Notatki z AiSD. Nr 5. 3 kwietnia 2007

SORTOWANIE

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

Na dzisiejszym wykładzie poznamy algorytmy, sortujące w czasie niższym niż wynika to z dolnego ograniczenia poznanego na poprzednim wykładzie. Jest to możliwe z dwóch powodów. Po pierwsze algorytmy te zakładają pewne ograniczenia na postać danych, a po drugie wykonują one na sortowanych elementach operacje inne niż porównania.

1 Counting Sort

POSTAĆ DANYCH: ciąg $A[1..n]$ liczb całkowitych z przedziału $\langle 1, k\rangle$.

IDEA: $\forall_{x\in A[1..n]}$ obliczyć liczbę $c[x] = |\{y \ : \ y \in A[1..n] \ \& \ y \leq x\}|$.

procedure *Counting − Sort*$(A[1..n], k, \mathbf{var} B[1..n])$
  for $i \leftarrow 1$ **to** k **do** $c[i] \leftarrow 0$
  for $j \leftarrow 1$ **to** n **do** $c[A[j]] \leftarrow c[A[j]] + 1$
  for $i \leftarrow 2$ **to** k **do** $c[i] \leftarrow c[i] + c[i-1]$
  for $j \leftarrow n$ **downto** 1 **do** $B[c[A[j]]] \leftarrow A[j]$
    $c[A[j]] \leftarrow c[A[j]] - 1$

UWAGA: W oczywisty sposób powyższa procedura może być zmodyfikowana do sortowania rekordów, w których klucz $A[j]$ jest jednym z wielu pól.

Definition 1 *Metodę sortowania nazywamy* stabilną*, jeśli w ciągu wyjściowym elementy o tej samej wartości klucza pozostają w takim samym porządku względem siebie w jakim znajdowały się w ciągu wejściowym.*

Fakt 1 *Counting − sort jest metodą stabilną.*

KOSZT: $\Theta(n + k)$.

2 Sortowanie kubełkowe (bucket sort).

POSTAĆ DANYCH: Ciąg $A[1..n]$ liczb rzeczywistych z przedziału $\langle 0, 1)$ wygenerowany przez generator liczb losowych o rozkładzie jednostajnym.

IDEA: Podzielić przedział $\langle 0, 1)$ na n odcinków ("kubełków") jednakowej długości; umieścić liczby w odpowiadających im kubełkach; posortować poszczególne kubełki; połączyć kubełki.

procedure *bucket − sort*$(A[1..n])$
  for $i \leftarrow 0$ **to** $n-1$ **do** $B[i] \leftarrow \emptyset$
  for $i \leftarrow 1$ **to** n **do** dołącz $A[i]$ do listy $B[\lfloor nA[i]\rfloor]$
  for $i \leftarrow 0$ **to** $n-1$ **do** posortuj procedurą *insert − sort* listę $B[i]$
  połącz listy $B[0], B[1], \ldots, B[n-1]$

KOSZT: Oczekiwany czas działania: $\Theta(n)$.

3 Sortowanie leksykograficzne ciągów jednakowej długości (radix sort).

Definition 2 *Niech S - zbiór uporządkowany liniowo oraz $s_1, \ldots, s_p, t_1, \ldots, t_q \in S$.*

$$(s_1, \ldots, s_p) \leq (t_1, \ldots, t_q) \stackrel{df}{\iff} \begin{array}{ll} (1) & \exists_{1 \leq j \leq \min(p,q)}\ s_j < t_j \ \&\ \forall_{i<j}\ s_i = t_i \\ albo & \\ (2) & p \leq q \ \&\ \forall_{1 \leq i \leq p}\ s_i = t_i. \end{array}$$

POSTAĆ DANYCH: $A_1, \ldots, A_n$ - elementy $\{0, \ldots, k-1\}^d$.

```
procedure radix − sort(A1, .., An)
   for i ← d downto 1 do
       metodą stabilną posortuj ciągi wg i−tego elementu
```

KOSZT: Jeśli w procedurze $Radix\!-\!sort$ zastosujemy $counting\!-\!sort$, to jej koszt wyniesie $O((n+k)d)$. Jest to koszt liniowy, gdy $k = O(n)$.

4 Sortowanie leksykograficzne ciągów niejednakowej długości.

POSTAĆ DANYCH: $A_1, \ldots, A_n$ ciągi liczb całkowitych $\in \langle 0, .., k-1 \rangle$.

Niech l_i-długość A_i, $l_{max} = \max\{l_i \ :\ i = 1, .., n\}$.

4.1 Pierwszy sposób

IDEA: Uzupełnić ciągi specjalnym elementem (mniejszym od każdego elementu z S), tak by miały jednakową długość i zastosować algorytm z poprzedniego punktu.

KOSZT: $\Theta((n+k) \cdot l_{max})$.

