

# 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

## Pogląd

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

## Notacja asymptotyczna

Niech  $f, g$  będą funkcjami

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

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

def Mówimy, że  $f$  jest  $O(g(n))$

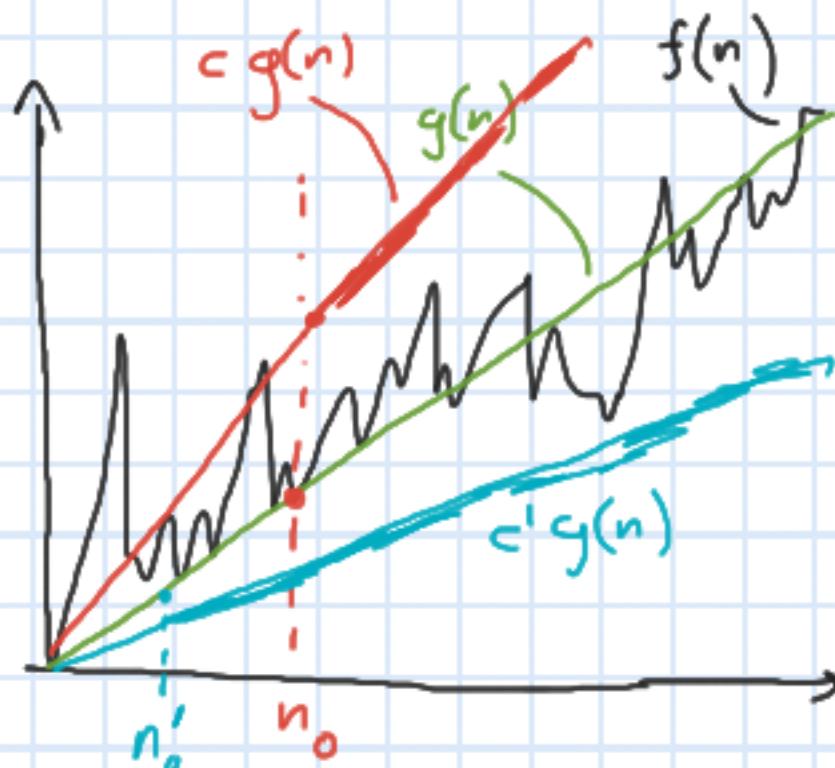
jeśli:

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

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

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

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



## Problem sortowania

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

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

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

## Uwagi

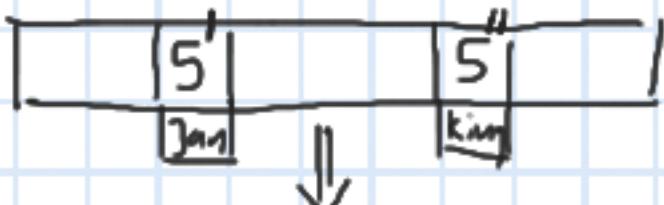
Reprezentacja danych — tablica

— listy      1 - kier.  
                2 - kier.

— plik

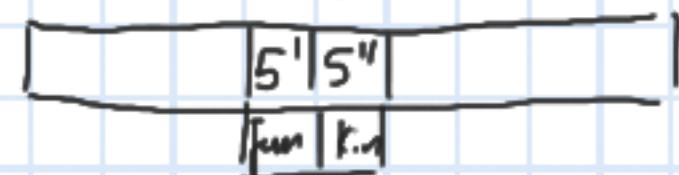
Stabilność sortowania

— wy elementy "inne"  
możą zmienić kolejność?

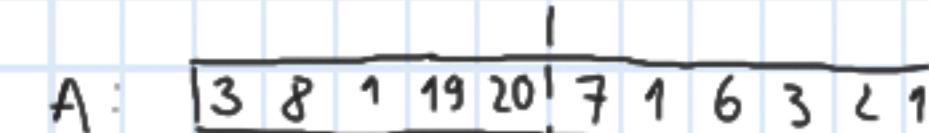


Szybkość sortowania  
↳ prosty  $O(n^2)$

↳ szybkie  $O(n \log n)$

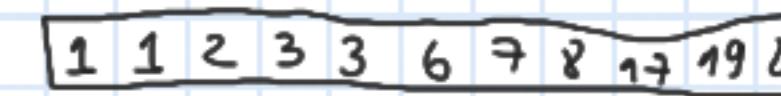
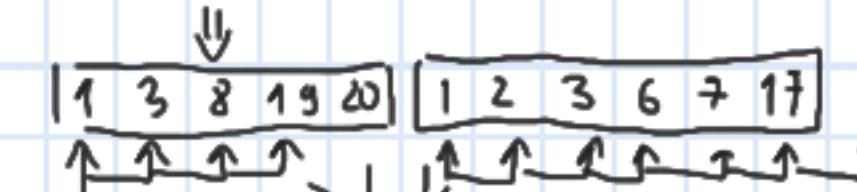


## ① Sortowanie przez scalanie (Merge sort)



rek. sout.  
lewy frag.

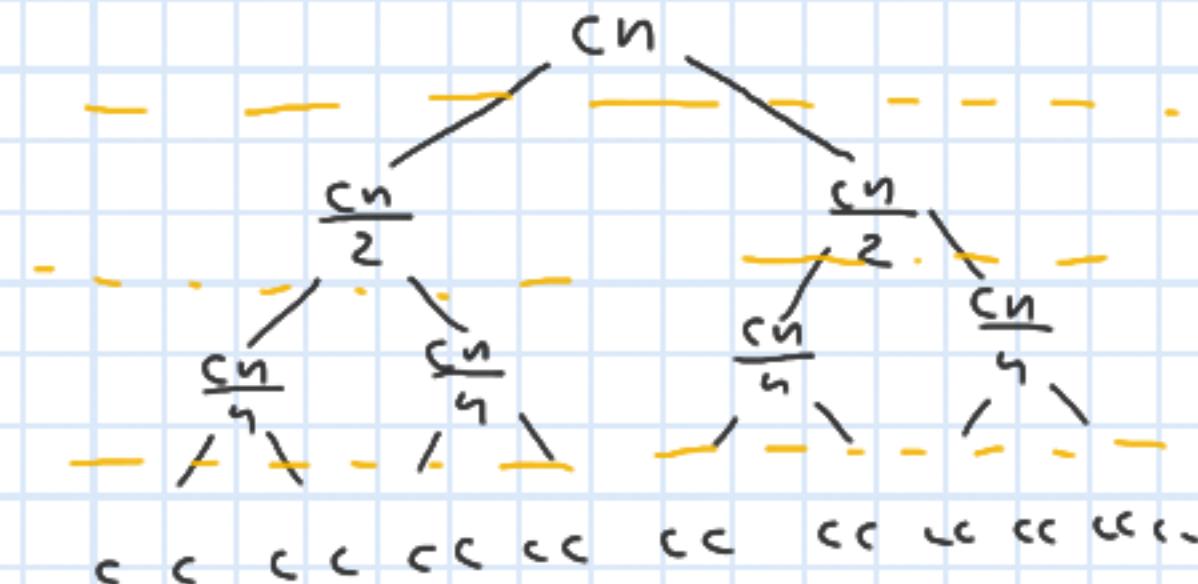
rek. sout.  
prawy frag.



scalanie

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

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

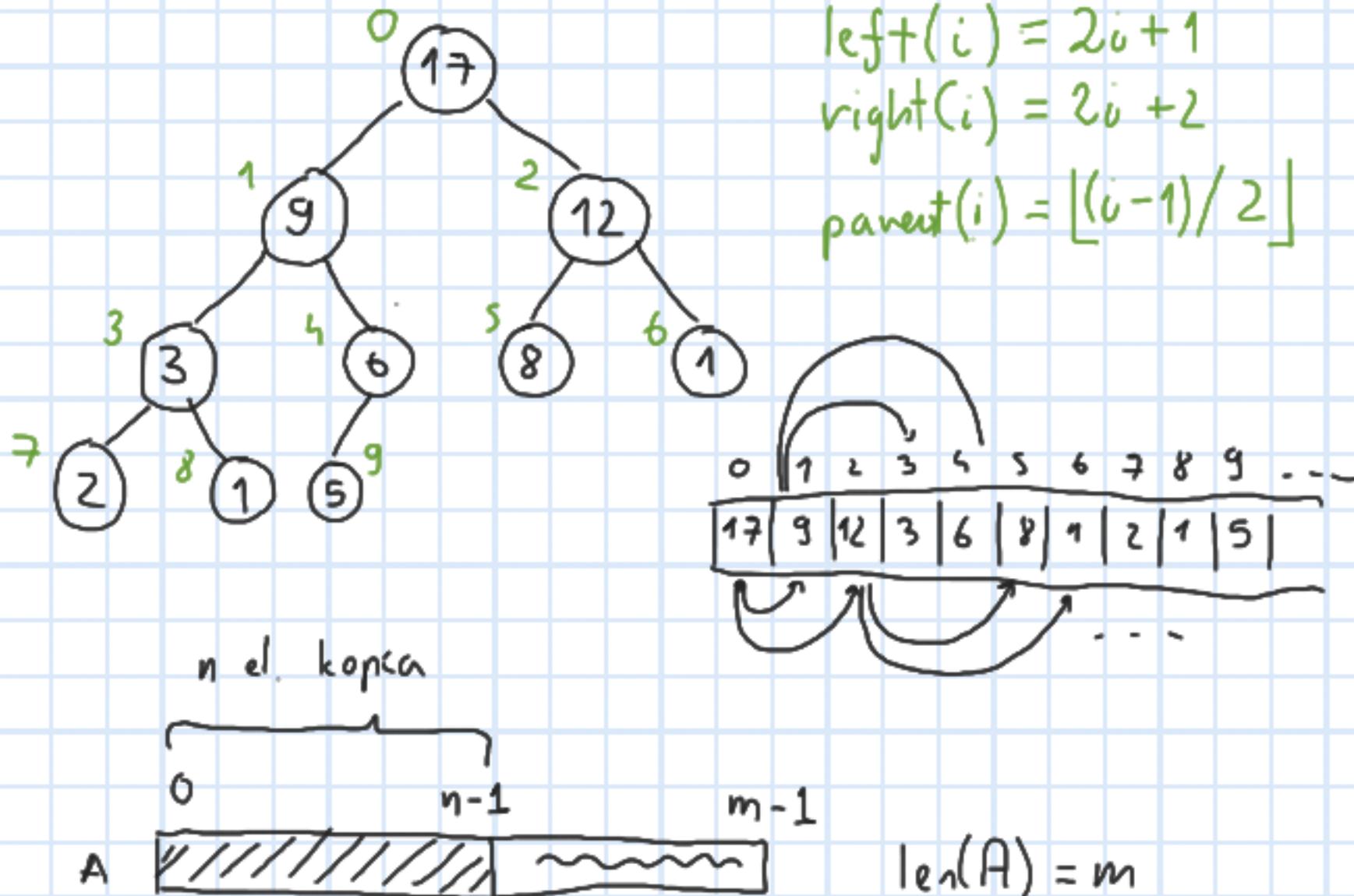


$$\sum cn \log n$$

## ② Soutovanie kopcov (Heapsort)

Kopiec — drevu binarne, u ktorym u každym výzle A prichádzajúceho vektoru  $\geq n$  je vlastnosť  $v[i] \geq v[2i+1], v[2i+2]$

Punktad



```
def heapify(A, n, i)
```

$l = \text{left}(i)$ ,  $\text{max\_ind} = i$

$r = \text{right}(i)$

if  $l < n$  and  $A[l] > A[\text{max\_ind}]$ :  
 $\text{max\_ind} = l$

if  $r < n$  and  $A[r] > A[\text{max\_ind}]$ :  
 $\text{max\_ind} = r$

if  $\text{max\_ind} \neq i$ :

$\text{swap}(A[:], A[\text{max\_ind}])$

$\text{heapify}(A, n, \text{max\_ind})$

$O(n \log n)$

$\Theta(n)$

```
def build_heap(A):
```

$n = \text{len}(A)$

for  $i$  in range( $\text{parent}(n-1), -1, -1$ ):

$\text{heapify}(A, n, i)$

$O(\log n)$

W tablicy mamy n elementów

$\frac{n}{2}$  tych elementów tworzą kopiec o wys. 0

$\frac{n}{4}$  tych el., tworzą kopiec o wys. 1

$\frac{n}{8}$  tych el., tworzą kopiec o wys. 2

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

Koszt build heap ujemiamy jako:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h = O\left(\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n \cdot h}{2^{h+1}}\right) \\ = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

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

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

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

def heapsort(A):

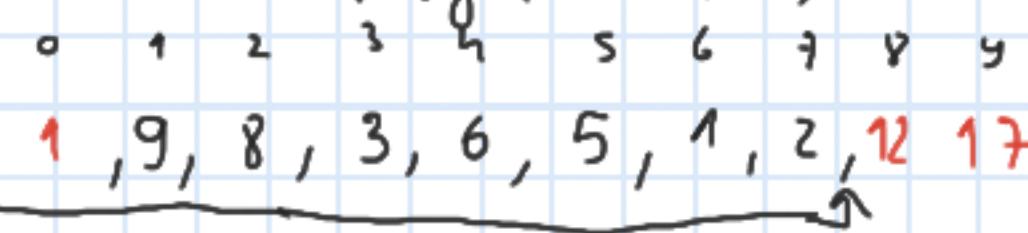
n = len(A)

buildheap(A)

for i in range(n-1, 0, -1):

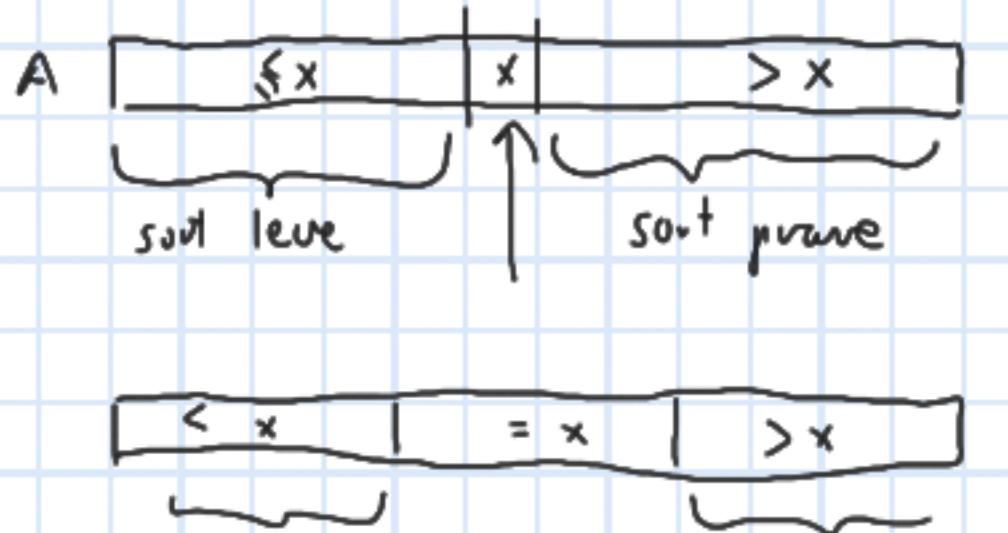
swap(A[i], A[0])

heapify(A, i, 0)  $\longleftarrow O(\log n)$



$\Theta(n \log n)$

### ③ Quick sort

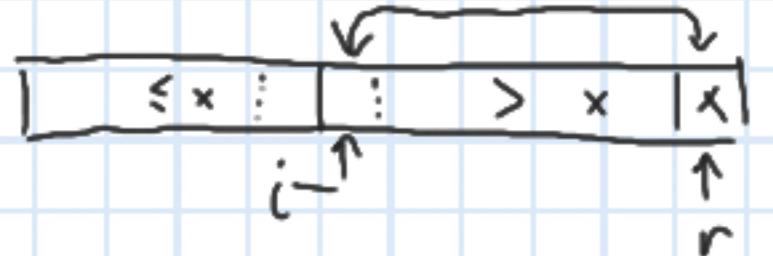
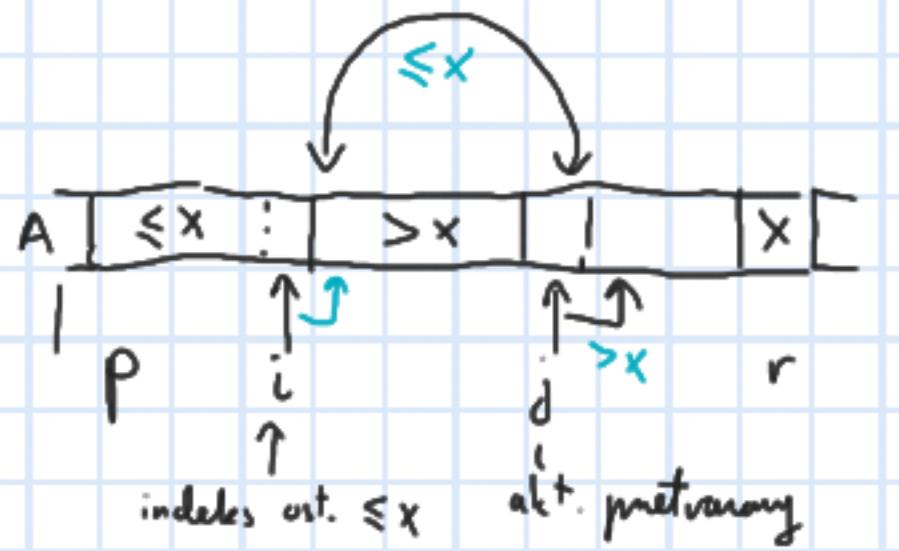
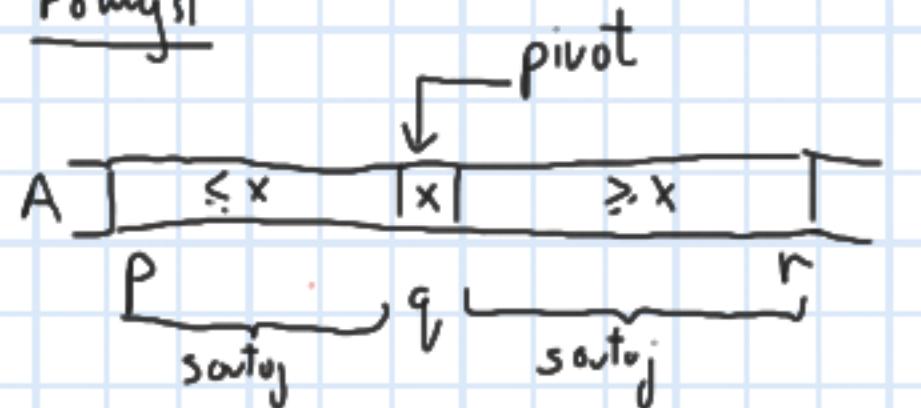


# Algorytmy i Struktury Danych

## Wykład 3

### Algorytm QuickSort

Pomyśl



```
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
```

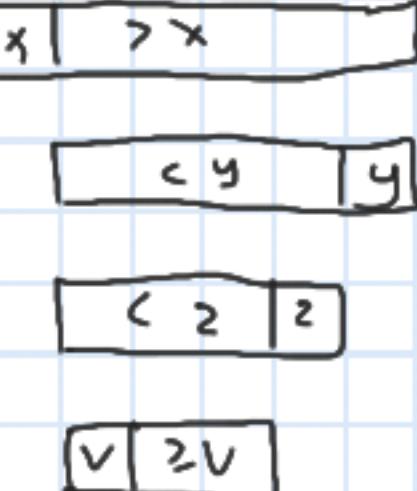
Złożoność obliczeniowa  
Idealne podziały

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

$$\bar{T}(n) = \Theta(n \log n)$$

Pechowe podziały

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



Algorytm Lomuto

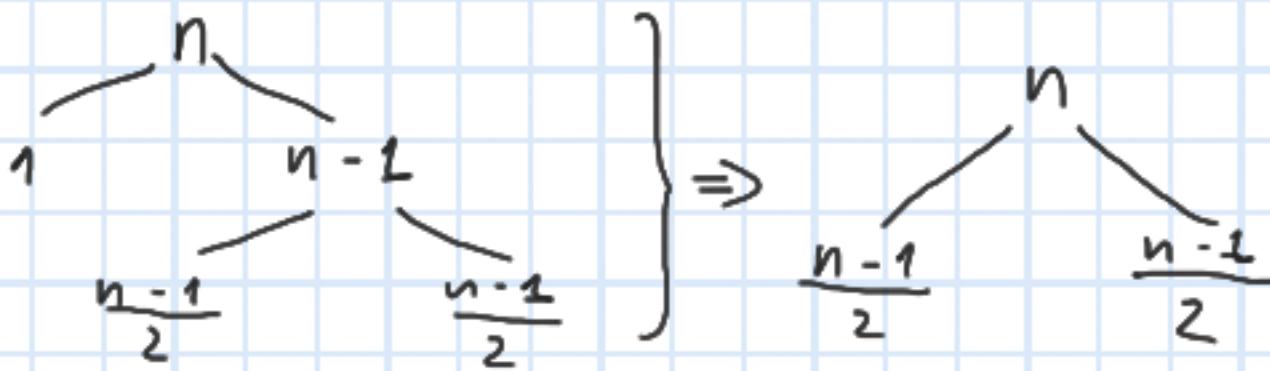
Algorytm Hoare'a



$$\begin{aligned} \bar{T}(n) &= T(n-1) + cn \\ &= T(n-2) + c(n-1) + cn \\ &\vdots \\ &= c + 2c + 3c + \dots + c(n-1) + cn \\ &= c \left( \frac{n(n+1)}{2} \right) = \Theta(n^2) \end{aligned}$$

## Mieszane podziały

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



## Quicksort przez rekurencji ogólny

def quicksort( A, p, r ) :

    while  $p < r$  :

$q = \text{partition}(A, p, r)$

        quicksort( A, p, q-1 )

$p = q + 1$

## Statystyki pozycyjne

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

min/max — algorytm  $\Theta(n)$

medianą — ??

def select( A, p, r, k ):

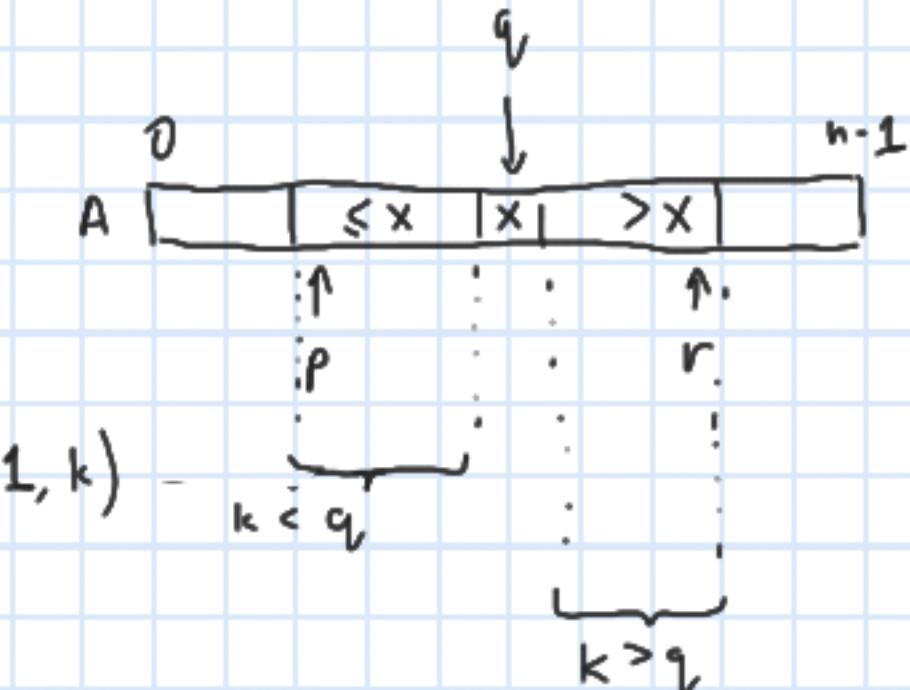
    if  $p == r$ : return  $A[p]$

$q = \text{partition}(A, p, r)$

    if  $q == k$ : return  $A[q]$

    elif  $k < q$ : return select( A, p, q-1, k )

    else: select( A, q+1, r, k )  
        return



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

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

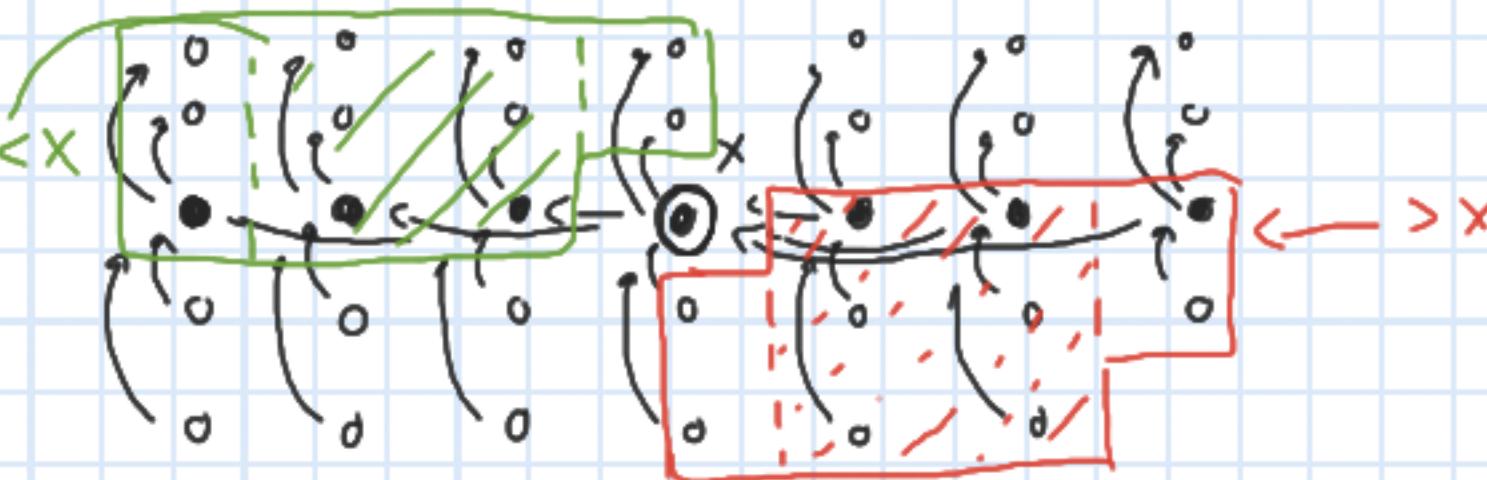
# Magiczne piętki – statystyczne przybliżenie u ciasie liniowym

## Algorytm

- ① podziel tablicę na  $\lceil \frac{n}{5} \rceil$  grup po 5 elementów, u których wyznacz medianę

- ② rekurencyjnie wyznacz  $x$  jako medianę median

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



ile jest elementów większych od  $x$ ?

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

## Złożoność czasowa

$$T(n) = \begin{cases} \Theta(1) & , n \leq \text{perna stała} \\ T\left(\lceil \frac{n}{5} \rceil\right) + \underbrace{T\left(\frac{7n}{10} + 6\right)}_{\substack{\text{med.} \\ \text{median}}} + \Theta(n) & , n \geq ! \end{cases}$$

Twierdzenie, że  $T(n) \leq cn$  dla pewnej stałej  $c$   
Dowód indukcyjny:

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

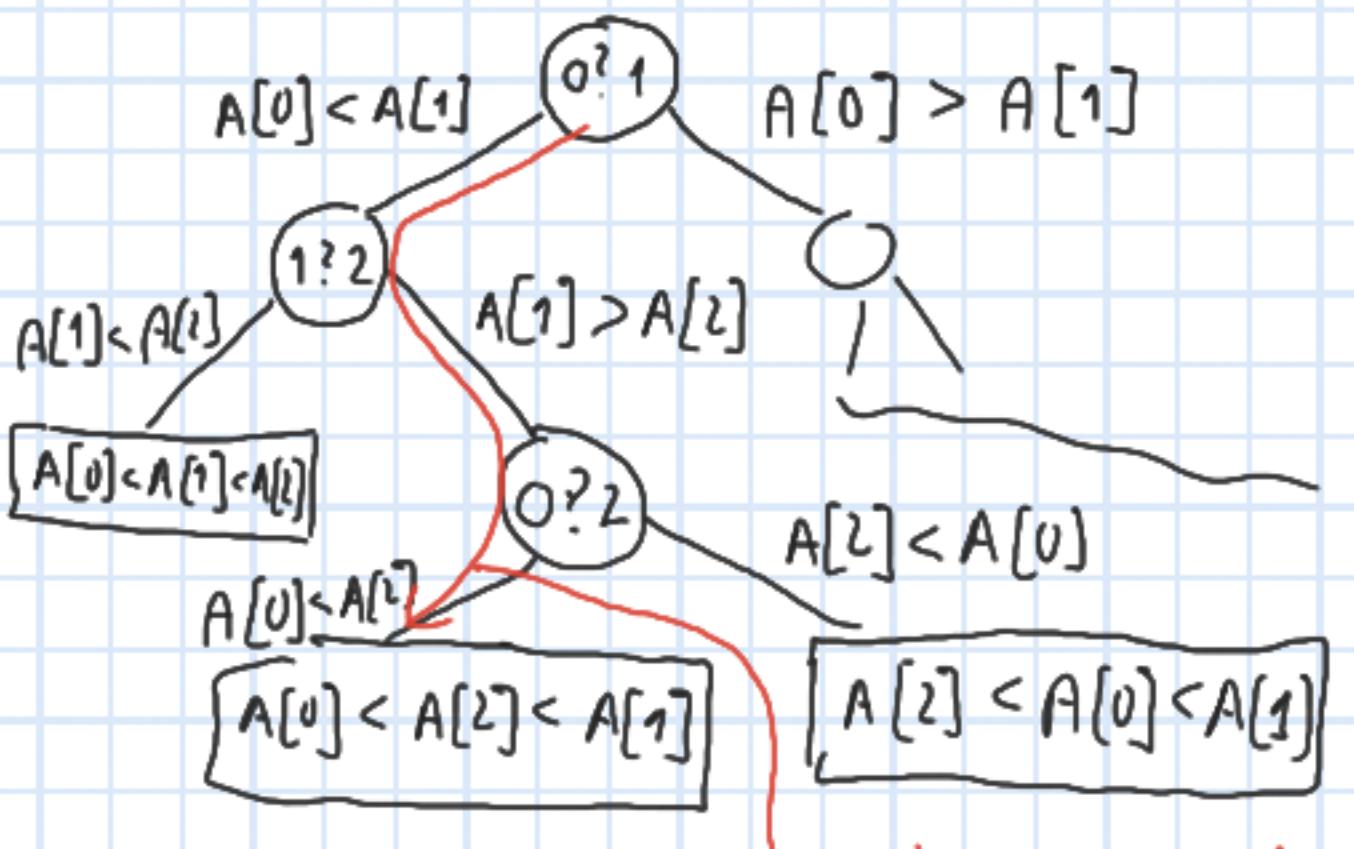
$$\leq \frac{2cn}{10} + \frac{7cn}{10} + 6c + an + c$$

