Algorytmy i Struktury Danych

Wykład 2

Złożoność czasowa — to funkcja opisująca liczbę elementarnych operacji w zależności od rozmiaru danych

- pesymistyczna — liczba operacji w najgorszym przypadku
- optymistyczna — minimalna liczba operacji
- oczekiwana — liczba operacji w przypadku średnim

Złożoność pamięciowa — funkcja opisująca liczbę wykorzystywanych komórek pamięci

Przykład

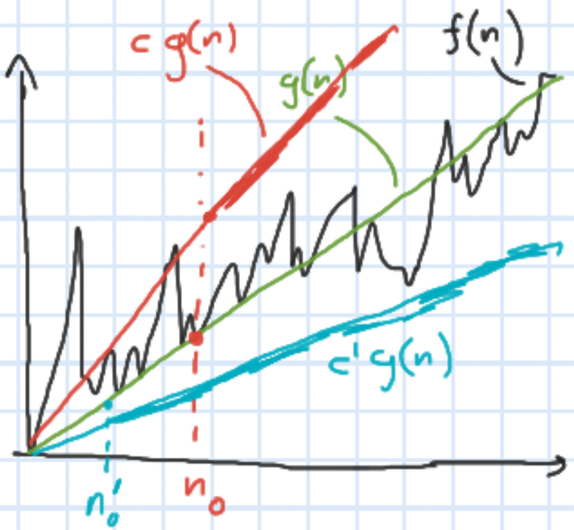
$7n^3 + 10n^2 + 3n\log n$ jest $\Theta(n^3)$

Notacja asymptotyczna

Niech f i g będą funkcjami

$f : \mathbb{N} \to \mathbb{N}$

$g : \mathbb{N} \to \mathbb{N}$



def Mówimy, że f jest $O(g(n))$ jeśli:

$$(\exists c>0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\left[f(n) \leq c \cdot g(n)\right]$$

Mówimy, że f jest $\Omega(g(n))$ jeśli

$$(\exists c>0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\left[f(n) \geqslant c \cdot g(n)\right]$$

Mówimy, że f jest $\Theta(g(n))$ jeśli f jest $O(g(n))$ jak i $\Omega(g(n))$

Problem sortowania

Dane: ciąg $a_0, \dots, a_{n-1}$ danych
razem z operatorem $\leqslant$

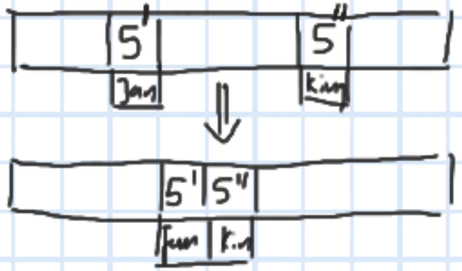
Wynik: permutacja $a_{\pi(0)}, \dots, a_{\pi(n-1)}$
ciągu wejściowego, taka że

$$a_{\pi(0)} \leqslant a_{\pi(1)} \leqslant \dots \leqslant a_{\pi(n-1)}$$

Uwagi

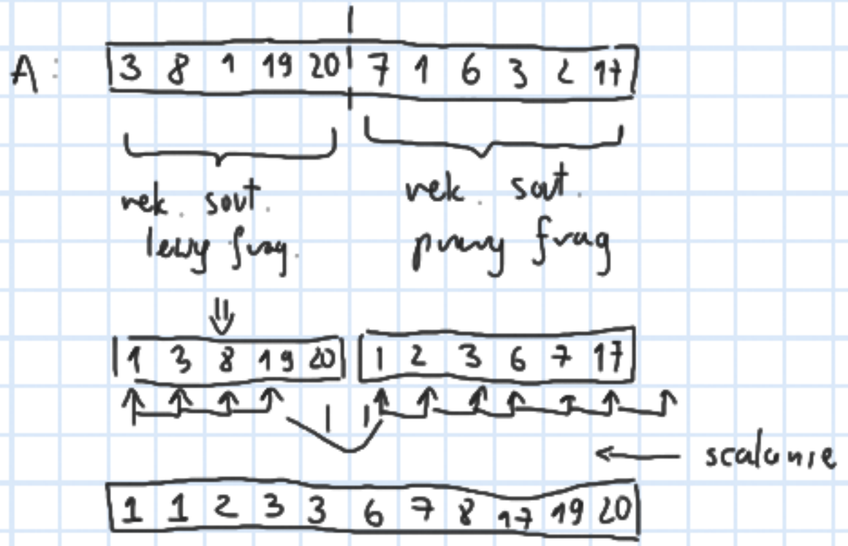
Reprezentacja danych — tablica
— listy 1-kier.
2-kier.
— plik

Stabilność sortowania — czy elementy "równe" mogą zmienić kolejność?



