Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Wykład 7.
Zaawansowane algorytmy sortowania

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Zaawansowane algorytmy sortowania

Proste algorytmy sortowania (o dużej złożoności obliczeniowej) są często wystarczające dla małych i średnich rozmiarów danych, ale (ze względu na tę złożoność obliczeniową) są niewystarczające do sortowania dużych porcji danych oraz wszędzie tam, gdzie czas sortowania danych ma charakter krytyczny.

Zaawansowane algorytmy sortowania wprowadzają różne elementy usprawniające proces porządkowania, między innymi:

- zmniejszenie nakładu pracy na operacje przemieszczania w porządkowanych ciągach elementów, znajdujących się daleko od swoich pozycji docelowych, poprzez wprowadzenie rozwiązań szybkiego przemieszczania elementów na znaczne odległości,

- zastosowanie metody "dziel-i-zwyciężaj".

Metoda "dziel-i-zwyciężaj" (ang. *divide and conquer principle*)

Metoda projektowania algorytmów.

Polega na podziale problemu na mniejsze problemy, które na ogół są tym samym problemem (mają taki sam charakter), ale dla danych o mniejszych rozmiarach, i przezwyciężeniu go w ten sposób. Następnie składa się rozwiązania podproblemów w rozwiązanie problemu.

Przykłady takiego podejścia występują w algorytmach sortowania, w których np.:

- w każdym kroku porządkowany ciąg jest dzielony na dwie części,
- elementy w tych częściach są porządkowane,
- uporządkowane części są łączone w jedną uporządkowaną całość.

Taki charakter mają sortowanie przez scalanie (łączenie, ang. *mergesort*) i szybkie sortowanie (ang. *quicksort*).

Przy okazji warto odnotować (przypomnieć), że szczególnym przypadkiem tej metody jest strategia, w której na każdym kroku rozwiązywania problemu jest on dzielony na co najmniej dwie części i do dalszych obliczeń pozostaje tylko jedna z nich. Przykładem takiego działania jest poszukiwanie elementu (lub miejsca dla niego) w ciągu uporządkowanym – jest to tzw. poszukiwanie przez połowienie, często stosowane w słownikach i encyklopediach. Inny przykład (omawiany wcześniej): poszukiwanie miejsca zerowego funkcji ciągłej f(x) w przedziale [a,b].

Algorytmy wykorzystujące tę metodę (najczęściej rekurencyjne) charakteryzują się często niską złożonością obliczeniową.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Sortowanie metodą Shella (ang. *Shellsort*) [Shell, 1959]

Polega na podziale dużej listy na wiele mniejszych podlist, z których każda jest sortowana przez wstawianie (InsertSort).

W kolejnych krokach sortowania liczba podlist systematycznie maleje a zwiększa się sukcesywnie długość każdej podlisty, aż do uzyskania jednej podlisty (czyli listy) o dużym stopniu uporządkowania (prawie posortowanej).

W ostatnim kroku pojedyncza podlista jest sortowana przez wstawianie (jest to, jak wiemy, algorytm, który bardzo sprawdza się dla prawie-uporządkowanych list).

Działanie algorytmu wykorzystuje koncepcję *H-sortowania* – sortowania kolejno podciągów elementów odległych od siebie o H_i (i=1,2,...,n).

Ciąg wartości $H_1,...,H_n$ musi spełniać warunek: $H_n=1$ (czyli w ostatnim, n-tym kroku muszą uczestniczyć wszystkie elementy ciągu) oraz $H_i > H_{i+1}$ dla i=1,2,...,n-1 (czyli ciąg przyrostów musi być malejący).

Algorytm Shellsort:

Dla i od 1 do n wykonać:

- przyjmij przyrost h_i,
- uporządkuj wszystkie podciągi elementów odległych od siebie o h_i (w zasadzie dowolną prostą metodą sortowania, ale zalecana jest metoda InsertSort – j.w.).