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

jakaś stała

wyliczam  $c$  tak  
że, że ta wartość  
jest ujemna dla  
odpowiednich  $n$

## Dolne ograniczenie na skuteczność sortowania



najdłuższa ścieżka w takiem dwusem  
to przynajmniej liczb porównań do wykonania

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



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

jeśli A ma n elementów to

dwo ro musi mieć  $\geq n!$  lisu

Dwo o wysokości h ma najwyżej  $2^h$  lisu  
liniarne

$$n! \leq 2^h$$

$$h \geq \log n!$$

# Algorytmy i Struktury Danych

## Układ

Sortowanie użycie liniowym — sortowanie  
przez zliczanie

```
def counting_sout( 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[ i ] = B[ i ]
```

tabela zlicza liczby naturalne  
ze zbiorem  $\{0, \dots, k-1\}$

wartość  $C[i]$  zlicza ilość  
elementów z  $A$ , które są  $\leq i$

$k = 5$

0	1	2	3	4	5	6	7
1	2	0	2	3	0	4	0



0	1	2	3	4
3	1	2	1	1
3	4	6	7	8

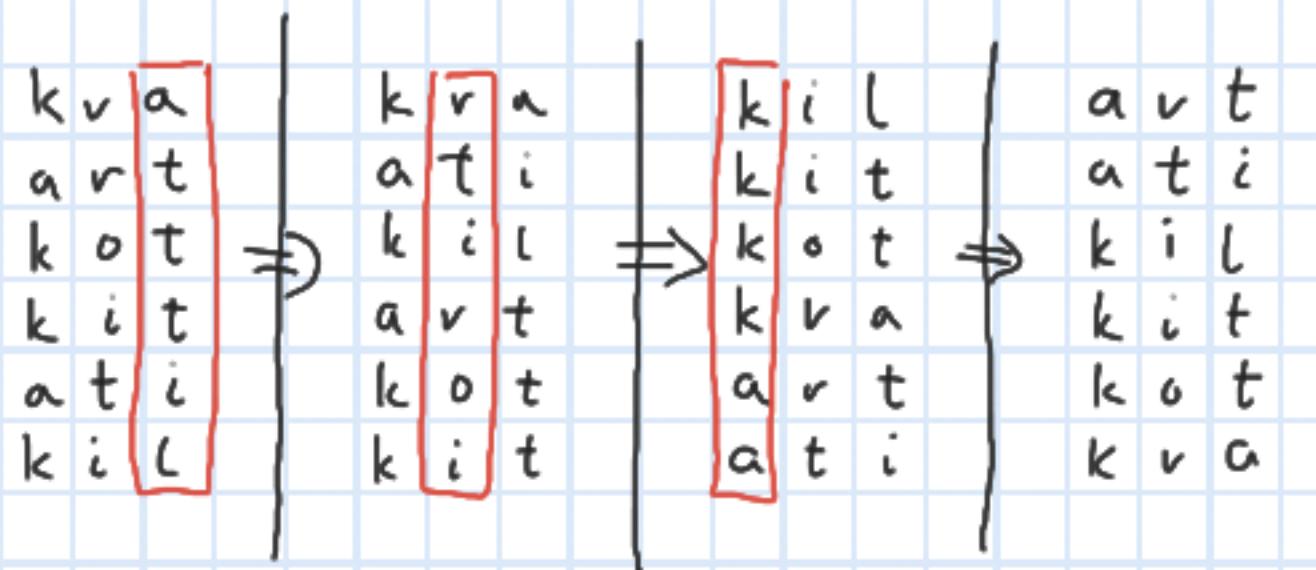
0	1	2	3	4	5	6	7
0	0	0	1	2	2	3	4

0	1	2	3	4
0	3	4	6	7

złożoność  
counting sout

$O(n+k)$

## Sztorcowanie przyjazne - Radix sort

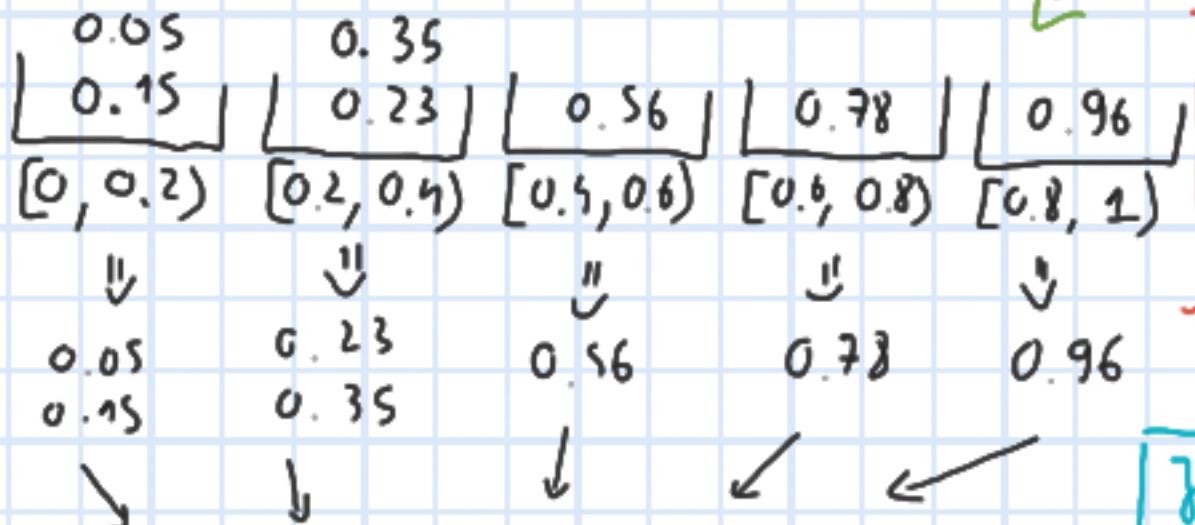


## Sztorcowanie kuletkowe - Bucket Sort

- sztorcowanie listy ujemnych, ze zbioru  $[0, 1]$

wygenerowane z rozkładu jednostajnego

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



kuletkowe porządkowanie

$\Theta(n)$

wymian danych  
do sztorcowania

sztormanie kuletek

np. przez ustalanie/

jeli kuletki dory, to Quicksort

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

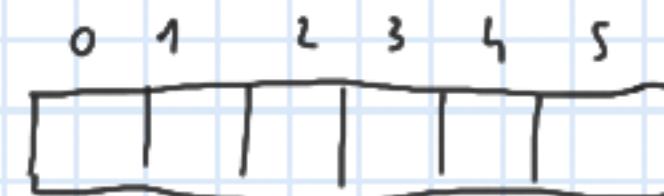
jeli kuletkow jut n, to  $A[i]$

umieszczenie w kuletkach  $[n \cdot A[i]]$

## Abstrakcyjne struktury danych

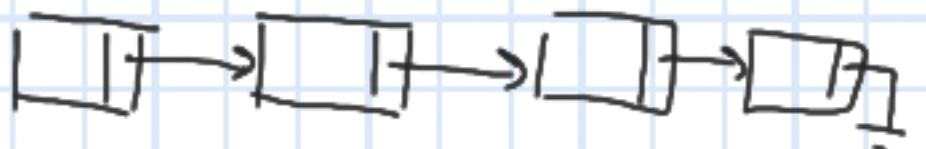
- "kontrakt" między strukturą a jej użytkownikiem — zbiór operacji
- realizacja "fizyczna"

### Tablica



"Coś, co powala odwzorować się do komórek po idzie numerach"

## Lista jedno/dwukierunkowa



class DNode:

def \_\_init\_\_(self, val):

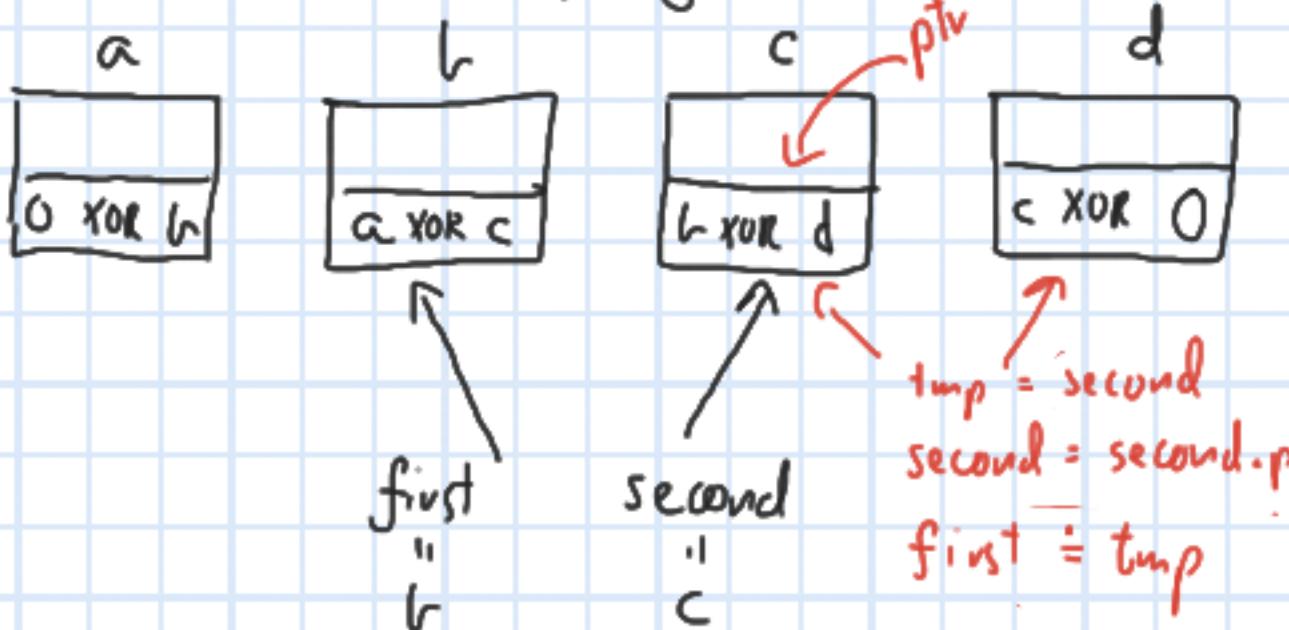
self.prev = None

self.next = None

self.val = val

ugret listy dwukierunkowej

## Lista dwukierunkowa "na jednym ukladaniu"



$$\begin{array}{r}
 1100110 \\
 0101100 \\
 \hline
 1001010
 \end{array}$$

XOR

$$\begin{array}{r}
 1001010 \\
 0101100 \\
 \hline
 1100110
 \end{array}$$

## Stos

"Cos, co powraca na odhodowane na wiecholce  
i zdejmowane z niego"

S - stos

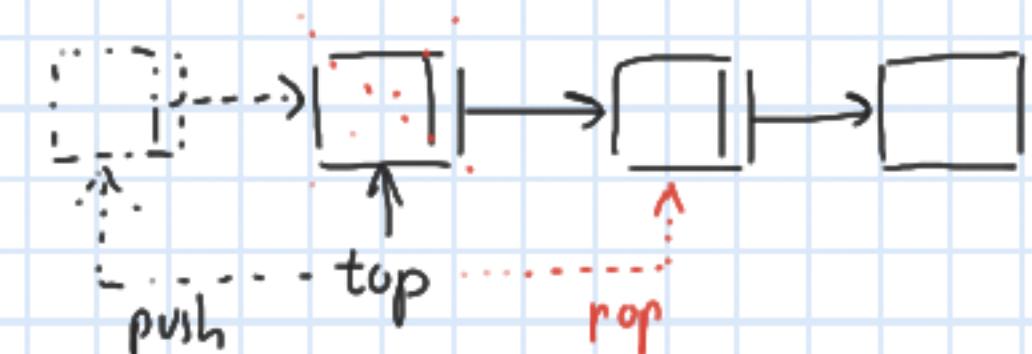
S.push(x)

S.pop()

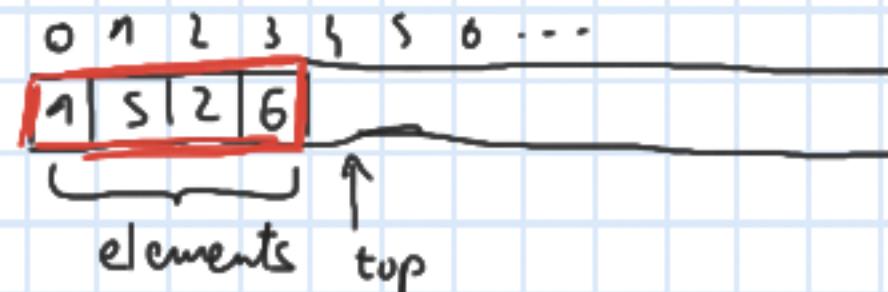
S.isEmpty()



## Implementacja listowa



## Implementacja tablicowa



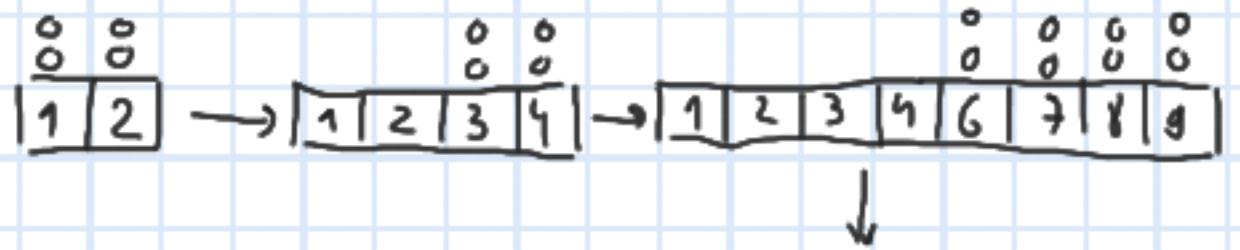
Dva stosy?



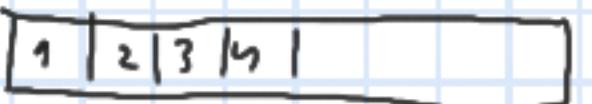
Có zrobic, gdy stos "tablicowy" się przepięci?

↳ zaalokowac nową, 2x większą tablicę i  
tum skopiować stos

- push - 3 zł
- pop - 1 zł



Mogę też zrealizować



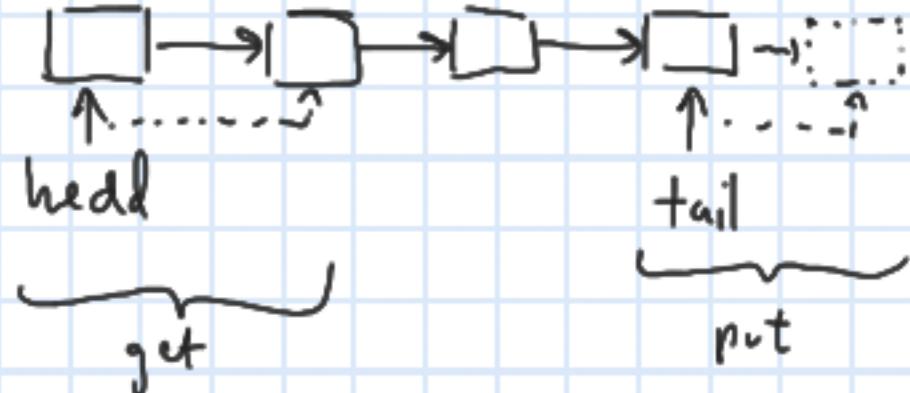
tablicę, ale np. dopiero jak wypróbowam powinny być  
pojemniejsze

## Kolejka

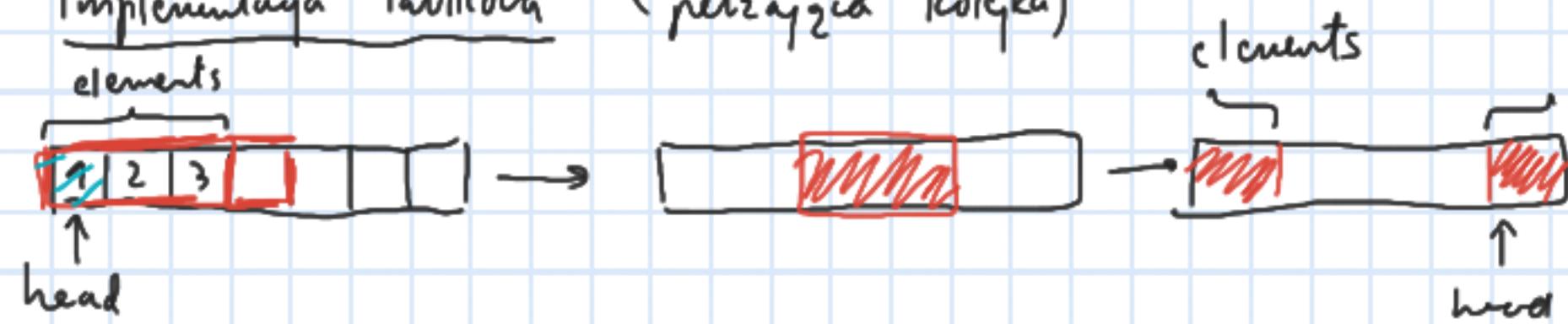
"Coś, co pozwala doliczać elementy na koniec i  
zaliera je z parą tuzów"

- enqueue (put)
- dequeue (get)

## Implementacja listowa



## Implementacja tablicowa (petrzajęca kolejka)



# Algorytmy i struktury danych

## Układ 5

### Algorytmy grafowe

Graf skierowany:

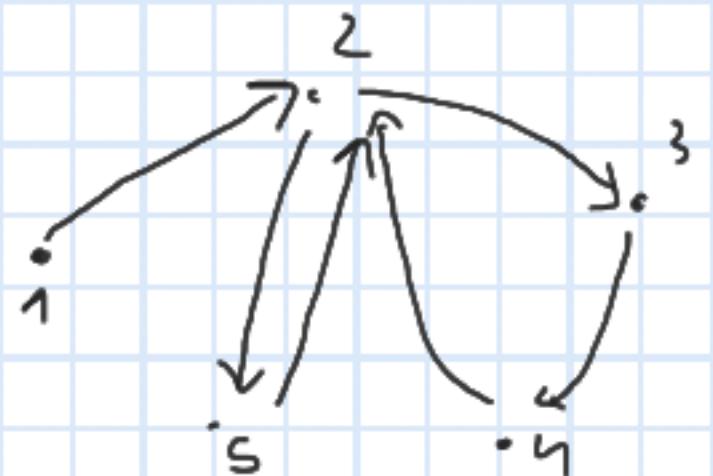
$$G = (V, E), \text{ gdzie}$$

$V = \{v_1, \dots, v_n\}$  – zbiór wierzchołków

$E = \{e_1, \dots, e_m\}$  – zbiór krawędzi

$$E \subseteq V \times V$$

na ogół nie  
dopuszczamy  
petli  $(v, v)$

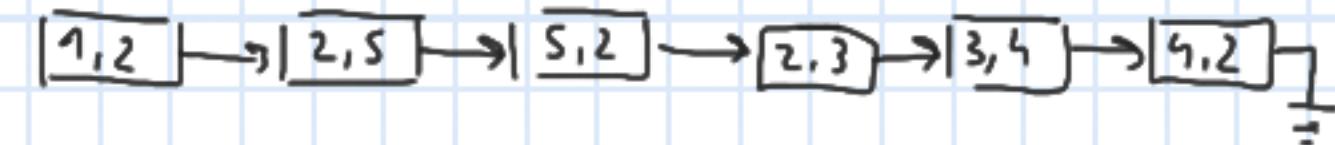


2 krawędzie / wierzchołki  
można związać dodatkowe  
informacje (np. waga lub  
długość krawędzi)

Grały nieskierowane – graf w którym krawędzie to zbiory  
elementów (często reprezentowane  
jako grały skierowane gdzie wszystkie krawędzie  
są w obie strony)

### Reprezentacja grafów

#### ① lista / tablica krawędzi



#### ② Reprezentacja macierza

	1	2	3	4	5
1	-	1	0	0	0
2	0	-	1	0	1
3	0	0	-	1	0
4	0	1	0	-	0
5	0	1	0	0	-

Zalety

- prostota

- dostęp  $O(1)$  do dowolnej krawędzi

Uady

- rozłożenie pamięci  $\Theta(V^2)$

- dostęp w czasie  $O(V)$  do krawędzi

z danego wierzchołka

#### ③ Listy sąsiedztwa

1	→ 2
2	→ 5 → 3
3	→ 4
4	→ 2
5	→ 2

Zalety

- rozłożenie pamięci  $O(V+E)$

- dostęp do krawędzi wychodzącej z  
wierzchołka w czasie  $O(d(v))$

Uady

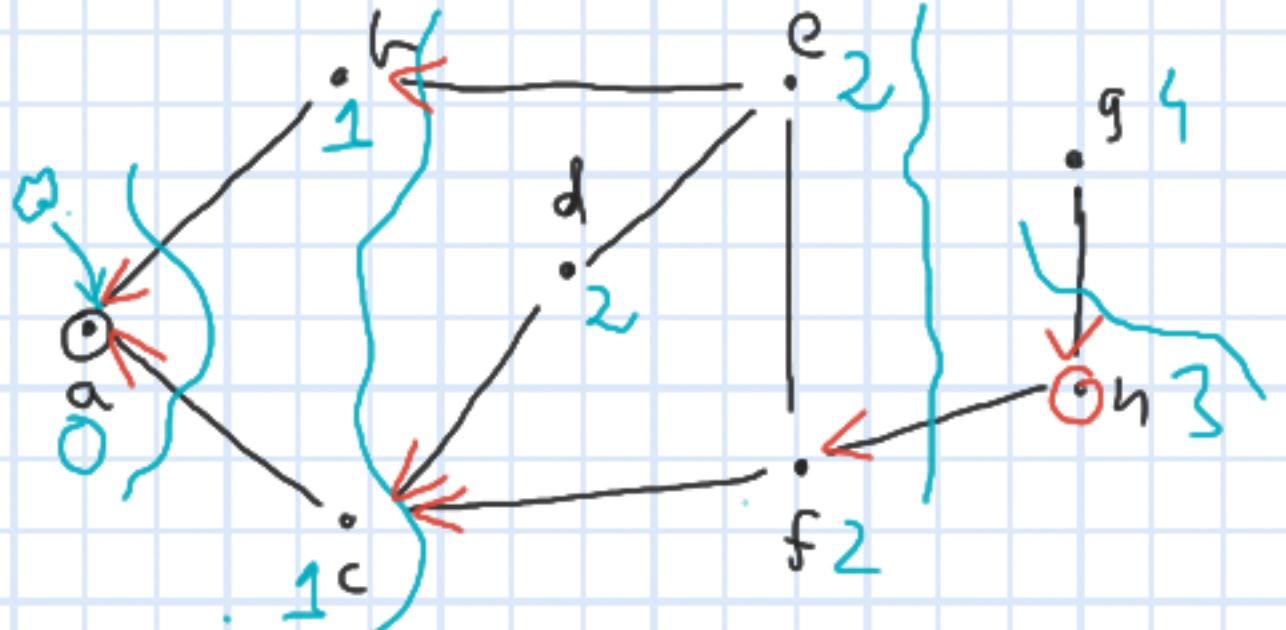
- brak dostępu w czasie

$O(1)$  do konkretnej krawędzi

stopień wierzchołka

# Algorytm przeszukiwania grafu w szerszy

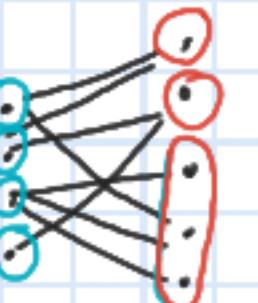
(Breadth-First Search; BFS)



$Q: \underline{a} | \underline{b} \underset{0}{\text{---}} c \underset{1}{\text{---}} \underline{e} \underset{2}{\text{---}} d \underset{2}{\text{---}} f \underset{2}{\text{---}} h \underset{3}{\text{---}} g \underset{4}{\text{---}}$

## Zastosowania BFS

- najkrótsze ścieżki z danego źródła
- spójność
- wykrywanie cykli
- testowanie dwuspojnosci



def BFS( G, s )

#  $G = (V, E)$

$Q = \text{Queue}()$

for  $v \in V$ :  $v.\text{visited} = \text{False}$

s.d = 0

s.visited = True

s.parent = None

Q.put(s)

while not Q.is\_empty():

$u = Q.get()$

for  $v$  jest sąsiadem  $u$ :

if not  $v.\text{visited}$ :

$v.\text{visited} = \text{True}$

$v.d = u.d + 1$

$v.parent = u$

Q.put(v)

typowa implementacja

$n = \text{len}(V)$

$d = [-1 \text{ for } v \text{ in } \text{range}(n)]$

$\text{visited} = [\text{False} \text{ for } v \text{ in } \text{range}(n)]$

$\text{parent} = [\text{None} \text{ for } v \text{ in } \text{range}(n)]$

$\downarrow$   
-1

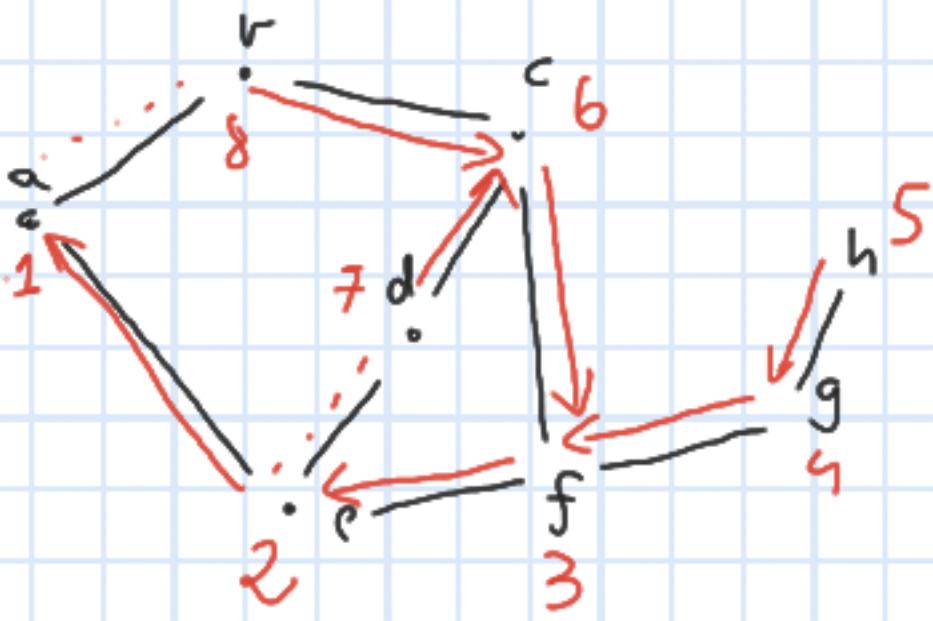
## Złożoność BFS

- lista:  $\Theta(V+E)$

- macierz:  $\Theta(V^2)$

← return  $d, \text{parent}, \text{visited}$

# Algorytm przeszukiwania w głębi (Depth-First Search; DFS)



```

def DFS(G):
    # G = (V,E)
    for v ∈ V:
        v.visited = False
        v.parent = None
    time = 0
    for u ∈ V:
        if not u.visited:
            DFSVisit(G, u)
    
```

```

def DFSVisit(G, u):
    nonlocal time
    
```

```

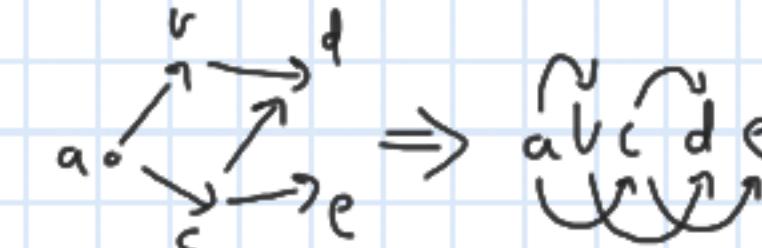
time += 1 ← czas odwiedzenia
u.visited = True
for v - sąsiad u:
    if not v.visited:
        v.parent = u
        DFSVisit(G, v)
    time += 1 ← czas poświetlenia
    
```

## Złożoność

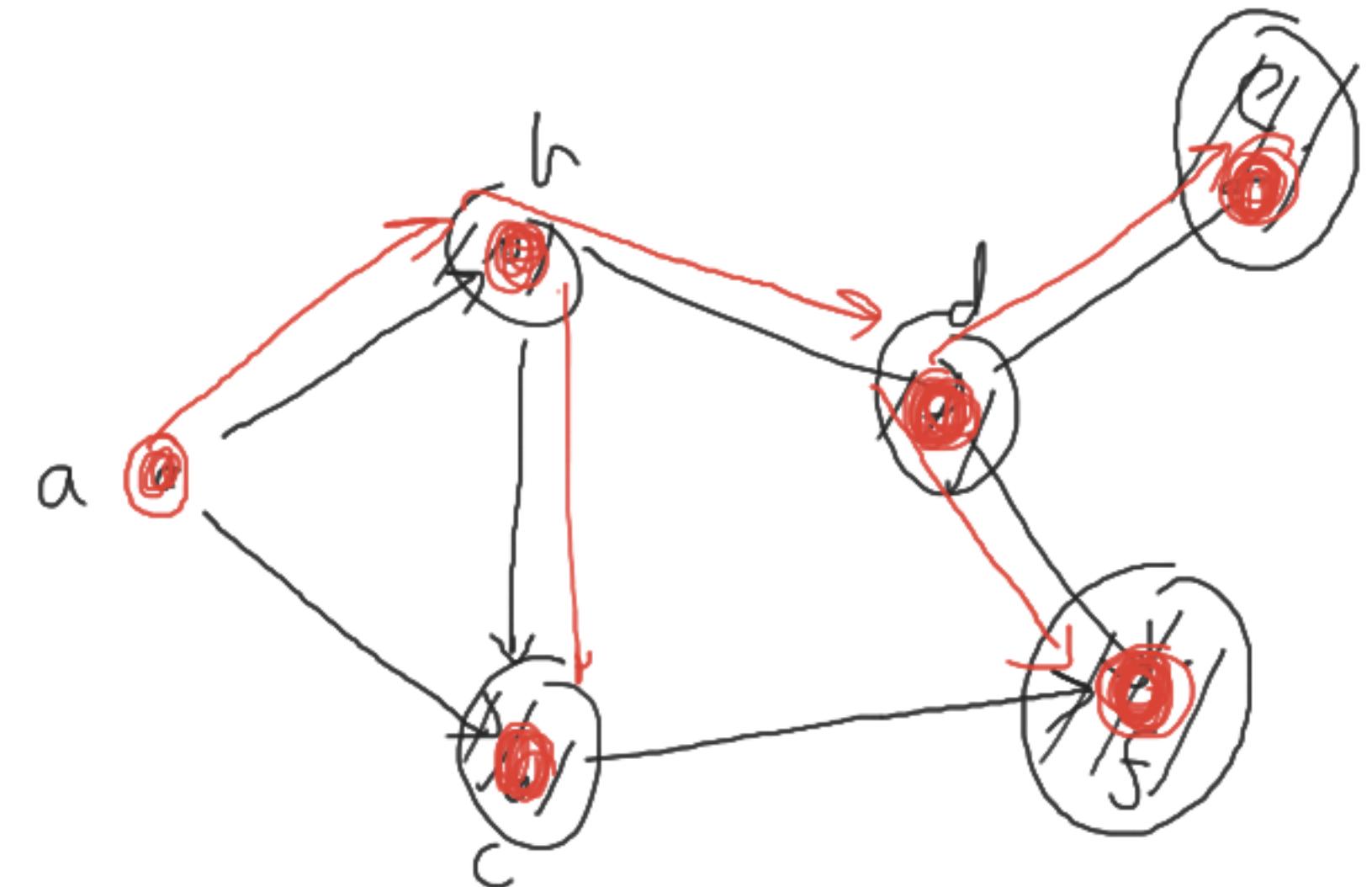
rep. listowa  $\Theta(V+E)$   
rep. maciowa  $\Theta(V^2)$

