Teraz gdybyśmy umieli zrobić otoczkę up liniowo, to umiemy posortować liniowo, bo posortowane elementy nie porobią. Pytanie, gdzie jest pułapka?

Oczywiście jest takie, że w modelu drzew decyzyjnych możemy wykonywać wyłącznie porównania. Chodzi o pewną ostrożność, aby uwierzyć, czy dany nie pewien warunek jest legalny w modelu i możemy korzystać z możliwości tego modelu. My nie pokazaliśmy, że sortowanie wymaga $\Omega(n \log n)$ oprócz tylko $n \log n$ porównań

SORTOWANIE

Problem (IZOMORFIZM DRZEW)

Dane: T_1, T_2 - drzewa nieskierowane

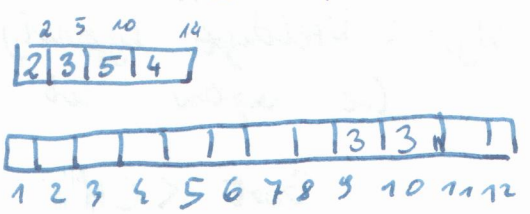
Pytanie: czy $T_1 \cong T_2$ (są izomorficzne)

Cel: Algorytm $O(n)$

Sortowanie przez zliczanie

Dane: $a_1 \dots a_n$ $\forall_i a_i \in \{1 \dots k\}$ 2 2 4 3 4 1 2 4 1 3 3 4 3 3

Zał: k nie jest duże



$O(n)$ • $\forall_{i=1 \dots n}$ $c_i \leftarrow 0$ (zerujemy i)

$O(n)$ • for $(i=1 \dots n)$ c_{a_i}++

$O(k)$ • for $(i=1 \dots k)$ $c_i \leftarrow c_i + c_{i-1}$

$O(n)$ • wypisz elementy idąc od prawej do lewej

$O(n+k)$

Def: Metoda jest stabilna jeśli zachowuje względną kolejność równych kluczy (z wejścia)

FAKT: Podana metoda sortowania jest stabilna

Sortowanie kubełkowe:

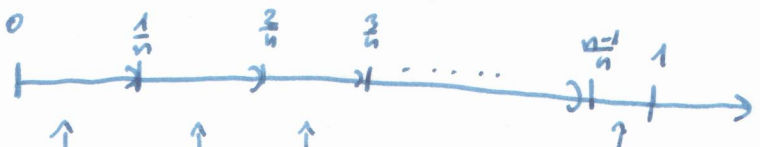
Dane: $x_1 \dots x_n$ - losowe dane z $[0,1]$

1.) Rozrzuć liczby do n kubełków: i-ty kubełek $O(1)$

odp: $[\frac{i-1}{n}, \frac{i}{n})$

2.) Sortujemy kubełki niezależnie (metodą insert-sort)

3.) Konkatenujemy otrzymane wyniki $O(n)$



to wpada: r_1 r_2 r_3 r_n elementów

ilość r_i poszczególnych kubełków

$\sum_{i=1}^{n} r_i = n$

$\sum_{i=1}^{n} O(r_i^2)$

Jaki jest czas oczekiwany? $O(n)$, ale nie będziemy tego dokładnie: RNO pokaże

Sortowanie leksykograficzne

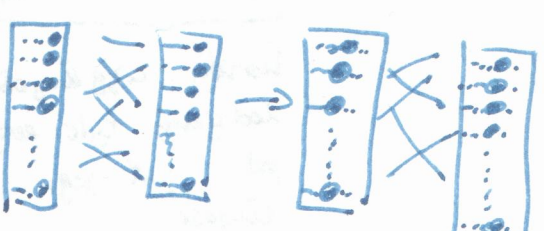
wersja I: ciągi jednakowej długości

Dane: $A_1 \dots A_n$ A_i - ciągi długości d z Σ (sygnatura) $(|\Sigma| = k)$

Zadanie: wypisz leksykograficznie

for $i = d, d-1, \dots, 1$

~~sortuj~~ metodą stabilną posortuj ciągi wg i-tej pozycji:



1. Rozrzuć ciągi do kubełków
2. Skonkatenuj zawartość kubełków

pomysł: sortowanie jest stabilne do wyjścia

~~czas~~ czas $O(d(n+k))$

Wersja II: ciągi mogą (mogą mieć) różną długość

Niech $k_i = |A_i|$

$l_{max} = \max_i \{k_i\}$

Pomysł 1:

A_1 ——————

A_2 —— ~~~

A_3 - ~~~~~~

A_4 ——— ~

A_5 —— ~~~

Dopychamy do l_{max} jakimiś lipnymi znakami (np. spacje), różnie

$\Theta(l_{max}(n+k))$

Pomysł 2:

(jak poprzednio)

l_{max} faz; w i-tej fazie:

1. Rozrzucamy do kubełków słowa o długości $\geq l_{max} - i + 1$
2. Konkatenujemy kubełki

(Preprocesing):

grupujemy ciągi o tej samej długości:

$c[i]$ - ciągi o długości i

$c[1], c[2], c[3], \ldots, c[l_{max}]$

Jak długo to działa?

