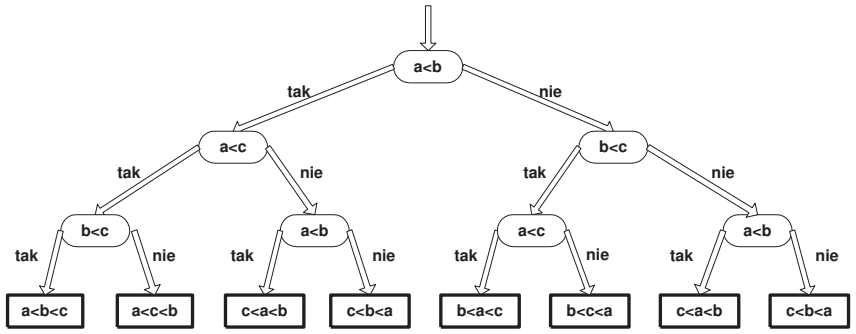
3 Dolne granice.

Rozważając dolne ograniczenia na złożoność problemu sortowania ograniczymy się, podobnie jak w przypadku problemu jednoczesnego znajdowania minimum i maksimum, do klasy algorytmów, które na elementach ciągu wejściowego wykonują jedynie operacje porównania. Działanie takich algorytmów można w naturalny sposób reprezentować *drzewami decyzyjnymi*. Niezbyt formalnie można je zdefiniować jako skończone drzewa binarne, w których każdy wierzchołek wewnętrzny reprezentuje jakieś porównanie, każdy liść reprezentuje wynik obliczeń a krawędzie odpowiadają obliczeniom wykonywanym przez algorytm pomiędzy kolejnymi porównaniami.

Ponieważ od drzew decyzyjnych wymagamy by były skończone, jedno drzewo nie może reprezentować działania algorytmu dla dowolnych danych. Z reguły przyjmujemy, że algorytm reprezentowany jest przez nieskończoną rodzinę drzew decyzyjnych $\{D_i\}_{i=1}^{\infty}$, gdzie drzewo D_n odpowiada działaniu algorytmu na danych o rozmiarze n.

PRZYKŁAD
Rysunek 7 przedstawia drzewo decyzyjne odpowiadające działaniu algorytmu *SelectSort* na ciągach 3-elementowych.

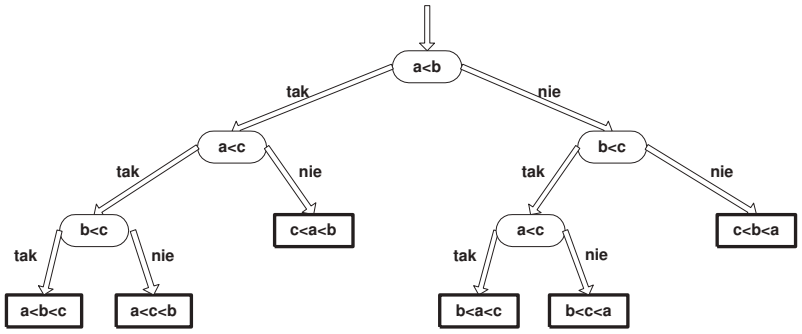


Rysunek 7: *Drzewo D_3 dla algorytmu sortowania przez selekcję.*

□

Jak łatwo zauważyć, algorytm Select Sort wykonuje niektóre porównania niepotrzebnie. Są to porównania "a<b" znajdujące się w odległości 2 od korzenia. Po ich usunięciu otrzymamy inne, mniejsze, drzewo decyzyjne dla sortowania ciągów 3-elementowych (patrz Rysunek 8). Pod względem liczby liści jest ono optymalne.

