

# Algorytmy i Struktury Danych

## Wykład 5

Najdłuższy rosnący podciąg

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

Zadanie: Znaleźć długość najdłuższego

(włącznie z pojęciem spójnego) podciągu rosnącego

$A: 13, 7, 21, 42, 8, 2, 44, 53$   
 $f: 1 \quad 1 \quad 2 \quad 3 \quad 2 \quad 1 \quad 4 \quad 5$

Programowanie dynamiczne

① określenie funkcji

② zapis rekurencyjny

③ implementacja

① funkcja

$f(i) = \text{dl. najdłuższego rosnącego}$

dla ciągu  $A[0], \dots, A[i]$

$f(n-1)$

wynik

$f(i) = \text{dl. najdłuższego rosnącego podciągu kończącego się na } A[i]$

② sformułowanie rekurencyjne

$f(-i) = 0$

$f(0) = 1$

$A: 1, 2, 4, 2, 3$   
 $f: 1 \quad 2 \quad 3 \quad 3 \quad ? \quad 3$

$$f(i) = \max_{t > 0} \{ f(t) + 1 \mid A[t] < A[i] \}$$

$A': 1, 2, 4, 0, 1, 2, 3$   
 $f: 1 \quad 2 \quad 3 \quad 3 \quad 3 \quad 3 \quad ? \quad 4$

```

def lis(A):
    n = len(A)
    F = [1] * n
    P = [-1] * n
    for i in range(1, n):
        for j in range(i):
            if A[j] < A[i] and F[j]+1 > F[i]:
                F[i] = F[j]+1
                P[i] = j
    return (max(F), F, P)

```

$\mathcal{O}(n^2)$

$\mathcal{O}(n \log n)$

$A: 13, 7, 21, 42, 8, 2, 14, 53$ $F: 1, 1, 2, 3, 2, 1, 4, 5$ $P: -1, -1, 1, 2, 1, 1, 3, 6$	
--	---

```

def printSolution(A, P, i):
    if P[i] != -1:
        printSolution(A, P, P[i])
    print(A[:i])

```

$index\_max = \max(\text{range}(\text{len}(A)), \underline{\text{key}} = A.\_.\_getitem\_\_(-))$

$\text{print}(A[:]) \Leftrightarrow \text{print}(A.\_.\_getitem\_\_(-)(i))$

# Problem plecakowy (dyskretny)

Dane: n przedmiotów

$w[i]$  - waga  $i$ -go przedmiotu

$P[i]$  - zysk z  $i$ -go przedmiotu (profit)

$\text{Max}W$  - dopuszczalna waga plecaka

Zadanie: Wybrać przedmioty o jak największym sumarycznym zysku, nie przekraczając ograniczenia wagi  $\text{Max}W$

Przykład

	0	1	2	3	4	5	
P	10	8	4	5	3	7	26
w	4	5	12	9	1	13	19
$\text{Max}W$	24						

## ① funkcja

$f(i, w) = \text{największy zysk jaki można osiągnąć wybierając spośród przedmiotów od } 0 \text{ do } i \text{ nie przekraczając wagi } w$

## ② zależność rekurencyjna

$$f(0, w) = \begin{cases} 0 & , w[0] > w \\ P[0] & , w[0] \leq w \end{cases}$$

$$f(i, 0) = 0 // \text{przedmioty nie mają wagi } 0$$

$$f(i, w) = \max \left\{ f(i-1, w), f(i-1, w-w[i]) + P[i] \right\}$$

zakładam, że  $w \geq w[i]$

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

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

```
F = [None] * n
```

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

```
F[i] = [0] * (MaxW + 1)
```

```
for w in range(W[0], MaxW + 1):
```

```
F[0][w] = P[0]
```

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

```
for w in range(1, MaxW + 1):
```

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

```
if w ≥ W[i]:
```

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

```
return F[n-1][MaxW], F
```