## Zastosowania

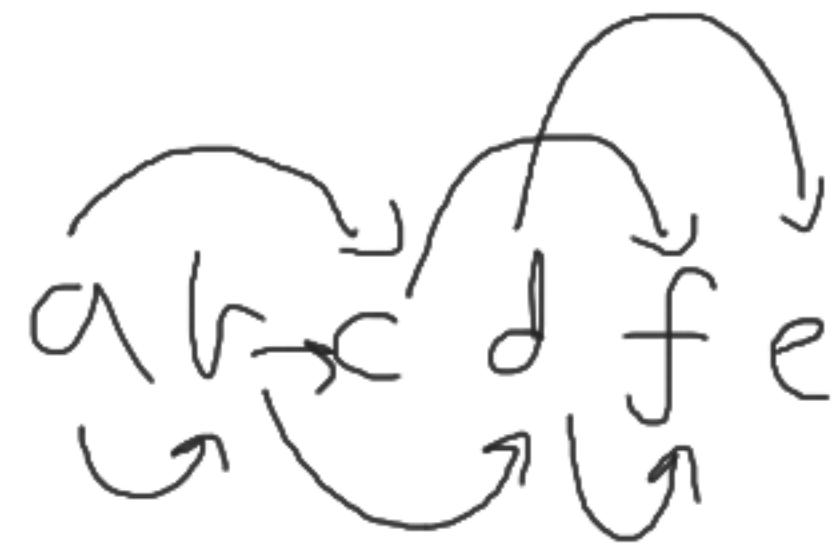
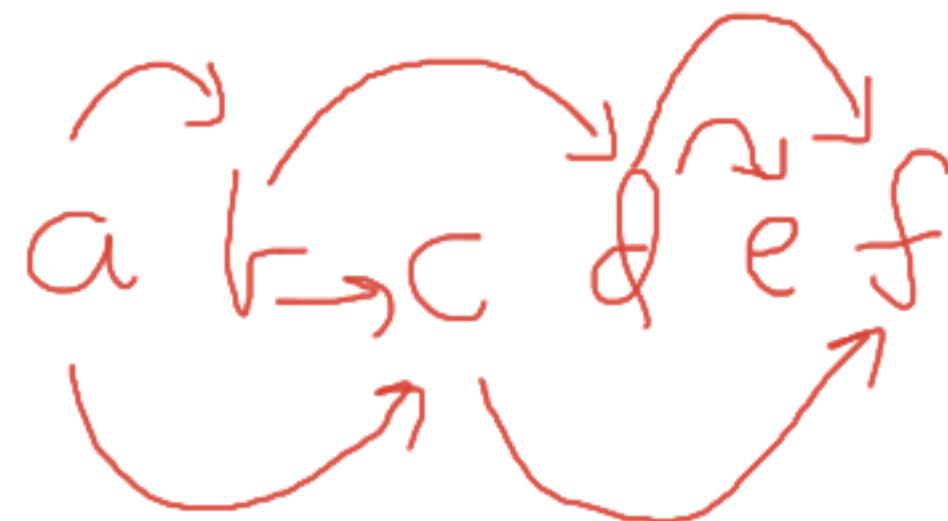
- spójność
- dwudzielność
- wykrywanie cykli
- sortowanie topologiczne
- silnie spójne składowe
- cykl Eulera
- mosty / punkty antykulagi



DAG



$O(V^2)$



# Algorytm : Struktury Danych

## Układ 6

### Zastosowania algorytmu DFS

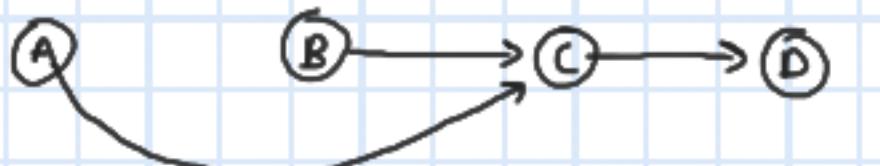
#### ① Sortowanie topologiczne DAGu

DAG - directed acyclic graph

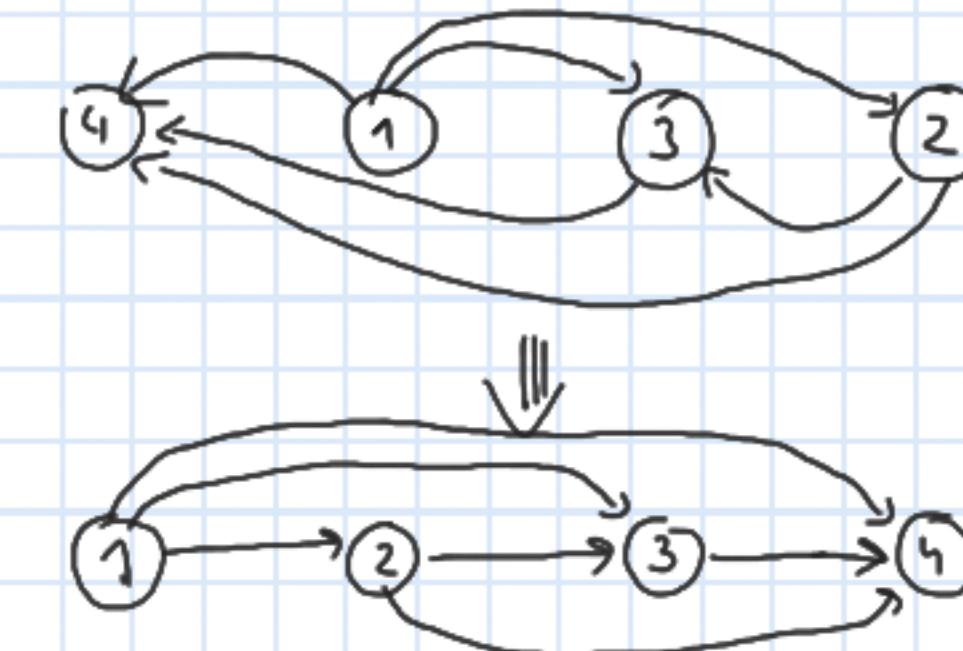
Sortowanie topologiczne - ułożenie wierszów krawędzi w grafie w taki kolejności, że krawędzie wskazują tylko "z lewej na prawą"

#### Poglądy zastosowania

- wyznaczenie kolejności realizacji zadań



- liniaryzacja częściowego porządku

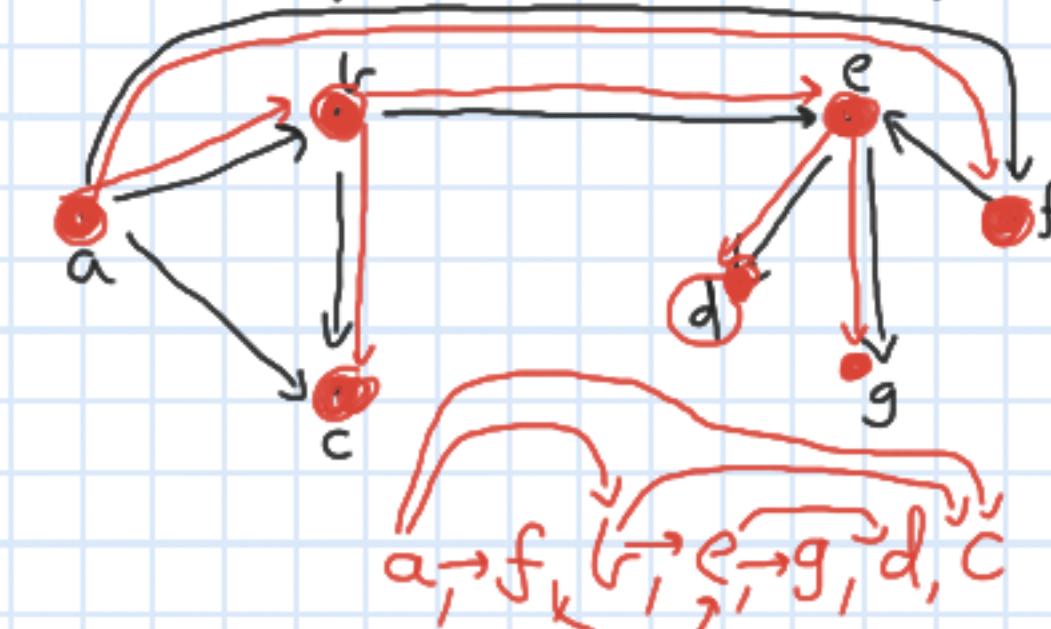


#### Algorytm

- uruchamiamy DFS

- po "przebrojeniu" danego wierszka dopisujemy go na początek转弯的 listy

- lista daje posortowanie kolejności



## ② Cykl Eulera

Cykl Eulera to cykl przechodzący przez każdy krawędź w grafie

tw Graf nieskierowany, spójny posiada cykl Eulera utw. gdy krawędzie wieniaków ma stopień parzysty  
dowód

Wierzchołek końcowy - oznaczenie

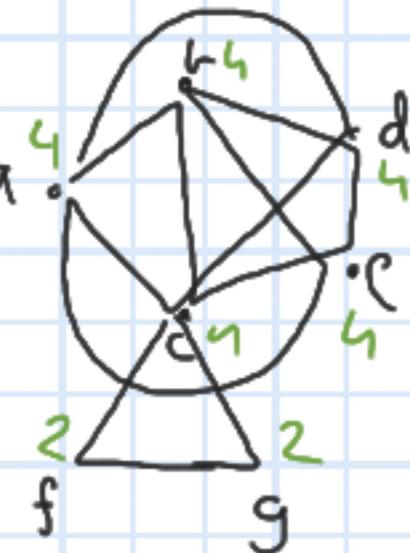
Wierzchołek wystarczający:

- zaczynając od dowolnego wybranej wierzchołka  $v$ , wdrapujemy dowolne cykle w krawędziach (bez powtarzania)

- w pewnym momencie musimy wrócić do  $v$

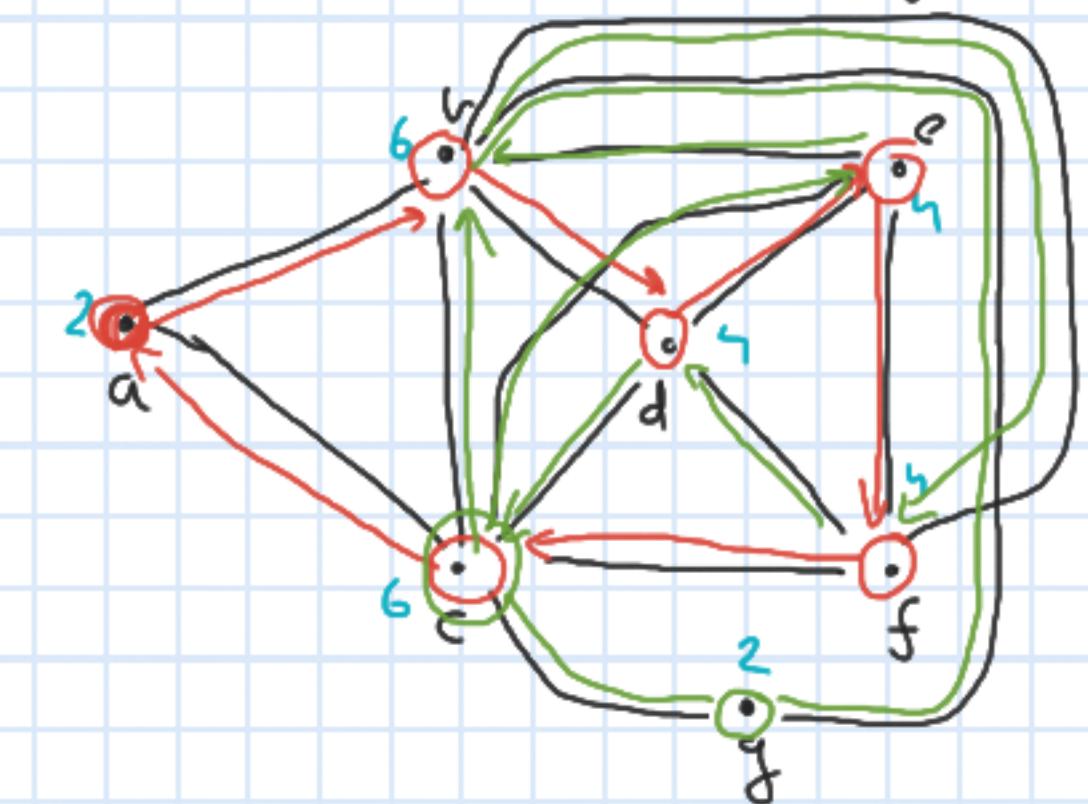
- [ jeśli odwiedziliśmy wszystkie krawędzie, to mamy cykl Eulera ]

- [ jeśli, to możemy "dokleić" fragment cyklu startując od nowego wierzchołka, z którego wychodzą nieodwiedzone krawędzie ]

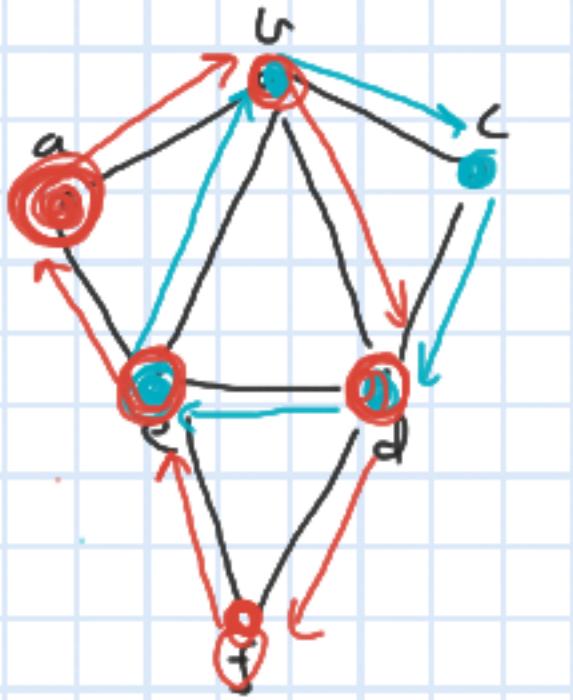


## Algorytm

- wykonujemy DFS, "usuwając" na bieżąco odwiedzone krawędzie i nie zahamując wielokrotnego wejścia do tego samego wierzchołka
- po zwiszeniu wierzchołka dodajemy go na koniec tworzącego cyklu



a b d e f [ c b f d c e b g c a ]



alr<sub>d</sub>f  
e b c d e a

Cykl Hamiltona — cykl przebiegający dokładnie raz przez każdy wierzchołek grafu



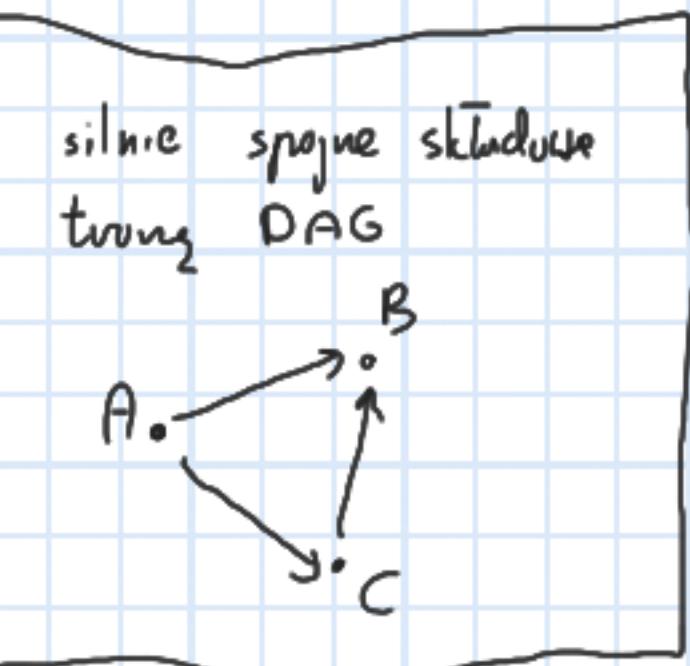
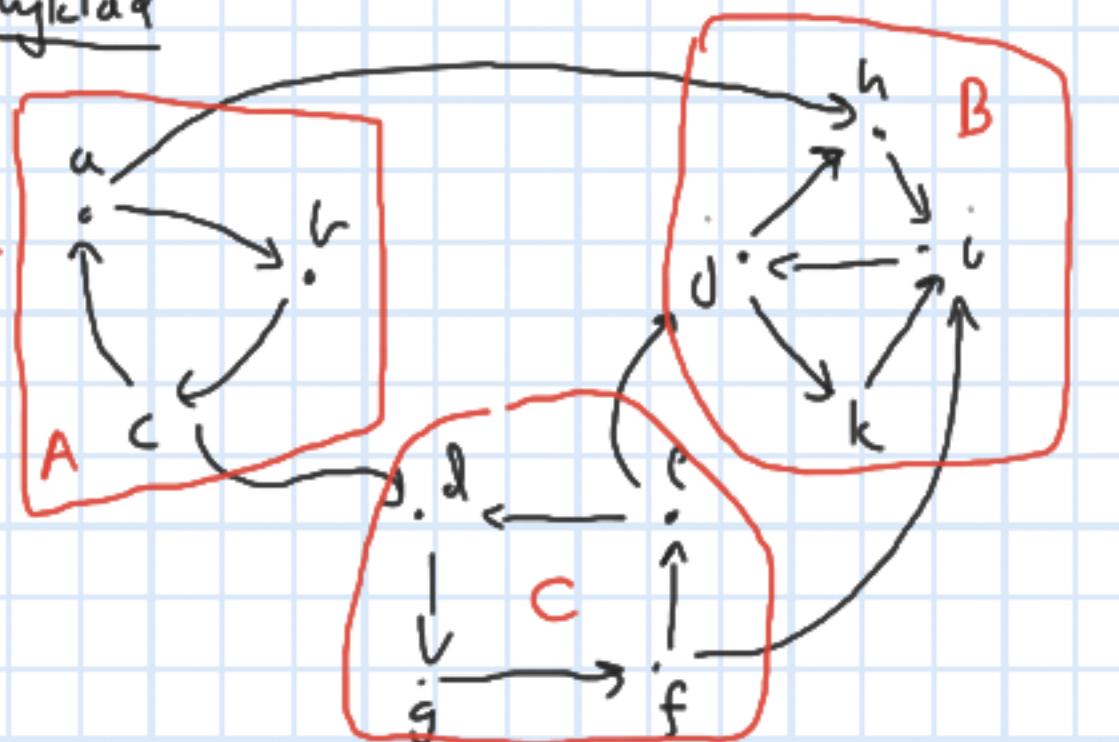
stwierdzenie, iż cykl Hamiltona istnieje, jest problemem NP-zupełnym

③ Silne spójne składowe w grafie skierowanym

dcf Niech  $G = (V, E)$  będzie grafem skierowanym

Ukierunkotki  $u$  i  $v$  należą do tej samej silnie spójnej składowej jeśli są wzajemnie osiągalne środkami zgodnie z kierunkiem krążku

PonkTajd



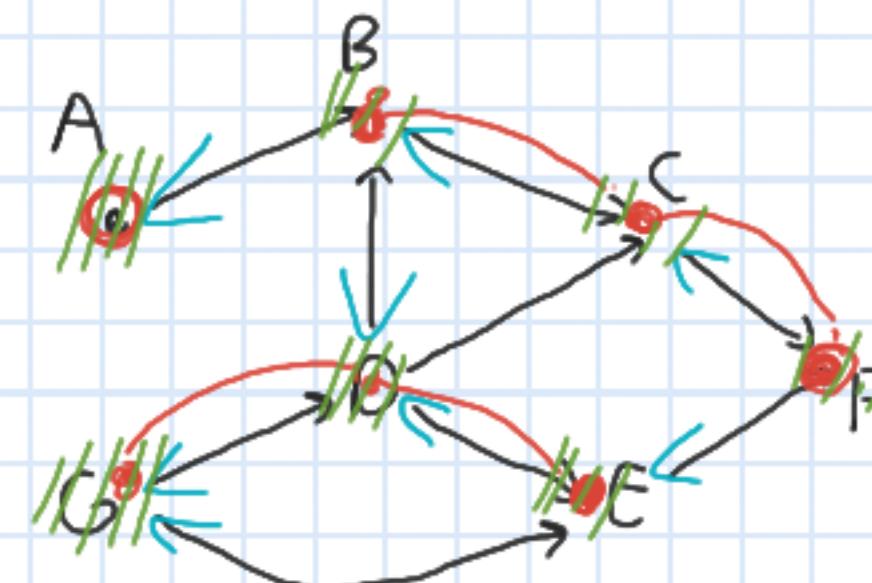
Algorytm

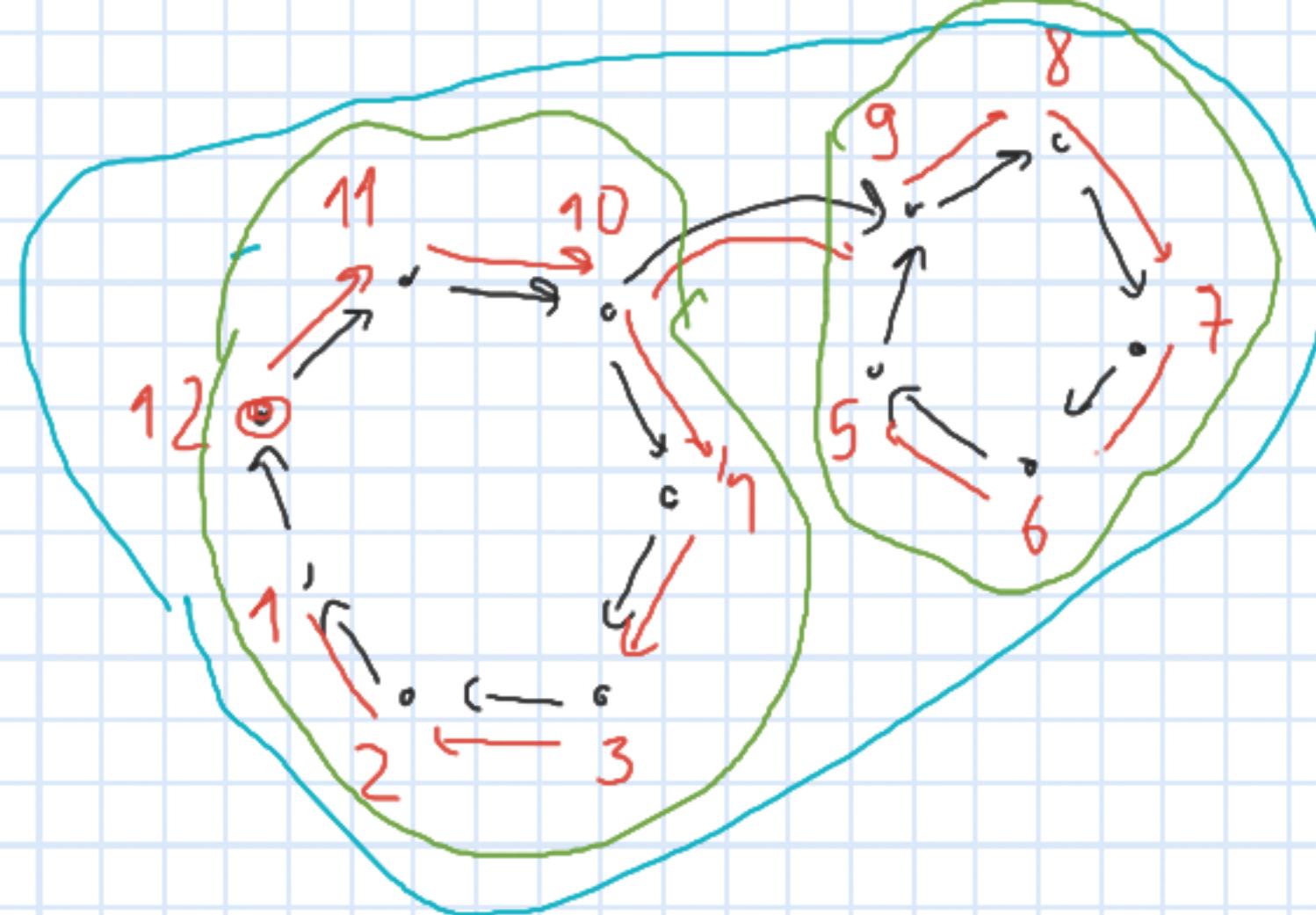
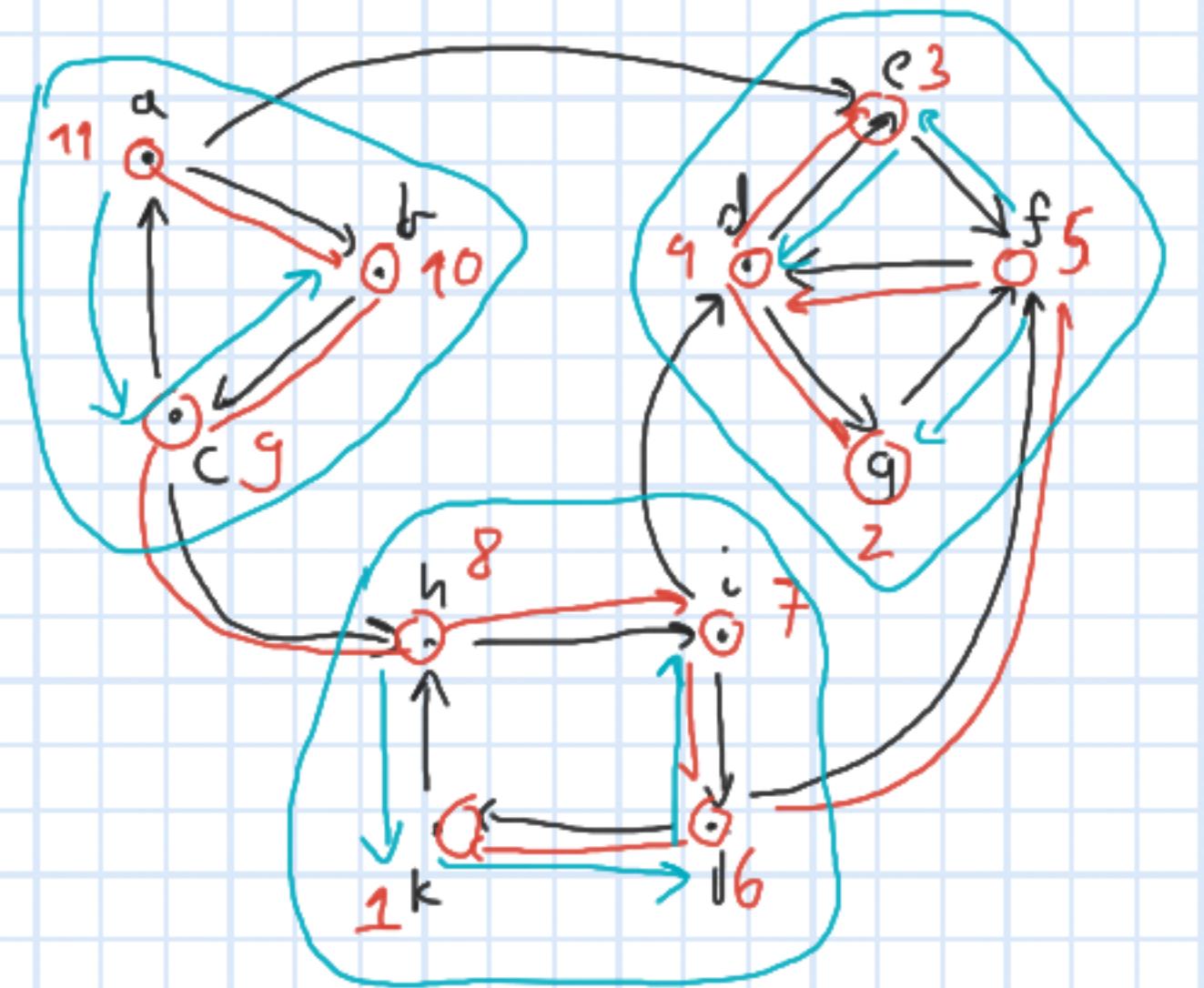
1. Wykonaj DFS zapisując rany pośrodku ujemnego ujemników

2. Odwrócić kierunek krążku

3. Wykonaj DFS ponownie, w kolejności malejących czasów pośrodku pojęcia z pierwotnego DFS

Intuicja





#### ④ Mosty w grafach nieskierowanych

def  $G = (V, E)$  - graf nieskierowany. Krawędź  $e \in E$  nazywamy mostem jeśli jej usunięcie spowoduje, że graf stanie się niespojny



tw Krawędź  $e$  jest mostem itw. gdy nie leży na żadnym cyklu prostym w grafie

dowód

$e = \{u, v\}$  jest mostem  $\Rightarrow e$  nie leży na cyklu bo

innejszej po jej usunięciu

ścieżki bieżącej ścieżki z  $u$  do  $v$

$e$  nie leży na żadnym cyklu  $\Rightarrow$  jest mostem bo pojęty

usunięta nie ma drogi z  $u$  do  $v$

#### Algorytm

1. Wykonaj DFS, zapisując dla każdego wierzchołka  $v$  czas odnalezienia  $d(v)$

2. Dla każdego wierzchołka  $v$  oblicz:

$$\text{low}(v) = \min(d(v), \min_{u \text{ - jest Nachborkiem}} d(u))$$

krzyżki do odniesionego, osiągalnego krawędzią ustępującą z  $v$   
nieprzezownego wierzchołka  
(poza krawędzią do rodzica)

$$\min_{w \text{ jest dzieciem } v} \text{low}(w)$$

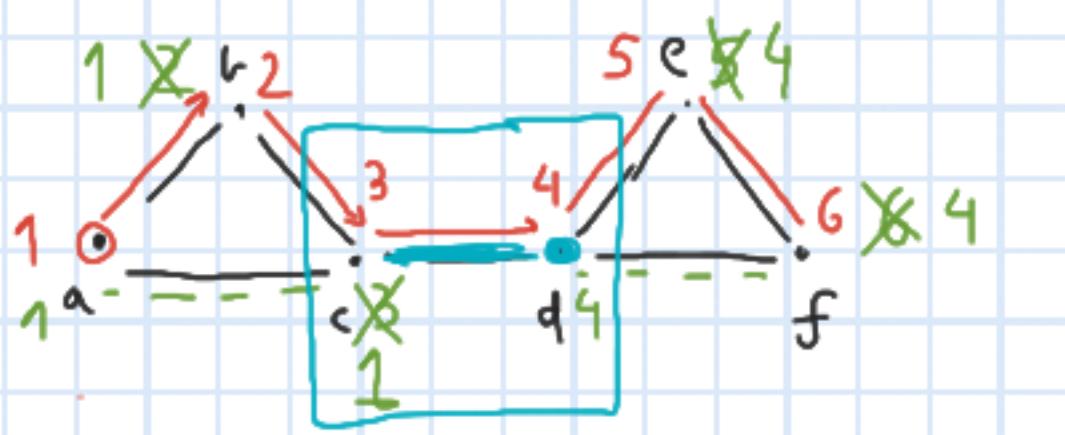
w duńcze DFS

3. Mosty to krawędzie

$$\{v, p(v)\}$$

gdzie  $d(v) = \text{low}(v)$

$\uparrow$   
rodzic  $v$



$d(v)$

$|out(v)|$

# Algorytmy i Struktury Danych

## Wykład 7

Nost w grafie - krawędź, której usunięcie rozspojąłoby graf

Krawędź c jest mostem, jeśli gdy nie leży na żadnym cyklu prostym

### Algorytm

① DFS z zapisywaniem czasu odwiedzenia ( $d(v)$  - czas odw. v)

② Dla każdego  $v$  obliczamy:

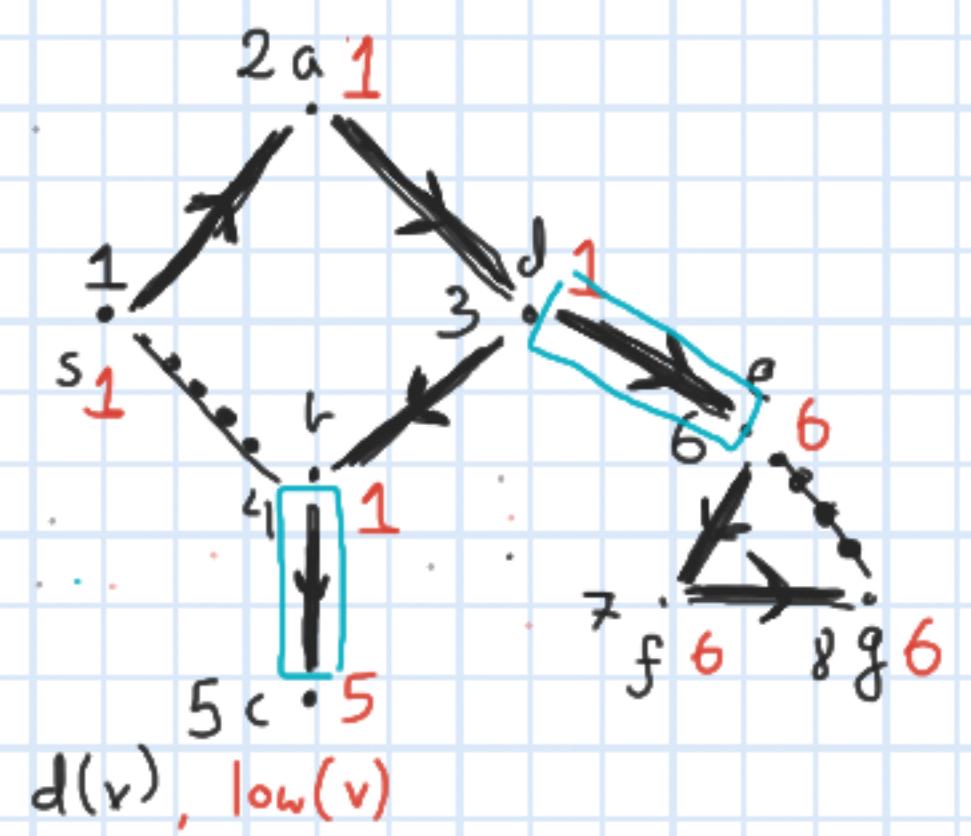
$$low(v) = \min \left( d(v), \min_{\substack{\text{istniego} \\ \text{krawędzi u} \\ \text{z } v \text{ do } u}} d(u), \min_{\substack{\text{w jąt dzieckiem} \\ v \in \text{drzewo DFS}}} low(u) \right)$$

③ Mosty to krawędzie

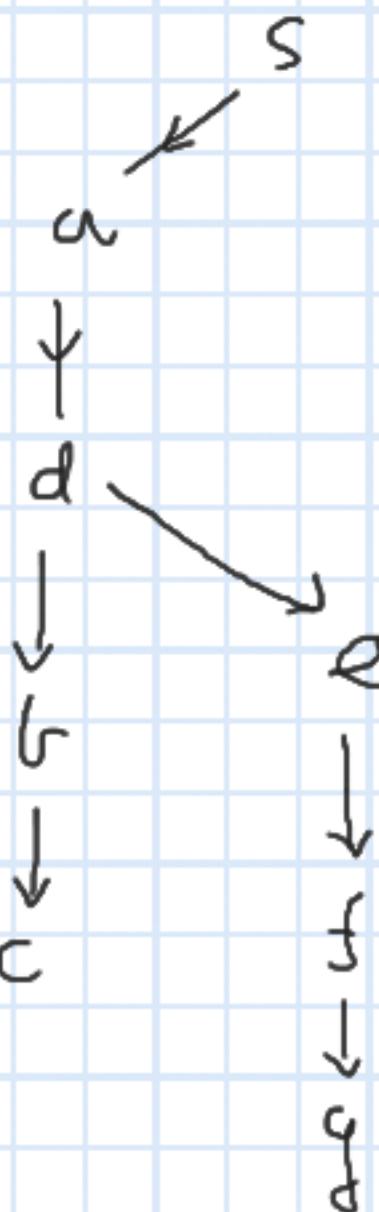
$$\{v, parent(v)\}$$

gdzie  $d(v) = low(v)$





$d(v)$ ,  $low(v)$



Jesli  $d(v) = low(v)$  to krawdzi  $\{v, parent(v)\}$  jest mostem

Dowód (nie uprost)



Jesli  $\{v, parent(v)\}$  nie jest mostem to istnieje sieka z  $v$  do  $parent(v)$  n.c. uzywajaca tej krawdzi

— Ta sieka musi zawiadzić krawędzi łączną (do wierzchołka  $x$ , dla którego  $d(x) < d(v)$ )

$$low(v) \leq d(x) < d(v)$$

Jesli  $d(v) \neq low(v)$  to  $\{v, parent(v)\}$  nie jest mostem



$$low(v) \neq d(v)$$

$low(v) < d(v)$  — to oznacza, że z  $v$  da się osiągnąć niedostępne  $x$ , t.z.  $d(x) < d(v)$

## Znajdowanie najkrótszych ścieżek w grafach ważonych

### Reprezentacja

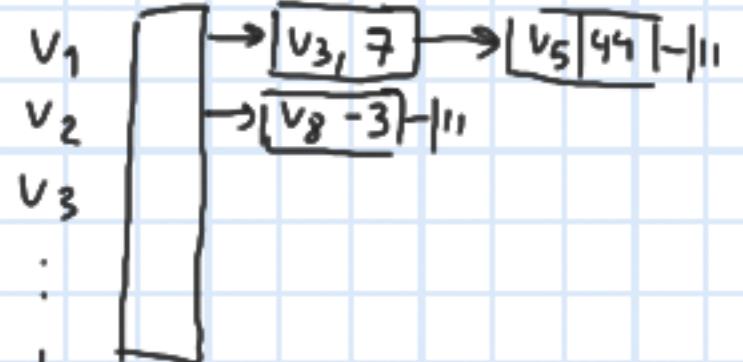
$$G = (V, E) - \text{graf}$$
$$w: E \rightarrow \mathbb{N} \quad (\mathbb{Z}, \mathbb{R})$$

### Reprezentacja macierzowa

$w$  - macierz wag

$w[i][j]$  - waga krawędzi między  $v_i$  i  $v_j$   
 $\infty$  gdy nie ma krawędzi

### Reprezentacja listowa

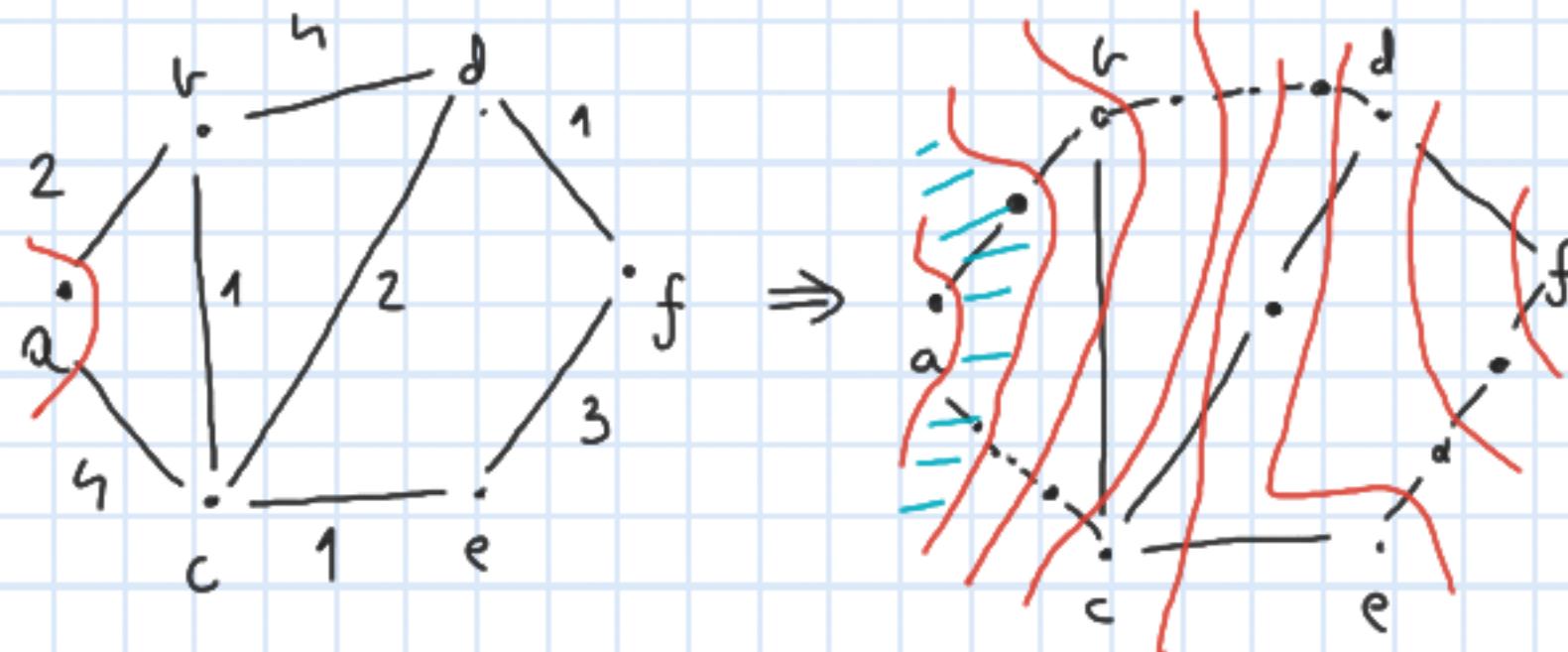


## Uwagę o problemie najkrótszych ścieżek

- 1-1 - na elementarnym poziomie trudne do wykonania
  - 1-uszyscy
  - uszyscy - uszyscy
- } standardowe wersje problemu

### Podejście elementarne (BFS z użyciem wiendotek)

długości/wagi krawędzi to małe liczby naturalne



trwanie sztucznych wiendotek

- nie tworząc ich fizycznie,  
tylko w kolejce BFS przedstawiając  
parę (wiendotek, licznik)

## Algorytm Dijkstry - algorytm elementarny,

ale u każdym kroku od razu skierowany do najbliższego  
przedzięnego wierzchołka

$\left\{ \begin{array}{l} \text{nie wymaga wag naturalnych, ale wymaga} \\ \text{wag nieujemnych} \end{array} \right.$

## Notacja

$$G = (V, E)$$

$w(u, v)$  - odległość z  $u$  do  $v$

$u.d$  - oszacowanie odległości z źródła do  $u$

$u.parent$  - poprzednik  $u$  na najkrótszej ścieżce

**Złożoność obliczeniowa**  
 $O(E \log V)$

## Algorytm (start z $s \in V$ )

① Umieść wszystkie wierzchołki w kolejce. } w prawidłyce często tylko  
tytu minimum → priorytetowej z oszacowaniem odległości } ustalonych pole u.d  
najmniej  $\infty$

② Ustalony  $s.d = 0$

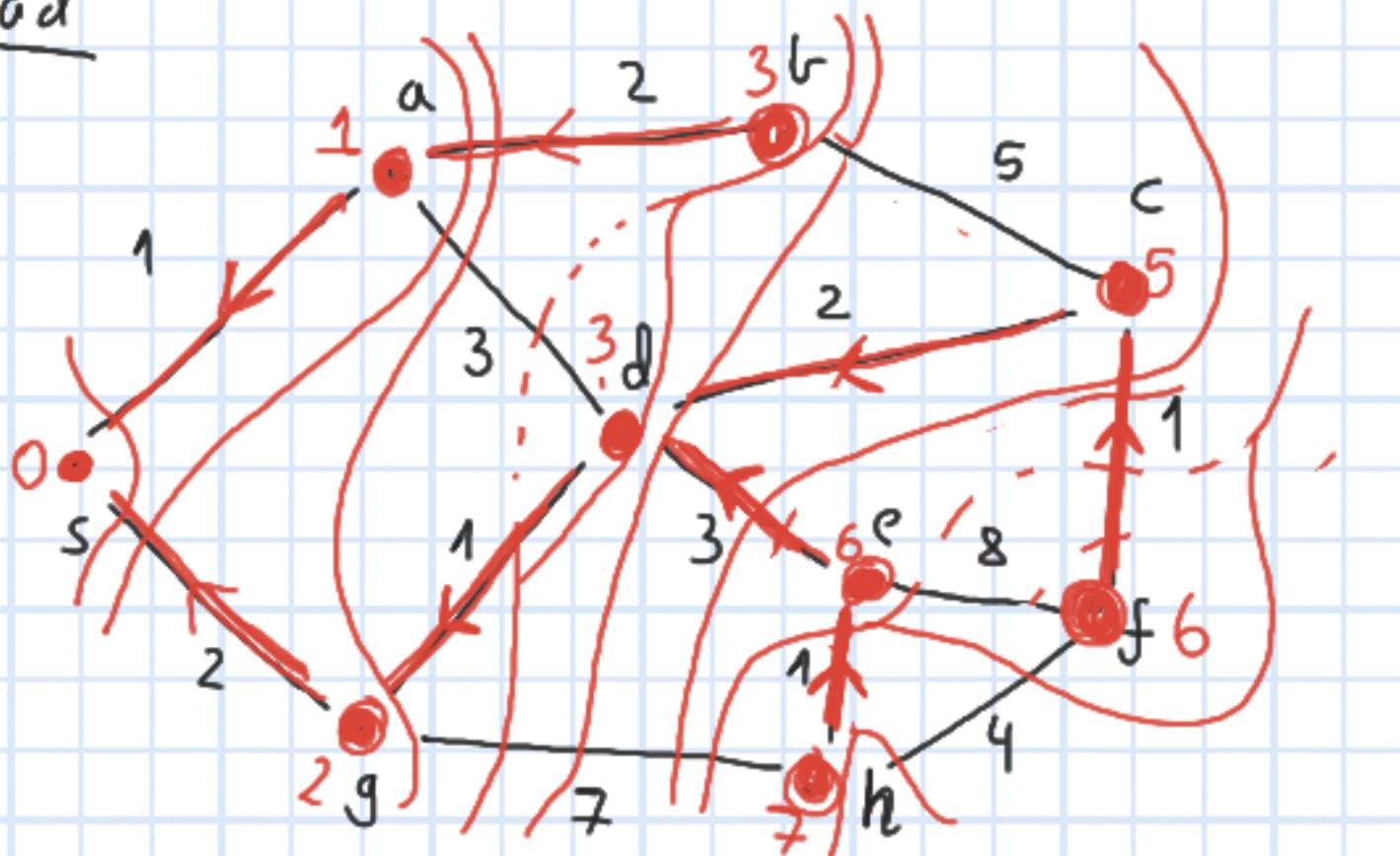
③ Póki kolejka nie jest pusta:

- usiąmij z kolejki u o minimalnej wartości u.d
- dla każdej krawędzi  $\{u, v\}$  dla grafu skierowanego  
wykonaj relaksację

wymaga aktualizacji  
kolejki !

```
def relax(u, v):
    if v.d > u.d + w(u, v):
        v.d = u.d + w(u, v)
        v.parent = u
```

Pnyktod



# Algorytm i Struktury Danych

## Układ 8

Szukamy najwótszych ścieżek w grafie gdy wagi krawędzi mogą być ujemne

Algorytm Bellmana - Forda (ścieżka wychodząca z s)  
 $G = (V, E)$

### ① Inicjalizacja

for  $v \in V$ :

$$v.d = \infty$$

$$v.parent = \text{None}$$

$$s.d = 0$$

### ② Relaksacja

for  $i$  in range( $|V| - 1$ ):

for  $(u, v) \in E$

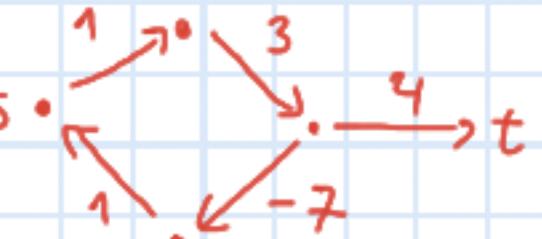
Relax( $u, v$ )

### ③ Weryfikacja

dla każdego  $(u, v) \in E$  sprawdzamy czy

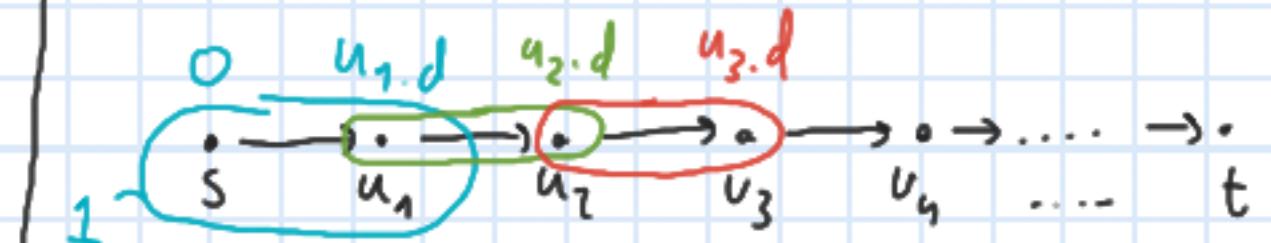
$$v.d \leq u.d + w(u, v)$$

→ rozwiązane moze nie istnieć (nie ma żadne zdefiniowane) gdy istnieje cykl o ujemnej wadze



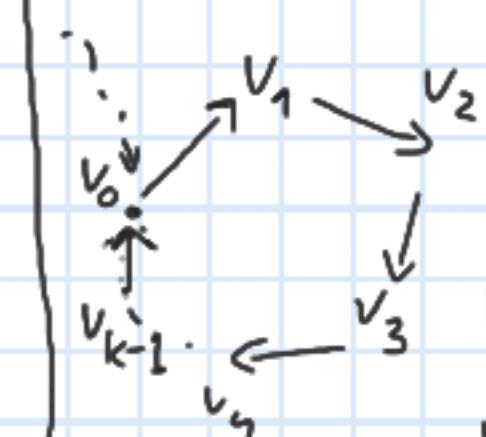
Czy ten algorytm jest poprawny?

Ugólniamy soląc przez najkrótszą ścieżkę z s do t



Po każdej iteracji głównej pętli u nasu ② algorytmu comuz dłuższy przebieg tej ścieżki ma poprawne wartości d

Jak działa wykrywanie ujemnych cykli?



$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0$$

Mamy powiększący cykl o ujemnej wadze, ale wykrywanie ujemnego cyklu zaawansowało.

Ale co innego zakończy:

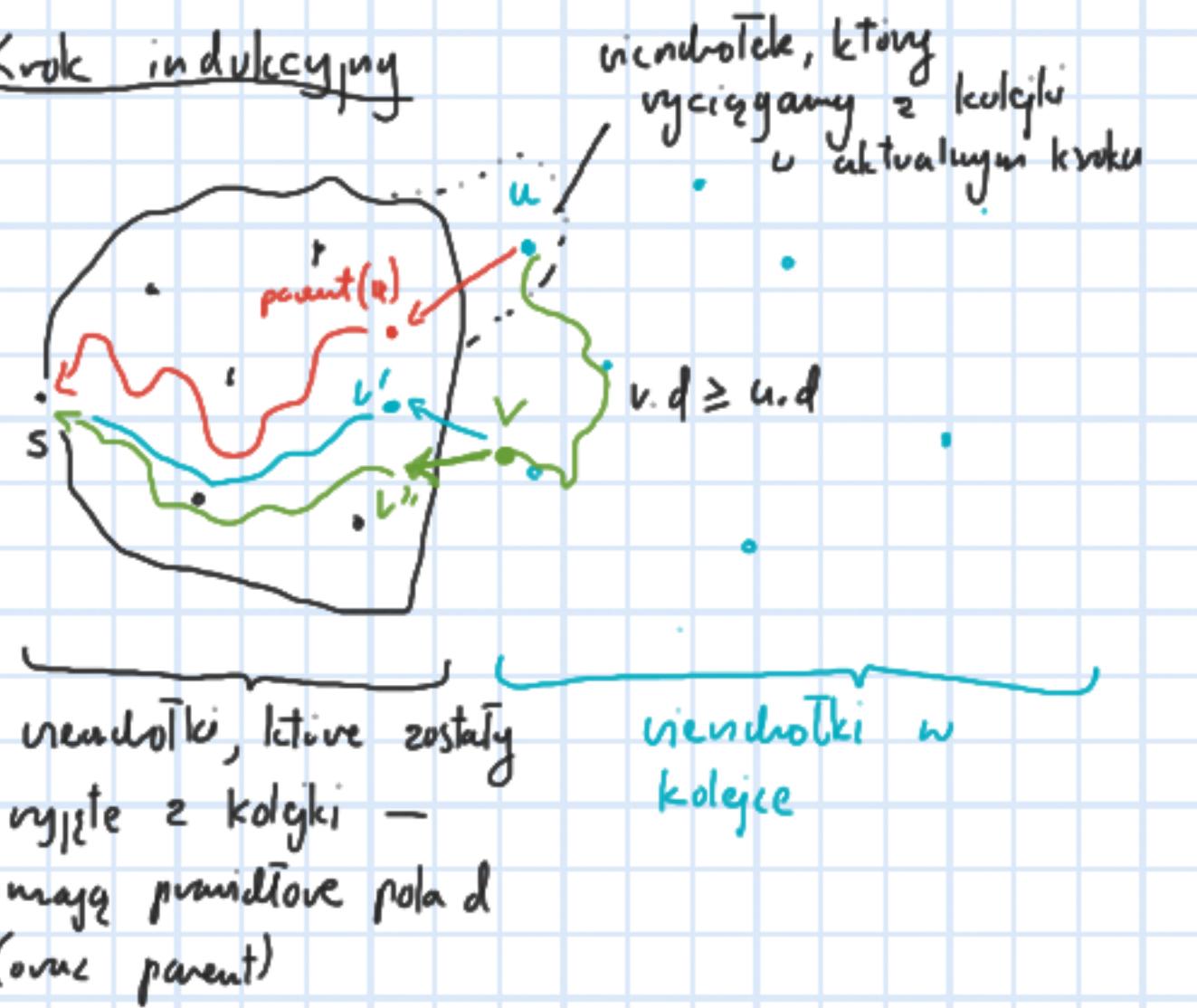
$$\sum_{i=1}^k v_i.d \leq \sum_{i=1}^k (v_{i-1}.d + w(v_{i-1}, v_i))$$

$$0 \leq \sum_{i=1}^k w(v_{i-1}, v_i) \rightarrow \text{Spłoszczenie } \nabla$$

## Poprawność algorytmu Dijkstry

Dowód przez indukcję – dla pierwszego kroku sytuacja jest jasna, bo jedynie wyciągnięty z kolejki wierzchołek to s i jego wartość  $s.d = 0$

### Krok indukcyjny



## Najkrótsze ścieżki między każdą parą wierzchołków

- ①  $|V|$  wywołani algorytmu Dijkstry  $O(VE \log V)$
- ②  $|V|$  wywołani algorytmu Bellmana-Forda  $O(V^2 E)$

### Konwencja

W specjalizowanych algorytmach dla najkrótszych ścieżek między każdą parą wierzchołków stosujemy reprezentację macienną

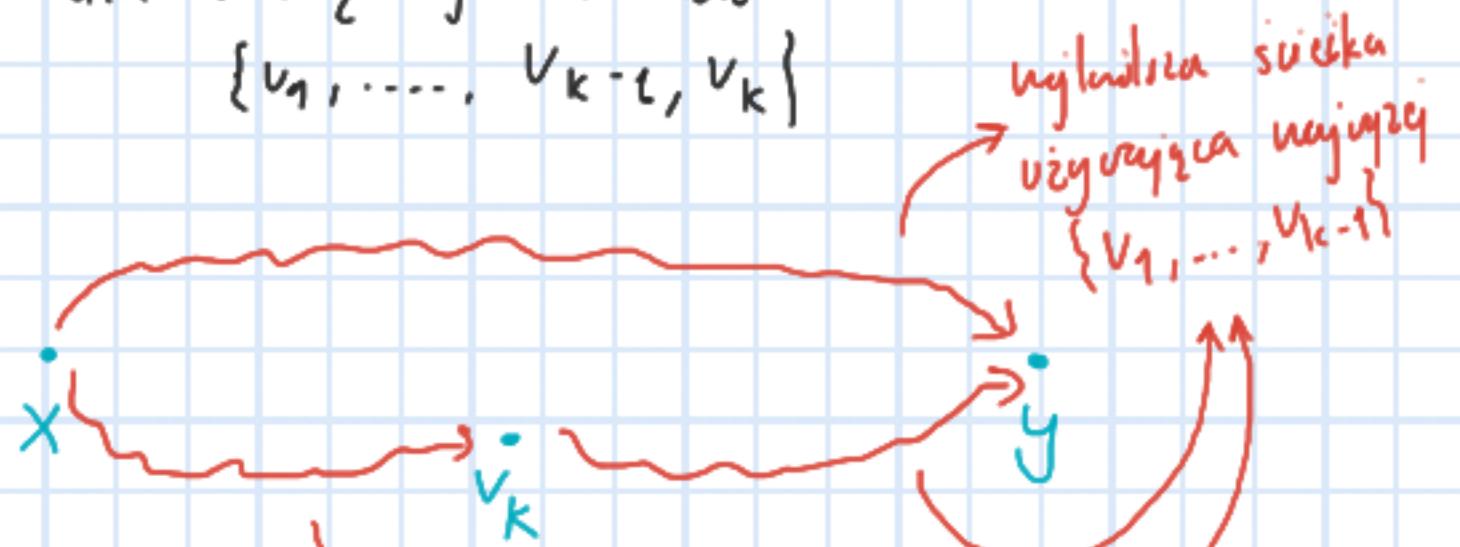
- i tak szukamy maciennych najkrótszych odległości  
 $D[u][v]$  to dł. najkrótszej ścieżki z  $u$  do  $v$

## Algorytm Floyda-Warshalla

Idea: Jeśli znamy najkrótsze ścieżki między każdą parą wierzchołków, które wierzchołki znajdują się tylko tych ze zbioru  $\{v_1, \dots, v_{k-1}\}$

to możemy osiągnąć podobne najkrótsze ścieżki dla reszty wierzchołków

$$\{v_1, \dots, v_{k-1}, v_k\}$$



Złożoność  
 $\Theta(V^3)$

$$V = \{v_1, \dots, v_n\}$$

$S^{(k)}$  - macierz długosci najkrótszych ścieżek konystruowanych z  $\{v_1, \dots, v_k\}$  jako wierzchołków

$S^{(0)}$  - macierz długosci krawędzi w grafie

## Algorytm

for  $k$  in range ( $1, n+1$ ):

for  $x \in V$ :

for  $y \in V$ :

$$S^{(k)}[x][y] = \min(S^{(k-1)}[x][y], S^{(k-1)}[x][v_k] + S^{(k-1)}[v_k][y])$$

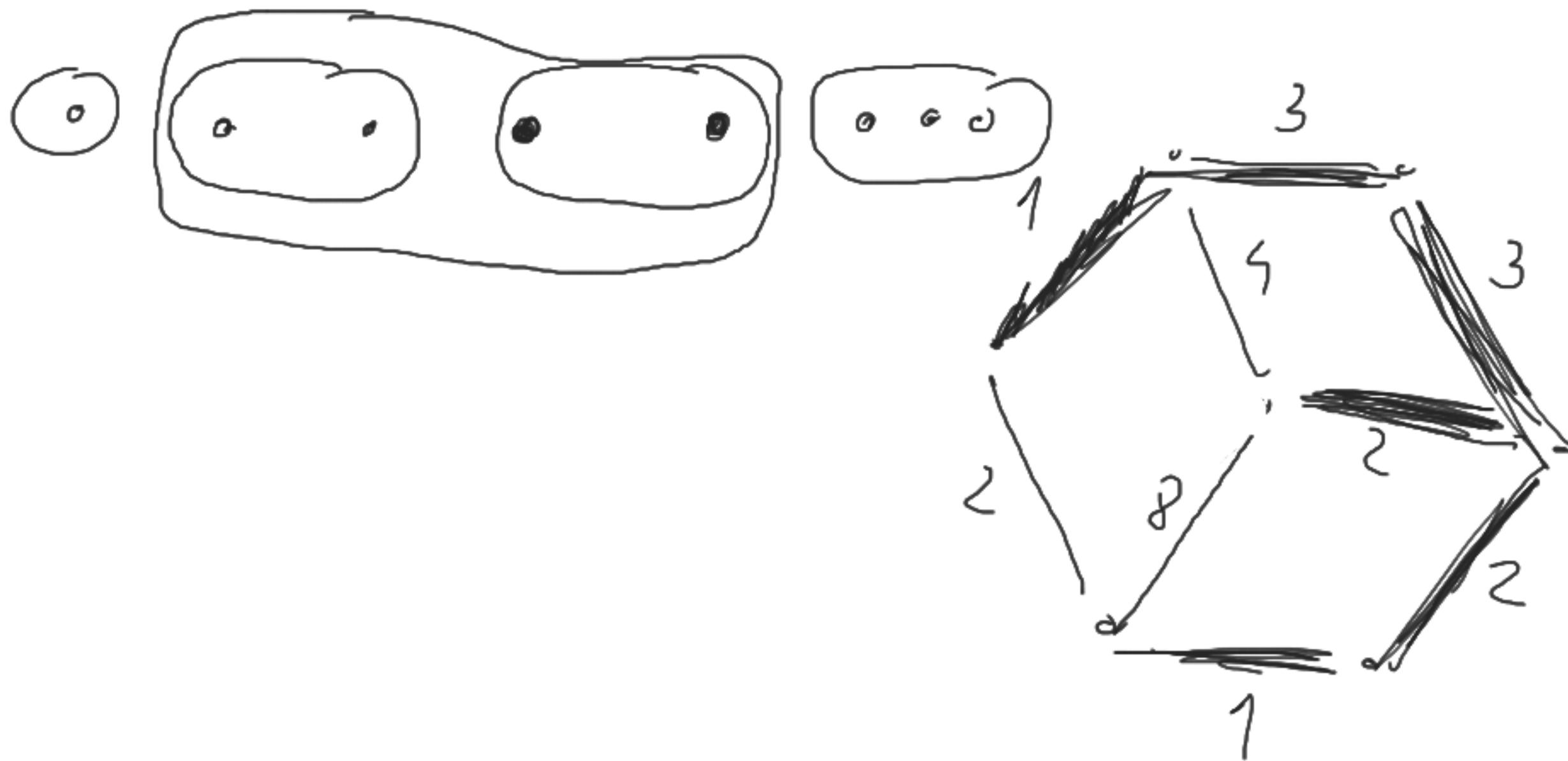
$$D = S^{(n)}$$

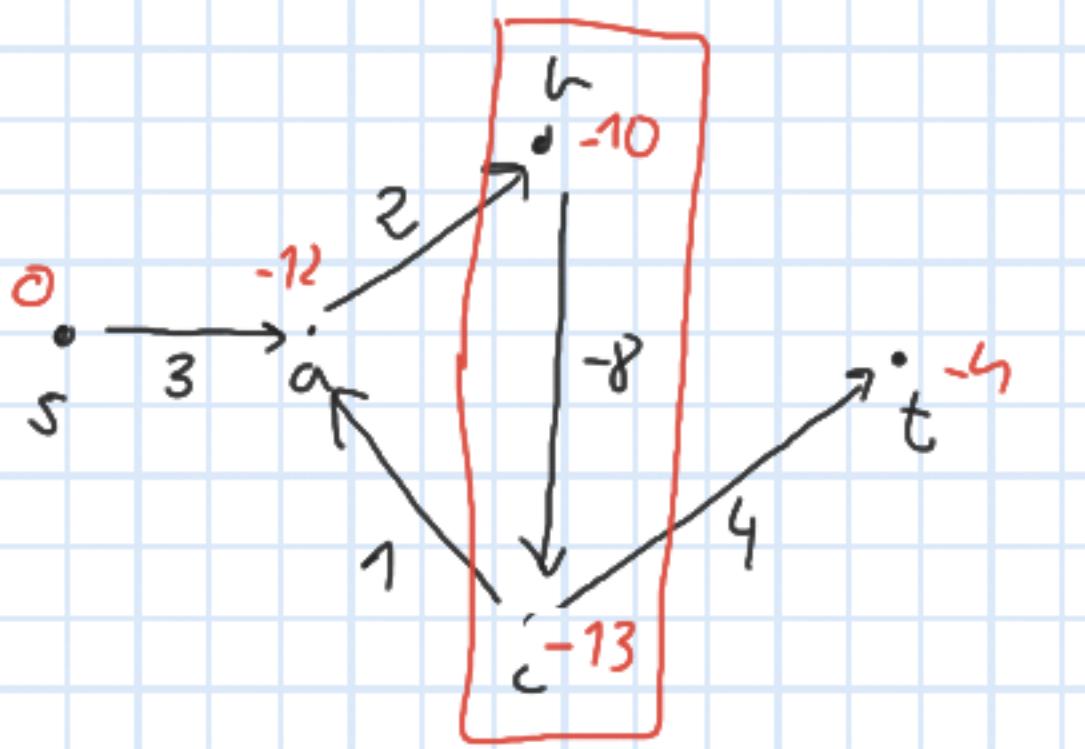
można przenieść!