Fakt 12 *Niech $\mathcal{A}$ będzie algorytmem sortującym, a $\{D_i\}_{i=1}^{\infty}$ – odpowiadającą mu rodziną drzew decyzyjnych. Wówczas drzewo D_n posiada co najmniej $n!$ liści, dla każdego n.*



Rysunek 8: *Optymalne drzewo decyzyjne dla algorytmów sortujących ciągi 3-elementowe.*

Uzasadnienie: Każda permutacja ciągu wejściowego może być wynikiem, a każdy liść drzewa D_n odpowiada jednemu wynikowi. □

Wprost z Faktu 12 mamy następujące:

Twierdzenie 3 *Niech $\mathcal{A}$ będzie algorytmem sortującym, a $\{D_i\}_{i=1}^{\infty}$ – odpowiadającą mu rodziną drzew decyzyjnych. Wówczas drzewo D_n ma wysokość co najmniej $\Omega(n \log n)$.*

Uzasadnienie: Drzewo binarne o $n!$ liściach (a takim jest D_n) musi mieć wysokość co najmniej $\log(n!)$. Ze wzoru Stirlinga, $n!$ możemy z dołu oszacować przez $(n/e)^n$, co daje nam:

$$\log n! \geq n\,(\log n - \log e) \geq n \log n - 1.44n$$

□

Ponieważ wysokość drzewa D_n odpowiada liczbie porównań wykonywanych w najgorszym przypadku przez algorytm A dla danych o rozmiarze n, otrzymujemy dolne ograniczenie na złożoność czasową (w najgorszym przypadku) algorytmów sortowania.

Wniosek 1 *Każdy algorytm sortujący za pomocą porównań ciąg n - elementowy wykonuje co najmniej $cn \log n$ porównań dla pewnej stałej $c > 0$.*

3.1 Ograniczenie na średnią złożoność

Działanie algorytmu sortowania, który dane wykorzystuje wyłącznie w porównaniach, zależy jedynie od względnego porządku pomiędzy elementami. W szczególności nie zależy ono od bezwzględnych wartości elementów. Dlatego badając złożoność takich algorytmów możemy ograniczać się do analizy zachowania algorytmu na permutacjach zbioru $\{1, 2, \ldots, n\}$, a średnia złożoność algorytmu na danych rozmiaru n może być policzona jako suma:

$$\sum_{\sigma - permutacja\ zbioru\ \{1,2,\ldots,n\}} P[\sigma]c(\sigma),$$

gdzie $P[\sigma]$ jest prawdopodobieństwem wystąpienia permutacji σ jako danych wejściowych, a $c(\sigma)$ jest równa liczbie porównań wykonywanych na tych danych. W języku drzew decyzyjnych można ją wyrazić jako średnią wysokość drzewa, tj.

$$\sum_{v - liść\ T} p_v d_v,$$

gdzie p_v oznacza prawdopodobieństwo dojścia do liścia v, a d_v - jego głębokość.

Teraz łatwo widać, że dla wielu rozkładów danych średnia złożoność algorytmu także wynosi $\Omega(n \log n)$. Wystarczy bowiem, by istniały stałe c i d takie, że prawdopodobieństwa dojścia do liści znajdujących się na głębokości nie mniejszej niż $cn \log n$ sumują się do wartości nie mniejszej d. W szczególności otrzymujemy:

Twierdzenie 4 *Jeżeli każda permutacja ciągu n-elementowego jest jednakowo prawdopodobna jako dana wejściowa, to wówczas każde drzewo decyzyjne sortujące ciągi n-elementowe ma średnią głębokość co najmniej* $\log n!$.