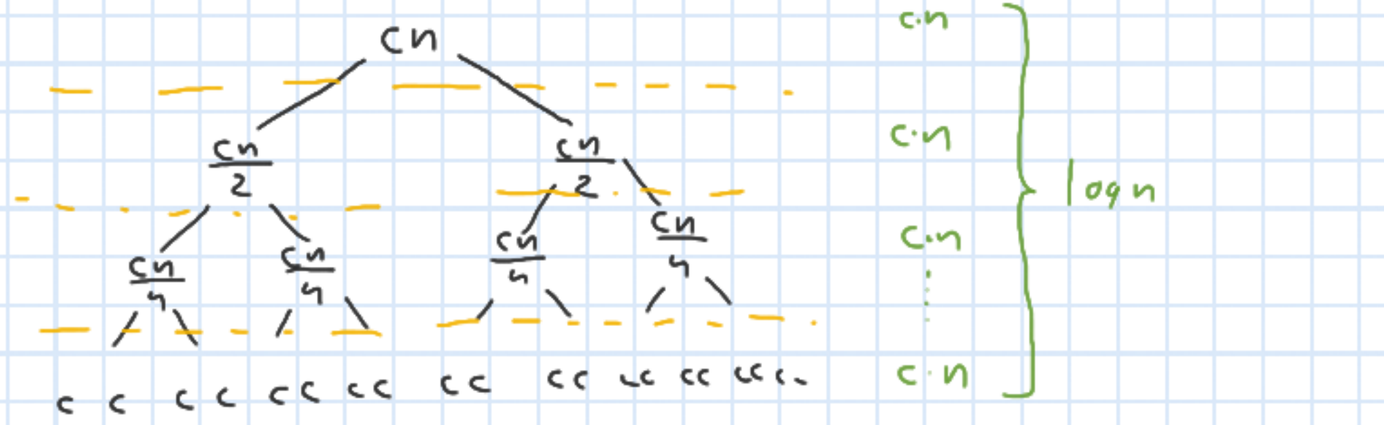
Szybkie sortowanie
↳ proste $O(n^2)$
↳ szybkie $O(n \log n)$

① Sortowanie przez scalanie (merge sort)



$$T(n) = \begin{cases} c, & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn, & n > 1 \end{cases}$$

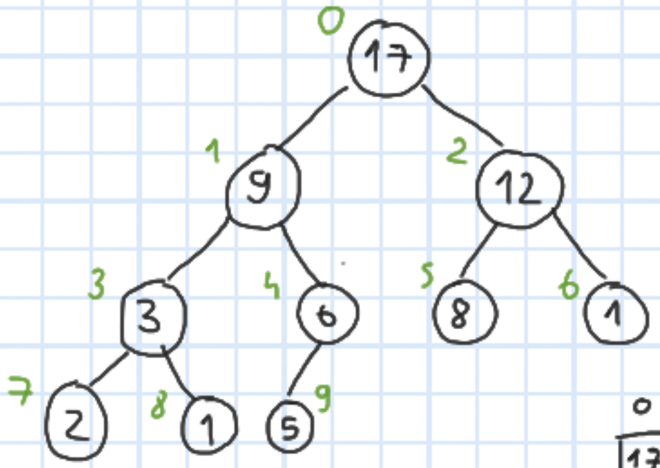
$T(n) = \Theta(n \log n)$



② Sortowanie kopcowe (Heapsort)

Kopiec — drzewo binarne, w którym w każdym węźle A przechowywany wartość ≥ niż wartości w jego dzieciach