możemy na tym etapie:

obliczać macierz P "parentów"

$P[x][y]$  - ostatni wierzchołek przed  $y$  na ścieżce z  $x$  do  $y$

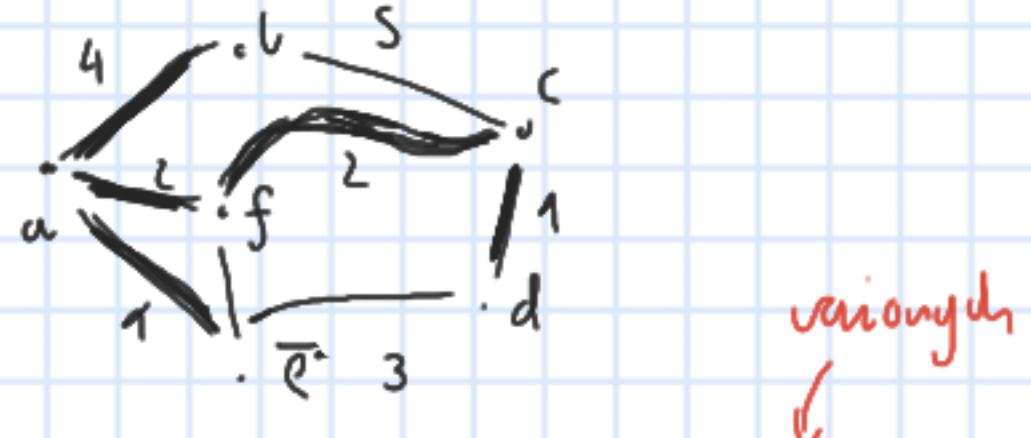




# Algorytmy i Struktury Danych

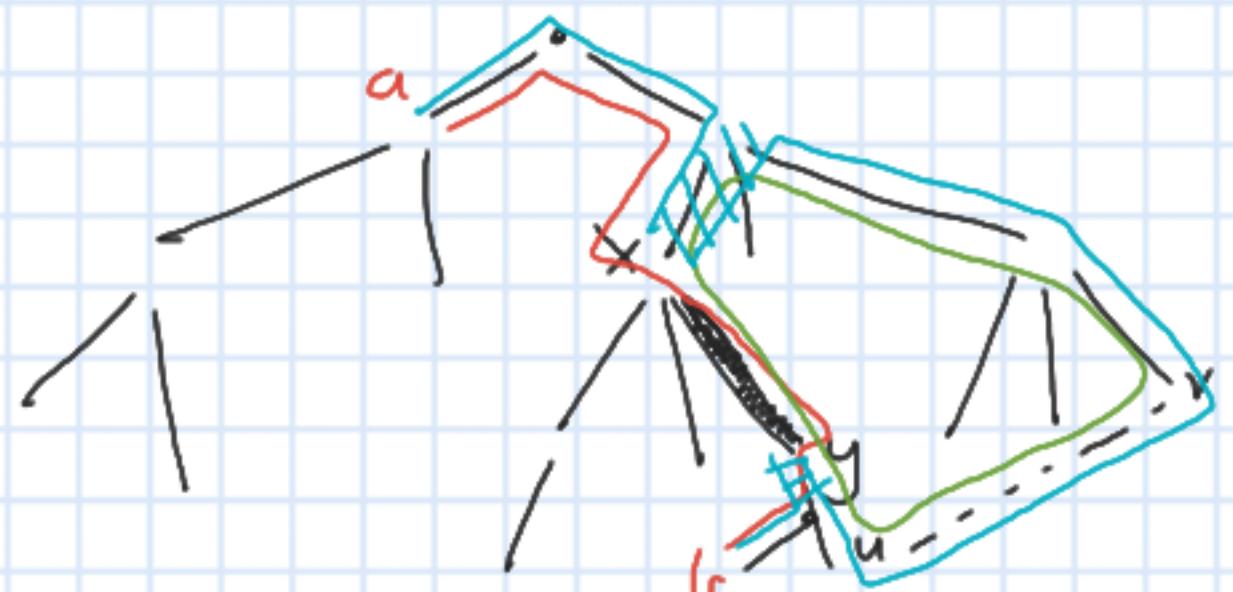
## Vykład 9

Minimalne drzewo rozpinające – zbiór krawędzi, które łączą wszystkie wierzchołki danego grafu i których suma wag jest minimalna.



(Problem dotyczący grafów spójnych, nieskierowanych)

MST – minimal spanning tree



## Obserwacja

$$G = (V, E), w: E \rightarrow \mathbb{R}$$

Jżeli  $A \subseteq E$  jest podzbiorem krawędzi pewnego MST dla  $G$  oraz  $e = \{u, v\}$  jest krawędzią taką, że:

a)  $e \notin A$

b)  $A \cup \{e\}$  nie zawiera cyklu

c)  $e$  ma minimalną wagę wśród krawędzi łączących parzyste wierzchołki

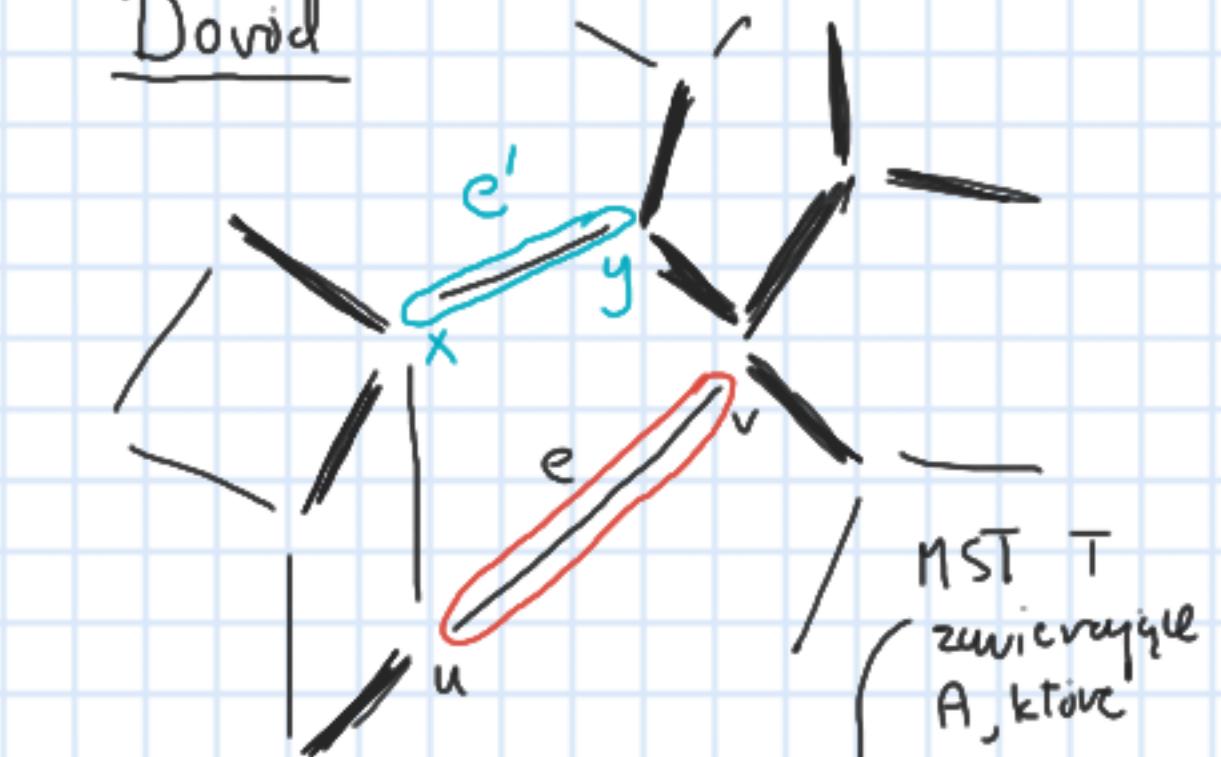
Wówczas  $A \cup \{e\}$  jest podzbiorem krawędzi pewnego MST

dla każdej pary wierzchołków istnieje ścieżka jej sumą

$a \rightsquigarrow x \rightsquigarrow y \rightsquigarrow b$

$a \rightsquigarrow x \rightsquigarrow \{u, v\} \rightsquigarrow y \rightsquigarrow b$

## Dowód



Jżeli nie istnieje MST rozpinający A overz zawierający  $e$ , to istnieje zapewniająca ścieżkę z  $u$  do  $v$  jakaś inną krawędź  $e' = \{x, y\}$  spoza  $A$

Trzecim, że  $T' = (T / \{e'\}) \cup \{e\}$  również jest MST

Zauważmy, że:

$$\textcircled{1} \quad w(T) \geq w(T') \quad \text{bo } w(e') \geq w(e)$$

←  $\textcircled{2} \quad T'$  uciski jest drzewem

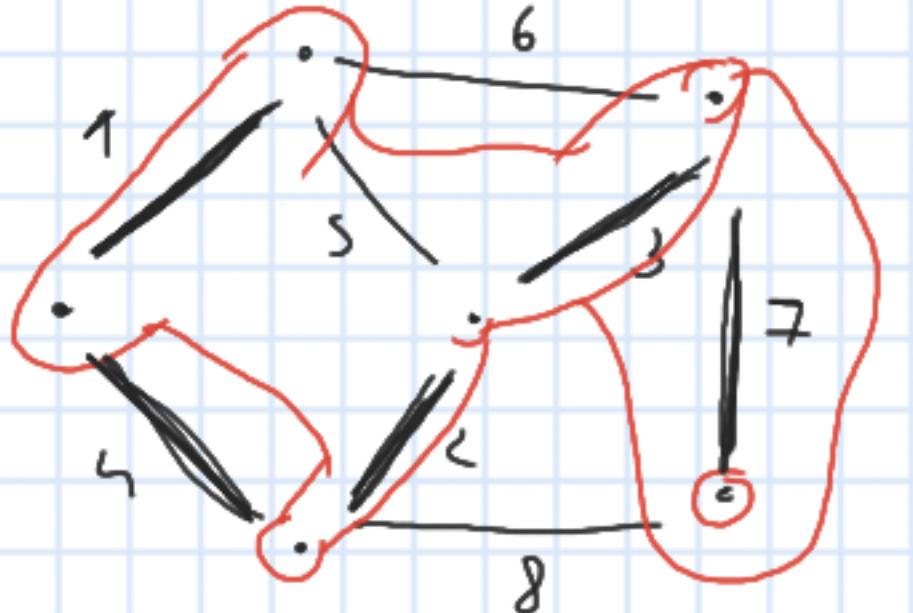
## Algorytm Kruskala znajdowania MST

Na wejściu mamy  $G = (V, E)$ ,  $w: E \rightarrow \mathbb{R}$

- ① Posortuj krawędzie po wagach  $O(E \log E)$
- ②  $A := \emptyset$   $O(1)$
- ③ Przeglądaj krawędzie w kolejności niemalejącej wag  
(rozważana krawędź to e)
 

Jeśli  $A \cup \{e\}$  nie tworzy cyklu to

$O(E)$  op.  
na Find-Union
- ④ Zwrócić A

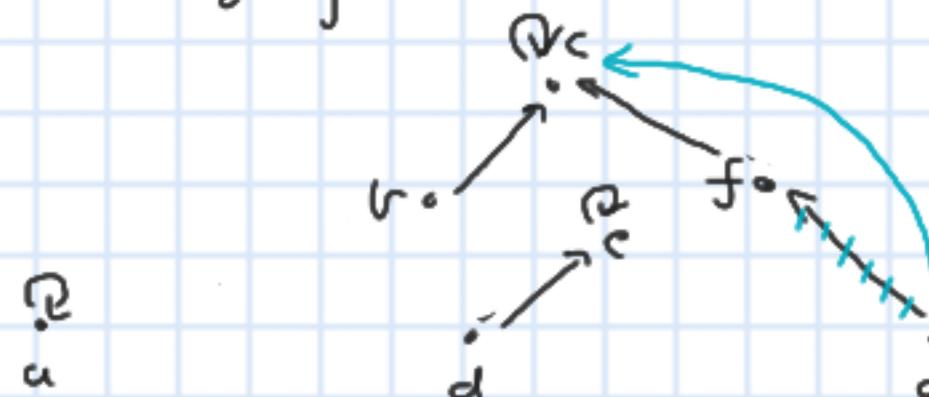


cały algorytm  
ma złożoność  
 $O(E \log E) =$   
 $O(E \log V)$

## Struktura Find-Union dla zbiorów rozłącznych



Struktura Find-Union realizująca jądro lasu zbiorów rozłącznych



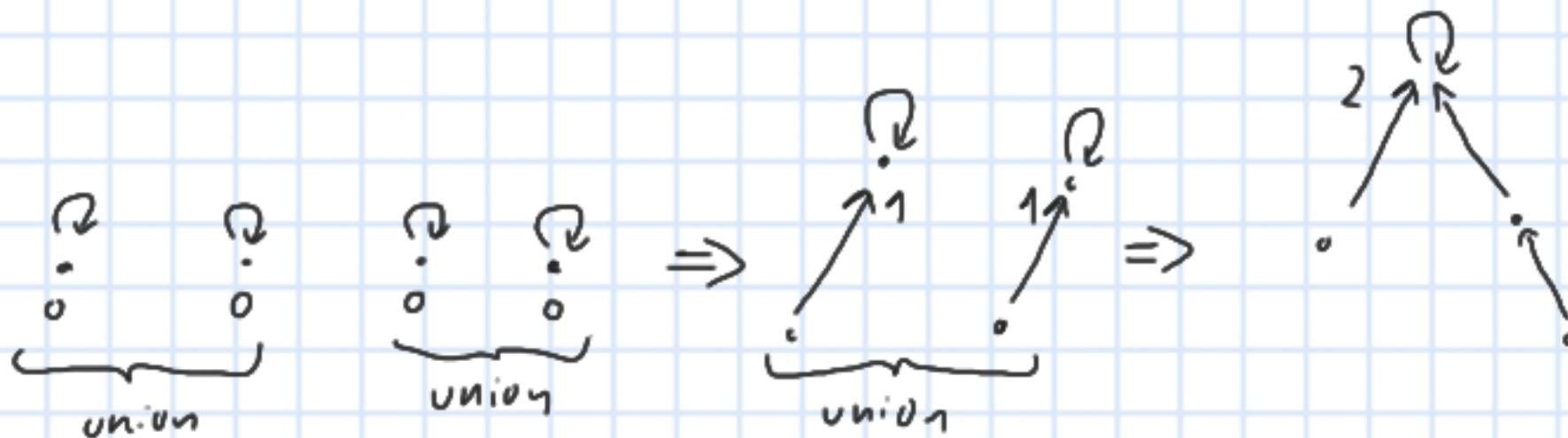
class Node:

```
def __init__(self, value):
    self.val = value
    self.parent = self
    self.rank = 0
```

```

def find( x ):
    if x.parent != x:
        x.parent = find(x.parent)
    return x.parent

```



```
def union( x, y ):
```

```

x = find(x)           if x == y: return
y = find(y)           / 
if x.rank > y.rank:
    y.parent = x
else:
    x.parent = y
    if x.rank == y.rank:
        y.rank += 1

```

gdzie  $\log^* n$  to to ile razy należy zastosować log, aby argument spadł do 1 w unii

### Obserwacja

Dlugość ranka k ma co najwyżej  $2^k$  elementów, oznacza to, że ścieżka o długości co najwyżej k

### Obserwacja

Jśli mamy n elementów to najwyższa runga wynosi  $\log n$ , co znaczy, że każda operacja find ma złożoność  $O(\log n)$ .  
Zatem czas całkowity na wykonanie m operacji find wynosi  $O(m \log n)$ .

### Fakt

Jśli wykonujemy m operacji na strukturze Find-Union zarządzającej n elementów, to ich czas całkowy wynosi  $O(m \log^* n)$ .

z Tymczasem według  
rungi i kompresji  
ścieżki

## Algorytm Prima znajdowania MST

$$G = (V, E), w: E \rightarrow \mathbb{R}$$

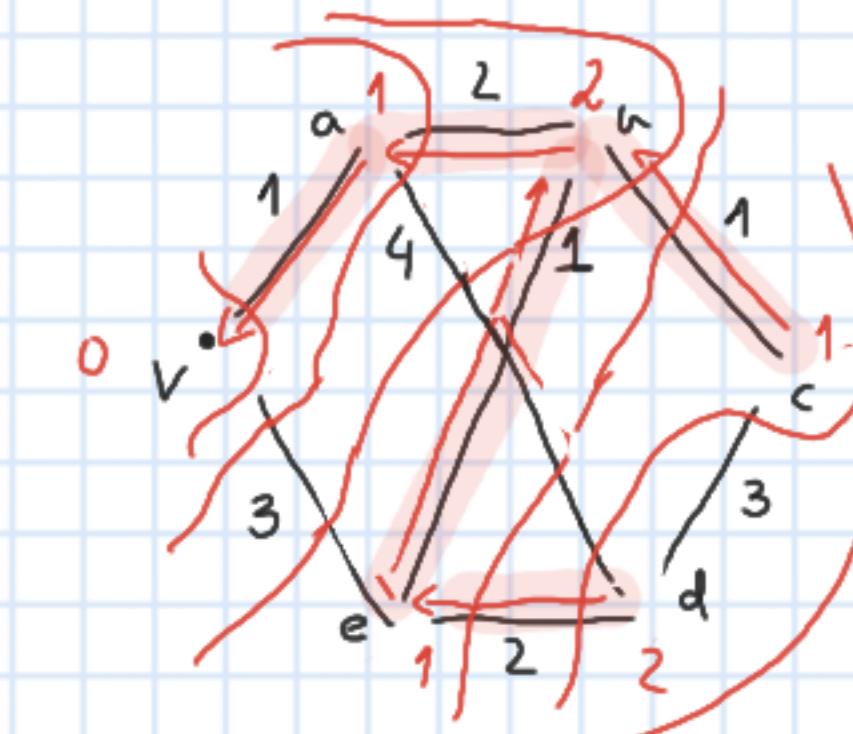
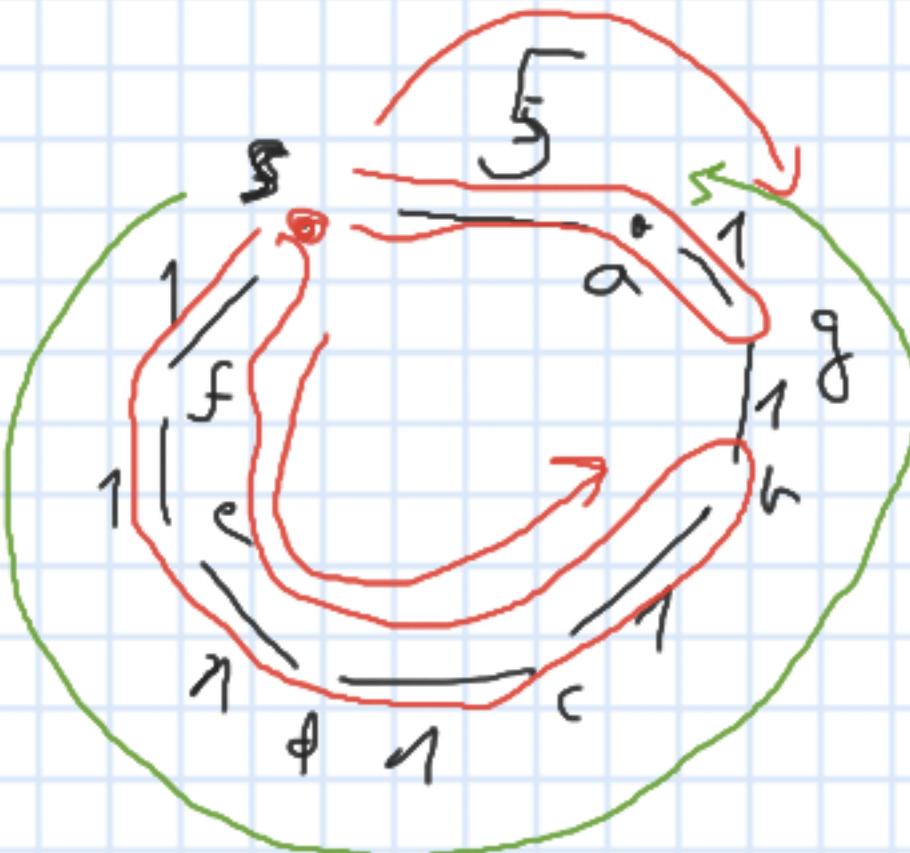
① Unieść wszystkie wierzchołki w kolejce, z wagą  $\infty$

② Zmienią wagę wierzchołka  $v$  na 0  
 ↗ wierzchołek startowy, zadany na wejściu

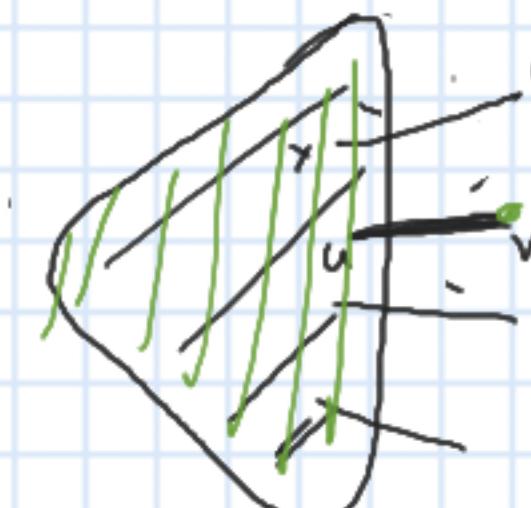
③ Jak daleko są wierzchołki w kolejce:

- ujmij wierzchołek  $t$  o minimalnej wadze z kolejki
- dla każdej krawędzi  $e = \{t, u\}$ , jeśli waga  $u$  jest  $\geq w(t, u)$  to zmniejsz ją do  $w(t, u)$

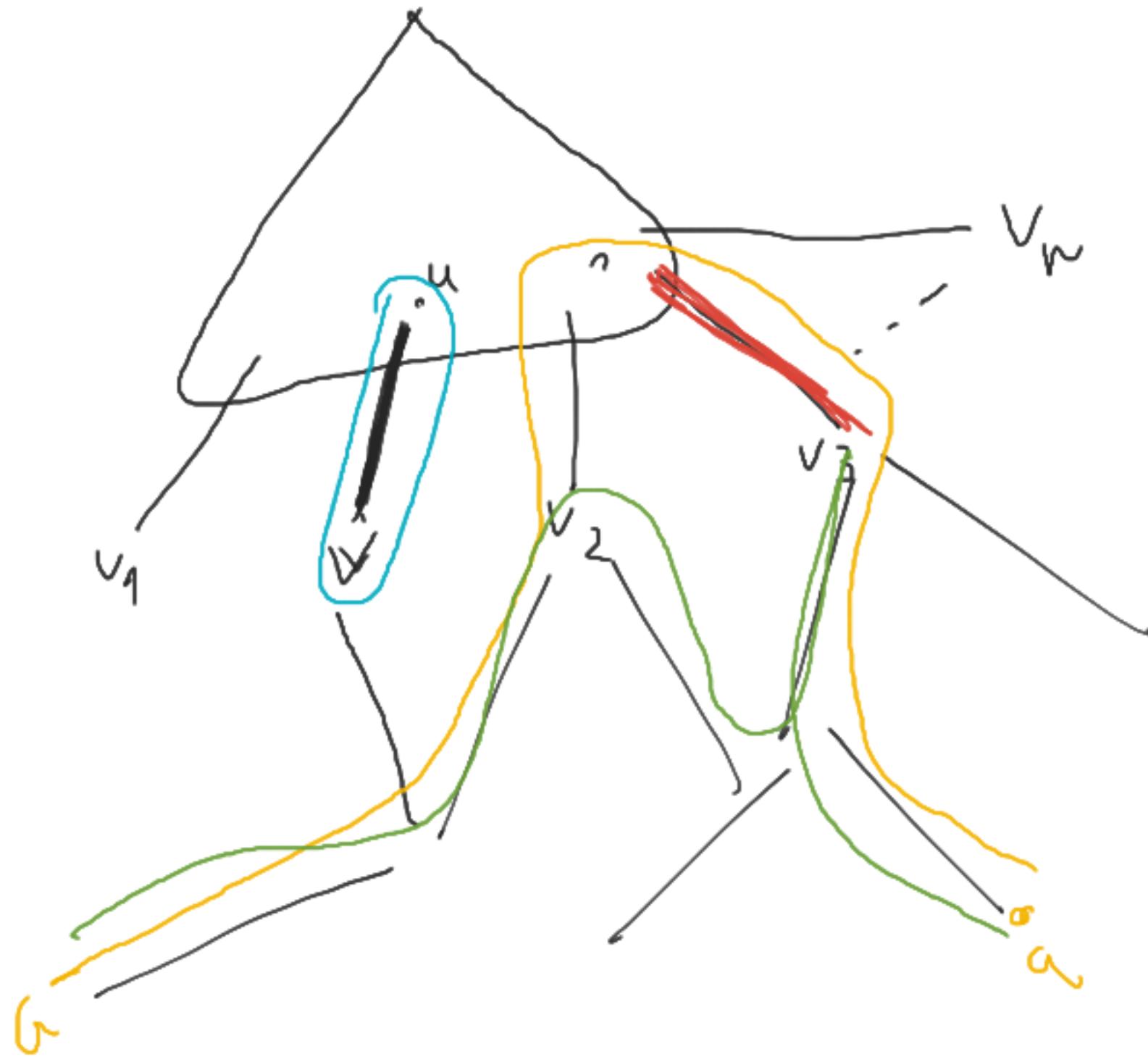
i ustaw  $u.parent$  na  $t$



$O(E \log V)$



$e = \{u, v\}$  o minimalnej  
wadze, wychodzącej  
z wierzchołka ją powołanego  
przez alg. Prima



# Algorytmy i struktury danych

## Wykład 10 - Metody konstrukcji algorytmów

- dziel i złączaj ← Quick Sort, Merge Sort
- zachowane algorytmy ← alg. Kruskala
- programowanie dynamiczne ← metoda zamiany  
wykładniowych algorytmów rekurencyjnych na  
iteracyjne wielomianowe, przez spamiętywanie  
wyników wcześniejszych

## Punkkład elementarny

$$F_0 = 1$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

} ciąg Fibonacciego

```
def fib(n):
```

```
    if n <= 1: return 1
```

```
    return fib(n-1) + fib(n-2)
```

## Uwaga: dynamika

```
def fib_dyn(n):
```

$$F = [0] * (n + 1)$$

$$F[0] = 1$$

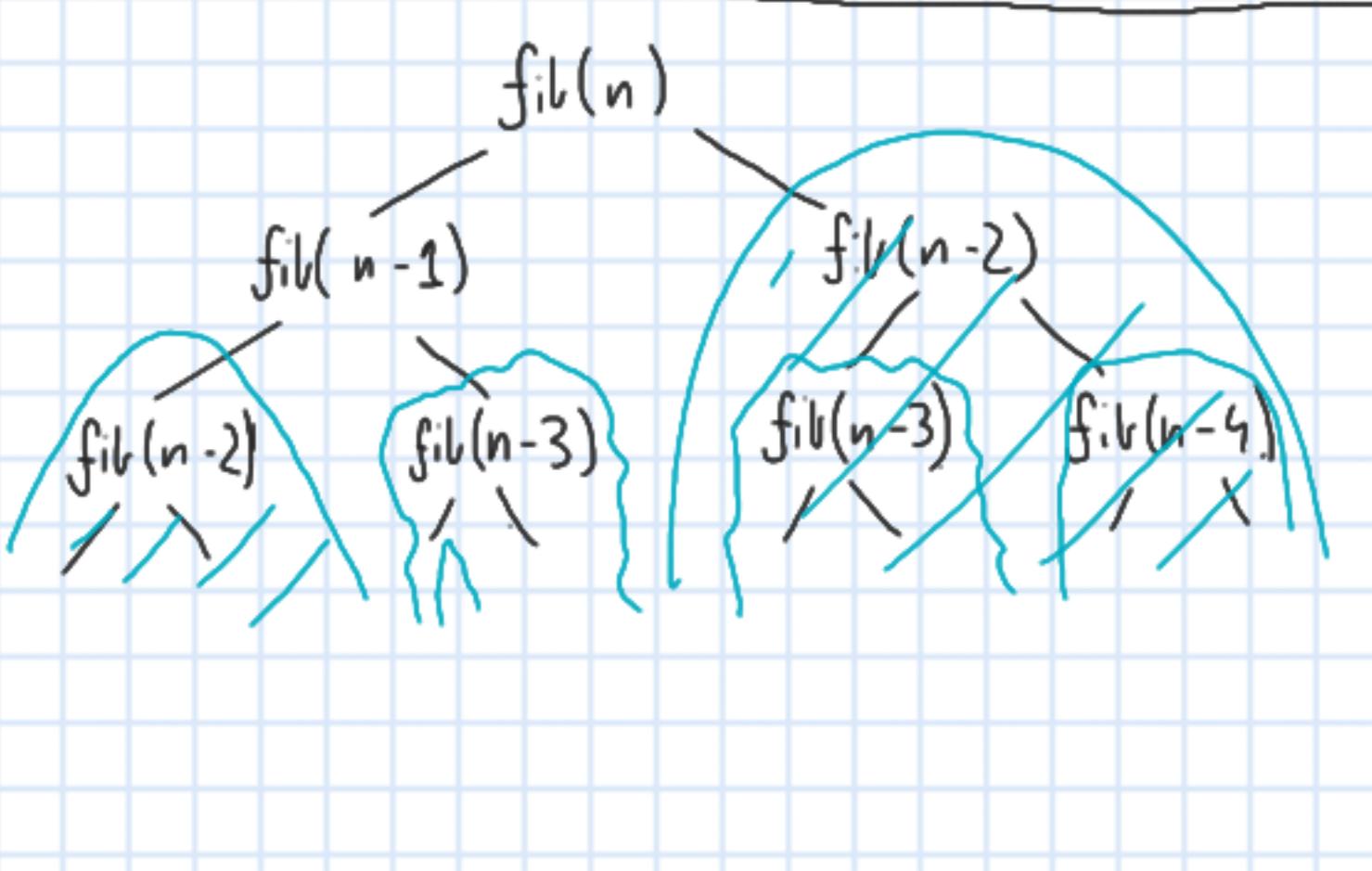
$$F[1] = 1$$

```
for i in range(2, n+1):
```

$$F[i] = F[i-1] + F[i-2]$$

```
return F[n]
```

} wartość nie potrzebujemy  
czyścić tablicy  $F$ , tylko  
dwie ostatnie linie



## Punktad niebanalny

longest  
increasing  
subsequence

Najdłuższy rosnący podciąg (LIS)

Dane:  $A[0], \dots, A[n-1]$  - ciąg lin

Zadanie: Znaleźć długość najdłuższego (niekoniecznie spójnego) rosnącego podciągu