Uzasadnienie: Na głębokości nie większej niż $\log(n/e)^n - 1$ znajduje się mniej niż $n!/2$ liści. Tak więc co najmniej $n!/2$ liści osiągalnych z prawdopodobieństwem $1/n!$ leży na głębokości większej, co implikuje, że średnia wysokość drzewa decyzynego jest większa niż $(1/n!)(n!/2)\log((n/e)^n)$. □

4 Quicksort

O algorytmie *Quicksort* wspomnieliśmy omawiając strategię dziel i zwyciężaj. Podany tam schemat algorytmu można zapisać w następujący sposób:

```
procedure quicksort(A[1..n], p, r)
    if r − p jest małe  then insert − sort(A[p..r])
    else choosepivot(A, p, r)
         q ← partition(A, p, r)
         quicksort(A, p, q)
         quicksort(A, q + 1, r)
```

Kluczowe znaczenie dla efektywności algorytmu mają wybór *pivota*, tj. elementu dzielącego, dokonywany w procedurze *choosepivot*, oraz implementacja procedury *partition* dokonującej przestawienia elementów tablicy A.

4.1 Implementacja procedury partition

Zakładamy, że w momencie wywołania $partition(A, p, r)$ pivot znajduje się w $A[p]$. Procedura przestawia elementy podtablicy $A[p..r]$ dokonując jej podziału na dwie części: w pierwszej – $A[p..q]$ – znajdą się elementy nie większe od pivota, w drugiej – $A[q + 1, r]$ – elementy nie mniejsze od pivota. Granica tego podziału, wartość q, jest przekazywana jako wynik procedury.

```
procedure partition(A[1..n], p, r)
    x ← A[p]
    i ← p − 1
    j ← r + 1
    while i < j do
        repeat j ← j − 1 until A[j] ≤ x
        repeat i ← i + 1 until A[i] ≥ x
        if i < j then zamień A[i] i A[j] miejscami
                 else return j
```

Fakt 13 *Koszt procedury partition*$(A[1..n], p, r)$ *wynosi* $\Theta(r-p)$.

4.2 Wybór pivota

Istnieje wiele metod wyboru pivota implementowanych w procedurze *quicksort*. Decydując się na którąś z nich musimy dokonać kompromisu między jakością pivota a czasem działania algorytmu. Nierozważne wybory pivotów mogą w skrajnym przypadku prowadzić do takich podziałów tablicy A, w których jedna z podtablic jest jednoelementowa, a to implikuje liniową głębokość rekursji i, w konsekwencji, kwadratowy czas działania procedury *quicksort*.

Wydawać się może, że idealnym pivotem jest mediana[3], ponieważ daje zrównoważone podziały tablicy A, co ogranicza głębokość rekursji do $\log n$. Ponadto istnieją algorytmy wyznaczające medianę w czasie liniowym (poznamy je pó"xniej), więc czas działania procedury *quicksort* wyraża się równaniem $t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Theta(n)$, co daje optymalnie asymptotyczny czas $\Theta(n \log n)$. Problem w tym, że stała ukryta pod Θ jest zbyt duża, by taki algorytm był praktyczny.

4.2.1 Prosta metoda deterministyczna

Najprostszą, dość często stosowaną, metodą jest wybór pierwszego elementu tablicy $A[p..r]$ jako elementu dzielącego. W naszym algorytmie sprowadza się ona do pominięcia wywołania *choosepivot*(A, p, r).

Metoda ta oczywiście może prowadzić do nierównomiernych podziałów. W szczególności, czas kwadratowy jest osiągany, gdy dane wejściowe są uporządkowane. Z drugiej strony, na losowych danych algorytm działa bardzo szybko.

4.2.2 Prosty wybór zrandomizowany

Jako pivot obieramy losowy element spośród elementów $A[p..r]$.

[3]Medianą zbioru S nazywamy taki jego element, który jest większy od dokładnie $\lfloor |S|/2 \rfloor$ elementów zbioru S. Definicja w naturalny sposób uogólnia się na wielozbiory.

```
procedure choosepivot(A[1..n], p, r)
    i ← random(p, q);
    zamień A[p] i A[i] miejscami
```

Przy takim wyborze pivota również może się zdarzyć, że algorytm będzie działać w czasie kwadratowym, jednak prawdopodobieństwo takiego zdarzenia jest zaniedbywalnie małe.