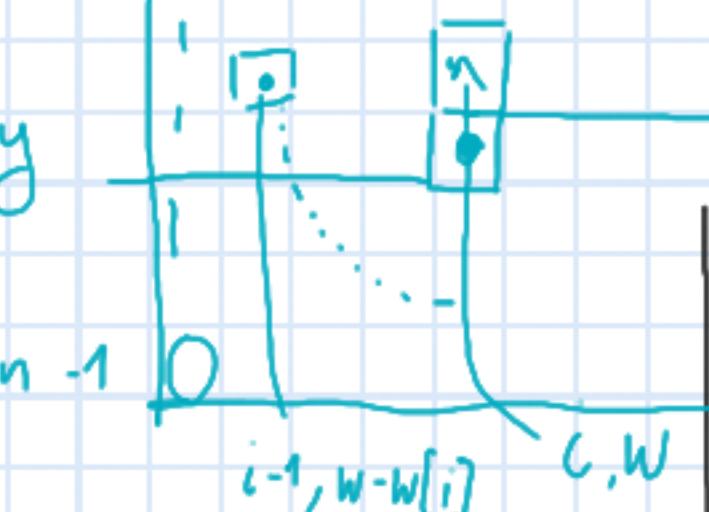
$$F[i][w] = f(i, w)$$

$$F = [[0] * (MaxW + 1) \text{ for } i \text{ in range } n]$$

vogl:

	0	W[0]	MaxW
0	0 0 0 0	P[0] - - - P[0]	
1	0		

nedmity



```
def getSolution(F, W, P, i, w):
```

```
f i < 0: return []
```

```
if i == 0:
```

```
if w ≥ W[0]: return [0]
```

```
return []
```

+P[]

```
if w ≥ W[i] and F[i][w] == F[i-1][w-W[i]]:
```

```
return getSolution(F, W, P, i-1, w-W[i]),
```

```
return getSolution(F, W, P, i-1, w) + [i]
```

## Zadanie obowiązkowe

Proszę przedstawić algorytm dla problemu

Knapsack działający w czasie

$$O(n \cdot \left( \sum_{i=0}^{n-1} P[i] \right))$$

## Zadanie obowiązkowe 2

W problemie sumy podzielnego mamy

dany ciąg liczb  $A[0], \dots, A[n-1]$  oraz

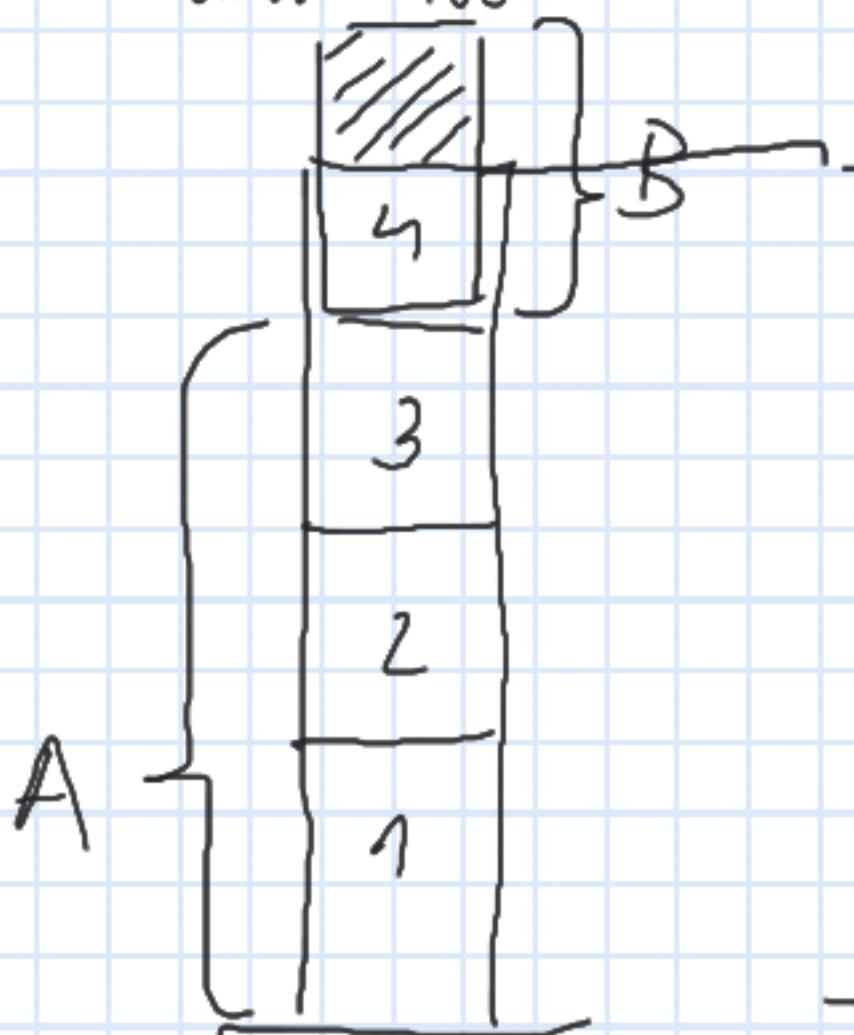
liczba  $T$ . Należy stwierdzić czy istnieje