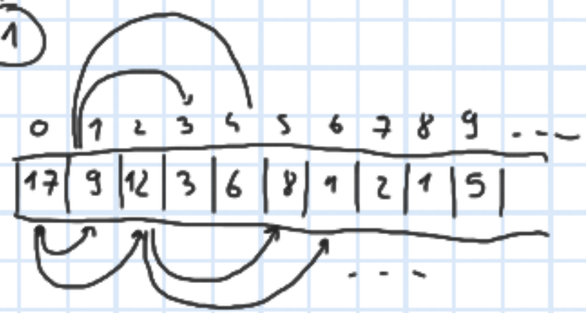
Przykład

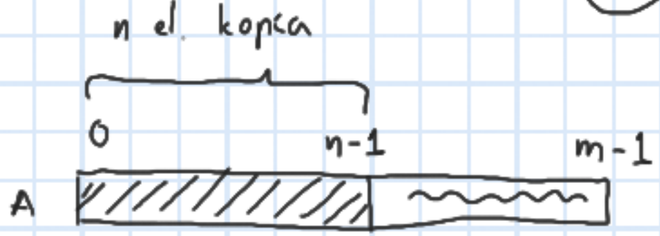


$left(i) = 2i + 1$

$right(i) = 2i + 2$

$parent(i) = \lfloor (i-1)/2 \rfloor$





$len(A) = m$

```
def heapify(A, n, i)
    l = left(i)      , max_ind = i
    r = right(i)
    if l<n and A[l] > A[max_ind]:
        max_ind = l
    if r<n and A[r] > A[max_ind]:
        max_ind = r
    if max_ind ≠ i:
        swap(A[i], A[max_ind])
        heapify(A, n, max_ind)
```

$O(\log n)$

```
def build_heap(A):
    n = len(A)
    for i in range(parent(n-1), -1, -1):
        heapify(A, n, i)
```

$O(n \log n)$

$\Theta(n)$

W tablicy mamy n elementów

$\frac{n}{2}$ tych elementów tworzy kopce o wys. 0

$\frac{n}{4}$ tych el., tworzy kopce o wys 1

$\frac{n}{8}$ tych el. tworzy kopce o wys. 2

$\vdots$

$\left\lceil \frac{n}{2^{h+1}} \right\rceil$ tych el tworzy kopce o wys. h

Koszt build heap wyrażamy jako:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h = O\left(\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n \cdot h}{2^{h+1}} \right)$$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

$$f(x) = 1 + x + x^2 + x^3 + \ldots = \frac{1}{1-x}$$

$$f'(x) = 1 + 2x + 3x^2 + \ldots = \frac{1}{(1-x)^2}$$

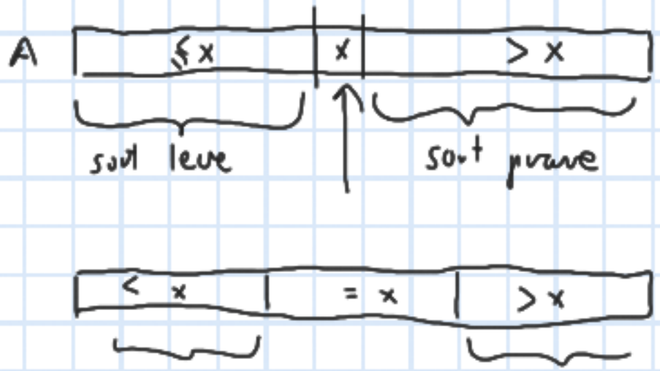
$$x f'(x) = x + 2x^2 + 3x^3 + \ldots = \frac{x}{(1-x)^2} \xrightarrow{x = \frac{1}{2}} 2$$

```
def heapsort(A):
    n = len(A)
    buildheap(A)
    for i in range(n-1, 0, -1):
        swap(A[i], A[0])
        heapify(A, i, 0)        <-- O(log n)
```