Zasadnicza różnica w stosunku do metody deterministycznej polega na tym, że teraz przebieg algorytmu zależy nie tylko od danych wejściowych, ale także od generatora liczb losowych (pseudolosowych). W szczególności teraz nie istnieją dane wejściowe lepsze i gorsze. Na każdych algorytm może działać jednakowo szybko i na każdych może się zdarzyć, że będzie działać w czasie kwadratowym.

4.2.3 Mediana z małej próbki

Często stosowaną metodą jest wybieranie jako pivota mediany z trzech losowo wybranych elementów tablicy. To prowadzi do istotnego zmniejszenia prawdopodobieństwa nierównomiernych podziałów. Ceną jest konieczność wykonania dwóch dodatkowych porównań i przede wszystkim dwóch dodatkowych wywołań generatora liczb losowych.

"Medianę z trzech" stosuje się także w wersji deterministycznej. Najczęściej wybiera się ją wówczas spośród pierwszego, środkowego i ostaniego elementu tablicy.

Eksperymentalnie stwierdzono, że zastosowanie "mediany z trzech' zamiast prostego wyboru pivota prowadzi do przyspieszenia *quicksortu* o kilka do kilkunastu procent (zależnie od zastosowanej wersji wyboru elementów i sprawności implementacyjnej przeprowadzającego eksperymenty).

Metodę tę można rozszerzać na liczniejsze próbki, jednak uzyskane zyski czasowe są znikome.

4.3 Średni koszt algorytmu

Załóżmy, że jako pivot wybierany jest z jednakowym prawdopodobieństwem dowolny element tablicy. Pokażemy, że przy tym założeniu średni koszt algorytmu *quicksort* wynosi $\Theta(n \log n)$. Dla uproszczenia analizy założymy ponadto, że wszystkie elementy sortowanej tablicy są różne.

Niech $n = r - p + 1$ oznacza liczbę elementów w $A[p..r]$ i niech

$$rank(x, A[p..r]) \stackrel{df}{=} |\{j : p \leq j \leq r \text{ i } A[j] \leq x\}| .$$

Ponieważ w momencie wywoływania procedury *partition* w $A[p]$ znajduje się losowy element z $A[p..r]$, więc wówczas

$$\forall_{i=1,..,n} \quad \Pr[rank(A[p], A[p..r]) = i] = \frac{1}{n}.$$

Wynik procedury *partition* w oczywisty sposób zależy od wartości $rank(A[p], A[p..r])$. Gdy jest ona równa i (dla $i = 2, \ldots, n$), wynikiem *partition* jest $p + i - 2$. Ponadto, gdy $rank(A[p], A[p..r]) = 1$, wynikiem jest p. Tak więc zmienna q z procedury *quicksort* przyjmuje wartość p z prawdopodobieństwem $2/n$, a każdą z pozostałych wartości (tj. $p + 1, p + 2, \ldots, r - 1$) z prawdopodobieństwem $1/n$. Stąd oczekiwany czas działania procedury *quicksort* wyraża się równaniem

$$\begin{cases} T(1) = & 1 \\ T(n) = & \frac{1}{n}\left[(T(1) + T(n-1)) + \sum_{d=1}^{n-1}(T(d) + T(n-d))\right] + \Theta(n) \end{cases}$$

Zmienna $d = q - p + 1$ oznacza długość pierwszej z podtablic.

Ponieważ $T(1) = \Theta(1)$ a $T(n-1)$ w najgorszym przypadku jest równe $\Theta(n^2)$, więc

$$\frac{1}{n}(T(1) + T(n-1)) = O(n).$$

To pozwala nam pominąć ten składnik, ponieważ będzie on uwzględniony w ostatnim członie sumy. Tak więc:

$$T(n) = \frac{1}{n}\sum_{d=1}^{n-1}(T(d) + T(n-d)) + \Theta(n).$$

W tej sumie każdy element $T(k)$ jest dodawany dwukrotnie (np. $T(1)$ raz dla $q = 1$ i raz dla $q = n - 1$), więc możemy napisać:

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \tag{1}$$

Ponieważ mamy silne przesłanki, by przypuszczać, że rozwiązanie tego równania jest rzędu $\Theta(n \log n)$, ograniczymy się do sprawdzenia tego faktu. Niech

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \leq an \log n + b$$

dla pewnych stałych $a, b > 0$. Naszym zadaniem jest pokazanie, że takie stałe a i b istnieją.