Sortowanie metodą Shella (ang. *Shellsort*) [Shell, 1959]

Przykład:

H	Ciąg wejściowy														InsertSort	Ciąg wyjściowy													
4	**2**	7	4	12	**8**	7	6	5	**10**	3	1	13	**9**		=>	**2**	7	4	12	**8**	7	6	5	**9**	3	1	13	**10**
4	2	**7**	4	12	8	**7**	6	5	9	**3**	1	13	10		=>	2	**3**	4	12	8	**7**	6	5	9	**7**	1	13	10
4	2	3	**4**	12	8	7	**6**	5	9	7	**1**	13	10		=>	2	3	**1**	12	8	7	**4**	5	9	7	**6**	13	10
4	2	3	1	**12**	8	7	4	**5**	9	7	6	**13**	10		=>	2	3	1	**5**	8	7	4	**12**	9	7	6	**13**	10
3	**2**	3	1	**5**	8	7	**4**	12	9	**7**	6	13	**10**		=>	**2**	3	1	**4**	8	7	**5**	12	9	**7**	6	13	**10**
3	2	**3**	1	4	**8**	7	5	**12**	9	7	**6**	13	10		=>	2	**3**	1	4	**6**	7	5	**8**	9	7	**12**	13	10
3	2	3	**1**	4	6	**7**	5	8	**9**	7	12	**13**	10		=>	2	3	**1**	4	6	**7**	5	8	**9**	7	12	**13**	10
2	**2**	3	**1**	4	**6**	7	**5**	8	**9**	7	**12**	13	**10**		=>	**1**	3	**2**	4	**5**	7	**6**	8	**9**	7	**10**	13	**12**
2	1	**3**	2	**4**	5	**7**	6	**8**	9	**7**	10	**13**	12		=>	1	**3**	2	**4**	5	**7**	6	**7**	9	**8**	10	**13**	12
1	**1**	**3**	**2**	**4**	**5**	**7**	**6**	**7**	**9**	**8**	**10**	**13**	**12**		=>	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**7**	**8**	**9**	**10**	**12**	**13**

W przykładzie widać, że ciąg częściowo uporządkowany wystąpił wcześniej, niż dla H=1. Można zatem wcześniej (dla H>1) posortować ciąg metodą InsertSort.

Oznacza to praktycznie możliwość (i potrzebę) ustalenia ciągu kolejno stosowanych przyrostów, czyli wartości H_i .

Sortowanie metodą Shella (ang. *Shellsort*) [Shell, 1959]

Wartości ciągu przyrostów przyjmuje się arbitralnie, w zależności od konkretnego przypadku danych do sortowania.

Często stosuje się odwrotność ciągu:

$H_{j+1} = 3 * H_j + 1$, i=1,2,...k, gdzie: liczba przyrostów *k* jest ograniczona warunkiem, by wydzielone podciągi zawierały co najmniej 3 elementy.

Dla danej długości ciągu można więc łatwo (algorytmicznie, w programie) wyznaczyć ciąg przyrostów.

Przykład:
Dla ciągu zawierającego 4000 elementów mamy ciąg przyrostów:

1, 4, 13, 40, 121, 364, 1093

które będą stosowane w odwrotnej kolejności, czyli:

$H_1 = 1093$, $H_2 = 364$, $H_3 = 121$, $H_4 = 40$, $H_5 = 13$, $H_6 = 4$, $H_7 = 1$.

Stąd inna nazwa algorytmu: sortowanie z malejącymi przyrostami.

Przykład implementacji algorytmu ShellSort:

```
import lists.List;
public class ShellSortListSorter implements ListSorter {      // ListSorter posiada
          // metodę public List sort(List list), której wynikiem jest lista posortowana
  private final Comparator  _comparator;
  private final int[]  _increments = {121, 40, 13, 4, 1}; // przyjęte arbitralnie

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

  public List sort(List list) {
    assert list != null : "Nie określono listy";
    for (int i=0; i < _increments.length; ++i) {
       int _h = _increments[i];
       hSort(list, _h);  // sortowanie podlisty określonej przyrostem _h
    }
    return list;
  }
  private void hSort(List list, int h) {
     if (list.size() < h*2) { return; }
     for (int i=0; i<h; ++i) { sortSublist(list, i, h); }
  }
// c.d.n.
```

// c.d.:

```
  private void sortSublist(List list, int startIndex, int h) {      // sortowanie podlisty
                                                                    // od indeksu startindex, z przyrostem h
    for ( int i=startIndex + h; i<list.size(); i+=h ) {
      Object value = list.get(i);
      int j;
      for ( j=i; j>startIndex; j -= h ) {
        Object previousValue = list.get(j-h);
        if (_comparator.compare(value, previousValue) >= 0) {break; }
        list.set(j,previousValue);
      }
      list.set(j, value);
    }
  }
} // ShellSortListSorter
```