## Punktad

$\circ \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7$   
 $A: 2, 1, 4, 3, 5, 8, 5, 7$   
 $f: 1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 4 \quad 5$   
 $P: -1 \quad -1 \quad 0 \quad 0 \quad 3 \quad 4 \quad 6$

Schemat działania algorytmu dynamicznego

- ① ustalić funkcję, której będziemy obliczać
- ② zapisać powyższą funkcję w postaci rekurencyjnej
- ③ konwersja zapisu rekurencyjnego do iteracyjnego (z wykorzystaniem spamiętywania wyników)
- ④ jak odczytać wynik

① Funkcja, którą będziemy obliczać

$f(k) = \text{długość najdłuższego podciągu kończącego się na } A[k]$   
wynik dla problemu to  $\max_k f(k)$

② Zapis rekurencyjny

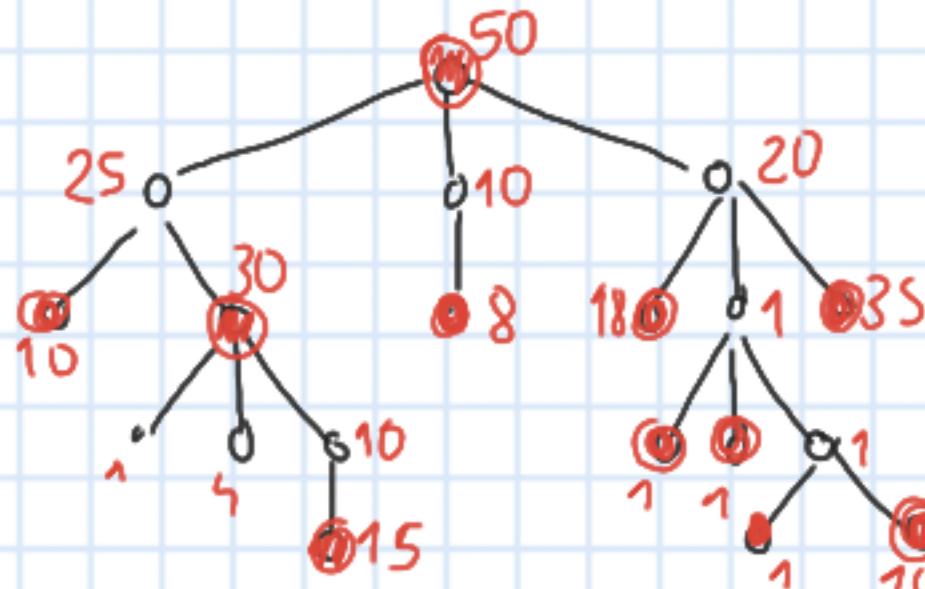
$$f(k) = \max \left\{ \underbrace{f(t)+1}_{=0 \text{ dla } t \text{ ujemnych}} \mid t < k \wedge A[t] < A[k] \right\}$$

③ Implementacja

```
def lis(A):
    n = len(A)
    F = [1] * n
    P = [-1] * n
    for k in range(1, n):
        for t in range(k):
            if A[t] < A[k] and F[k] < F[t] + 1:
                F[k] = F[t] + 1
                P[k] = t
    return max(F) // F, P
```

```
def print_sol(A, P, k):
    if P[k] != -1:
        print_sol(A, P, P[k])
    print(A[k])
```

## Problem imprezy firmowej



class Employee:

```
def __init__(self, fun):
    self.emp = []
    self.fun = fun
    self.f = -1
    self.g = -1
```

V = Employee(50)

W = Employee(10)

X = Employee(23)

V.emp.append(W)

V.emp.append(X)

① Jak opisać funkcję rozwiązywania

v - ugザt dneva

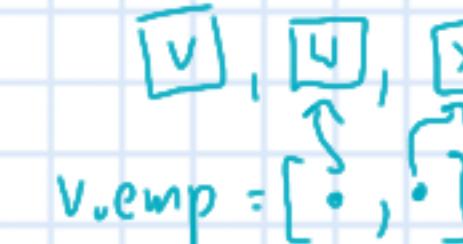
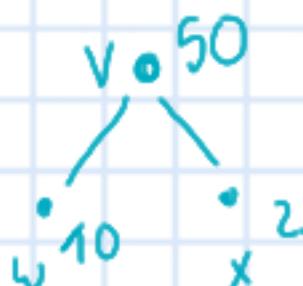
f(v) - wartość najlepszej imprezy dla poddneva ukoncentrowanego w v

g(v) - j.w., gdy v nie jest zaproszony

② Zapis rekurencyjny

$$g(v) = \sum_{u - \text{pracownik } v} f(u)$$

$$f(v) = \max \left\{ g(v), \text{fun}(v) + \sum_{u - \text{pracownik } v} g(u) \right\}$$



③ Implementacja

def f(v):

if v.f ≥ 0: return v.f

x = g(v)

y = v.fun

for u in v.emp:

y += g(u)

v.f = max(x, y)

return v.f

def g(v):

if v.g ≥ 0: return v.g

v.g = 0

for u in v.emp:

v.g += f(u)

return v.g

# Algorytmy i Struktury Danych

## Układ 11

Problem plecakowy (ang. Knapsack)

Dane:  $I = \{0, \dots, n-1\}$  - przedmioty

$w: I \rightarrow \mathbb{N}$  - wagi przedmiotów

$p: I \rightarrow \mathbb{N}$  - ceny przedmiotów

$B \in \mathbb{N}$  - maksymalna waga

Zadanie: Znaleźć podzbiór  $I$  o maksymalnej sumarycznej cenie oraz wadze nie przekraczającej  $B$ .

① Funkcja, której będziemy obliczać

$f(i, b) = \text{maksymalna suma cen przedmiotów z } \{0, \dots, i\}, \text{ których waga nie przekracza } b$

{ wynik to  $f(n-1, B)$

② Sformułowanie rekurencyjne

nie liczymy  $i$ -go przedmiotu

$$f(i, b) = \max \left( \begin{array}{l} f(i-1, b), \\ f(i-1, b - w(i)) + p(i) \end{array} \right)$$

nie liczymy  $i$ -ty przedmiot
jak nie, to ten człon pomijamy

$$f(0, b) = \begin{cases} p(0) & , w(0) \leq b \\ 0 & , w(0) > b \end{cases}$$

### ③ Implementacja

```
def knapsack(W, P, B):
```

```
    n = len(W)
```

```
    F = [[0 for b in range(B+1)] for i in range(n)]
```

```
    for b in range(W[0], B+1):
```

```
        F[0][b] = P[0]
```

```
    for b in range(B+1):
```

```
        for i in range(1, n):
```

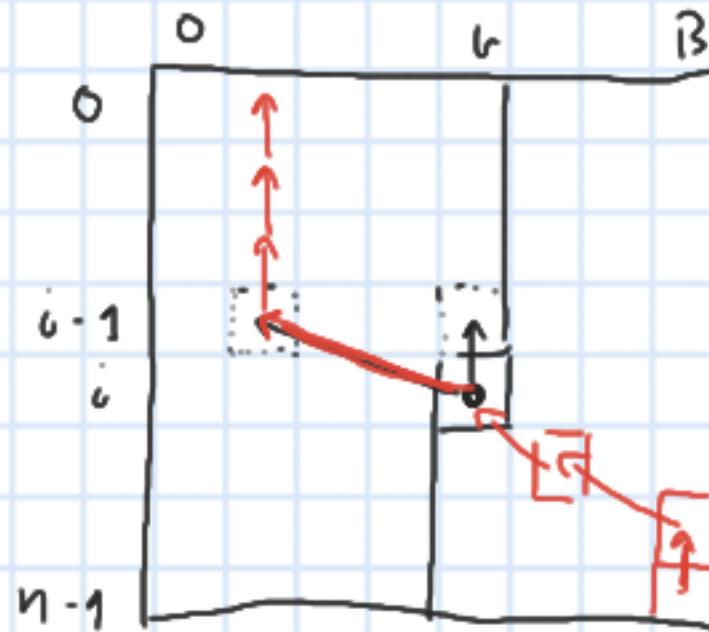
```
            F[i][b] = F[i-1][b]
```

```
            if b - W[i] >= 0:
```

```
                F[i][b] = max(F[i][b], F[i-1][b-W[i]] + P[i])
```

```
    return F[n-1][B]
```

Odtwarzanie rozwiązań



# Problem komiwojazna (TSP, traveling salesperson problem)

Dane:  $C = \{0, \dots, n-1\}$  - zbiór miast

$d: C \times C \rightarrow \mathbb{R}_+$  - odległość między miastami

Zadanie: Znaleźć kolejność odwiedzania miast tak, by

zamknić się w 0, odwiedzić wszystkie

miasta i pokonać możliwie jak najkrótszą  
trase

Ogólnie problem jest NP-zupełny

→ jeśli  $d$  nie jest metryką to nie ma żadnych algorytmów

→ jeśli  $d$  jest metryką  $\rightarrow \frac{3}{2}$ -aproxymacyjna  
jeśli miasta w przestrzeni euklidesowej - PTAS

$\hookrightarrow \mathbb{R}^2$  i trasa liczbowa  $\rightarrow$  istnieje alg.  
 $\xrightarrow{\text{wielomianowy}}$

## Wersja ogólna problemu

Brute-force: sprawdza kolejność kolejowici miast  
 $\Theta(n!)$

## Algorytm dynamiczny

### ① Określenie funkcji

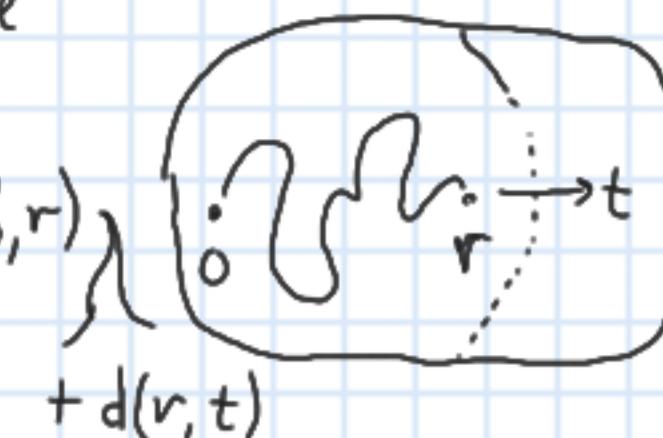
$S \subseteq C$  - podzbiór miast,  $0 \in S$ ,  $t \in S$

$f(S, t) =$  długość (w sensie  $d$ ) najkrótszej trasy z 0 do  $t$ ,  
odwiedzającej wszystkie miasta z  $S$ : żadnego innego

$$\left\{ \begin{array}{l} \text{rozważanie} \\ \min_{t \in C \setminus \{0\}} f(C, t) + d(t, 0) \end{array} \right.$$

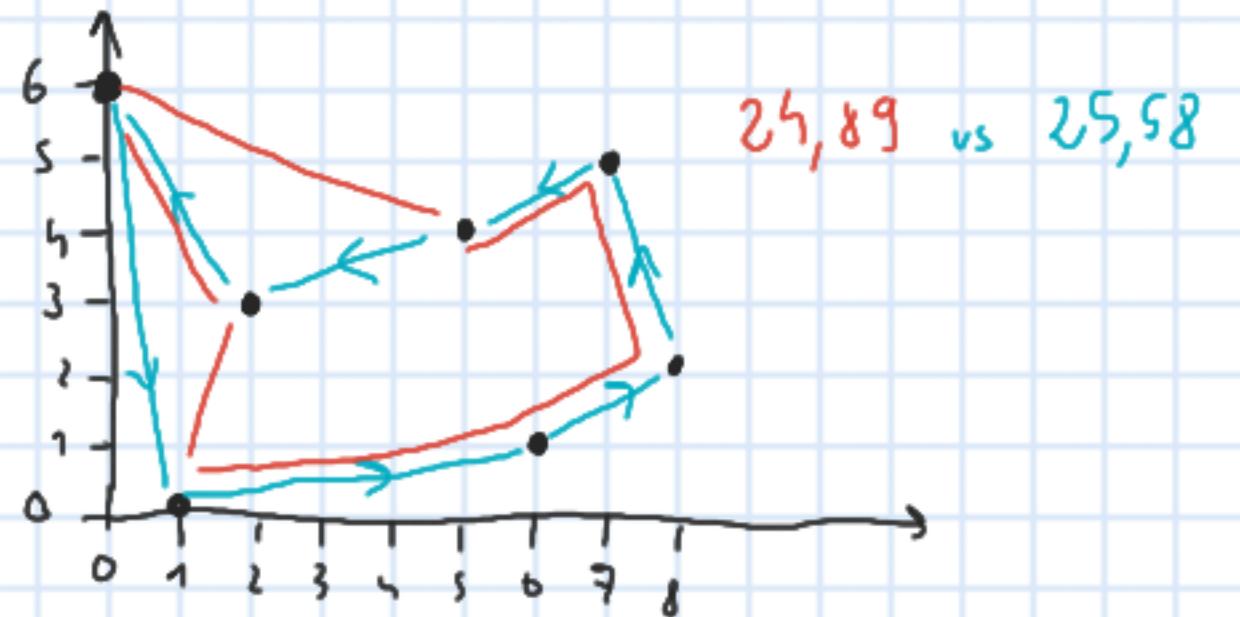
### ② Sformułowanie rekurencyjne

$$f(S, t) = \min_{r \in S \setminus \{t\}} f(S \setminus \{t\}, r)$$



## Uwaga litoniana problemu komisuracji

Niasta sę w  $\mathbb{R}^2$ , najpiękniej udrugujemy  
tylko w prawo, potem tylko w lewo



## Algorytm dynamiczny

## ① Okreslenie funkcji

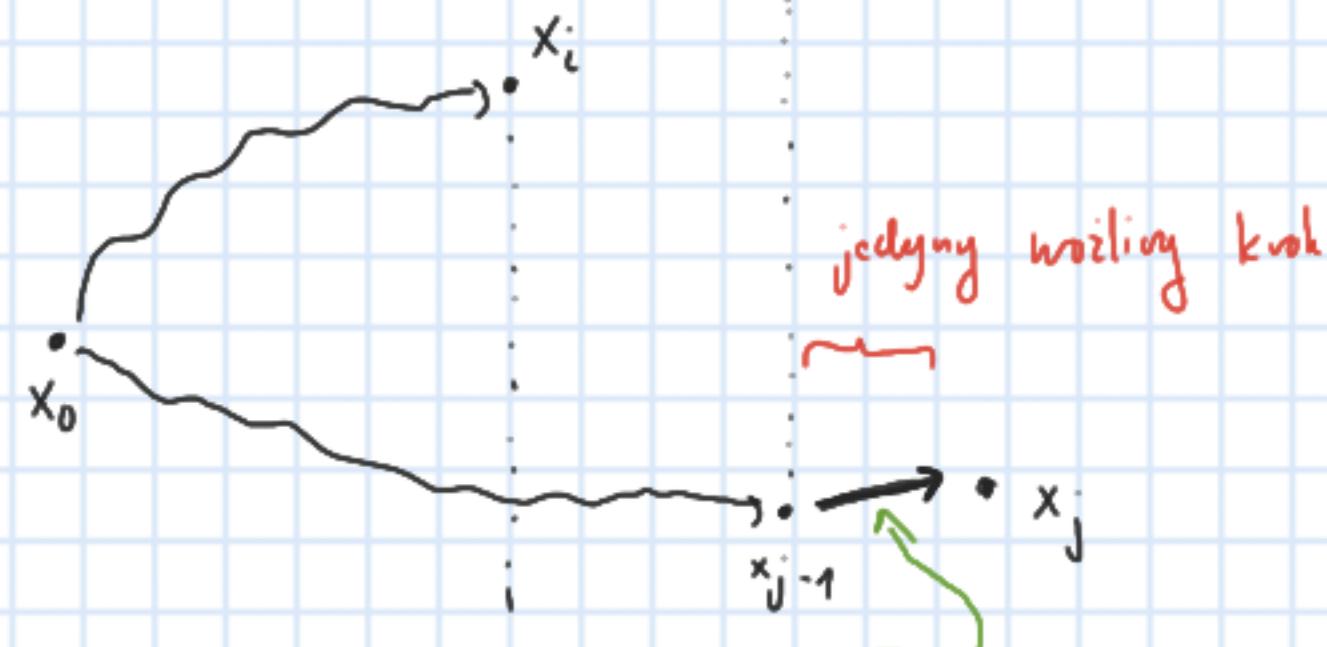


$f(i, j) = \sqrt{\text{kontynuacja z } x_0 \text{ do } x_i \text{ oraz z } x_0 \text{ do } x_j}$ , która wynosi

wszystkich miast  $x_0, \dots, x_j$ , ale żadnego nie pertanają

② Zapis rekurencyjny funkcji f

a)

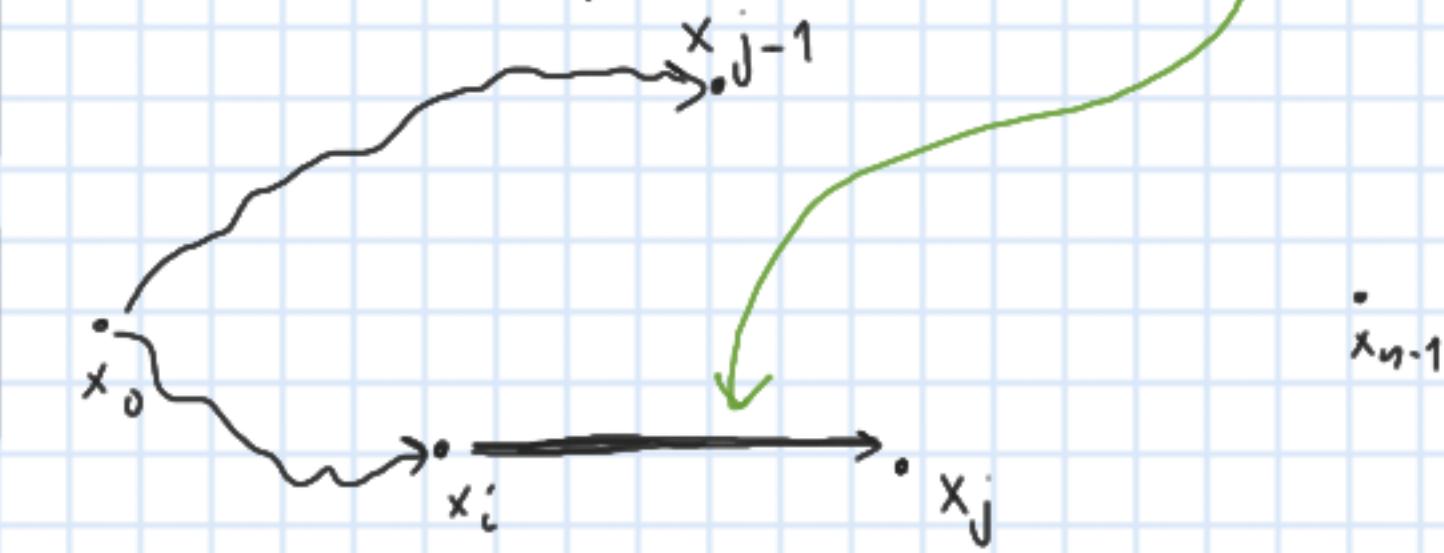


$$f(i, j) = f(i, j-1) + d(x_{j-1}, x_j)$$

$i < j - 1$

$$l) \leftarrow j - 1$$

$$f(j-L, j) = \min_{i < j-1} f(i, j-1) + d(x_i, x_j)$$



## Uzyskanie wyniku

### ③ Implementasi

$$D[i][j] = d(x_i, x_j)$$

$$F = [[\inf] * n \text{ for } i \text{ in range}(n)]$$

$O(n^3)$

```
def tspf(i, j, F, D):
    if F[i][j] != inf: return F[i][j]
    if i == j - 1: // b)
        best = inf
        for k in range(j - 1):
            best = min(best, tspf(k, j - 1, F, D) + D[k][j])
        F[j - 1][j] = best
    else: // a)
        F[i][j] = tspf(i, j - 1, F, D) + D[j - 1][j]
    return F[i][j]
```

# Algorytmy i Struktury Danych

## Układ 12 - Algorytmy zaktanne

Algorytmy zaktanne to takie, które podejmują serie lokalnie "optimalnych" decyzji, licząc że da to globalnicze optimalne rozwiązanie

### Uwagi

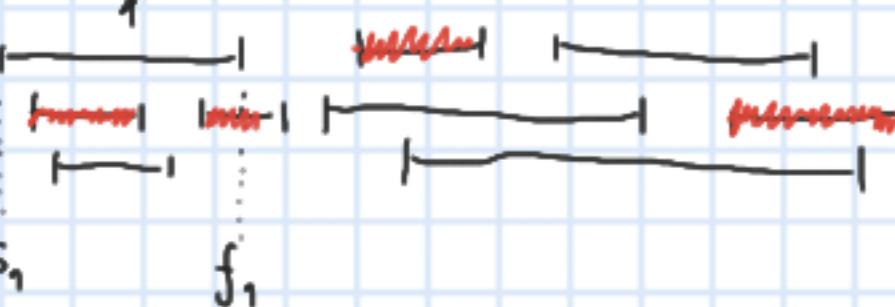
- często nie dają zw. optimalnego

↳ ale czasem tak i na ogólną liczbę szybkie

- często dają rozwiązania przyblżone

### Problem wybranego zadania

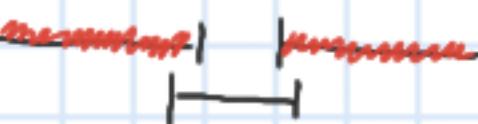
Dane:



Zadanie: Ułożyć jak najwięcej przedziałów, takie że żadny z nich nie pokrywa się

### Algorytm

① Najpierw największy przedział



② Uybieramy przedział, który przecina się z największą liczbą pozostałych



**rozwiązania  
lub dne**

### Poprawny algorytm zaktanney

- posortuj przedziały po punktach końca
- wybieraj od najnowszej końca poprzedniego się

### Dowód poprawności

Zawsze możemy zacząć od najbliższego końca poprzedniego się przedziału

① Uczarny dozwolne rozwiązanie optimalne

② jeśli zaktanuje najbliższego końca się przedział to "jest OK"

③ jeśli nie, to mówimy do niczego dodać najbliższego końca się przedział i usunąć tylko jeden nadrodnego z rozwiązania



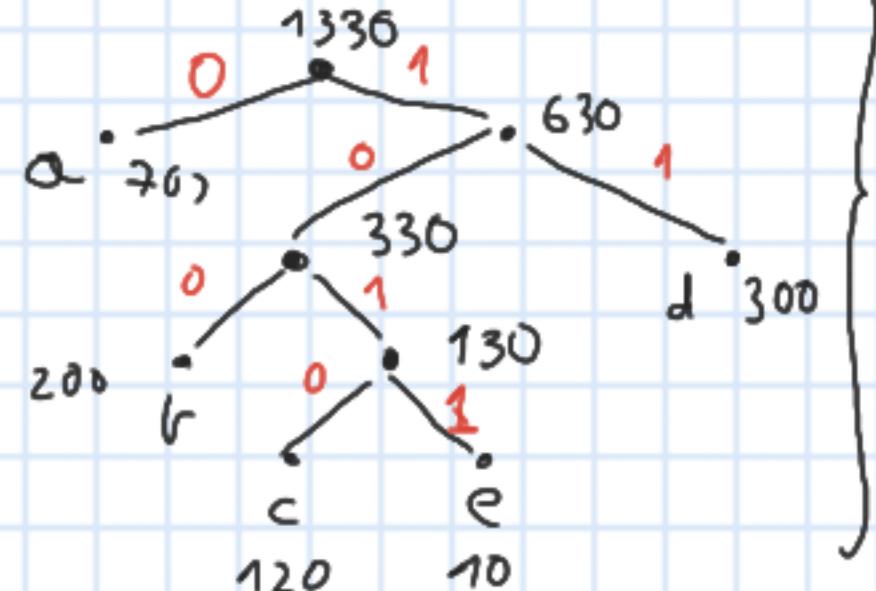
najnowszy koniec się przedział

✓ zw. opt.

Kody Huffman

Symbol:	a	b	c	d	e
Codetoč:	700	200	120	300	10
	000	001	010	011	100

$$\text{długość po : } 1330 \times 3 = 3990 \\ \text{Lekarzanie} \quad \quad \quad 1336$$



700	200	120	300	10
a	b	c	d	R
0	100	1010	11	1011

$$\begin{array}{r}
 \text{Dívizio: } 700 + 3 \cdot 200 + 4 \cdot 120 + 2 \cdot 300 + 4 \cdot 10 \\
 = 700 + 600 + 480 + 600 + 40 \\
 \underbrace{\hspace{1cm}}_{1300} \quad \underbrace{\hspace{1cm}}_{1080} \quad \underbrace{\hspace{1cm}}_{40} \\
 \underbrace{\hspace{3cm}}_{2380} \\
 = 2420
 \end{array}$$

Kodování Huffmana

Dane: alfabet symboli i dla każdego symbolu s, częstotliwość  $f(s)$  jego wystąpienia

Zadanie: Znaleźć drezyna, który mały elementy całkowicie

zakon liczący, dającce kodowana o najmniejszej  
koszcie

$$B(T) = \sum_s f(s) \cdot d_T(s)$$

d<sub>T</sub>(s) kodu s w dnevici T

## Algorytm zaktanym

- vez dva symbole o najmniejszej częstotliwości ( $x, y$ )
  - potem je u nowy symbol (wzetr dwojka) o sumie częstotliwości

$$f(z) = f(x) + f(y)$$

- postanij ten krok dla alfabetu gdzie zastępuje  $x$  i  $y$

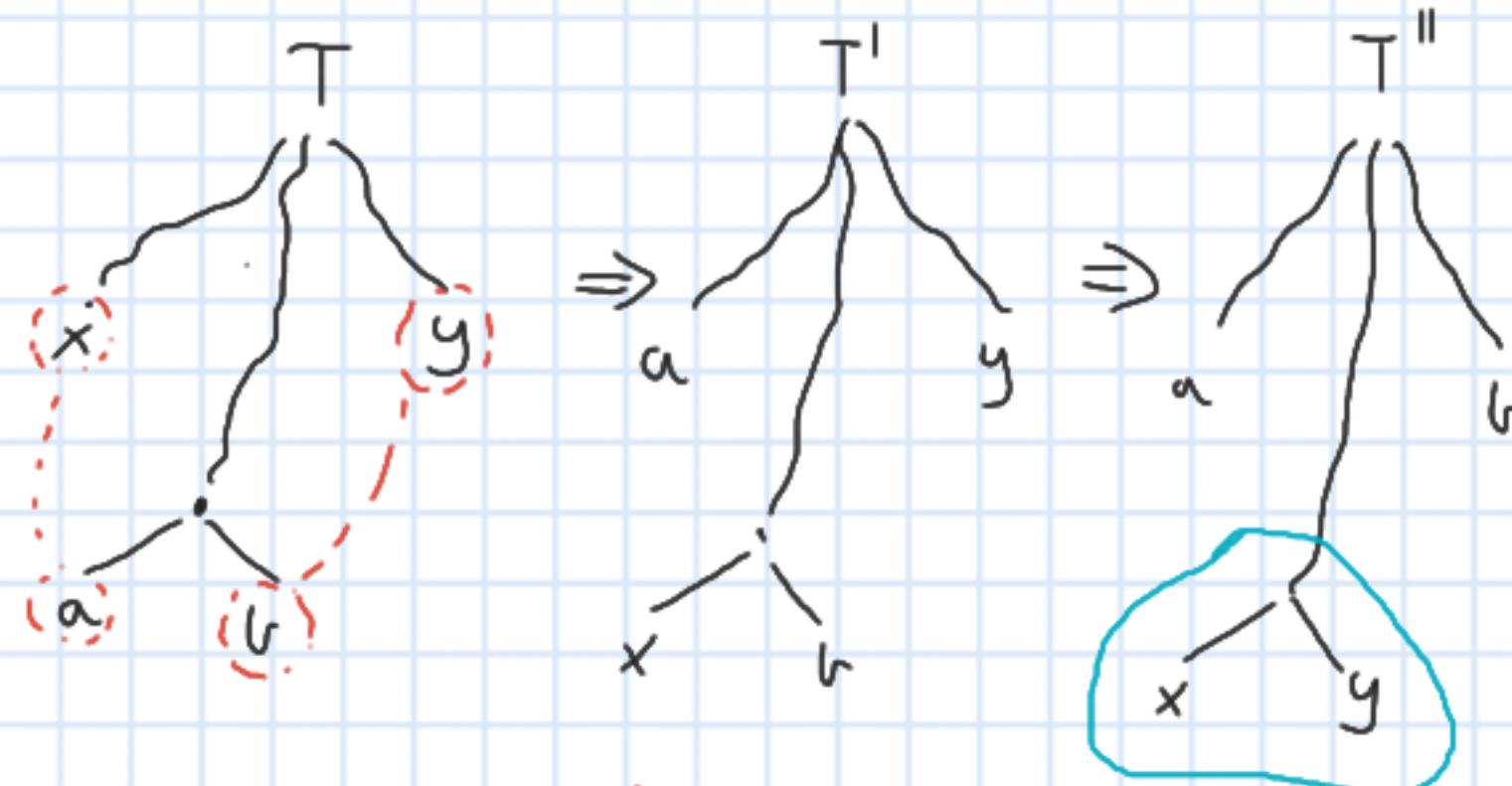
## Dowód poprawności

Krok 1: Dwa najbardziej symboliczne umieszczone w spójnym węźle

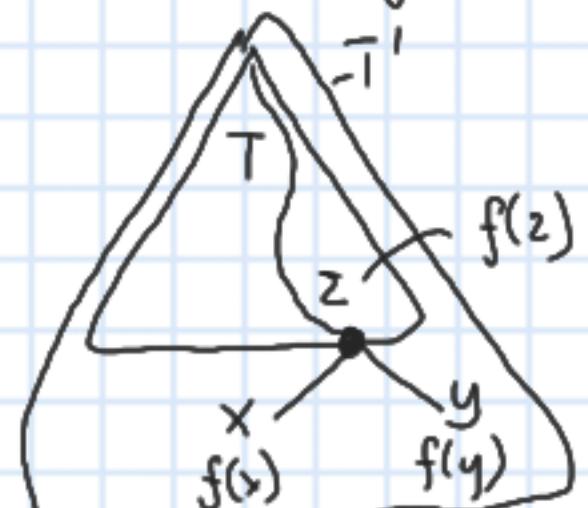
$T$  - opt. drzewo

$a, b$  - sąsiednie symbole na najniższej głębokości w  $T$

$x, y$  - symbole o największej częstotliwości



Krok 2: Optymalna podstruktura



$$f(z) = f(x) + f(y)$$

$$B(T') = B(T) + f(x) + f(y)$$

Dnieso  $T$  dla alfabetu z  
dwoma symbolami  $x$  i  $y$   
musi być optymalne

$$\begin{aligned} B(T') &= B(T) - f(a) d_T(a) \\ &\quad + f(a) d_T(x) \\ &\quad - f(x) d_T(x) \\ &\quad + f(x) d_T(a) \\ &= B(T) + (f(x) - f(a)) d_T(a) - (f(x) - f(a)) d_T(x) \\ &= B(T) + \underbrace{(f(x) - f(a))}_{\leq 0} \underbrace{(d_T(a) - d_T(x))}_{\geq 0} \\ &\leq B(T) \end{aligned}$$

$$B(T'') \leq B(T') \leq B(T)$$

$$\downarrow \\ B(T'') = B(T)$$

# Problem plecakowy w wersji ciągłej

Dane: Ciecie  $c_1, \dots, c_n$ , gdzie dla każdej  $c_i$

masy:  $p(c_i)$  - kost/zysk

$v(c_i)$  - objętość

$L$  - objętość, którą mamy zatrzymać

Zadanie: Ułożyć ile jakaś cięciu zatrzymać, aby maksymalizować zysk

której cięciu mamy uzyskać ilość utamkową, optymalizując odpowiedni utamkowy zysk