rodzaj sumującą się dokładnie do  $T$

naturzysko

$$\max \left( \begin{array}{c} \begin{array}{c|cc} P & 2 & 100000 \\ \hline W & 1 & 100000 \end{array} \\ \text{MaxW} = 100\ 000 \end{array} \right) \geq \frac{1}{2} \bar{OPT}$$

$$\begin{array}{c} \begin{array}{c|ccccc} P & 2 & 1 & 1 & 1 & \dots & 1 \\ \hline W & 100 & 1 & 1 & 1 & & 1 \end{array} \\ \text{MaxW} = 100 \end{array}$$



$$\bar{OPT}_{\text{ciggle}} \geq \bar{OPT}$$

$$\underline{A + B} \geq \underline{OPT}_{\text{ciggle}}$$

# Algorytmy i Struktury Danych

## Wykład 6

Problem komiwojażera

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

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

Zadanie: Znaleźć taką kolejność odwiedzania miast, że startując w 0, każde miasto odwiedzamy jednokrotnie, kończąc w mieście 0: suma

przelocionych odległości jest minimalna

nic mniejsze 0,  
u którego koniec,

Złożoność

$$O(2^n \cdot n^2)$$

## Algorytmy

① Próbujemy wszystkich możliwości  
 $O^*(n!)$

② Algorytm dynamiczny

$f(X, i)$  = koszt najkrótszej ścieżki startującej w mieście 0, przebiegającej wszystkie miasta z  $X$ ; nie więcej, i kończącej w  $i$   
zbiór miast  
zarównież 0  
oraz  $i$

