Państwowa Wyższa Szkoła Zawodowa w Kaliszu

Zbigniew Szpunar

wykład 6.
Algorytmy sortowania

Algorytmy sortowania

Sortowanie (porządkowanie) danych według pewnego kryterium jest elementem (nierzadko podstawowym) wielu algorytmów przetwarzania danych uporządkowanych.

Algorytmy przetwarzania posortowanych danych są stosowane często intuicyjnie, przez każdego, np..:

- wyszukiwanie osoby na liście alfabetycznej (by dotrzeć do danych tej osoby, np. do numeru telefonu tej osoby, w książce telefonicznej z danego województwa, umieszczonej na półce z książkami telefonicznymi wszystkich województw),
- szukanie haseł w encyklopedii, słowniku,
- wyszukanie oceny z kolokwium na liście uporządkowanej wg rosnących wartości numeru albumu,
- itp.

Algorytmy, działając na na uporządkowanych danych, zyskują na efektywności.

Dane mogą być różnorodne, a o sposobie uporządkowania decyduje kryterium dostosowane do postaci tych danych.

Warunkiem stosowania sortowania danych jest dostępność struktury danych zdolnej do ich przechowania, z zachowaniem względnej kolejności tych danych, np.:

- listy,
- tablice.

Algorytmy sortowania

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

- porównaniu elementów (w celu stwierdzenia zgodności ich względnej kolejności w odniesieniu do kryterium sortowania),
- przesuwaniu elementów na pozycje określone przez kryterium sortowania.

Istnieje wiele różnych algorytmów sortowania. Ich porównywanie może być dokonywane według różnorodnych kryteriów, np.:

- stopnia skomplikowania algorytmu,
- złożoności obliczeniowej (optymistycznej, przeciętnej i pesymistycznej – rzutuje to na "szybkość" sortowania),
- złożoności pamięciowej (zapotrzebowania na dodatkową pamięć),
- stabilności algorytmu (zachowywaniu pierwotnej kolejności elementów o jednakowych kluczach),
- przydatności do sortowania w odniesieniu do różnych struktur danych (tablice, listy, pliki).

Każdy z algorytmów sortowania może mieć wiele różnych (alternatywnych) realizacji.

Nie ma, w zasadzie, jednego, najlepszego algorytmu sortowania.

Państwowa Wyższa Szkoła Zawodowa w Kaliszu
Zbigniew Szpunar

Zagadnienie doboru algorytmu sortowania

Dobór algorytmu (oraz jego realizacji) oznacza konieczność uwzględnienia różnych czynników, w tym między innymi:

- rodzaj i wielkość porządkowanej struktury (tablica, lista, plik),
- złożoność operacji porównania kluczy,
- złożoność operacji przesuwania elementów (złożoność struktury, wielkość elementu),
- stan początkowy uporządkowania danych, np.:
  - dane uporządkowane zgodnie z przyjętym kryterium porządkowania,
    { 1 3 4 6 9 } (przed sortowaniem: "rosnąco")
  - dane uporządkowane losowo,
    { 1 4 3 9 6 }
  - dane prawie uporządkowane,
    { 1 4 3 6 9 }
  - dane odwrotnie uporządkowane,
    { 9 6 4 3 1 }
  - dane o równych wartościach),
    { 1 2 1 3 2 }
- wielkość dostępnej pamięci operacyjnej,
- wymagania dotyczące stabilności,
- ...

Operacja porównania w algorytmach sortowania

Większość algorytmów ustala kolejność elementów na podstawie porównań ich wartości.

Porównywanie jest dokonywane na podstawie przyjętego kryterium, dostosowanego do postaci (struktury) sortowanych danych.

Porównywanie w dziedzinie podstawowych typów danych jest oczywiste – działają tu operatory relacyjne. W przypadku porównywania złożonych obiektów wygodnie jest zdefiniować mechanizm porównania: komparator.

Komparator wykonuje jedną operację: określa wzajemną kolejność dwóch elementów (obiektów), w sensie przyjętego kryterium sortowania.

Wynikiem operacji jest:

- wartość ujemna (np. –1), jeśli pierwszy z dwóch porównywanych elementów jest mniejszy, niż drugi,
- zero (0), jeśli są równe,
- wartość dodatnia (np. 1), jeśli pierwszy z dwóch porównywanych elementów jest większy, niż drugi,

Jeśli typ któregokolwiek z porównywanych dwóch elementów wyklucza możliwość porównania, wówczas próba porównania powoduje wystąpienie wyjątku ClassCastException.

Interfejs komparatora

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

Metoda compare ma sens uogólnionego operatora porównania (określenia porządku) dwóch obiektów: lewego i prawego: czy lewy jest "mniejszy", "równy" czy "większy" od prawego, w sensie przyjętego kryterium porównania.