Sumaryczny koszt kroku 1:

[redacted]

słów o długości $\geq l_{max}$ + # słów o długości $\geq l_{max-1}$ + ... + # słów o długości ≥ 1

$= \underbrace{l_1 + l_2 + \ldots + l_n}_{\text{każdy ciąg długości dodajemy tyle razy ile jest jego długość}} =$ rozmiar danych

Aby kody 2 bit wykonywały szybko musimy wiedzieć które kubełki w poszczególnych fazach są niepuste.

Tworzymy ciąg par postaci $\langle i, a \rangle$

i – pozycja, a – litera na tej pozycji

Przykład:

$A_1 = abcaa$ $\langle 1,a\rangle \langle 2,b\rangle \langle 3,c\rangle \langle 4,a\rangle \langle 5,a\rangle$

$A_2 = bbacd$ $\langle 1,b\rangle \langle 2,b\rangle \langle 3,a\rangle \langle 4,c\rangle \langle 5,d\rangle$

$A_3 = aacaa$ $\langle 1,a\rangle \langle 2,a\rangle \langle 3,c\rangle \langle 4,a\rangle$

$A_4 = caica$ $\langle 1,c\rangle \langle 2,a\rangle \langle 3,c\rangle \langle 4,a\rangle$

← to jest jeden ciąg!

tych par jest $\sum_{i=1}^{n} l_i$; posortujmy te pary leksykograficznie:

$\underbrace{\langle 1,\dots\rangle \langle 1,\dots\rangle \langle 1,\dots\rangle \dots}_{\langle 1,a\rangle\langle 1,a\rangle\langle 1,b\rangle\langle 1,c\rangle} \langle 2,\dots\rangle\langle 2,\dots\rangle, \dots \langle l_{max},\dots\rangle$

← zauważmy, że kubełek 0 jest pusty :)

Koszt tego sortowania $2\left(\sum_{i=1}^{n} l_i + k + l_{max}\right)$

Ostateczny koszt: $l_{total} + k$

Teraz zajmiemy się zastosowaniem tego sortowania:

Problem: (IZOMORFIZM DRZEW)

Dane: $T_1 = (V_1, E_1)$, $T_2 = (V_2, E_2)$

r_1 - korzeń T_1
r_2 - korzeń T_2

Chcemy liniowo !!!

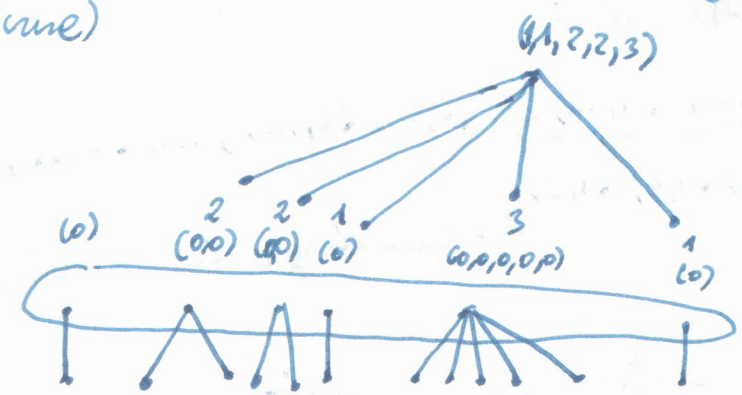
Pytanie: $T_1 \cong T_2$?

Zał, że:

T_1 i T_2 mają:
- tę samą wysokość
- na każdym poziomie równą # liści

Idea:

na każdym poziomie sprawdzamy, czy w T_1 i w T_2 są takie same same liczby podrzew tego samego typu (podrzewa są tego samego typu jeśli są izomorficzne)



$\forall_{v\text{-liść } T_i}$ $hash(v) = 0$

for $j = depth(T_i) \dots 1$
S_i - zbiór nieliści w T_i z poziomu j
$\forall v \in S_i$ $hash(v)$ = wektor $\langle i_1, \dots, i_k \rangle$ t.że
- $i_1 \leq \dots \leq i_k$
- v ma k synów $u_1, \dots, u_k$ t.że $hash(u_l) = i_l$

L_i ← uporządkowany ciąg wektorów opisujących wierzchołki z S_i

if $L_1' \neq L_2'$ return "nieizomorficzne"

$\forall w \in S_i$ hod(w) = 1 + rank(key(v) wśród z L_i)

Na początku L_i dostaje wszystkie liście z poziomu j

Pytanie: Jak szybko to zrobić?

Porządkowanie ciągów wektorów: liniowo od sumy długości wektorów

Długość L_i w trakcie k-tej iteracji = # wierzchołków na niższym poziomie

Stąd $\sum_{\text{iteracje}} |L_i| \leq O(\#\text{ wierzchołków w } T_i)$

Quicksort

Quicksort (A[1...n], p, r) ← sortujemy od elem p-tego do r-tego

```
if (r-p) - małe then insetsort (A[p...r])
else
        choose_pivot (A,p,r)
        q = partition (A,p,r)
        Quicksort (A, p, q)
        Quicksort (A, q+1, r)
```

Partition (A[1...n], p, r)

```
x ← A[p]
i ← p-1
j ← r+1
while i<j
        repeat  j ← j-1 until A[j] ≤ x
        repeat  i ← i+1 until A[i] ≥ x
        if (i<j) swap (A[i], A[j]
```