UWAGA: Jest to metoda nieefektywna, gdy ciągów długich jest niewiele.

4.2 Drugi sposób

IDEA:

for $i \leftarrow l_{max}$ **downto** 1 **do**
  metodą stabilną posortuj ciągi o długości $\geq i$ wg i-tej składowej

ALGORYTM:

1. Utwórz listy $nonempty[l]$ $(l = 1, .., l_{max})$ takie, że
   - $x \in nonempty[l]$ iff x jest $l-$tą składową jakiegoś ciągu A_i.
   - $nonempty[l]$ jest uporządkowana niemalejąco.
2. Utwórz listy $length[l]$ $(l = 1, .., l_{max})$ takie, że $length[l]$ zawiera wszystkie ciągi A_i o długości l.
3. $queue \leftarrow \emptyset$

```
   for j ← 0 to k − 1 do q[j] ← ∅
   for l ← l_max downto 1 do
      queue ← concat(length[l], queue)
      while queue ≠ ∅ do
         Y ← pierwszy ciąg z queue
         queue ← queue \ {Y}
         a ← l-ta składowa ciągu Y
         q[a] ← concat(q[a], {Y})
      for each  j ← nonempty[l] do
         queue ← concat(queue, q[j])
         q[j] ← ∅
```

Operacja $concat(K_1, K_2)$ dołącza kolejkę K_2 do końca kolejki K_1.

Twierdzenie 1 *Powyższy algorytm można zaimplementować tak, by działał w czasie* $O(k + \sum_{i=1}^{n} l_i)$.

UZASADNIENIE: Niech $l_{total} = \sum_{i=1}^{n} l_i$.
Jedynym niezupełnie trywialnym krokiem jest tworzenie list *nonempty*:

- tworzymy w czasie $O(l_{total})$ ciąg S zawierający wszystkie pary $\langle l, a\rangle$, takie, że a jest l-tą składową jakiegoś $A[i]$;
- sortujemy leksykograficznie w czasie $O(k + l_{total})$ ciąg S;
- przeglądając S z lewa na prawo tworzymy $O(l_{total})$ listy *nonempty*.

Krok 2 wymaga czasu $O(l_{total})$.

Wewnętrzna pętla while działa w czasie proporcjonalnym do sumarycznej (po wszystkich iteracjach pętli zewnętrznej) długości kolejek *queue*. Ponieważ w l-tej iteracji *queue* ma długość równą liczbie ciągów co najmniej l-elementowych, więc koszt while jest $O(l_{total})$.

Wewnętrzna pętla for działa w czasie proporcjonalnym do sumarycznej (po wszystkich iteracjach pętli zewnętrznej) długości list *nonempty*. Ponieważ w każdej iteracji *nonempty* jest nie dłuższa od *queue*, czas pętli for jest również $O(l_{total})$. □

4.3 Przykład zastosowania

PROBLEM:
Dane: T_1, T_2 -drzewa o ustalonych korzeniach,
Zadanie: sprawdzić czy T_1 i T_2 są izomorficzne.

IDEA: Wędrując przez wszystkie poziomy (począwszy od najniższego) sprawdzamy czy na każdym poziomie obydwa drzewa zawierają taką samą liczbę wierzchołków tego samego typu. (Wierzchołki są tego samego typu, jeśli poddrzewa w nich zakorzenione są izomorficzne.)

ALGORYTM:

Bez zmniejszenia ogólności możemy założyć, że obydwa drzewa mają tę samą wysokość.

1. $\forall_{v\ -\ li\ w\ T_i}\ kod(v) \leftarrow 0$
2. **for** $j \leftarrow depth(T_1)$ **downto** 1 **do**
3. $\quad S_i \leftarrow$ zbiór wierzchołków T_i z poziomu j nie będących liśćmi
4. $\quad \forall_{v \in S_i}\ key(v) \leftarrow$ wektor $\langle i_1, \ldots, i_k \rangle$, taki że
   - $i_1 \leq i_2 \leq \ldots \leq i_k$
   - v ma k synów $u_1, \ldots, u_k$ i $i_l = kod(u_l)$
5. $\quad L_i \leftarrow$ lista wierzchołków z S_i posortowana leksykograficznie według wartości key
6. $\quad L_i^{'} \leftarrow$ otrzymany w ten sposób uporządkowany ciąg wektorów
7. $\quad$ **if** $L_1^{'} \neq L_2^{'}$ **then return** ("nieizomorficzne")
8. $\quad \forall_{v \in L_i}\ kod(v) \leftarrow 1 + rank(key(v), \{key(u) \mid u \in L_i\})$
9. $\quad$ Na początek L_i dołącz wszystkie liście z poziomu j drzewa T_i
10. **return** ("izomorficzne")

Twierdzenie 2 *Izomorfizm dwóch drzew o n wierzchołkach może być sprawdzony w czasie $O(n)$.*