Interfejs Comparator wprowadza oddzielenie kryterium sortowania od samego algorytmu sortowania. Płyną stąd różne korzyści, np.:

- algorytm może być wyposażany (np. rozszerzany) w różne "wtyczki", określające różne kryteria sortowania,
- różne algorytmy mogą być miarodajnie porównywane pod względem wydajności (przy tej samej złożoności operacji porównywania elementów w tych algorytmach).

Możliwość definiowania różnych komparatorów prowadzi do uproszczenia kodu aplikacji.

Komparator naturalny

Komparator utworzony w oparciu o interfejs Comparable:

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

narzuca tzw. porządek naturalny, dlatego nazywa się go komparatorem naturalnym. Aby sortować w porządku naturalnym, elementy sortowane muszą implementować interfejs Comparable, czyli mieć zdefiniowaną metodę compareTo, dostarczającą:

- wartość int ujemną, gdy obiekt (this) jest mniejszy, niż other,
- wartość int 0, gdy obiekt (this) = other,
- wartość int dodatnią, gdy obiekt (this) jest większy, niż other.

Przykład definicji komparatora naturalnego:

```
package sorting;
public final class NaturalComparator implements Comparator {
   // wykorzystuje wzorzec singleton – jednoinstancyjnej klasy:
   public static final NaturalComparator INSTANCE = new NaturalComparator();
   private NaturalComparator() { } // konstruktor prywatny, by uniemożliwić
                                   // dowolne tworzenie dalszych instancji
   public int compare(Object left, Object right) throws ClassCastException
   { return ((Comparable) left).compareTo(right); }
}
```

Komparator odwrotny

Komparator odwrotny - komparator dostosowany do porządkowania w kolejności odwrotnej obiektów uporządkowanych w danej kolejności.

```
package sorting;
public class ReverseComparator implements Comparator {
// podstawowy komparator
    private final Comparator  _comparator;

    public ReverseComparator(Comparator comparator)
    { _comparator = comparator; }

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

Takie podejście uwalnia programistę od znajomości implementacji komparatora "oryginalnego", w tym: od znajomości szczegółów (np. atrybutów) porównywanych obiektów, gdy chce zdefiniować komparator odwrotny.

Komparator złożony

W przypadkach wymagających użycia złożonego komparatora można skomplikować metodę compare lub zbudować komparator złożony, korzystający z istniejących, prostych komparatorów, np.:

```
package sorting;
import iterators.Iterator; import lists.ArrayList; import lists.List;
public class CompoundComparator implements Comparator {
    // tablica komparatorów: od najważniejszego do najmniej ważnego:
    private final List _comparators = new ArrayList();

    public void addComparator(Comparator comparator)
    { _comparators.add(comparator); }

    public int compare(Object left, Object right) throws ClassCastException {
        int result = 0;
        Iterator i = _comparators.iterator();
        for (i.first();
              !i.isDone()&&(result=((Comparator) i.current()).compare(left, right))==0;
              i.next());
        return result;
    }
}
```

Algorytmy sortowania o kwadratowej złożoności średniej

1. Algorytm **sortowania przez zamianę** (BubbleSort, sortowanie bąbelkowe)

Ideę tego sortowania (oraz proste implementacje) znamy z poprzedniego semestru:

Powtarzaj, aż do całkowitego uporządkowania ciągu, operację:
"porównaj parę sąsiednich elementów i jeśli ich kolejność jest nieprawidłowa to zamień te elementy miejscami"

Podstawowa realizacja algorytmu:

Dla wygody zdefiniujmy interfejs ułatwiający wykorzystanie algorytmów sortowania :

```
package sorting;
import lists.List;

public interface ListSorter {
   public List sort(List list); // wynikiem jest lista posortowana
}
```

Algorytm sortowania bąbelkowego

```
package sorting; import lists.List;
public class BubbleSort implements ListSorter {
    private final Comparator _comparator;
    public BubbleSort(Comparator comparator) { _comparator = comparator; }
      // wynikiem jest posortowana lista pierwotna
      //  najbardziej prymitywna wersja – bez przyspieszania zakończenia
    public List sort(List list) {
        int size = list.size();
         for (int pass = 1; pass < size; ++pass) {
            for (int left = 0; left < (size - pass); ++left) {
                int right = left + 1;
                if (_comparator.compare(list.get(left), list.get(right)) > 0)
                   swap(list, left, right);
            }
         }
         return list;
     }
     private void swap(List list, int left, int right) {
        Object temp = list.get(left);
        list.set(left, list.get(right));
        list.set(right, temp);
    }
}
```

Algorytm sortowania bąbelkowego

Algorytm jest prosty, ale powolny.

Można go przyspieszyć:

- ograniczając pętlę zewnętrzną poprzez wykrycie zaistnienia uporządkowania wcześniej, niż po *size-1* krokach,
- ograniczając pętlę wewnętrzną przez zapamiętanie pozycji ostatniej zamiany w poprzednim przebiegu,
- zastępując ciąg zamian elementów ciągiem przepisań,
- przechodząc ciąg na przemian w obu kierunkach (ShakerSort).

Algorytm sortowania bąbelkowego

Po wprowadzeniu trzech pierwszych usprawnień:

```
// wynikiem jest posortowana lista pierwotna
// wersja ulepszona wykrywająca wcześniejsze uporządkowanie; tylko metoda sort()
    public List sort(List list) {
        int lastSwap = list.size() -1; // pozycja ostatniej zamiany
        while(lastSwap>0){
            int end=lastSwap;
            lastSwap=0;
            for (int left = 0; left < end; ++left) {
                if (_comparator.compare(list.get(left), list.get(left+1)) > 0)
                  { // ciąg zamian jest zastąpiony ciągiem przepisań
                    Object temp=list.get(left);
                    while(left<end && _comparator.compare(temp, list.get(left+1)) > 0)
                     { list.set(left, list.get(left+1)); left++; }
                    lastSwap=left;
                    list.set(left,temp);
                  }
            }
        }
        return list;
    }
```

Algorytm sortowania przez wybieranie

2. Algorytm **sortowania przez wybieranie** (SelectSort)

Idea porządkowania (dla wariantu: porządkowanie "malejące"):

Powtarzaj n–krotnie operacje:
"weź największy element z ciągu nieuporządkowanego
i wpisz go na koniec ciągu już uporządkowanego"

Cechy algorytmu:
- jest prosty i powolny,
- nie wymaga dodatkowej pamięci,
- ma kwadratową złożoność obliczeniową,
- ma najmniejszą możliwą liczbę przepisań,
- nie zależy od wstępnego uporządkowania ciągu,
- nie jest stabilny (nie zachowuje pierwotnej kolejności elementów równych).
- można go przyspieszyć znajdując jednocześnie elementy: maksymalny i minimalny.

Algorytm sortowania przez wybieranie

```
package sorting;
import lists.List;
public class SelectSort implements ListSorter {
    private final Comparator _comparator;
    public SelectSort(Comparator comparator)  { _comparator = comparator; }
    public List sort(List list) {
        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;
    }
    private void swap(List list, int left, int right) {
        if (left != right) { // sprawdzenie, czy to są dwa różne elementy (czy przestawiać)
            Object temp = list.get(left);
            list.set(left, list.get(right));
            list.set(right, temp);
        }
    }
}
```

Algorytm sortowania przez wstawianie

3. Algorytm **sortowania przez wstawianie** (InsertSort)

Idea porządkowania:

Powtarzaj n–krotnie operacje:
"weź kolejny element z ciągu nieuporządkowanego
i wstaw go na właściwe miejsce w ciągu uporządkowanym"

Zwany jest algorytmem karciarza (od układania kart w ręce, na zasadzie pobierania ich ze stosu kart i bieżące układanie według kolorów i wartości).

Cechy algorytmu:

- jest prosty i powolny (duża liczba porównań),
- nie wymaga dodatkowej pamięci,
- ma kwadratową złożoność obliczeniową,
- ma najmniejszą możliwą liczbę przepisań,
- mocno zależy od uporządkowania danych ("lubi" ciągi uporządkowane),
- jest stabilny (zachowuje pierwotną kolejność elementów równych),
- można go przyspieszyć zmniejszając liczbę porównań (poprzez zastosowanie wyszukiwania binarnego).

Algorytm sortowania przez wstawianie

```
package sorting;
import lists.LinkedList;     // założenie, że w pakiecie lists są zdefiniowane
import lists.List;           // klasy LinkedList i List, omawiane wcześniej
import iteration.Iterators;  // j.w. dla iteratora
public class InsertSort implements ListSorter {
    private final Comparator _comparator;
    public InsertSort(Comparator comparator)  { _comparator = comparator; }

    public List sort(List list) {
        final List result = new LinkedList();
        Iterator it = list.iterator();
        for (it.first(); !it.isDona(); it.next() {
            int slot = result.size();
            while (slot>0) {
              if (_comparator.compare(it.current(), result.get(slot-1)) >= 0) { break; }
             --slot;
            }
            result.insert(slot, it.current());
        }
        return result;
    }
}
```

Państwowa Wyższa Szkoła Zawodowa w Kaliszu
Zbigniew Szpunar

Złożone algorytmy sortowania – na następnym wykładzie