Rozwiązywanie

$$\min_{i \neq 0} f(C, i) + d(i, 0)$$

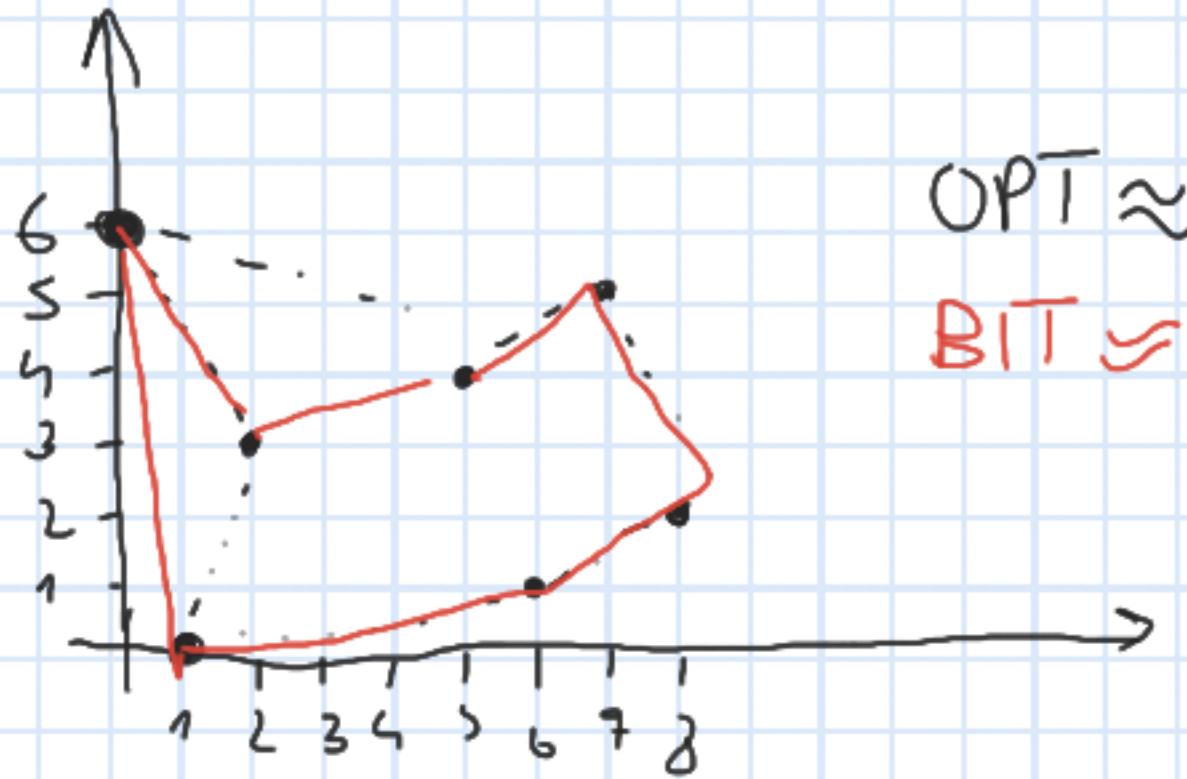
Rekurencja

$$f(X, i) = \min_{\substack{j \neq i \\ j \in X - \{i\}}} [f(X - \{i\}, j) + d(j, i)]$$

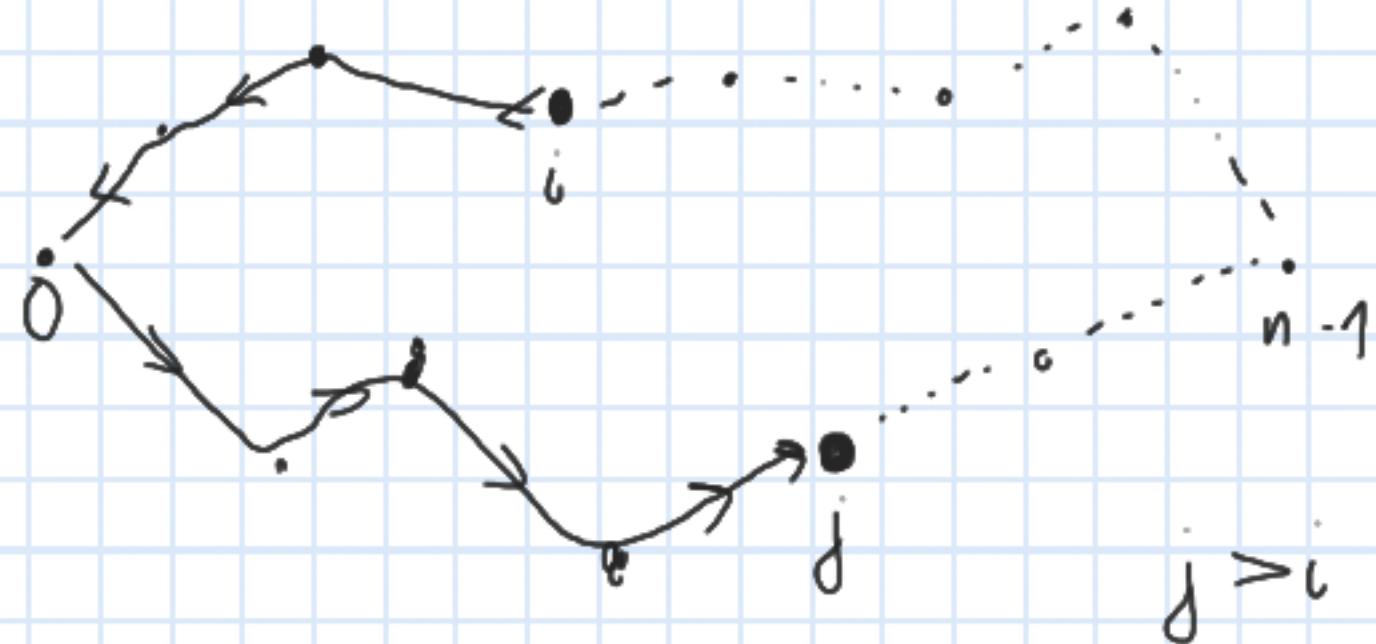
$$f(X - \{i\}, j) + d(j, i)$$

## Bitoniczny problem komiujazera

Miasta są w  $\mathbb{R}^2$  i stosujemy std. metrykę euklidesową. Wszystkie uspójne miasta są różne. Na tą samą trasę uspójne  $n$  miast najpierw rosną a potem maleją.