0	1	2	3	4	5	6	7	8	9
1	9	8	3	6	5	1	2	12	17

$\Theta(n \log n)$

(3) Quick sort

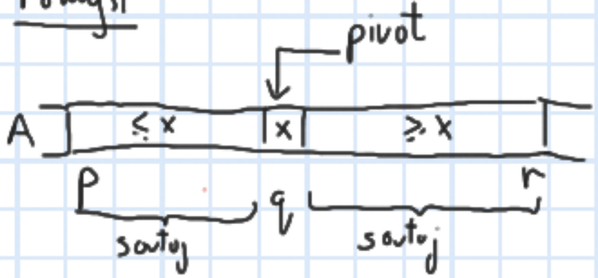


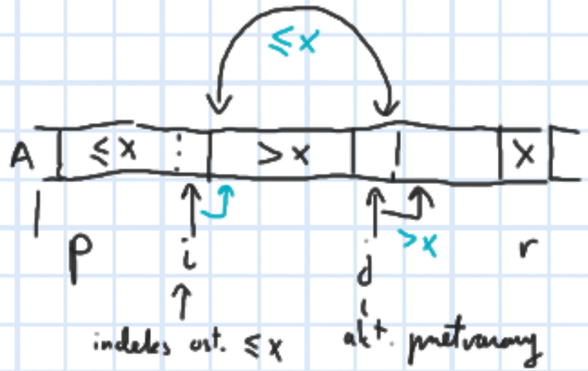
Algorytmy i Struktury Danych

Wykład 3

Algorytm Quick Sort

Pomysł






```
def quicksort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quicksort(A, p, q-1)
        quicksort(A, q+1, r)
```

```
def partition(A, p, r):
    x = A[r]
    i = p-1
    for j in range(p, r):
        if A[j] ≤ x:
            i += 1
            swap(A[i], A[j])
    swap(A[i+1], A[r])
    return i+1
```

Algorytm Lomuto

Algorytm Hoare'a

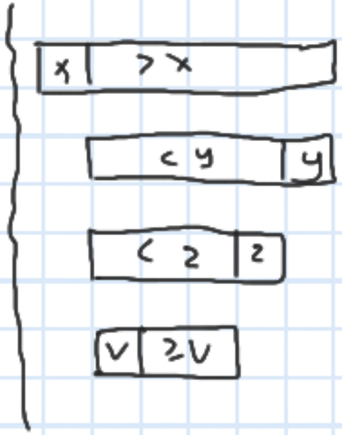
Złożoność obliczeniowa

Idealne podziały

$$T(n) = \begin{cases} c, & n \le 1 \\ 2T\left(\frac{n}{2}\right) + cn, & n > 1 \end{cases} \quad T(n) = \Theta(n \log n)$$

Pesymistyczne podziały

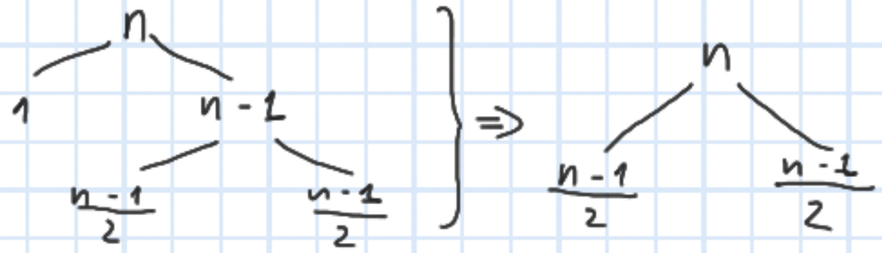
$$T(n) = \begin{cases} c, & n \le 1 \\ T(n-1) + cn, & n > 1 \end{cases}$$



$$T(n) = T(n-1) + cn$$
$$= T(n-2) + c(n-1) + cn$$
$$\vdots$$
$$= c + 2c + 3c + \ldots + c(n-1) + cn$$
$$= c\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Mieszane podziały

Na co drugim poziomie rekurencji podziały pechowe, a na co drugim idealne



Quicksort bez rekurencji ogonowej

```
def quicksort(A, p, r):
    while p < r:
        q = partition(A, p, r)
        quicksort(A, p, q-1)
        p = q+1
```