## Algorytm

- sortujemy według zysku

- bieremy ciecie o największym zysku, ostatnią ew. utamkową

	$c_1$	$c_2$	$\dots$	$c_{L+1}$
zysk	2	1	$\dots$	1
objętość	L	1	$\dots$	1

Rozważamy algorytmy zachowane dla wersji dyskretnej  
(którym cieciu bieremy w całości lub w ogóle)

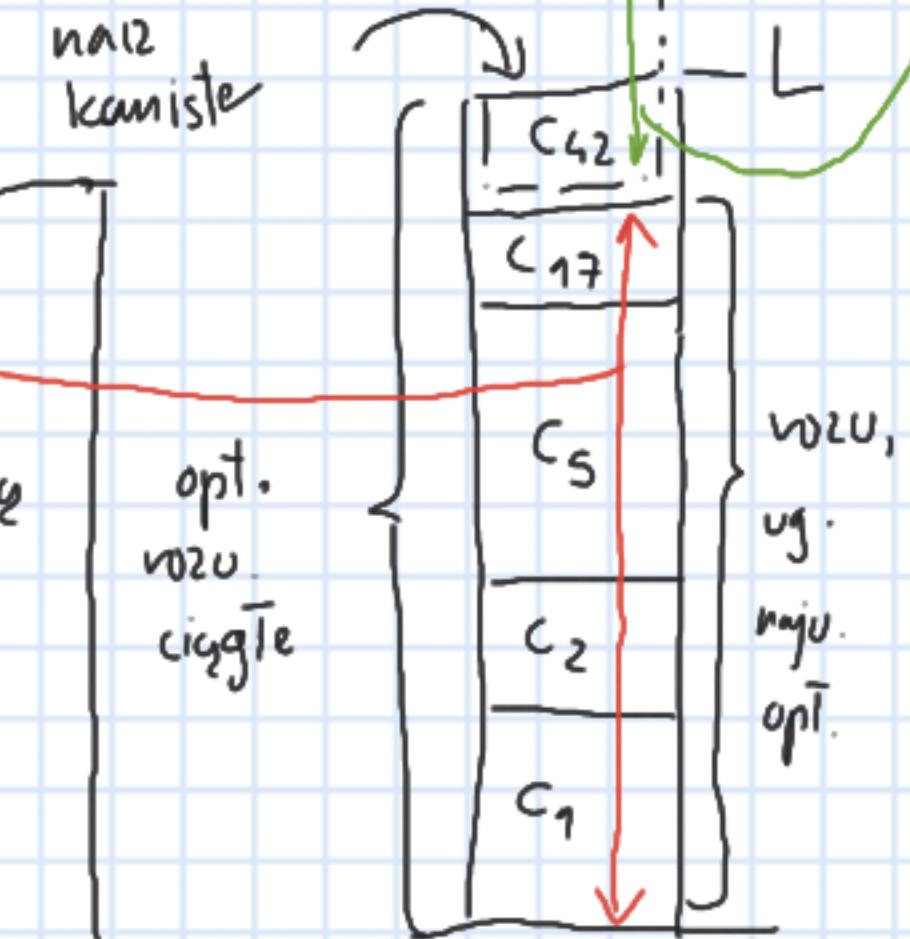
- alg. "największy zysk najpierw" popelnia nieograniony ciąg

- alg. "największa optymalność najpierw" ma ten sam problem

	$c_1$	$c_2$	$\dots$	$c_L$
zysk	2	L-1	$\dots$	1
obj.	2	L	$\dots$	<1

Jesli polujemy rozwiązańa obrotu ogramiennymi i wyliczamy lepsze, to będzie ono najwyżej 2x gorsze od optymalnego

- sortujemy według "optymalności" zysk/objętość
- bieremy najbliższej optymalne ciecie, ostatnią ew. utamkową



# Algorytmy i Struktury Danych

## WykTad 13

Realizacja tablicy anazyjnyjnych

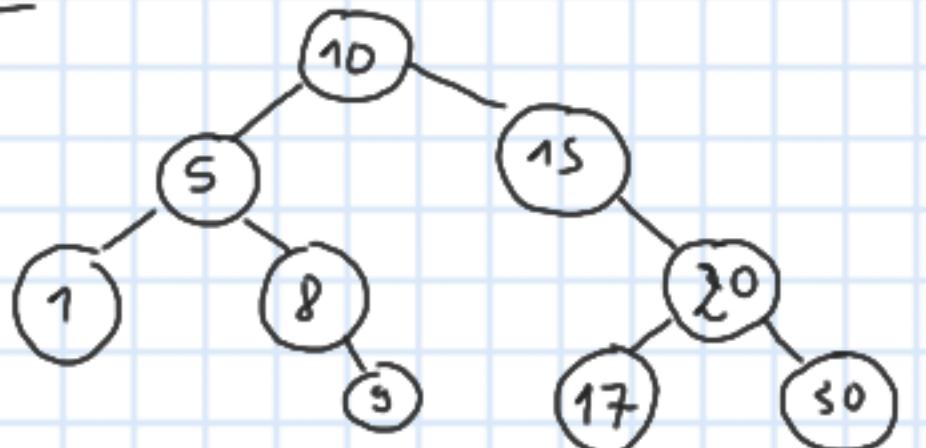
$T["kot"] = 7 \leftarrow \text{wartość}$   
 $\text{key}$

Dane w wyszukiwanie binarnych (BST - binary)



$y < x < z$

## PunkTad



### Operacje na drzewie BST

- utworzenie / usuwanie
- nastepnik / poprzednik
- min / max

```
class BSTNode:  
    def __init__(self, k, v):  
        self.left = None  
        self.right = None  
        self.parent = None  
        self.key = k  
        self.val = v
```

```
def search(node, key):
```

```
    while node != None:
```

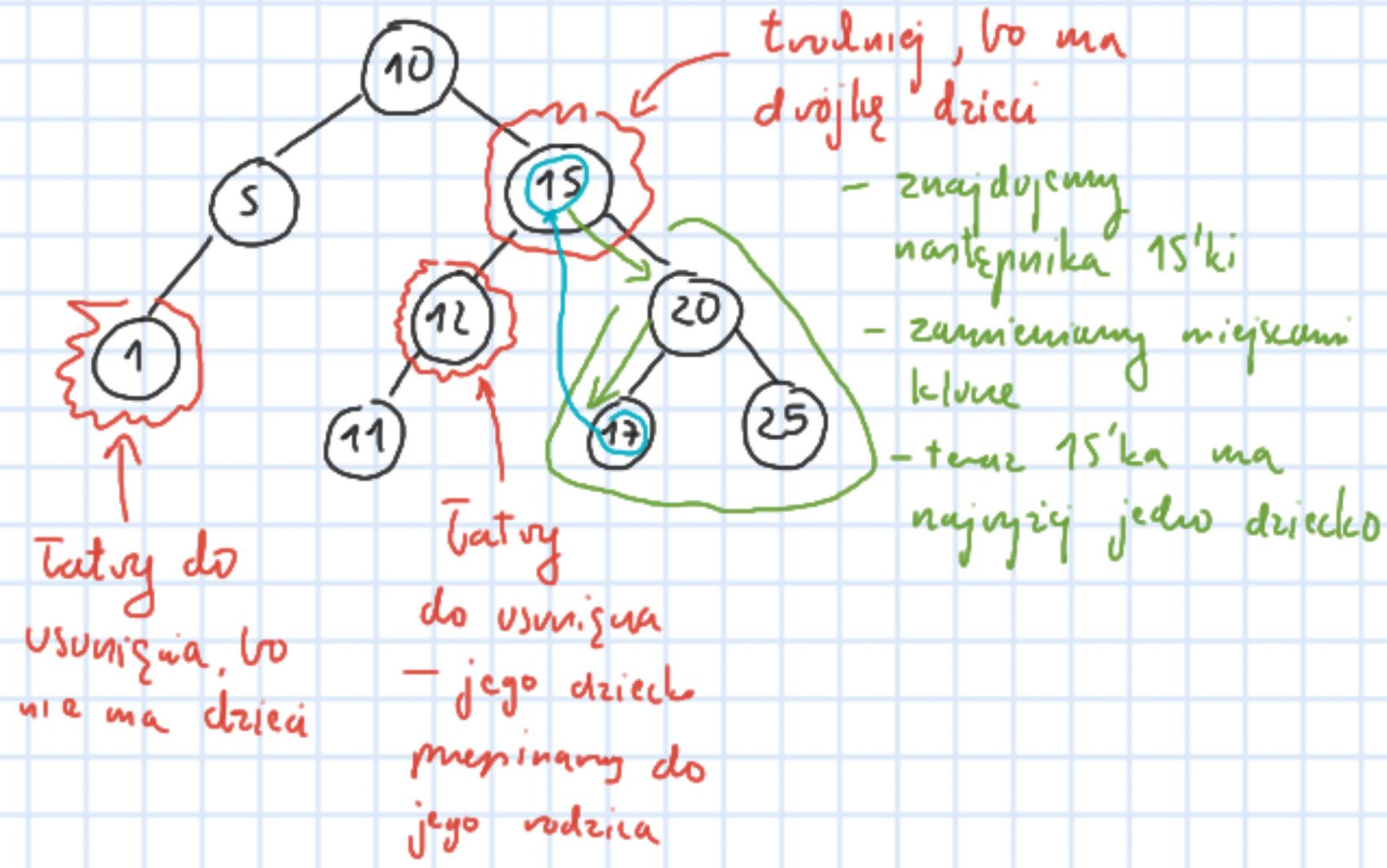
```
        if node.key == key: return node
```

```
        elif node.key < key: node = node.right
```

```
        else: node = node.left
```

```
    return None
```

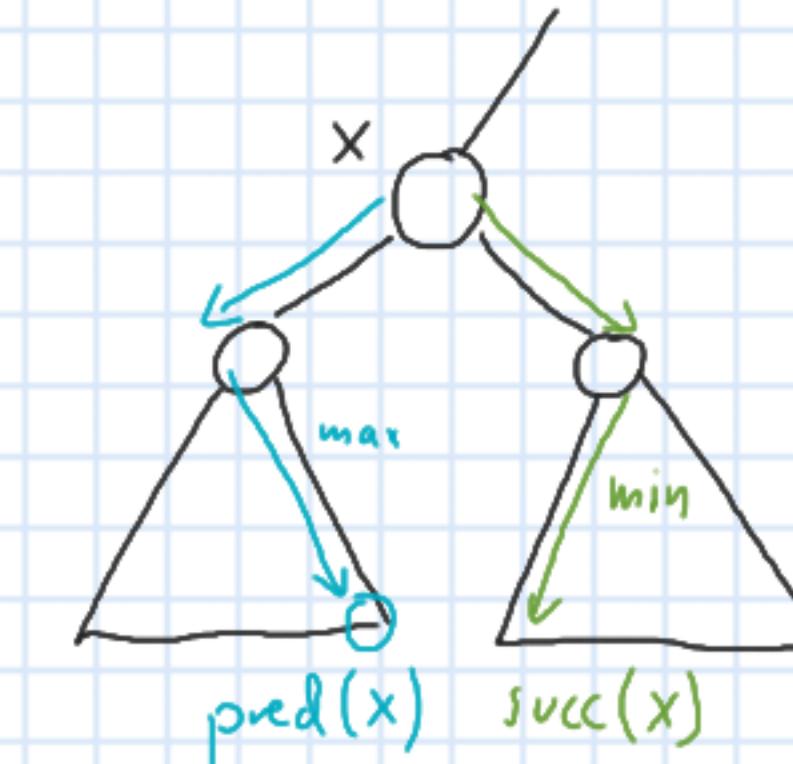
## Usuwanie z drzewa BST



## Wyszukiwanie min/max w drzewie BST

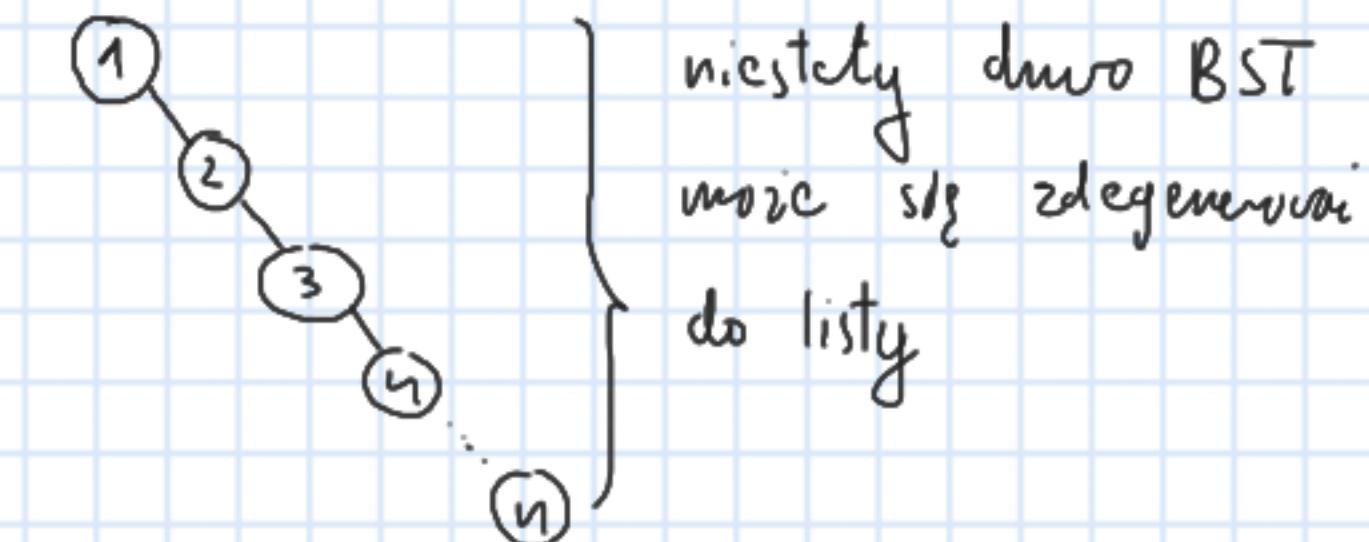


## Wyszukiwanie następnika i poprzednika



## Złożoność operacji na drzewach BST

$O(h)$ , gdzie  $h$  to wysokość drzewa



## Dnera czerwono - czarne

Sz to dnera BST gdzie obniżują

dodatkowe zasady:

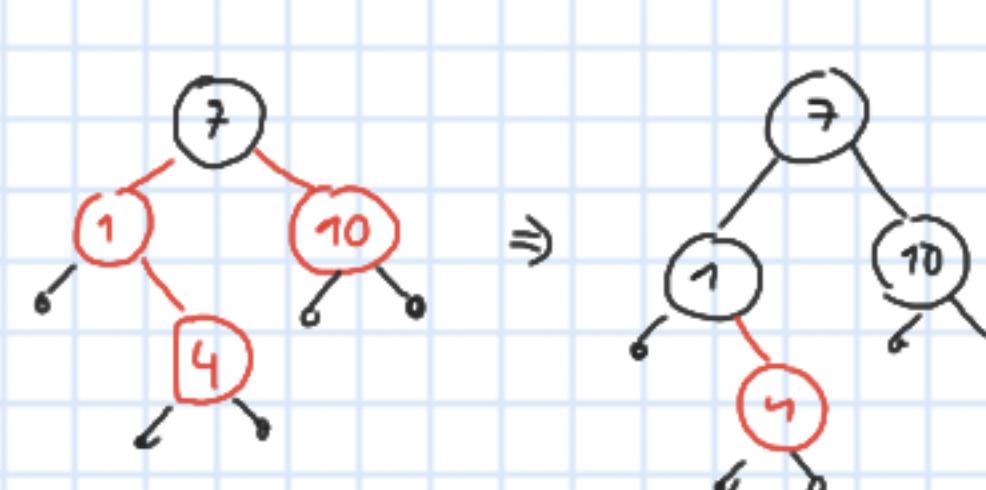
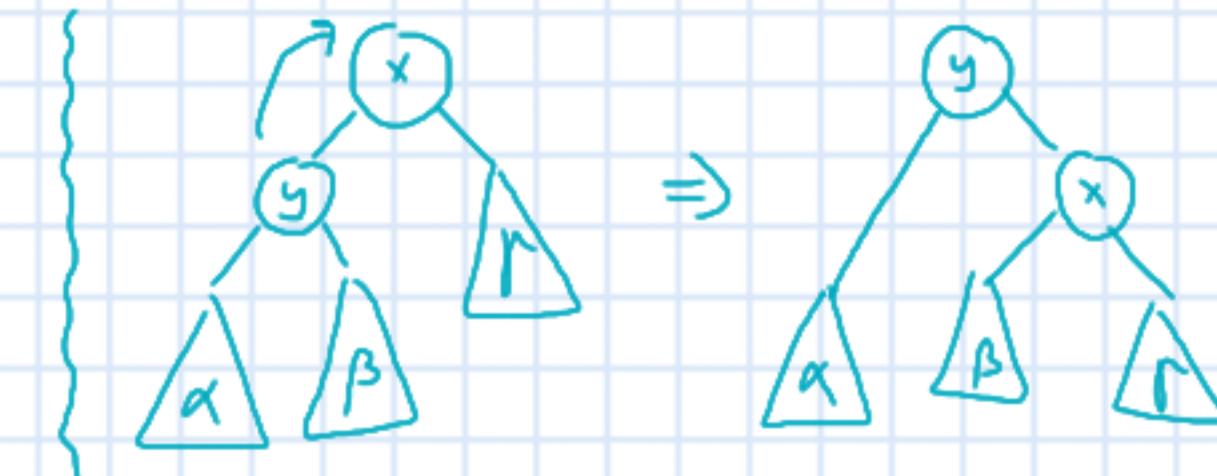
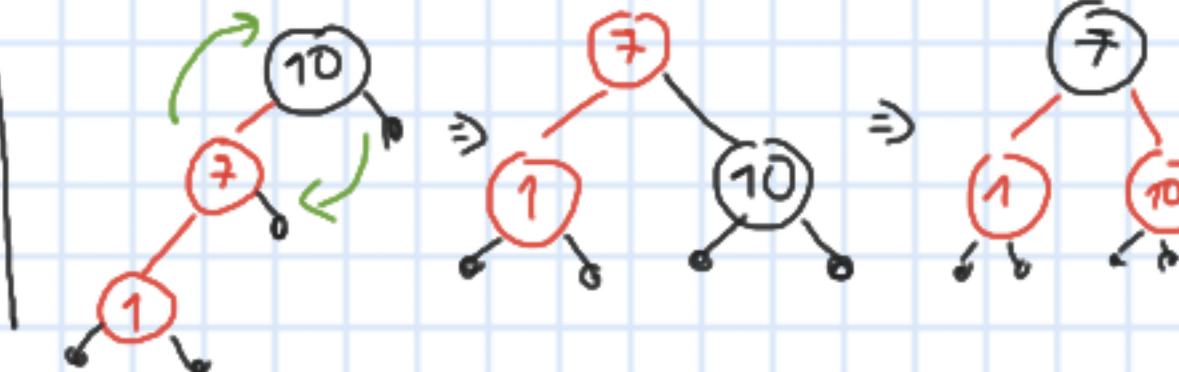
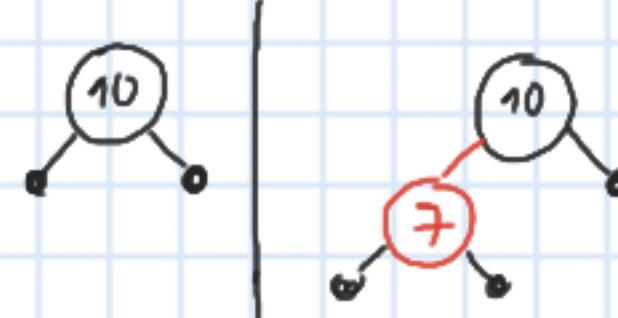
- ① każdy węzeł jest czerwony lub czarny
- ② koniec jest czarny
- ③ każdy liść (None) jest czarny
- ④ jeśli węzeł jest czerwony, to obaj jego synowie są czarni
- ⑤ każda prosta succelka z ustalonego węzła do } !  
liścia ma tyle samo czarnych węzłów

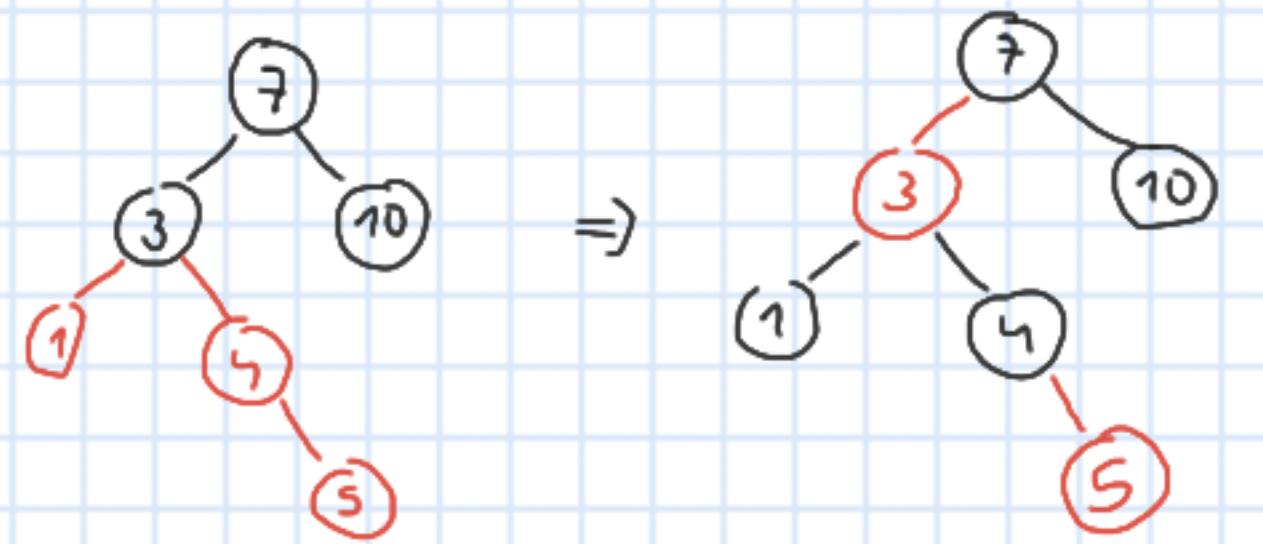
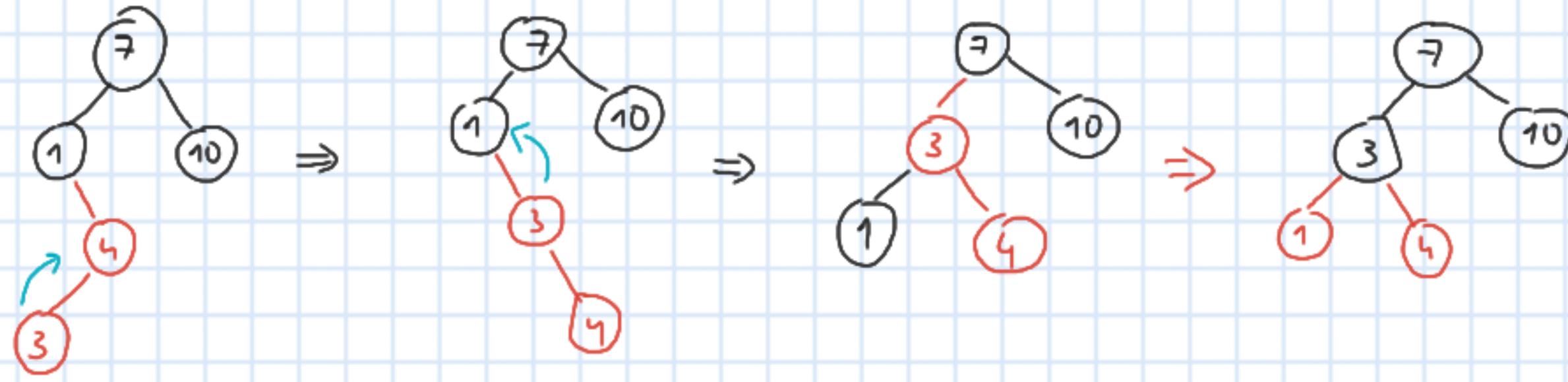
$BH(x)$  - czarna wysokość dnera zakończonego w  $x$

$H(x)$  - zagleba - || - -|| - -|| - w  $x$

$$BH(x) \leq H(x) \leq 2 BH(x)$$

10, 7, 1, 4, 3, 5





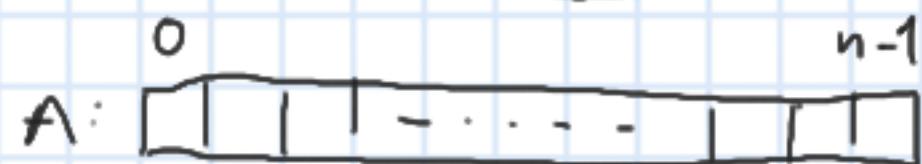
## Tablice haszujce

Counting sort



indeks w tablicy jest wartością diletu

## Idea tablicy haszujcej



indeks to "zahasszona" wartości klucza

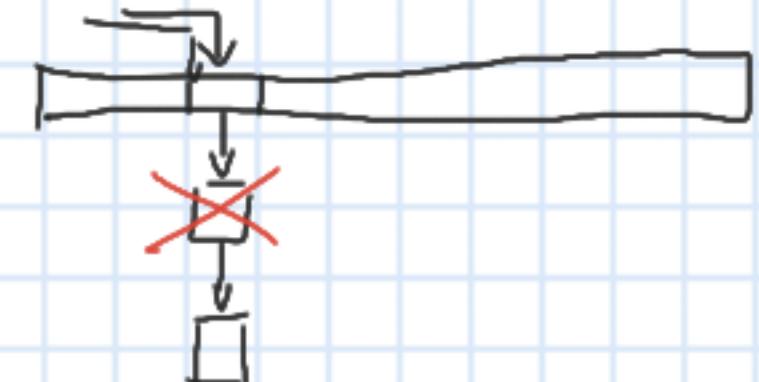
$h: \text{Dane} \rightarrow \mathbb{N}$  - funkcja haszująca

element  $d \in \text{Dane}$  umieszczamy pod indeksem  $h(d, \text{key}) \bmod n$

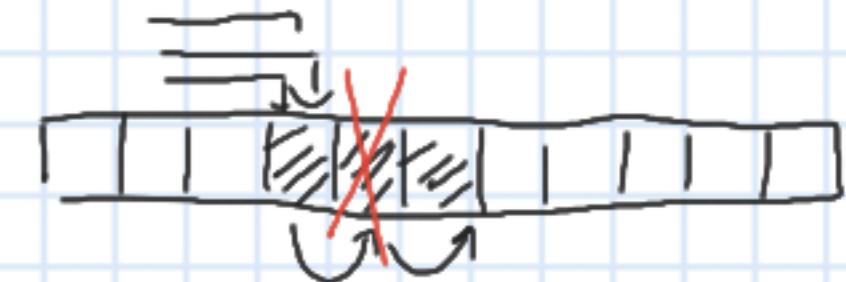
## Rozwiązywanie konfliktów



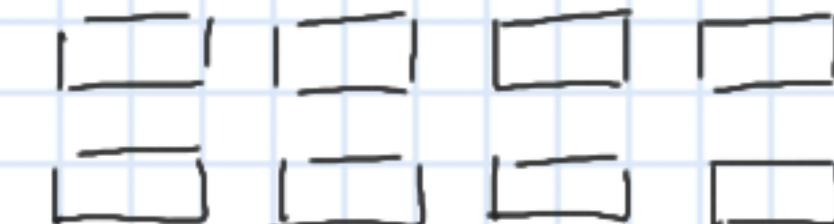
tablica haszująca jest tablica list



adresowanic otwarte



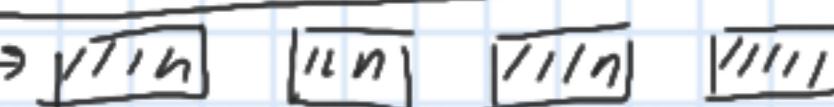
## Jak skonstruować funkcję haszującą



XOR



$k \rightarrow$



Haszująca uniwersalna

$p$ -duża liczba pierwsza

$k \in \{0, \dots, p-1\}$

$$h(k) = ((ak + b) \bmod p)$$

$$a \in \{1, \dots, p-1\}$$

$$b \in \{0, \dots, p-1\}$$

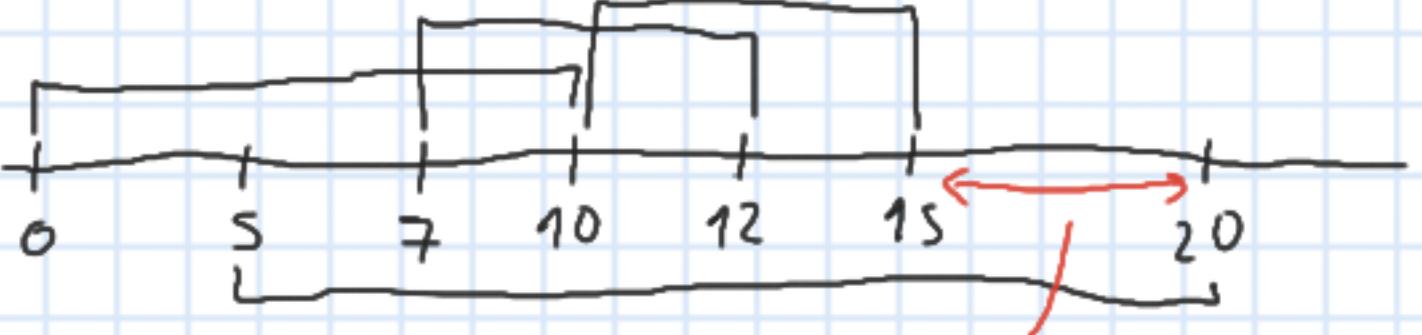
# Algorytm i Struktury Danych

## Wykład 14

Dwie przedziały - interesuje nas struktura danych przedziałów przedziałów i powtarzających się przedziałów zawierających dany punkt