① funkcja, którą będziemy obliczać



$f(i, j) = \text{minimalny koszt ścieżki :}$   
 $0 < i < j$

takich, że tą samą trasę ścieżki odwiedzają każde miasto ze zbioru  $\{1, \dots, j\}$   
dokładnie raz

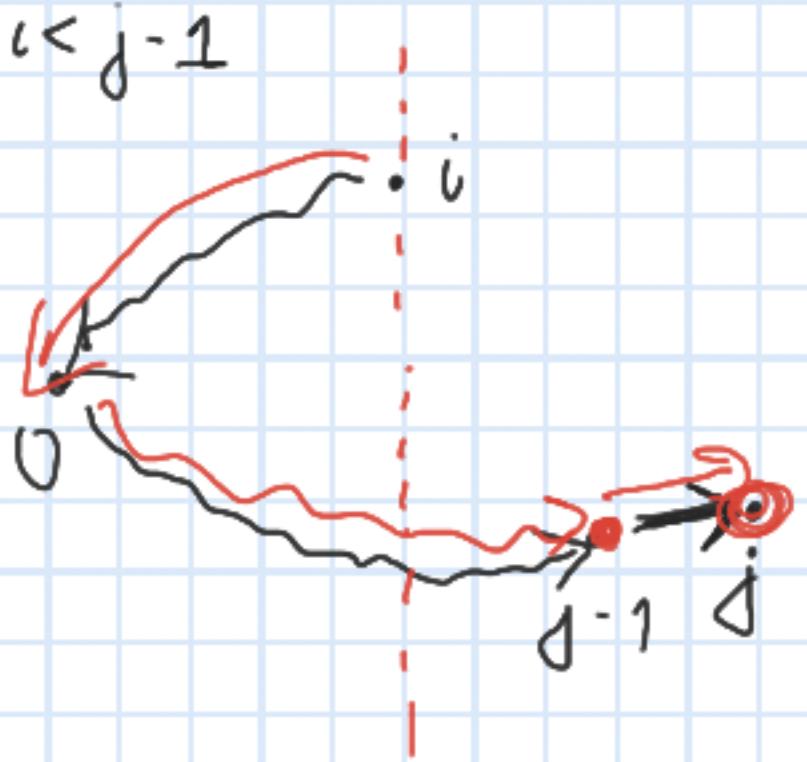
Jak odnaleźć rozwiązanie?

$$\min_i (f(i, n-1) + d(i, n-1))$$

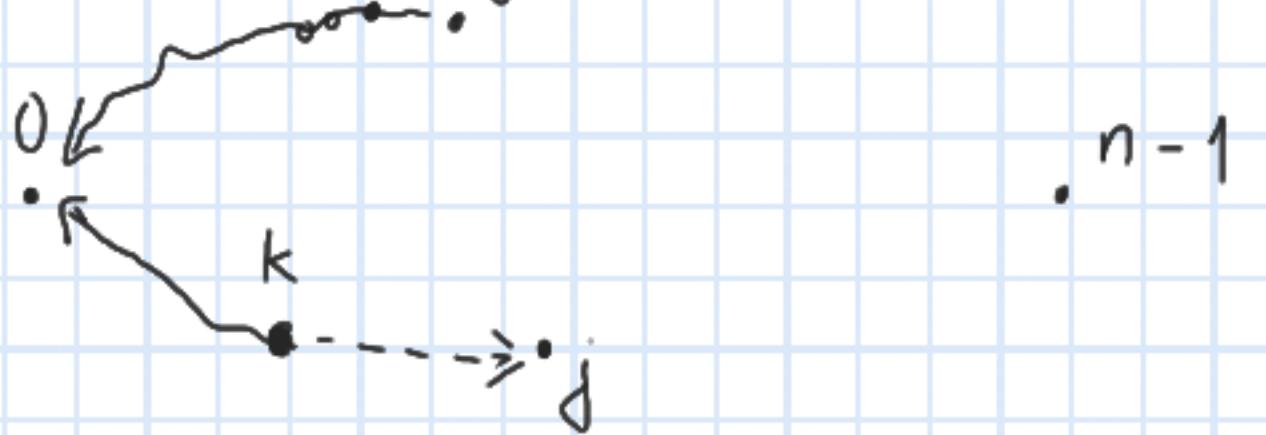


② Zapis rekurencyjny funkcji  $f$

$$a) f(i, j) = f(i, j-1) + d(j-1, j)$$



$$b) f(j-1, j) = \min_{k \in [j-1]} \left( f(k, j-1) + d(k, j) \right)$$



③ Implementacja

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

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

$$F[0][1] = D[0][1]$$

```
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
```