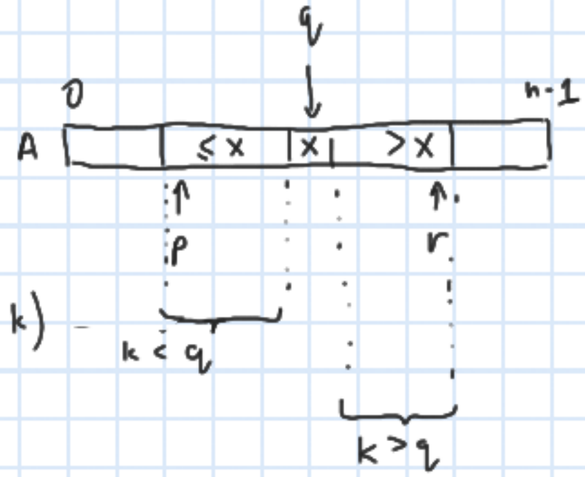
Statystyki pozycyjne

k-ta statystyka pozycyjna — element na pozycji k po posortowaniu

min/max — oczywisty algorytm $\Theta(n)$

mediana — ??

```
def select(A, p, r, k):
    if p == r: return A[p]
    q = partition(A, p, r)
    if q == k: return A[q]
    elif k < q: return select(A, p, q-1, k)
    else: return select(A, q+1, r, k)
```



$$T(n) = \begin{cases} c, & n \le 1 \\ T\left(\frac{n}{2}\right) + cn, & n > 1 \end{cases} \Rightarrow$$

$$\begin{aligned} T(n) &= cn + T\left(\frac{n}{2}\right) \\ &= cn + \frac{cn}{2} + T\left(\frac{n}{4}\right) \\ &\vdots \\ &= c\left(n + \frac{n}{2} + \frac{n}{4} + \ldots + 1\right) \\ &= 2cn = \Theta(n) \end{aligned}$$

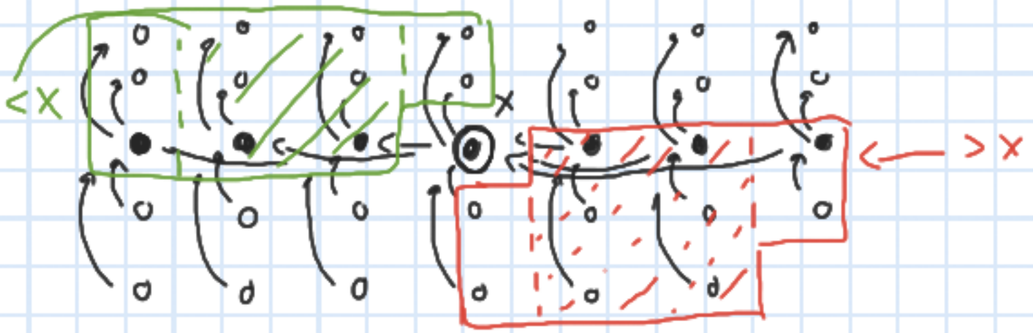
Magiczne piątki – statystyki pozycyjne w czasie liniowym

Algorytm

① podziel tablicę na $\lceil \frac{n}{5} \rceil$ grup po 5 elementów, w każdej wyznaczamy medianę

② rekurencyjnie wyznacz x jako medianę median

③ kontynuuj jak w zwykłym "select", ale używając x do wykonania partition (jako pivot)



Ile jest elementów większych od x?

$$3 \cdot \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$

Złożoność czasowa

$$T(n) = \begin{cases} \Theta(1) & , n \leq \text{pewna stała} \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6) + \Theta(n) & , n \geq \bullet \end{cases}$$

$T(\lceil \frac{n}{5} \rceil)$: med. median

$T(\frac{7n}{10} + 6)$: zajmuje jak w select

Twierdzimy, że $T(n) \leq cn$ dla pewnej stałej c

Dowód indukcyjny:

$$T(n) \leq c \lceil \tfrac{n}{5} \rceil + \frac{7nc}{10} + 6c + an$$

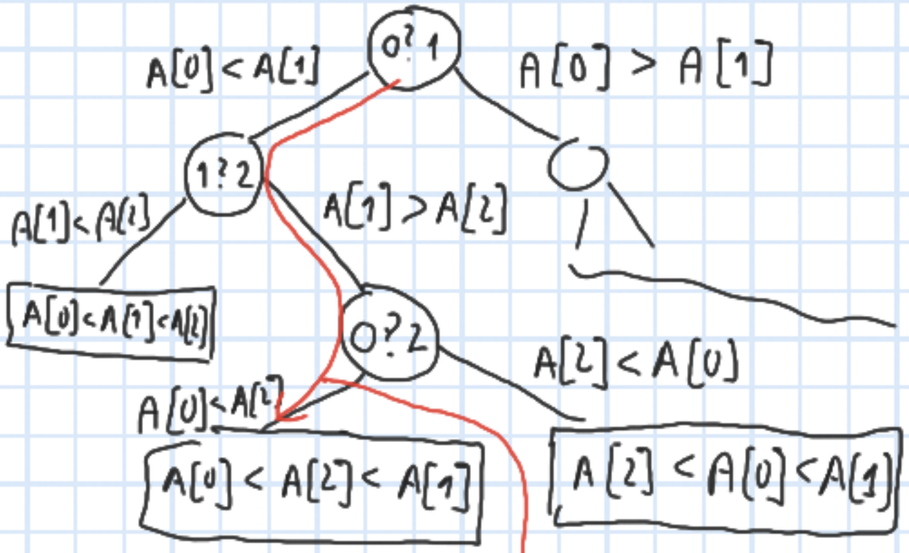
(an: jakaś stała)