Algorytm *ShellSort* ma złożoność obliczeniową $O(n^{3/2})$, więc można go uznać za lepszy, niż proste algorytmy sortowania.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Sortowanie szybkie (ang. *Quicksort*) [Hoare, 1962]

Rekurencyjny algorytm sortowania (choć można zrealizować jego wersję iteracyjną), wykorzystujący metodę "dziel-i-zwyciężaj".

Realizuje sortowanie listy dzieląc zadanie sortowania na rekurencyjne sortowanie coraz mniejszych fragmentów listy. Na każdym poziomie rekurencji przetwarzanie obejmuje trzy etapy:

- umieszczenie (arbitralnie) wybranego elementu ciągu na jego pozycji docelowej,
- umieszczenie na lewo od niego wszystkich elementów mniejszych od niego,
- umieszczenie na prawo od niego pozostałych elementów.

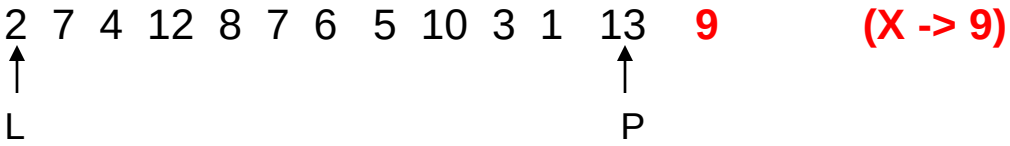
Wybrany w 1. kroku element jest więc elementem (X) dzielącym listę.

Wybór pierwszego elementu dzielącego dokonywany jest arbitralnie. Istnieje szereg strategii wyboru elementu dzielącego listę. Pomijając to zagadnienie można przyjąć, że tym elementem będzie ostatni element listy.

Kroki 2. i 3. wymagają zainicjowania wartości dwóch indeksów:

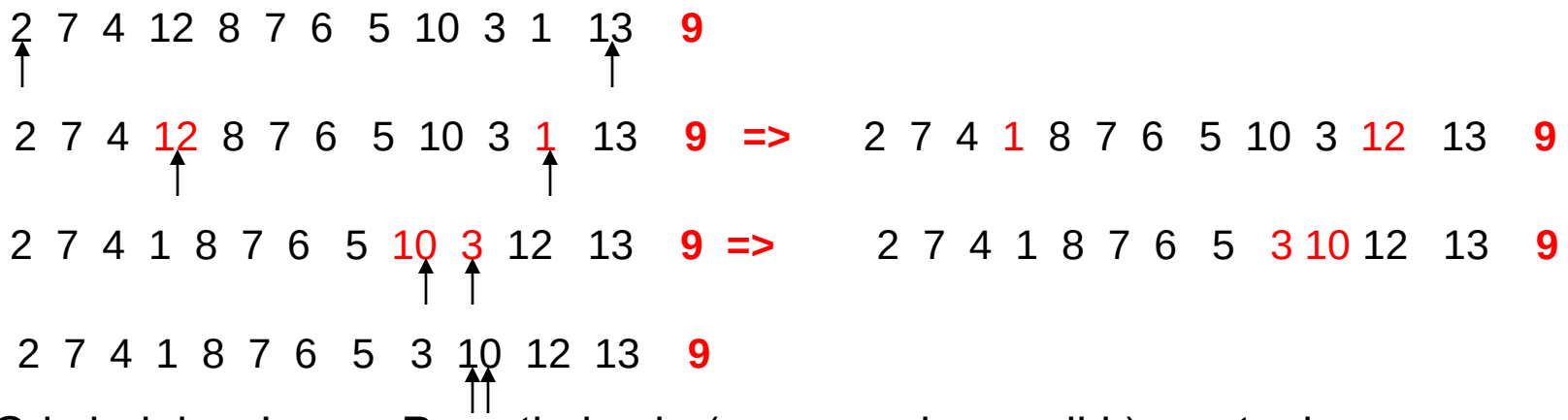
- pierwszy (L - "lewy") wskazuje (początkowo) na pierwszy element listy,
- drugi (P - "prawy") wskazuje na większy z dwóch elementów: element X lub element poprzedzający go.