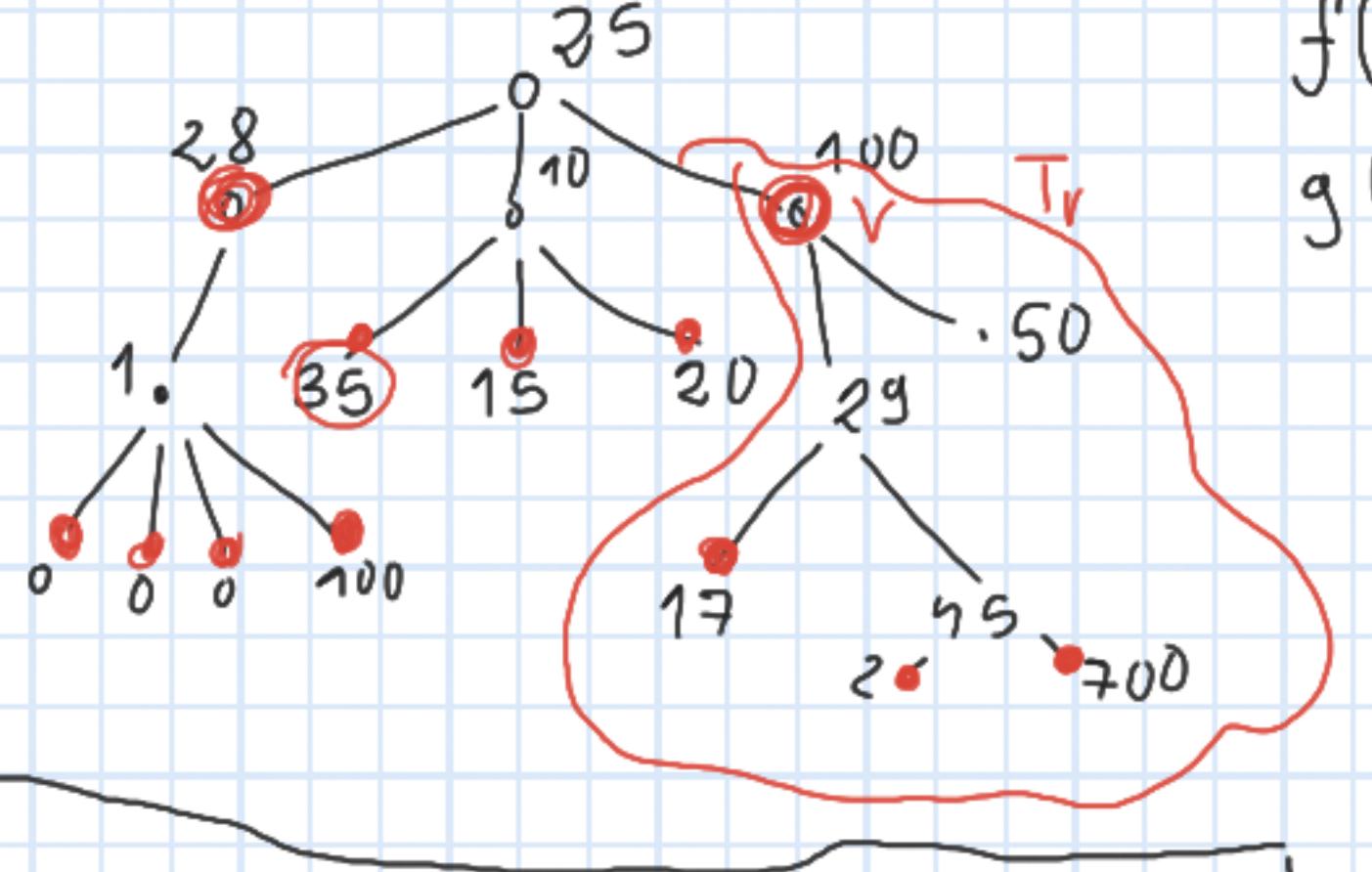
```
    else: // (a)
```

```
        F[i][j] = tspf(i, j-1, F, D) + D[j-1][j]
```

```
return F[i][j]
```