$$\leq \frac{2cn}{10} + \frac{7cn}{10} + 6c + an + c$$

$$= cn + \left(-\frac{1}{10} cn + 7c + an \right)$$

wybieramy c tak duże, że ta wartość jest ujemna dla odp. dużych n

Dolne ograniczenie na złożoność sortowania



najdłuższa ścieżka w tutaj drzewie
to pesymistyczna liczba porównań do wykonania

$$\frac{n}{2}(\log n - 1) = \frac{n}{2}\log\frac{n}{2} = \log\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq \log n! \leq \log n^n = n\log n$$

$$\downarrow$$

$$\Theta(n\log n)$$

Jeśli A ma n elementów to

drzewo musi mieć $\geq n!$ liści

Drzewo o wysokości h ma najwyżej 2^h liści
(binarne)

$$n! \leq 2^h$$

$$h \geq \log n!$$

Algorytmy i Struktury Danych

Wykład 4

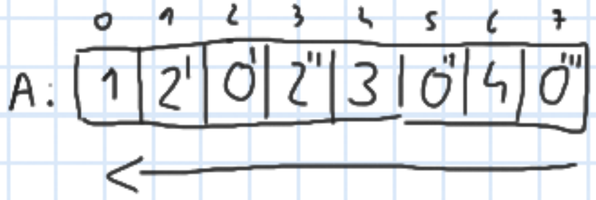
Sortowanie w czasie liniowym – sortowanie przez zliczanie

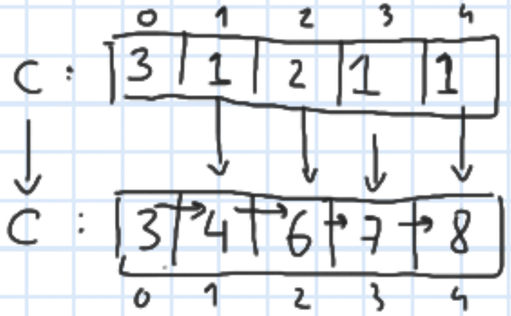
tablica zawiera liczby naturalne ze zbioru $\{0, \ldots, k-1\}$

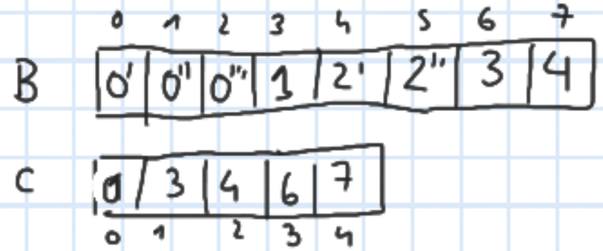
```
def counting_sort( A , k ):
    n = len (A)
    B = [None] * n
    C = [0] * k
    for x in A: C[x] += 1
    for i in range (1, k): C [i] += C[i-1]

    for i in range (n-1, -1, -1):
        B[C[A[i]]-1] = A[i]
        C[A[i]] -= 1
    for i in range (n):
        A[:] = B[:]
```