Bierzemy b wystarczająco duże by $T(1) \leq b$. Dla $n > 1$ mamy:

$$T(n) \quad = \quad \frac{2}{n}\sum_{k=1}^{n-1}(ak \log k + b) + \Theta(n) \leq \frac{2a}{n}\sum_{k=1}^{n-1} k \log k + \frac{2b}{n}(n-1) + \Theta(n)$$

Proste oszacowanie $\sum_{k=1}^{n-1} k \log k$ przez $\frac{1}{2}n^2 \log n$ nie prowadzi do celu, ponieważ musimy pozbyć się składnika $\Theta(n)$. Oszacujmy więc $\sum_{k=1}^{n-1} k \log k$ nieco staranniej:

Fakt 14 $\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$

Dowód. Rozbijamy sumę na dwie części:

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

Szacując $\log k$ przez $\log \frac{n}{2}$ dla $k < \lceil \frac{n}{2} \rceil$ oraz przez $\log n$ dla $k \geq \lceil \frac{n}{2} \rceil$, otrzymujemy:

$$\sum_{k=1}^{n-1} k \log k \leq ((\log n) - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k = \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \leq$$
$$\frac{1}{2} n(n-1) \log n - \frac{1}{2}(\frac{n}{2} - 1)\frac{n}{2} \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

□

Teraz możemy napisać

$$\frac{2a}{n}\left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \leq an \log n - \frac{a}{4} n + 2b + \Theta(n) =$$
$$an \log n + b + \left(\Theta(n) + b - \frac{a}{4} n\right)$$

Składową $\left(\Theta(n) + b - \frac{a}{4} n\right)$ możemy pominąć, dobierając a tak, by $\frac{a}{4} n \geq \Theta(n) + b$. Zauważmy, że taki dobór zależy jedynie od stałej b oraz od stałej ukrytej pod Θ, a więc za a można przyjąć odpowiednio dużą stałą.

To kończy sprawdzenie, że $T(n) \leq an \log n + b$ dla pewnych stałych $a, b > 0$.

4.4 Inne usprawnienia

Quicksort jest dość powszechnie uważany za najszybszą (a przynajmniej jedną z najszybszych) metodę sortowania. Jego znaczenie spowodowało, że wiele wysiłku włożono w opracowanie modyfikacji, mających na celu uzyskanie jak największej efektywności. Poniżej wymieniamy kilka z nich:

- Trójpodział. W przypadku, gdy spodziewamy się, że sortowane klucze mogą się wielokrotnie powtarzać (np. gdy przestrzeń kluczy jest mała), opłacalne może być zmodyfikowanie procedury *partition* tak, by dawała podział na trzy części: elementy mniejsze od pivota, równe pivotowi i większe od pivota. Oczywiście *quicksort* jest rekurencyjnie wywoływany jedynie do pierwszej i trzeciej części. W przypadku, gdy liczba elementów równych pivotowi jest znaczna, może to przynieść istotne przyspieszenie.

- Eliminacja rekursji.

  - Tak jak w przypadku wszystkich algorytmów opartych na strategii dziel i zwyciężaj, spory zysk można otrzymać, starannie dobierając próg na rozmiar danych, poniżej którego opłaca się zastosować prosty algorytm nierekurencyjny w miejsce rekurencyjnych wywołań procedury *quicksort*.
  - W wielu implementacjach *quicksortu* przeznaczonych do powszechnego użytku (np. w bibliotekach procedur) w ogóle wyeliminowano rekursję.

