania listy wejściowej nie powinna być zaskoczeniem. Jej przyczynę wyjaśnialiśmy wcześniej — jest nią szczególny sposób przeszukiwania listy wynikowej, począwszy od jej końca, nie początku.

Na koniec pozostaje porównanie zachowania się algorytmów sortowania dla typowej, nieuporządkowanej listy:

```
testRandomCaseBubblesort: 498501 wywołań
testRandomCaseSelectionSort: 498501 wywołań
testRandomCaseInsertionSort: 262095 wywołań
```

Algorytm sortowania przez wstawianie wykonuje, jak widać, dwukrotnie mniej porównań niż każdy z dwóch pozostałych algorytmów.

Jak interpretować wyniki tej analizy?

Z przeprowadzonej analizy porównawczej powinniśmy oczywiście wyciągnąć pewne wnioski, musimy jednak być przy tym świadomi warunków, w jakich analiza ta została przeprowadzona. Aby mianowicie poznać prawdziwe oblicze każdego z algorytmów, należałoby wzbogacić tę analizę o co najmniej następujące elementy:

- zliczanie także operacji przestawiania elementów, a nie tylko operacji ich porównywania,
- wykorzystanie różnych implementacji list, na przykład tablicowej, a nie tylko wiązanej,
- pomiar rzeczywistego czasu wykonania każdego z sortowań.

Mimo wspomnianych ograniczeń możemy jednak pokusić się o następujące ustalenia:

- Algorytmy sortowania bąbelkowego i sortowania przez wybieranie wykonują zawsze tę samą liczbę porównań dla tych samych danych wejściowych.
- Liczba operacji wykonywanych zarówno w sortowaniu bąbelkowym, jak i w sortowaniu przez wybieranie, jest niezależna od charakteru sortowanych danych.
- Liczba operacji wykonywanych w sortowaniu przez wstawianie jest w dużym stopniu zależna od charakteru sortowanych danych. W najgorszym przypadku liczba ta jest równa liczbie porównań wykonywanych przez dwa pozostałe algorytmy (dla tych samych danych), w najlepszym przypadku jest ona mniejsza od liczby sortowanych elementów.

Być może najważniejszym wnioskiem podsumowującym analizę jest niewrażliwość sortowania bąbelkowego i sortowania przez wybieranie na charakter sortowanych danych. W przeciwieństwie do nich sortowanie przez wstawianie wykazuje duże zdolności adaptacyjne: jeśli można posortować dane mniejszym nakładem pracy, algorytm istotnie wykorzystuje tę możliwość. Jest to główną przyczyną faworyzowania w praktyce sortowania przez wstawianie w stosunku do sortowania bąbelkowego i sortowania przez wybieranie.

Podsumowanie

W niniejszym rozdziale:

- zaimplementowaliśmy trzy proste algorytmy sortowania — bąbelkowe, przez wybieranie i przez wstawianie — i zweryfikowaliśmy poprawność ich implementacji za pomocą odpowiednich zestawów testowych,
- opisaliśmy koncepcję komparatora i zaimplementowaliśmy kilka komparatorów — komparator naturalny, komparator odwrotny i komparator zliczający,

- porównaliśmy liczbę porównań wykonywanych przez każdy z trzech wymienionych algorytmów sortowania dla trzech szczególnych rodzajów danych wejściowych: listy już posortowanej, listy posortowanej odwrotnie i listy o losowym układzie elementów oraz sformułowaliśmy ogólne wnioski na temat charakteru każdego z algorytmów,
- wyjaśniliśmy pojęcie stabilności algorytmu sortowania.

Lektura niniejszego rozdziału z pewnością pozwoli Czytelnikom lepiej zrozumieć znaczenie sortowania dla innych czynności algorytmicznych, na przykład wyszukiwania. Treść rozdziału jest ponadto dowodem na to, że rozmaite problemy algorytmiczne mogą być rozwiązywane na różne sposoby — w szczególności istnieje kilka różnych metod porządkowania elementów w zadanej kolejności. W następnym rozdziale poznamy inne, bardziej złożone metody sortowania, które dla bardzo dużych rozmiarów danych wejściowych okazują się znacznie efektywniejsze od metod dotychczas opisanych.

Ćwiczenia

1. Stwórz zestawy testowe weryfikujące poprawność sortowania — przez każdy z algorytmów — losowo wygenerowanej listy obiektów typu double.
2. Stwórz zestawy testowe udowadniające, że sortowanie bąbelkowe i sortowanie przez wstawianie (w implementacjach prezentowanych w niniejszym rozdziale) są stabilnymi metodami sortowania.
3. Skonstruuj komparator wyznaczający alfabetyczną kolejność łańcuchów, bez rozróżniania małych i wielkich liter.
4. Napisz program-sterownik zliczający liczbę przestawień obiektów w ramach każdego z opisywanych w rozdziale algorytmów sortowania.