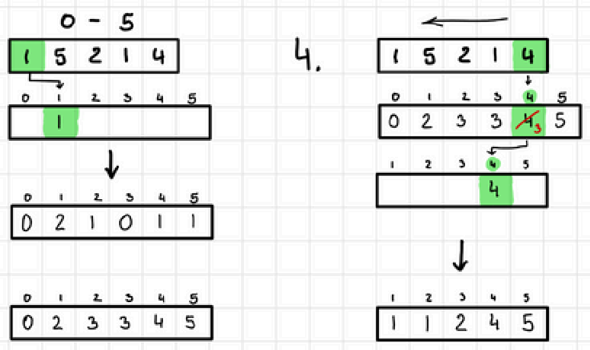
Sortowanie

```
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
```

1) Zainicjuj nową tablicę c (o długości zakresu zmiennych) zerami
2) Zlicz ilości występowania elementów w tablicy A do c (na i-tym indeksie w tablicy c znajduje się ilość występowań i w tablicy A
3) Do każdego elementu w tablicy c dodajemy poprzedni
4) Iterując od końca tablicy A wstawiamy jej element do nowej tablicy na miejsce odpowiadające c[element] i zmniejszamy tą komórkę o jeden tzn: c[e] = c[e] - 1
(Iterowanie od końca gwarantuje nam stabilność sortowania)

Przykład:



Charakterystyka:

✓ Algorytm działa w czasie $O(n+k)$ ← długość przedziału

✗ Dane są ograniczone

Counting Sort jest stabilny.

Def: Metodę sortowania nazywamy stabilną Jeśli elementy o tej samej wartości w ciągu wyjściowym pozostają w takiej samej kolejności względem siebie w jakiej były w ciągu wejściowym

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

```
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 select − sort listę B[i]
    połącz listy B[0], B[1], . . . , B[n − 1]
```

Sortowanie elementów 0-1 np. [0.41, 0.91, 0.49, 0.48, 0.24]

1. Stwórz listę n kubełków (array of linked lists)
2. Przypisz każdy element do kubełka wg. $\lfloor n \cdot arr[i] \rfloor$
3. Posortuj kubełki za pomocą insertSort ← który jest w miarę optymalny dla list wiązanych oraz jest stabilny.

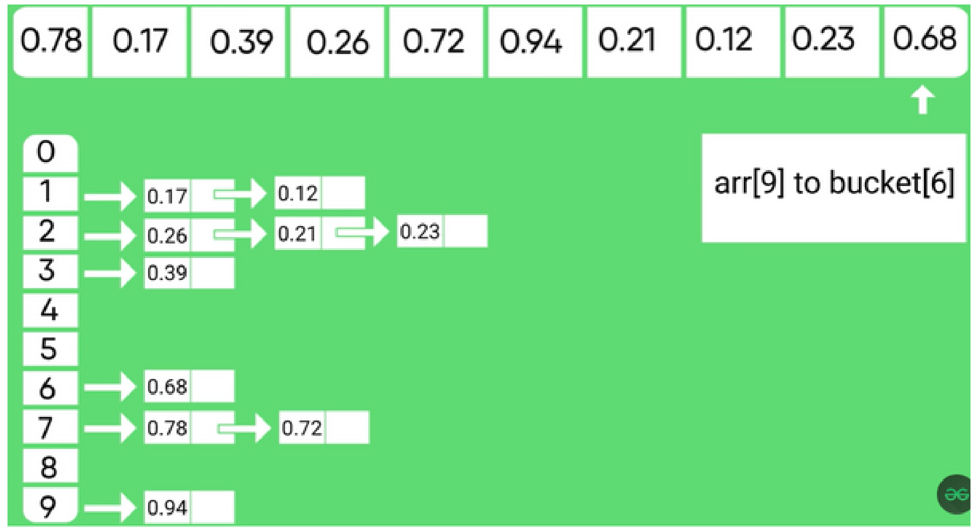
Charakterystyka:

✓ Złożoność oczekiwana $O(n)$

✗ dane z konkretnego przedziału

✗ zależy od danych (pesymistycznie $O(n^2)$)

Bucket Sort jest stabilny jeśli korzysta ze stabilnego alg. sortowania.



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

UZASADNIENIE: Niech Y będzie zmienną losową równą liczbie porównań wykonanych podczas sortowania kubełków. Mamy

$$Y = Y_1 + \ldots + Y_n,$$

gdzie Y_i jest zmienną losową równą liczbie porównań wykonanych podczas sortowania i-tego kubełka. Z liniowości wartości oczekiwanej mamy

$$E[Y] = \sum_{i=1}^{n} E[Y_i].$$

Niech X_i będzie zmienną losową równą liczbie elementów w i-tym kubełku. Oczywiście X_i ma rozkład dwumianowy, w którym prawdopodobieństwo sukcesu wynosi $1/n$. Ponieważ do sortowania kubełków stosujemy *select-sort*, mamy $Y_i = X_i^2$. Stąd wystarczy teraz oszacować $E[X_i^2]$. □

POSTAĆ DANYCH: $A_1, \ldots, A_n$ - ciągi elementów z $\Sigma = \{0, 1, \ldots, k-1\}$ o długości 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 $k = O(n)$.