- Optymalizacja pętli wewnętrznej, aż do zapisania jej w języku wewnętrznym procesora.

- W zastosowaniach, w których krytycznym zasobem jest pamięć (np. w układach realizujących sortowanie hardware'owo), stosowana bywa nierekurencyjna wersja (rekursja wymaga pamięci na stos wywołań) działająca "w miejscu", a więc wykorzystująca co najwyżej $O(1)$ komórek pamięci poza tymi, które zajmuje sortowany ciąg.

Notatki z AiSD. Nr ??. 16 marca 2005

Sortowanie c.d.

IIUWr. II rok informatyki. Przygotował: 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.

5 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, var B[1..n])
    for i ← 1 to k do c[i] ← 0
    for j ← 1 to n do c[A[j]] ← c[A[j]] + 1
    for i ← 2 to k do c[i] ← c[i] + c[i − 1]
    for j ← n downto 1 do  B[c[A[j]]] ← A[j]
                           c[A[j]] ← 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.

Definicja 10 *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 15 *Counting − sort jest metodą stabilną.*

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

6 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 ← 0 to n − 1 do B[i] ← ∅
    for i ← 1 to n do  dołącz A[i] do listy B[⌊nA[i]⌋]
    for i ← 0 to n − 1 do  posortuj procedurą insert − sort listę B[i]
    połącz listy B[0], B[1], ..., B[n − 1]
```

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

7 Sortowanie leksykograficzne ciągów jednakowej długości (radix sort).

Definicja 11 *Niech S - zbiór uporządkowany liniowo oraz $s_1, \ldots, s_p, t_1, \ldots, t_q \in S$.*

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

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

```
procedure radix − sort(A_1, .., A_n)
    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 $k = O(n)$.

8 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, $l_{max} = \max\{l_i\ :\ i = 1, .., n\}$.

8.1 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.

8.2 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 $j \leftarrow 0$ **to** $k-1$ **do** $q[j] \leftarrow \emptyset$
   for $l \leftarrow l_{max}$ **downto** 1 **do**
   $queue \leftarrow concat(length[l], queue)$
   while $queue \neq \emptyset$ **do**
   $Y \leftarrow$ pierwszy ciąg z $queue$
   $queue \leftarrow queue \setminus \{Y\}$
   $a \leftarrow l$-ta składowa ciągu Y
   $q[a] \leftarrow concat(q[a], \{Y\})$
   for each $j \leftarrow nonempty[l]$ **do**
   $queue \leftarrow concat(queue, q[j])$
   $q[j] \leftarrow \emptyset$

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

Twierdzenie 5 *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 $O(l_{total})$. □

8.3 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. $S_i \leftarrow$ zbiór wierzchołków T_i z poziomu j nie będących liśćmi
4. $\forall_{v \in S_i}\ key(v) \leftarrow$ 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. $L_i \leftarrow$ lista wierzchołków z S_i posortowana leksykograficznie według wartości key
6. $L_i' \leftarrow$ otrzymany w ten sposób uporządkowany ciąg wektorów
7. **if** $L_1' \neq L_2'$ **then return** ("nieizomorficzne")
8. $\forall_{v \in L_i}\ kod(v) \leftarrow 1 + rank(key(v), \{key(u) \mid u \in L_i\})$
9. Na początek L_i dołącz wszystkie liście z poziomu j drzewa T_i
10. **return** ("izomorficzne")

Twierdzenie 6 *Izomorfizm dwóch drzew o n wierzchołkach może być sprawdzony w czasie* $O(n)$.