### Przykład

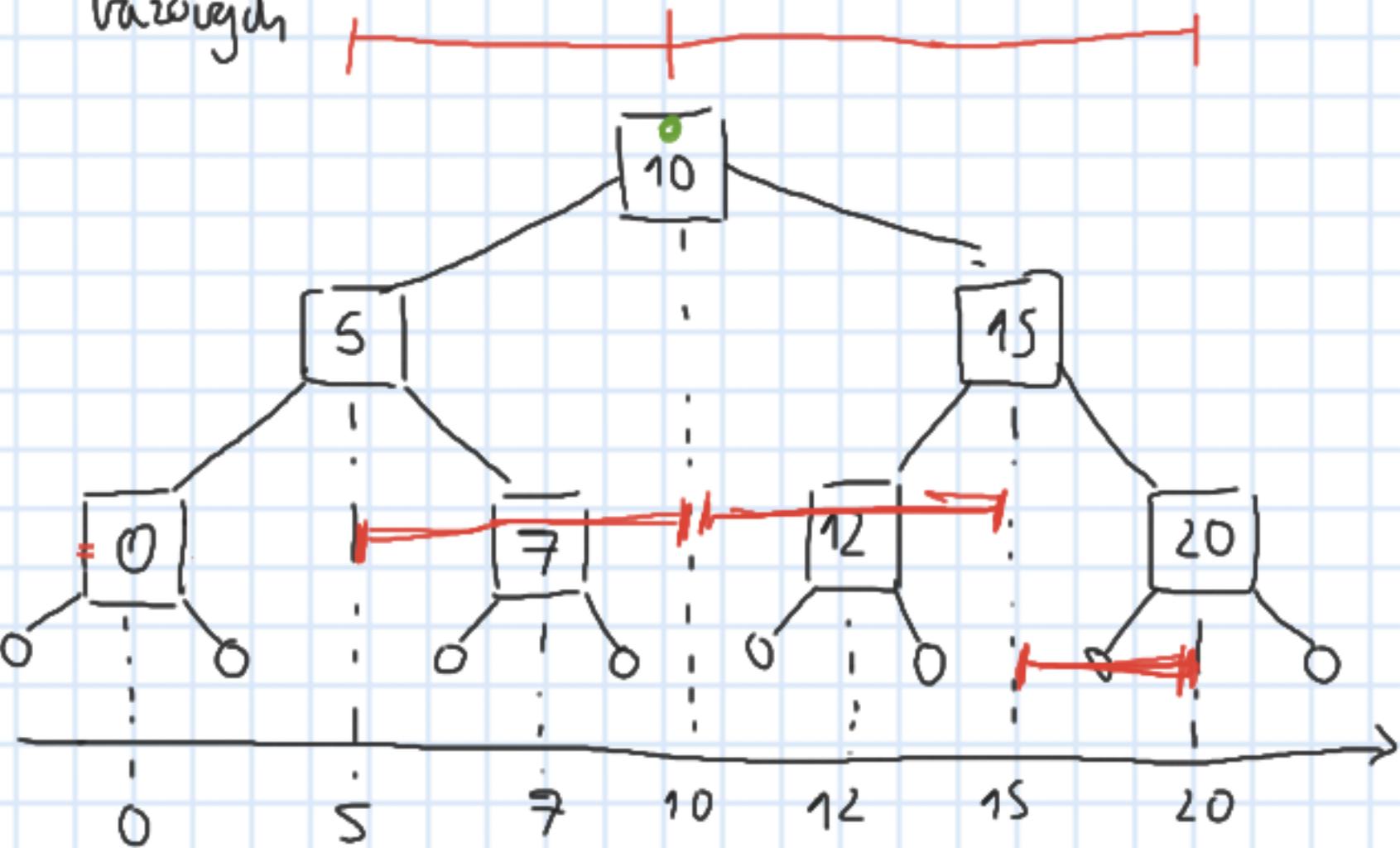
$[0, 10]$ ,  $[5, 20]$ ,  $[7, 12]$ ,  $[10, 15]$



punkt bazowy

Predział przedziwiający w tych węzłach, kiedy  
span całkowicie się w nim zauważa (a nie jest  
to prawa dla rodzica)

Tworzymy dno BST z punktami przedziałów  
bazowych



Kiedy węzeł dnia odpowiada za sumę przedziałów

bazowych u jego poddzieciach

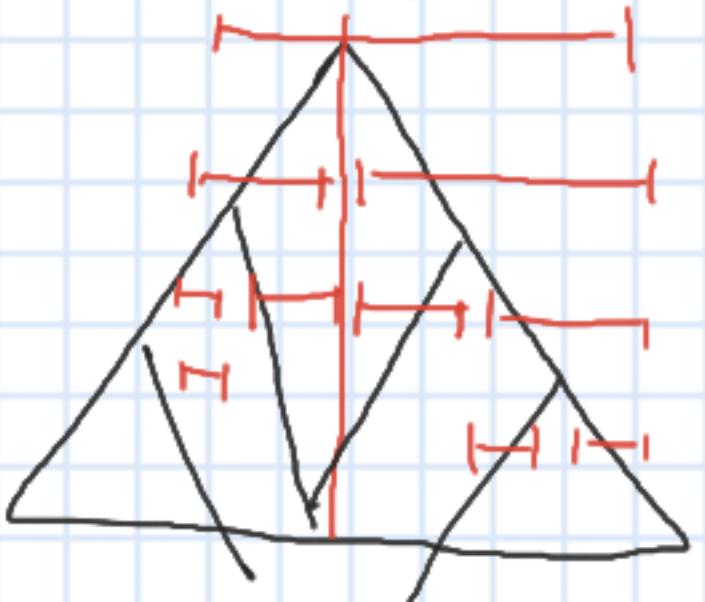
$$\text{span}(5) = [-\infty, 10], \quad \text{span}(12) = [10, 15]$$

## Ustanianie przedziału

Jesli pasuje idealnie u danym węźle, to ustaw

Jesli nie:

- przesuń ze span lewego poddusza ustaw "u lewo"
- przesuń ze span prawego poddusza ustaw "u prawo"



## Sprawdzanie przynależności punktu

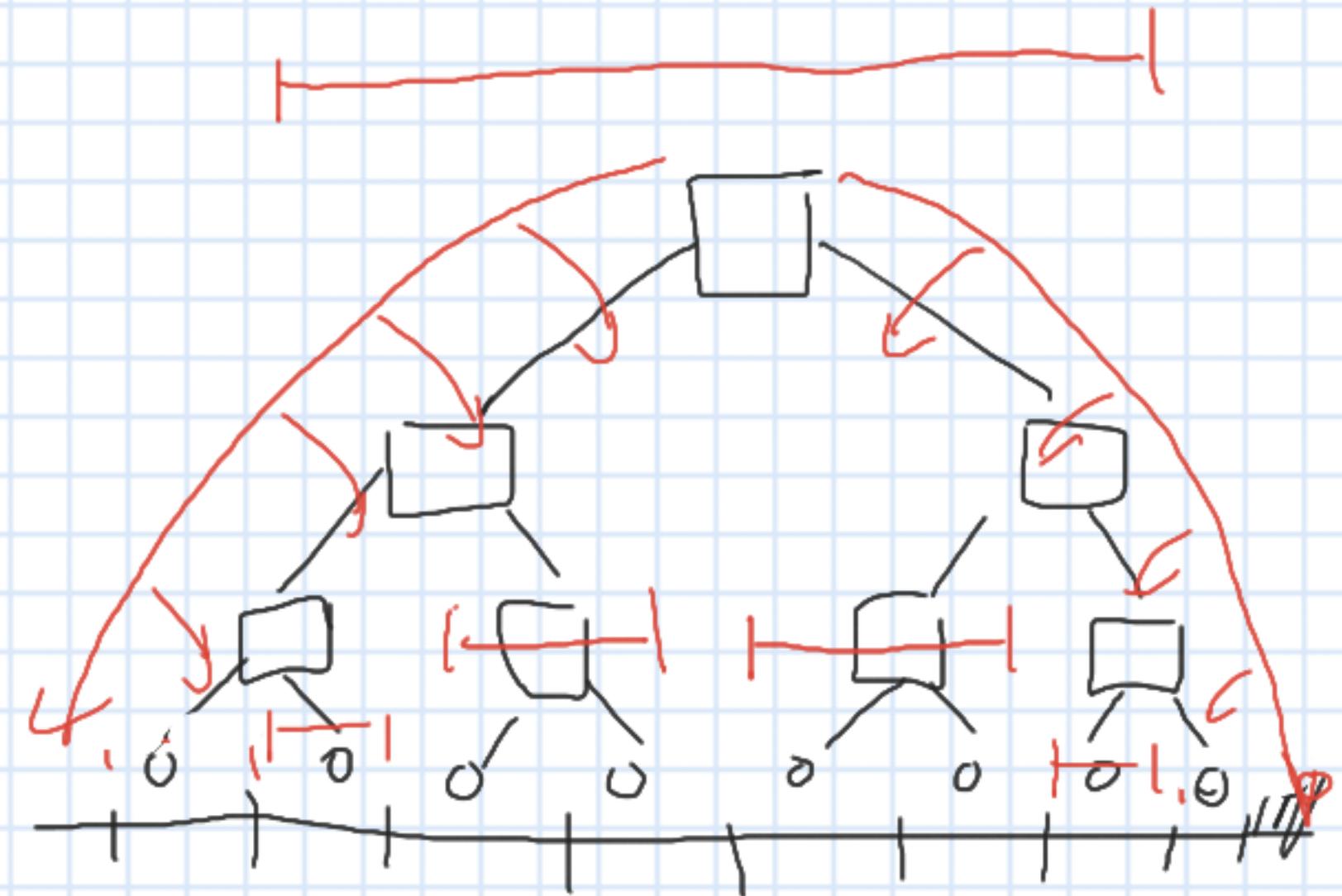
Załączamy od przedziału bazowego, do którego należy dany punkt, węzły w górnym

wypisując spotkane przedziały

## Złożoność pamięciowa

Dla n przedziałów bazowych, sumo  
ducho wymaga  $O(n)$  pamięci.

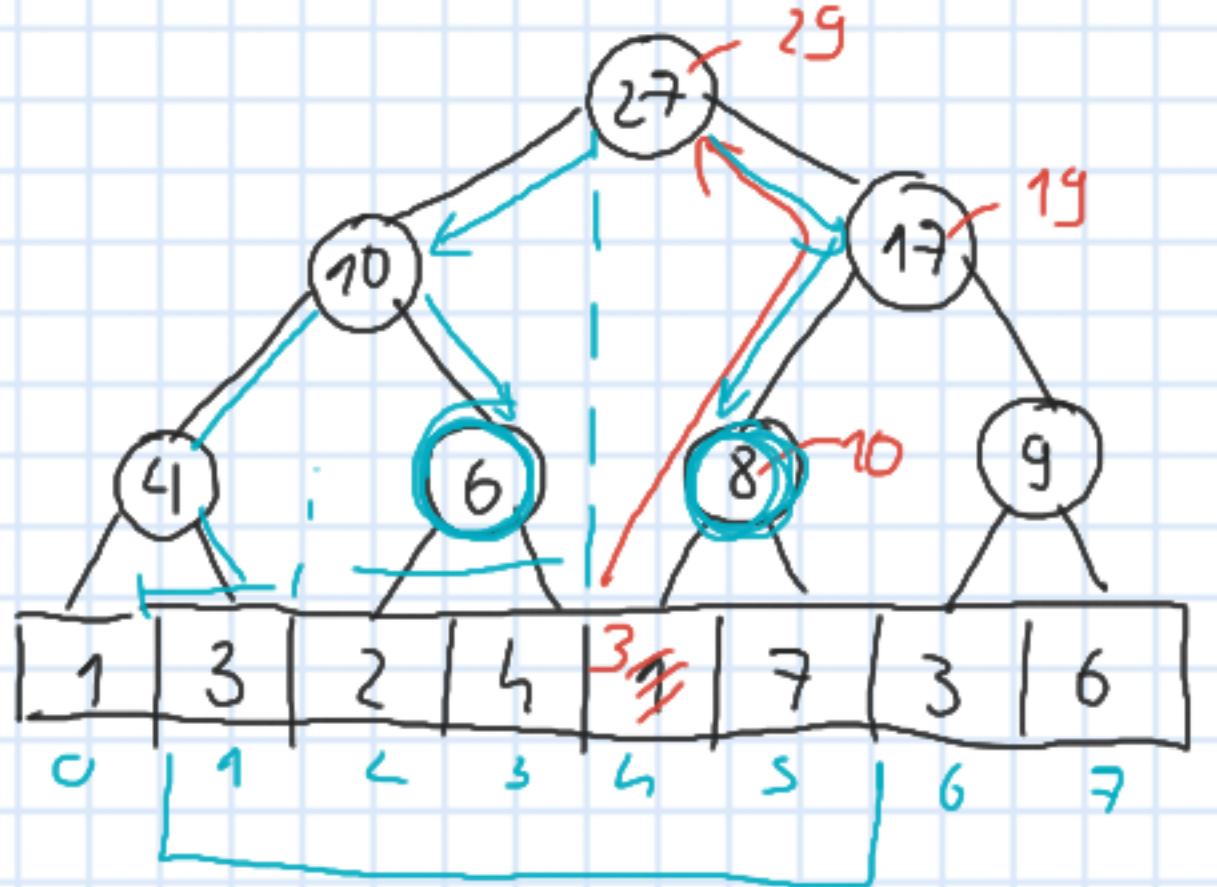
Predostosowanie pojedynczego przedziału wymaga  
 $O(\log n)$  pamięci



Drugi počítač – struktura danych, u

kterej mamy tabulku linkovou operace:

- změnu vlastnosti  $T[i]$
- občas sumy  $T[i] + T[i+1] + \dots + T[j]$



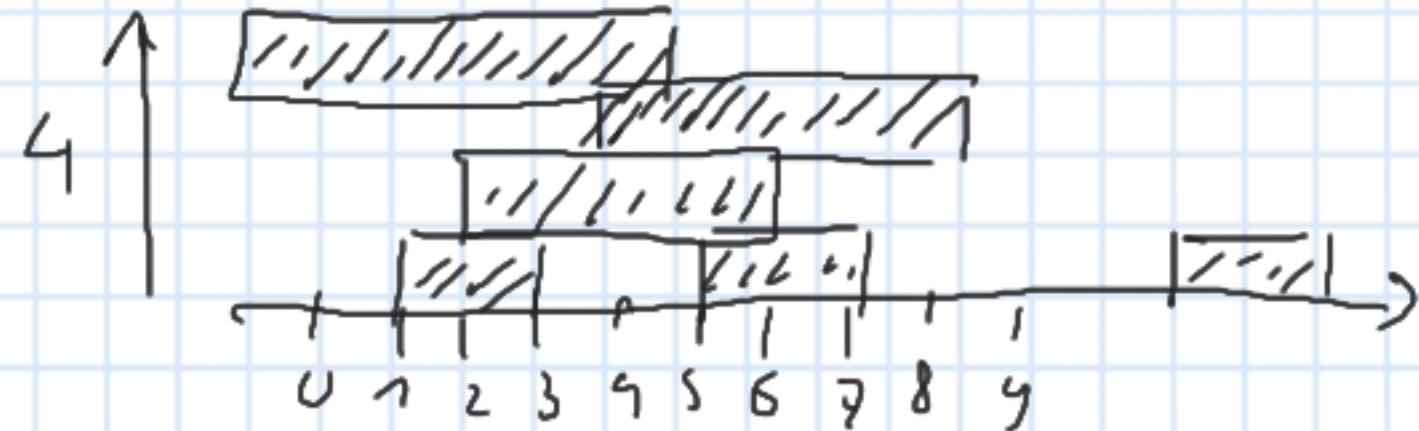
$$6 + 3 + 10 = 19$$

0	1	2	3	4	5	6

$$\text{left}(i) = 2i + 1$$

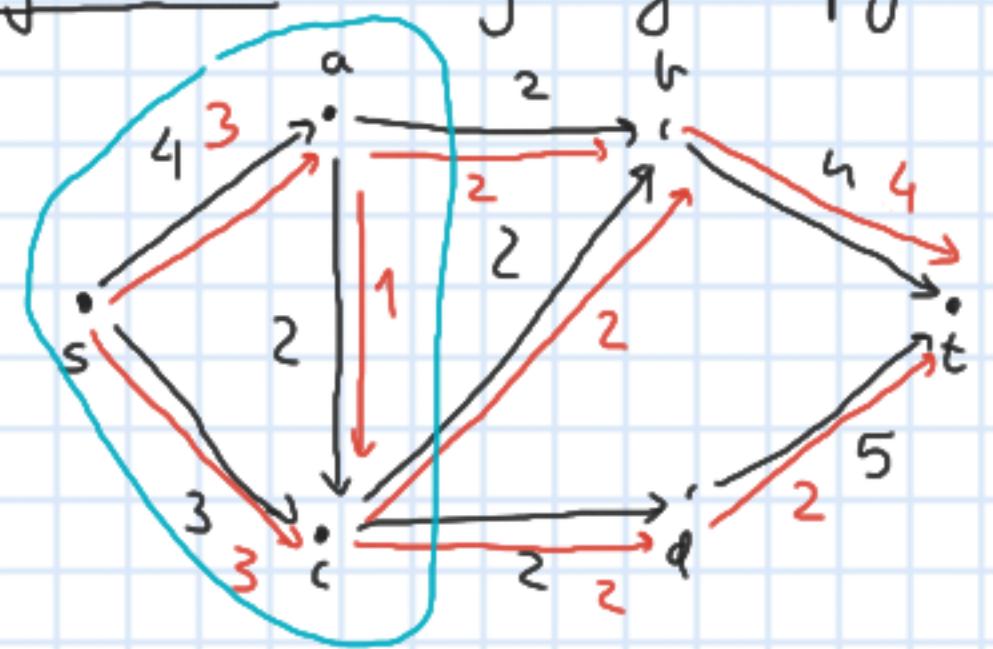
$$\text{right}(i) = 2i + 2$$

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



# Algorytmy i Struktury Danych

## Wykład 15 - Maksymalny przepływ



Ujęcie:  $G = (V, E)$  – graf skierowany  
 $\left\{ \begin{array}{l} (\forall u, v \in V) [\text{istnieje najwyżej jedna sprawa } (u, v), (v, u)] \end{array} \right.$

$$s, t \in V$$

$c: V \times V \rightarrow \mathbb{N}$  ← funkcja pojemności  
 $\left\{ \begin{array}{l} \text{jeli } (u, v) \notin E \text{ to } c(u, v) = 0 \\ c(u, u) = 0 \end{array} \right.$

Przepływ to funkcja  $f: V \times V \rightarrow \mathbb{N}$ , taka że:

$$(\forall u, v \in V) [f(u, v) \leq c(u, v)]$$

$$(\forall v \in V - \{s, t\}) \left[ \sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u) \right]$$

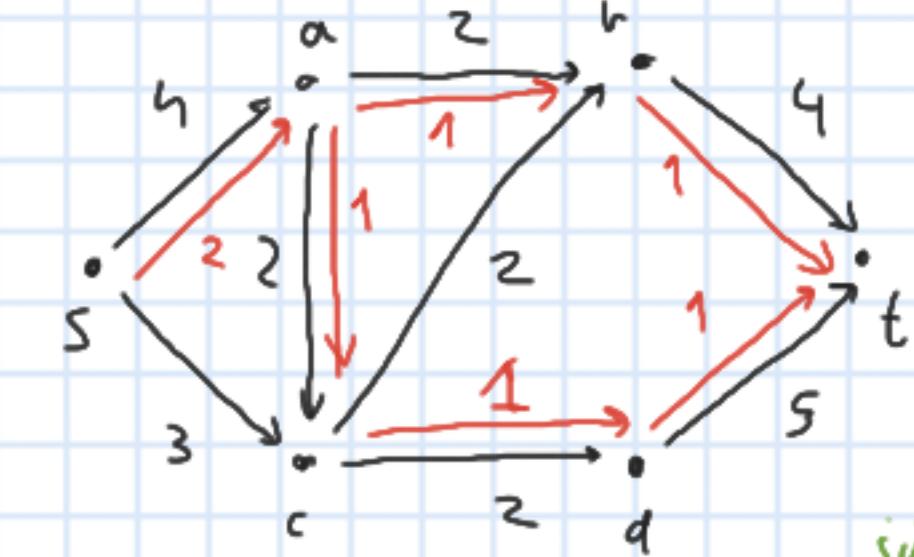
## Wartość przepływu

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Zadanie: Znaleźć przepływ o maksymalnej wartości

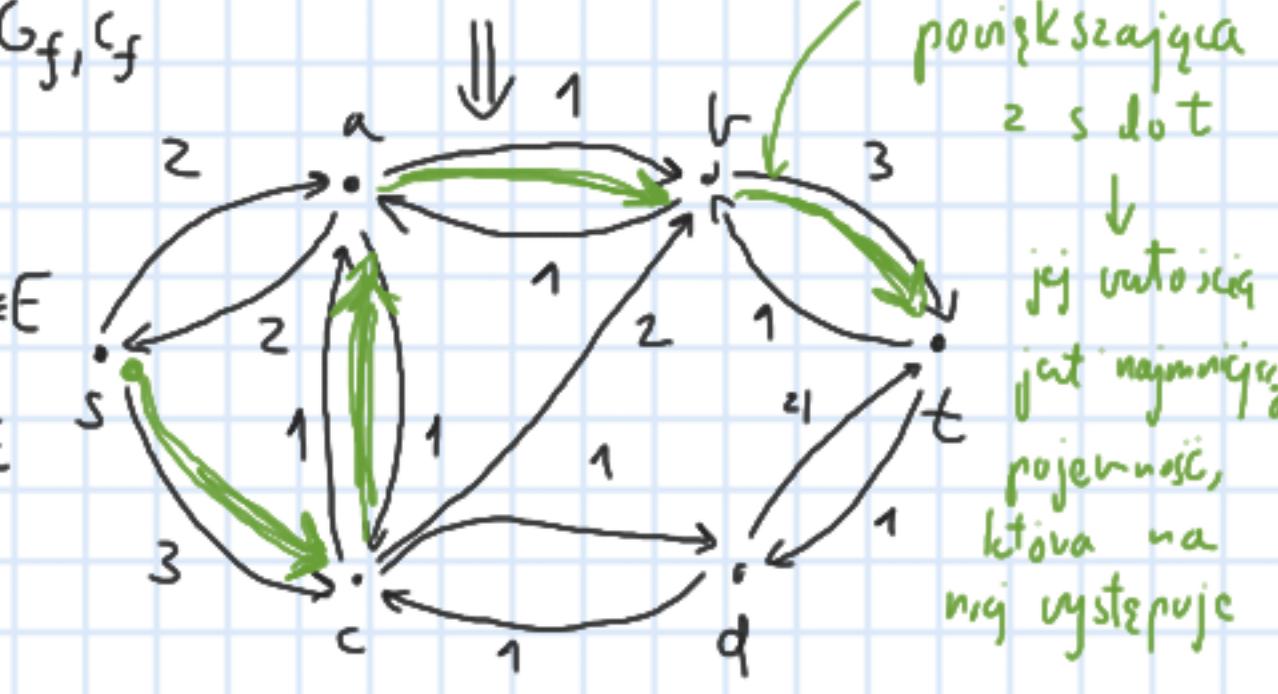
### Sieci residualne

$$\left. \begin{array}{l} G = (V, E), s, t \\ c: V \times V \rightarrow \mathbb{N} \\ f: V \times V \rightarrow \mathbb{N} \end{array} \right\} \begin{array}{l} \text{ujęcie} \\ \text{przepływ} \end{array}$$



Definiujemy sieci residualne  $G_f, c_f$   
 przez funkcje:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v), & (u, v) \in E \\ f(v, u), & (v, u) \in E \\ 0, & \text{up.p.} \end{cases}$$



## Obserwacja

Pnepłyniwa można powiększyć wzdłuż sieci powiększającej

## Metoda Forda - Fulkersona

1. Jeśli istnieje siećka powiększająca dla  $G, f$ , to powiększ  $f$  wzdłuż sieci

2. Powtórzyć krok 1

jak nie ma, to  
 $|f|$  jest maksymalny

## Lemat

Niech  $f$  będzie pnepłynem u sieci  $G$ , z pojemnościem  $c$ , źródłem  $s$ , ujściem  $t$ , a  $(S, T)$  będzie pnękniem.

Wówczas:

$$|f| = f(S, T)$$

dodatkiem 0

## Dowód

$$\forall u \in V - \{s, t\}$$

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0$$

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

$$+ \sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v, u)$$

$$= \sum_{u \in S} \sum_{v \in V} f(u, v) - \sum_{u \in S} \sum_{v \in V} f(v, u)$$

$$= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) +$$

$$\left( \sum_{u \in S} \sum_{v \in S} f(u, v) - \sum_{u \in S} \sum_{v \in S} f(v, u) \right) = 0$$

$$= f(S, T)$$

## Pnęknię u sieci

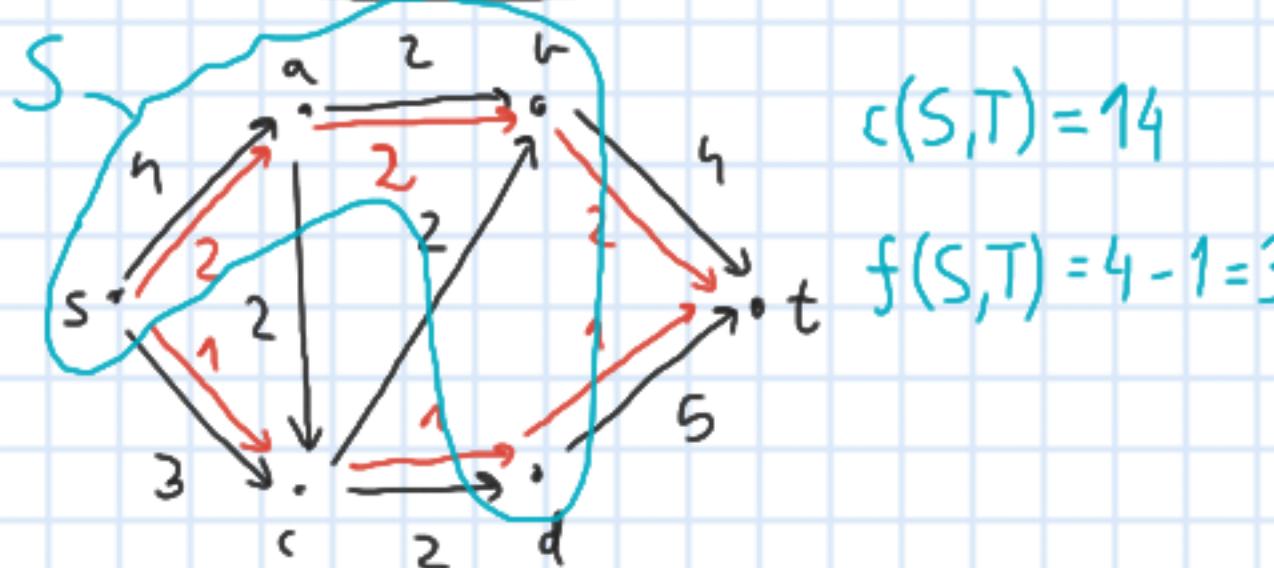
$$\begin{aligned} G &= (V, E), s, t \} \text{ sieci pnepłynowa} \\ c: V \times V &\rightarrow \mathbb{N} \\ f: V \times V &\rightarrow \mathbb{N} \end{aligned}$$

Pnęknię sieci to podzieli  $V$  na

$$S, T = V - S$$

gdzie  $s \in S, t \in T$

$\} S, T$  - zbiory nienekotkowe



## Przepustowość pnékniu

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

## Pnepływu netto

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

## Lemat

Dla każdego podkroju  $S, T$  i przepływu  $f$

zachodzi:  $|f| = f(S, T) \leq c(S, T)$

## Dowód

$$|f| = f(S, T)$$

$$= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

$$\leq \sum_{u \in S} \sum_{v \in T} f(u, v)$$

$$\leq \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T)$$

## Dowód (II, $\textcircled{2} \Rightarrow \textcircled{3}$ )

$$S = \{v \in V \mid \begin{array}{l} \text{istnieje ścieżka z } s \text{ do } v \\ \text{w sieci residualnej } G_f, c_f \end{array}\}$$

$s \in S, t \notin S$  - to innej istnieją

ścieżka powiększająca

$$T = V - S$$

## tu (max-flow/min-cut theorem)

Niech  $G = (V, E)$ ,  $s, t \in V$ ,  $c: V \times V \rightarrow \mathbb{N}$

Wydziel się z sieci przepływu. Niech  $f$  wydziel przepływem w tej sieci. Nasłępujące

warunki są równoważne:

## Dowód (I)

$\textcircled{1} \Rightarrow \textcircled{2}$  - oznacza

$\textcircled{3} \Rightarrow \textcircled{1}$  - oznacza

$\textcircled{1}$   $f$  jest maksymalnym przepływem  
w danej sieci

$\textcircled{2}$   $G_f, c_f$  nie zawiera ścieżek powiększających

$\textcircled{3}$  Dla każdego podkroju  $S, T$  zachodzi  
 $|f| = c(S, T)$

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in T} \left( \underbrace{f(u, v)}_{c(u, v)} - \underbrace{f(v, u)}_0 \right) = \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T)$$

Jeli  $(u, v) \in E$  to  $c_f(u, v) = 0$ , czyli  $f(u, v) = c(u, v)$

Jeli  $(v, u) \in E$  to  $c_f(u, v) = 0$ , czyli  $f(v, u) = 0$

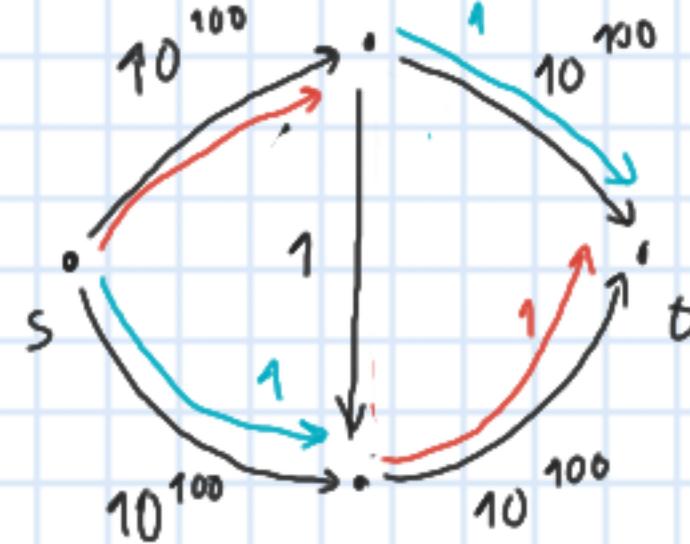
Jeli  $(u, v) \notin E$  oraz  $(v, u) \notin E$ :  $f(u, v) = 0 = c(u, v)$ ,  $f(v, u) = 0$

## Złożoność metody Forda - Fulkersona

$$O((V+E) |f^*|)$$

szukanie pojętynej ścieżki powiększającej  
wartość maksymalnego przepływu

Potencjalnie bardzo wolno!

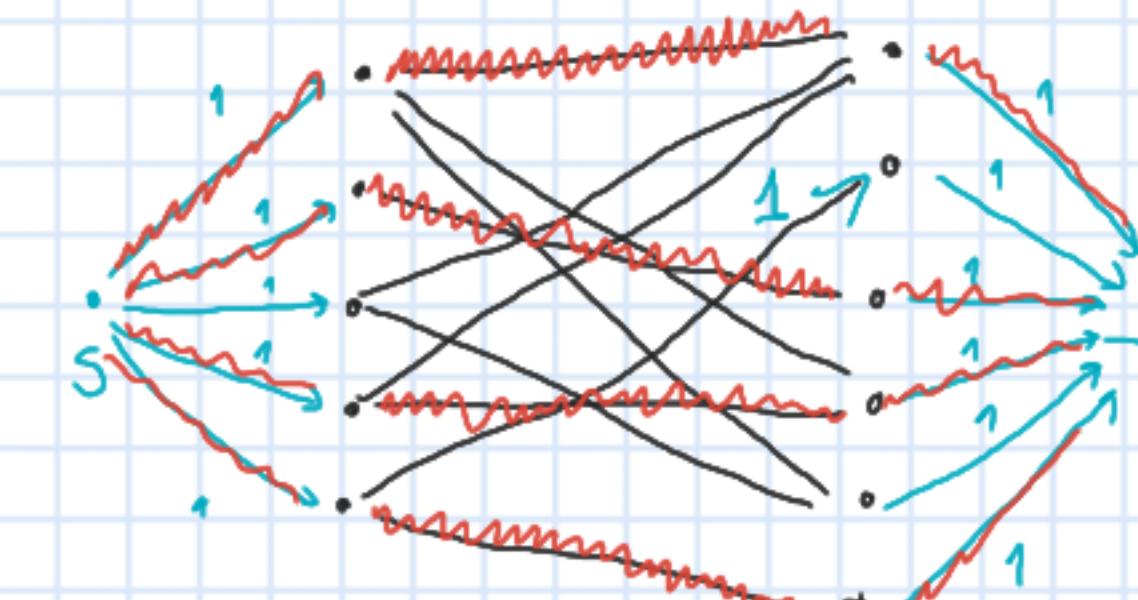


Jeli ścieżkę powiększającą szukam algorytmem BFS

to złożoność wynosi  $O(VE^2)$

{ istnieje algorytm  $O(V^3)$

Pnęty można wykorzystać do znalezienia maksymalnego skojarzenia w grafie diudzielnym



Złożoność

$$O(VE)$$