kwide C[i] zawiera liczbę wartości z A, które są $\leq i$

$k = 5$



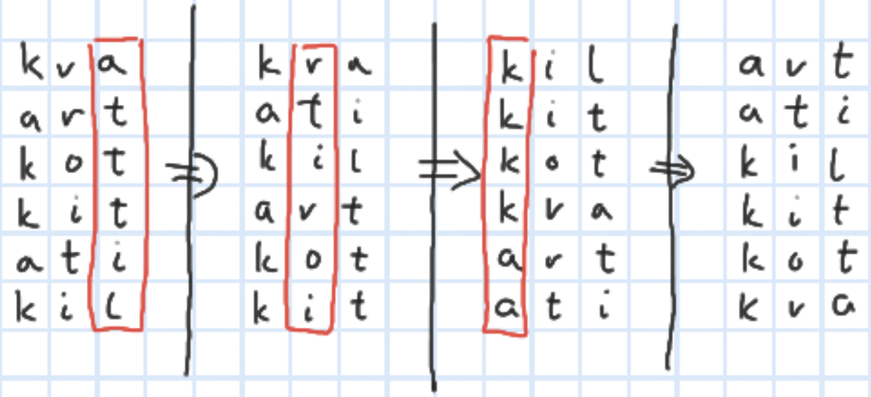




złożoność counting sort

$O(n+k)$

Sortowanie pozycyjne – Radix sort

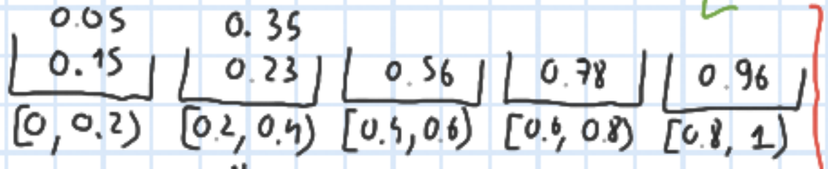


Sortowanie kubełkowe – Bucket Sort

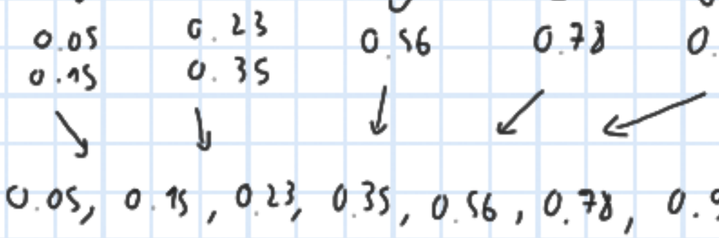
– sortowanie liczb wymiernych, ze zbioru $[0,1)$ wygenerowane z rozkładu jednostajnego

0.15, 0.23, 0.78, 0.56, 0.35, 0.96, 0.05

kubełków powinno być $\Theta(n)$ — rozmiar danych do sortowania



sortowanie kubełków np. przez ustawianie/ jeśli kubełek duży, to QuickSort



0.05, 0.15, 0.23, 0.35, 0.56, 0.78, 0.96

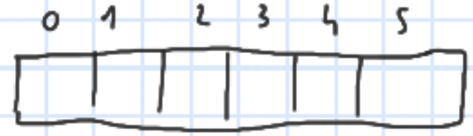
Jeśli kubełków jest n, to $A[i]$ umieszczamy w kubełku $\lfloor n \cdot A[i] \rfloor$

Abstrakcyjne struktury danych

↳ "kontrakt" między strukturą a jej użytkownikiem — zbiór operacji

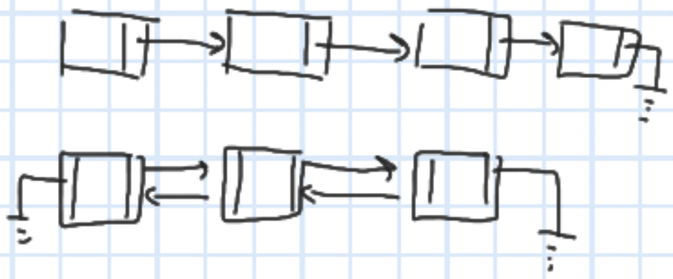
↳ realizacja "fizyczna"

Tablica



"Coś, co pozwala odwoływać się do komórek po ich numerach"

Lista jedno/dwukierunkowa



"Coś, co pozwala poruszać się po kolei po elementach w jedną lub dwie strony i wpinać lub usuwać elementy"

```
class DNode:
    def __init__(self, val):
        self.prev = None
        self.next = None
        self.val  = val
```

węzeł listy dwukierunkowej

Stos

"Coś, co pozwala na odkładanie na wierzchołek i zdejmowanie z niego"