Przykład:

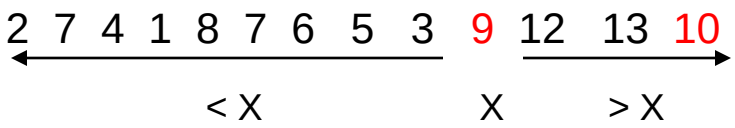


Sortowanie szybkie – c.d.

W czasie wykonywania kroków 2. i 3. indeksy przesuwają się ku sobie tak długo, aż indeks L napotka na element > X, a indeks P – na element < X. Wówczas elementy te zamienia się miejscami i kontynuuje się to działanie aż do chwili, gdy indeksy L oraz P spotkają się ze sobą na pewnej pozycji k.



Gdy indeksy L oraz P spotkają się (na pewnej pozycji k), zostanie wyznaczona nowa pozycja dotychczasowego elementu dzielącego X. Należy przestawić elementy X oraz element z pozycji k miejscami.



Element dzielący X trafił na swoją docelową pozycję. Należy teraz zastosować ten sam algorytm do posortowania (niezależnie od siebie) obu wydzielonych list.

Bazowy przypadek rekurencji ("koniec") następuje, gdy lista ma 1 element.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Przykład implementacji algorytmu Quicksort [na podstawie: Harris, Ross]:

```
import lists.List;
public class QuickSortListSorter implements ListSorter {
  private final Comparator _comparator;

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

  public List sort(List list) {                              // zwraca posortowaną listę
     quicksort(list, 0, list.size() - 1);
     return list;
  }
  private void quicksort(List list, int startIndex, int endIndex) {
     if (endIndex > startIndex) {
        Object value=list.get(endIndex);
        int partition = partition(list, value, startIndex, endIndex - 1);      // podział listy,
                                                  // wyznaczenie pozycji elementu dzielącego
        if (_comparator.compare(list.get(partition), value) < 0) ++partition;
        swap(list, partition, endIndex);
        quicksort(list, startIndex, partition - 1);
        quicksort(list, partition + 1, endIndex);
     }
  }
// c.d.n.
```