POSTAĆ DANYCH: $A_1, \ldots, A_n$ ciągi elementów z $\Sigma = \{0, 1, \ldots, k-1\}$.

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

4.1 Pierwszy sposób

IDEA: Uzupełnić ciągi specjalnym elementem (mniejszym od każdego elementu z Σ), 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

Chcemy opracować metodę sortującą w czasie liniowym względem rozmiaru danych, który jest rów $l_{total} = \sum_{i=1}^{n} l_i$.

IDEA:

```
for i ← l_max downto 1 do
    metodą stabilną posortuj ciągi o długości ≥ i wg i-tej składowej
```

ALGORYTM:

1. Utwórz listy *niepustekubelki*$[l]$ $(l = 1, .., l_{max})$ takie, że
   - $x \in$ *niepustekubelki*$[l]$ iff x jest l−tą składową jakiegoś ciągu A_i.
   - *niepustekubelki*$[l]$ jest uporządkowana niemalejąco.
2. Utwórz listy *ciagi*$[l]$ $(l = 1, .., l_{max})$ takie, że *ciagi*$[l]$ zawiera wszystkie ciągi A_i o długości l.
3. *kolslów* ← ∅

```
   for j ← 0 to k − 1 do q[j] ← ∅
   for l ← l_max downto 1 do
      kolslów ← concat(ciagi[l], kolslów)
       while kolslów ≠ ∅ do
          Y ← pierwszy ciąg z kolslów
          kolslów ← kolslów \ {Y}
          a ← l-ta składowa ciągu Y
          q[a] ← concat(q[a], {Y})
       for each  j ← niepustekubelki[l] do
          kolslów ← concat(kolslów, 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* $O(k + l_{total})$.

UZASADNIENIE:
Jedynym niezupełnie trywialnym krokiem jest krok 1, w którym tworzone są listy *niepustekubelki*:

- tworzymy w czasie $O(l_{total})$ ciąg P 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 P;
- przeglądając P z lewa na prawo tworzymy $O(l_{total})$ listy *niepustekubelki*.

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

Aby oszacować czas wykonania kroku 3, oszacujemy czas wykonania dwóch jego pętli wewnętrznych:

- Wewnętrzna pętla while działa w czasie proporcjonalnym do sumarycznej (po wszystkich iteracjach pętli zewnętrznej) długości kolejek *kolslw*. Ponieważ w l-tej iteracji *kolslw* 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 *niepustekubelki*. Ponieważ w każdej iteracji *niepustekubelki* jest nie dłuższa od *kolslw*, 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 będą tego samego typu, jeśli poddrzewa w nich zakorzenione będą izomorficzne).

ALGORYTM:

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

- wysokość,
- liczbę liści na każdym poziomie.

1. $\forall_{v \text{ – liść w } T_i}$ $kod(v) \leftarrow 0$
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
   − $i_1 \le i_2 \le \ldots \le i_k$
   − 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 2 *Izomorfizm dwóch ukorzenionych drzew o n wierzchołkach może być sprawdzony w czasie* $O(n)$.

Izomorfizm Drzew

1. Przejdź przez drzewo i przypisz każdemu węzłowi liczbę dzieci
2. Zacznij od korzenia
3. Wywołaj się rekurencyjnie - jeśli zwracamy jest false, to zwróć false.
4. Weź dzieci tego węzła w drzewie T_1 i T_2, a następnie posortuj je po ilości ich dzieci z kubełkami pierwszego.
5. Jeśli posortowane dzieci węzła z drzewa T_1 są równe tym z drzewa T_2 pod kątem liczby ich dzieci, to zwróć prawda wpp. false.

Który z poniższych algorytmów sortowania może w najgorszym przypadku wykonać $\Omega(n^2)$ porównań:

a) quicksort

b) mergesort

c) insertsort

Przypomnienie: $\Omega(n^2)$ oznacza nie mniej niż cn^2 dla pewnej stałej $c > 0$.

a) quicksort - jeżeli znajdowanie pivota jest źle zrobione, tzn. odcina stałą liczbę elementów przy każdym partition, np. stosunek $1 : (n-1)$, to wtedy złożoność to $n * O(n) = O(n^2)$

b) mergesort - worst case $O(n \log n)$

c) insertsort - posortowany lub odwrotnie posortowany ciag bedzie mieć $O(n^2)$

Treść

Jak uogólnić wykładowy algorytm szukania izomorfizmów drzew ukorzenionych na drzewa nieukorzenione.

Jaką ma złożoność nowy algorytm?

▼ Rozwiązanie

Ukorzeniamy drzewa w centroidzie.

Znalezienie centroidu robi się liniowo, np. BFS-em.

Dalej stosujemy algorytm wykładowy.

Treść

Przedstaw ideę szybkiego algorytmu sprawdzania izomorfizmu drzew. W jakim czasie działa ten algorytm?

Treść

Porównaj trudność problemu sprawdzania izomorfizmu drzew ukorzenionych i problemu sprawdzania izomorfizmu drzew nieukorzenionych.

2. (Z 2pkt) Ułóż algorytm sortujący stabilnie i w miejscu ciągi rekordów o kluczach ze zbioru $\{1, 2, 3\}$.

3. (1pkt) Podaj algorytm sprawdzający izomorfizm drzew nieukorzenionych.