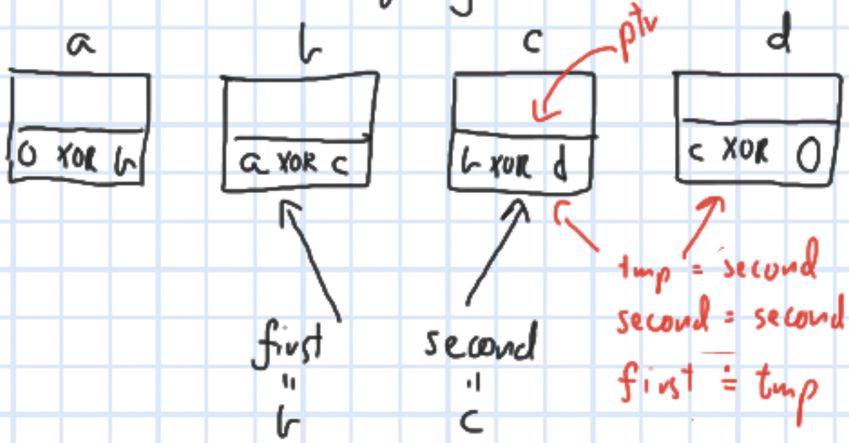
S - stos

S.push(x)

S.pop()

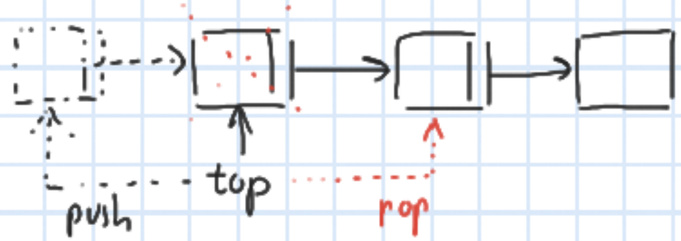
S.isempty()

Lista dwukierunkowa "na jednym wskaźniku"

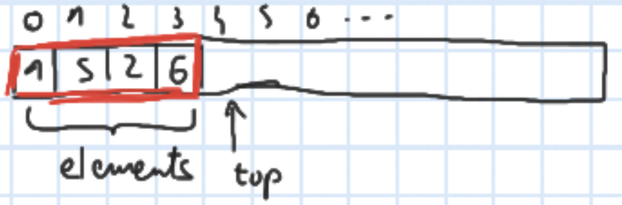



```
  1100110
  0101100
XOR
-------
  1001010

  1001010
XOR 0101100
-------
  1100110
```

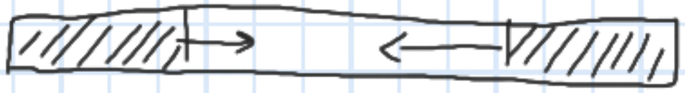
Implementacja listowa



Implementacja tablicowa



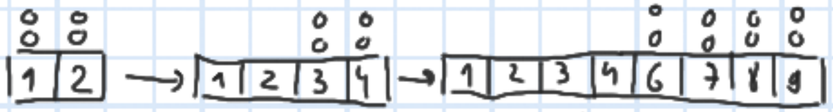
Dwa stosy?

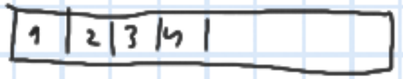


Co zrobić, gdy stos "tablicowy" się przepełni?

↳ zaalokować nową, 2x większą tablicę i tam skopiować stos

- push - 3 zł
- pop - 1 zł





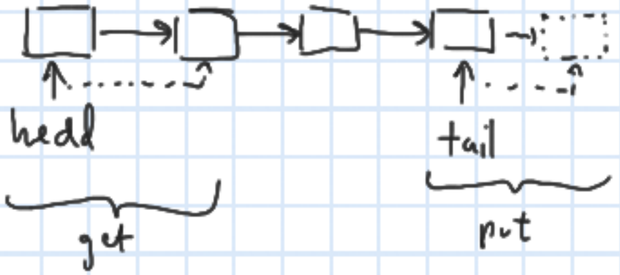
Mogę też zmniejszać tablicę, ale np. dopiero jak wypełniona poniżej 1/4 pojemności

Kolejka

"Coś, co pozwala dokładać elementy na końcu i zabierać je z początku"

- enque (put)
- deque (get)

Implementacja listowa



Implementacja tablicowa (pętlująca kolejka)