// c.d.:

```
private int partition(List list, Object value, int leftIndex, int rightIndex) {
   int left = leftIndex;
   int right = rightIndex;
   while (left < right) {
      if (_comparator.compare(list.get(left), value) < 0) { ++left; continue; }
      if (_comparator.compare(list.get(right), value) >= 0) { --right; continue; }
      swap(list, left, right);
      ++left;
    }
    return left;      // zwraca znalezioną pozycję elementu dzielącego
 }

 private void swap(List list, int left, int right) {  // zamiana elementów
    if (left != right) {
       Object temp = list.get(left);
       list.set(left, list.get(right));
       list.set(right, temp);
    }
   }
} // koniec QuickSortListSorter
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Sortowanie szybkie – c.d.

Można też przedstawić ideę algorytmu następująco:

1. Podziel ciąg na dwa podciągi (według zasady: <X oraz >=X).
2. Uporządkuj pierwszy podciąg tą samą metodą.
3. Uporządkuj drugi podciąg tą samą metodą.
4. Połącz podciągi zachowując uporządkowanie.

Algorytm Qiucksort:

- jest prosty, sortuje ciągi "w miejscu",
- jest bardzo szybki (liniowo-logarytmiczna złożoność obliczeniowa),
- potrzebuje pamięci dla wywołań rekurencyjnych (można uwolnić się od tej wady realizując odpowiednią wersję iteracyjną algorytmu),
- jest silnie zależny od początkowego uporządkowania ciągu (paradoksalnie, najgorszym przypadkiem jest ciąg już uporządkowany), działa wolno, gdy elementy ciągu powtarzają się,
- ma efektywność uzależnioną od równomierności podziału na podciągi; jest lepszy od innych algorytmów przy długich ciągach (można przerwać rekurencję przy ciągach krótszych i kontynuować sortowanie np. InsertSort-em).
- można częściowo uniezależnić się od negatywnych elementów rekurencji (rozrost stosu wywołań, szczególnie przy nierównomiernych podziałach ciągu) wprowadzając własny stos do odkładania dłuższego z podciągów przy równoczesnej, doraźnej obsłudze krótszego podciągu.

Sortowanie przez scalanie (łączenie), ang. *Mergesort*

Algorytm rekurencyjny, wykorzystuje ideę "dziel-i-zwyciężaj".

Idea algorytmu:

1. Podziel ciąg na dwa równoliczne podciągi (jeśli nie jest to możliwe ze względu na nieparzystą liczbę elementów – jeden z podciągów będzie o jeden element dłuższy, niż drugi).
2. Uporządkuj pierwszy podciąg tą samą metodą.
3. Uporządkuj drugi podciąg tą samą metodą.
4. Połącz oba podciągi zachowując uporządkowanie.

Zakończenie działań rekurencyjnych następuje, gdy ciąg ma mniej, niż dwa elementy.

Łączenie dwóch ciągów uporządkowanych może przebiegać następująco:

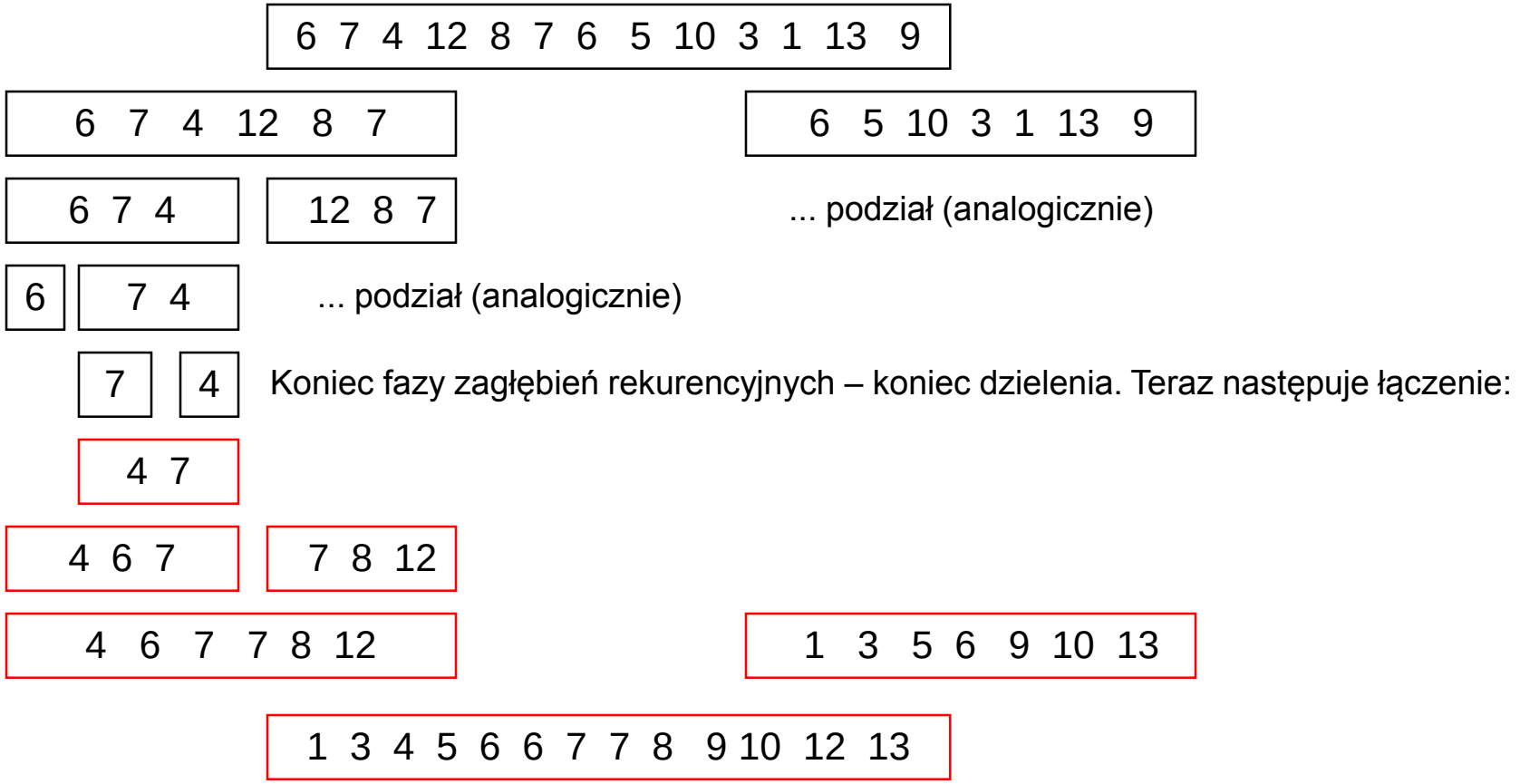
- porównujemy ze sobą pierwsze elementy z każdego z ciągów danych,
- mniejszy element wstawiamy do nowego ciągu i usuwamy z ciągu danych,
- powtarzamy te czynności, aż oba ciągi danych będą puste.

W ten sposób, w nowo utworzonym ciągu wszystkie elementy są uporządkowane, a (co najważniejsze) operacja ta wymaga wykonania niewielu porównań.

Sortowanie każdego z podciągów to rekurencyjne wywołanie tej samej metody, czyli: podział na podciągi, porządkowanie podciągów i ich łączenie, a więc praktycznie podział na elementarne podciągi i ich kolejne łączenie.

Sortowanie przez scalanie (łączenie) – c.d.

Przykład sortowania przez scalanie:



Łatwo zauważyć, że metoda zapewnia stabilność procesu sortowania.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Przykład implementacji algorytmu Mergesort [Harris, Ross]:

```
import iterators.Iterator;
import lists.ArrayList;
import lists.List;
public class MergeSortListSorter implements ListSorter {
    private final Comparator _comparator;

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

    public List sort(List list)                          // zwraca posortowaną listę
    { return mergesort(list, 0, list.size() - 1); }

    private List mergesort(List list, int startIndex, int endIndex) {
        if (startIndex == endIndex) {                    // lista jednoelementowa
            List result = new ArrayList();
            result.add(list.get(startIndex));
            return result;
        }
        int splitIndex = startIndex + (endIndex - startIndex) / 2;
        return merge(mergesort(list, startIndex, splitIndex),
                     mergesort(list, splitIndex + 1, endIndex));  // scalenie list w jedną listę
    }
// c.d.n.
```

// c.d.:

```
 private List merge(List left, List right) {
    List result = new ArrayList();
    Iterator l = left.iterator();  Iterator r = right.iterator();
    l.first(); r.first();
    while (!l.isDone() && !r.isDone()) {
        if (_comparator.compare(l.current(), r.current()) <= 0)
                { result.add(l.current()); l.next(); }
        else { result.add(r.current()); r.next(); }
    }
    while(!l.isDone())
       { result.add(l.current()); l.next(); }

    while(!r.isDone())
       { result.add(r.current()); r.next(); }
    return result;
 }
} // koniec MergeSortListSorter
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Uwagi o zaawansowanych i prostych algorytmach sortowania

Zaawansowane algorytmy sortowania mają znacznie mniejszą złożoność obliczeniową niż proste algorytmy sortowania, ale w niektórych sytuacjach (dla niektórych uporządkowań początkowych danych) są wolniejsze, niż algorytmy proste.

Podejmując decyzję o wyborze algorytmu w konkretnym przypadku należy przeanalizować charakter danych do sortowania i stosownie do wyników analizy wybrać algorytm.

Należy pamiętać, że liczba porównań w algorytmach sortowania nie jest jedynym czynnikiem wpływającym na wydajności algorytmów. Występują przecież inne operacje, których liczbę także należy brać pod uwagę.

Należy też brać pod uwagę zapotrzebowanie algorytmów na pamięć (np. Mergesort charakteryzuje się zwiększonym zapotrzebowaniem na pamięć, w stosunku do Quicksortu i metody Shella, na tworzenie kopii każdej sortowanej listy, co też wymaga czasu!).

Z omówionych, zaawansowanych metod sortowania, tylko Mergesort zapewnia stabilność sortowania. W przypadku pozostałych algorytmów należałoby zastosować dodatkowe mechanizmy zapewniające stabilność (np. komparatory złożone).

Istnieje jeszcze sporo innych algorytmów sortowania. O niektórych z nich będzie jeszcze mowa – na kolejnych wykładach.