Algorytmy i Struktury Danych, 5. ćwiczenia

2009-11-10

1 Plan zajęć

- izomorfizm drzew,
- d-kopce,

2 Izomorfizm drzew

Algorytm:

TreeIsomorphism(T1,T2,depth)

```
1: if T1.height > depth then
2:    return (T1.height = T2.height);
3: end if
4: if not TreeIsomorphism(T1,T2,depth+1) then
5:    return false;
6: end if
7: for v ∈ T1.nodes[depth + 1] ∪ T2.nodes[depth + 1] do
8:    {w porządku rosnących etykiet}
9:    dodaj value(v) do listy wierzchołka parent(v)
10: end for
11: posortuj leksykograficznie listy value(v) dla v ∈ T1.nodes[depth]
12: posortuj leksykograficznie listy value(v) dla v ∈ T2.nodes[depth]
13: porównaj czy listy są identyczne, jeśli nie to return false
14: zamień etykiety value(v) na liczby z zakresu 1, ..., n
15: return true
```

3 Izomorfizm drzew — algorytm dla drzew nieskierowanych

Znajdź w drzewach centroidy (każde drzewo zawiera co najwyżej 2 centroidy), dla każdej kombinacji ukorzeń drzewa w centroidach i uruchom poprzedni algorytm.

Niech $w(x) = max\{|subtree(t_i)| : t_i \in adj(x)\}$. *Centroid* — wierzchołek o minimalnej wadze $w(x)$.

Find(v)

```
1: niech c1, ..., c_k synowie wierzchołka v,
```

2: jeśli $subtree(c_i) \leq n/2$ dla $1 \leq i \leq k$, to **return** v,
3: wpp. niech c_j wierzchołek, taki, że $subtree(c_j) > n/2$ (jest tylko jeden o tej własności),
4: **return** FIND(c_j)

FINDCENTROID(V)

1: ukorzeń drzew w dowolnym wierzchołku r,
2: oblicz wartości $subtree(v)$ dla wszystkich wierzchołków,
3: **return** FIND(r)

4 d-kopce

d–kopiec do drzewo zupełne o stopniu d z porządkiem kopcowym (min w korzeniu). Należy pokazać, że poszczególne operacje wykonuje się w czasie:

- Min — $O(1)$
- DeleteMin — $O(d \cdot \log_d(n))$
- DecreaseKey — $O(\log_d(n))$

Koszt implementacji algorytmu Dijkstry, przy użyciu d–kopców: $O(nd \cdot \log_d(n) + m \cdot \log_d(n))$.

Zanalizować jak należy dobrać d w zależności od m i n (jeśli za d weźmiemy $max(2, \lceil m/n \rceil)$ to dostajemy $O(\frac{m \log n}{\log m/n})$).

5 Rozgłaszanie komunikatów

Dane drzewo T, należy obliczyć czas potrzebny na przesłanie komunikatów do wszystkich węzłów drzewa. Przesłanie komunikatu po jednej krawędzi zajmuje 1 jednostkę czasu.
Algorytm $O(n \log n)$:

- jeśli wierzchołek jest liściem to $czas = 0$,
- wpp. rekurencyjnie oblicz czas potrzebny na rozgłoszenie w poddrzewach,
- posortuj malejąco otrzymane czasy: $t_1, \ldots, t_k$
- $czas = max\{i + t_i : 1 \leq i \leq k\}$

Aby otrzymać algorytm $O(n)$ trzeba sprytnie obliczać wartości atrybutu $czas$.

- $Q = \{$ liście $T \}$,
- while $root \notin Q$ do
  - $x = Q.extractMin()$
  - dodaj $x.czas$ do kolejki $parent(x)$,
  - jeśli $parent(x)$ ma już pełną listę poddrzew, to policz $parent(x).czas$ i dodaj $parent(x)$ do kolejki.

Kolejkę Q można zaimplementować w tablicy (i–ty element tablicy zawiera listę wierzchołków o wartości $x.czas = i$). Sumarycznie operacje $extractMin$ zajmą czas $O(n)$. Dodawanie do kolejki zajmuje czas $O(1)$.