$O(n^2)$

# Impreza instytutowa II



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

## ① funkcja

$f(v) = \text{wartość najlepszej imprezy w } T_v$

$g(v) = \text{wartość najlepszej imprezy w } T_v, \text{ na której } v \text{ nie idzie}$

## ② rekurencja

$$g(v) = \sum_{u - \text{bezpr. podjedny } v} f(u)$$

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

Rozwiązań

$f(\text{root})$

dof  $f(v)$ :

if  $v.f \geq 0$ : return  $v.f$

$x = v.\text{fun}$

for  $u$  in  $v.\text{emp}$ :

$x += g(u)$

$y = g(v)$

$v.f = \max(x, y)$

return  $v.f$

dof  $g(v)$ :

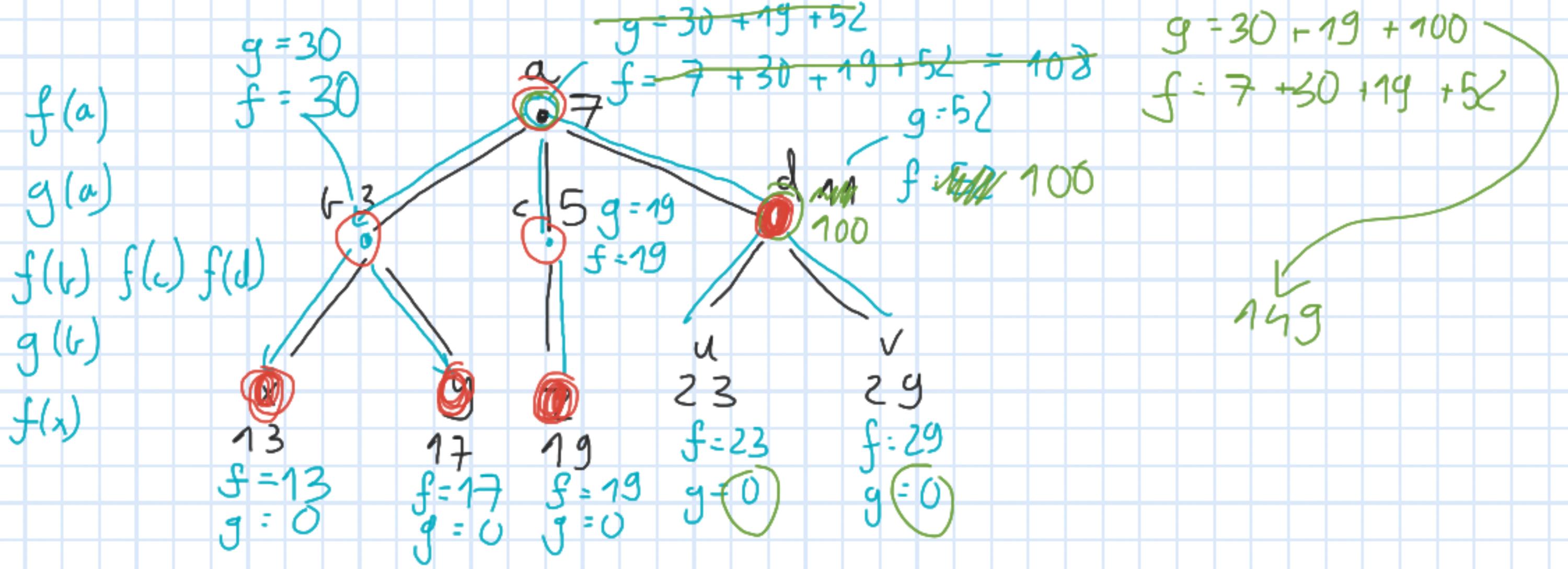
if  $v.g \geq 0$  return  $v.g$

$v.g = 0$

for  $u$  in  $v.\text{emp}$ :

$v.g += f(u)$

return  $v.g$



# Algorytmy i Struktury Danych

## Układ 7

Algorytmy zatrzymane

↳ algorytmy podejmujące lokalne  
optymalne decyzje

## Uwagi

- często nie są poprawne

↳ ale mogą dawać przyblizzone  
rozwiązań

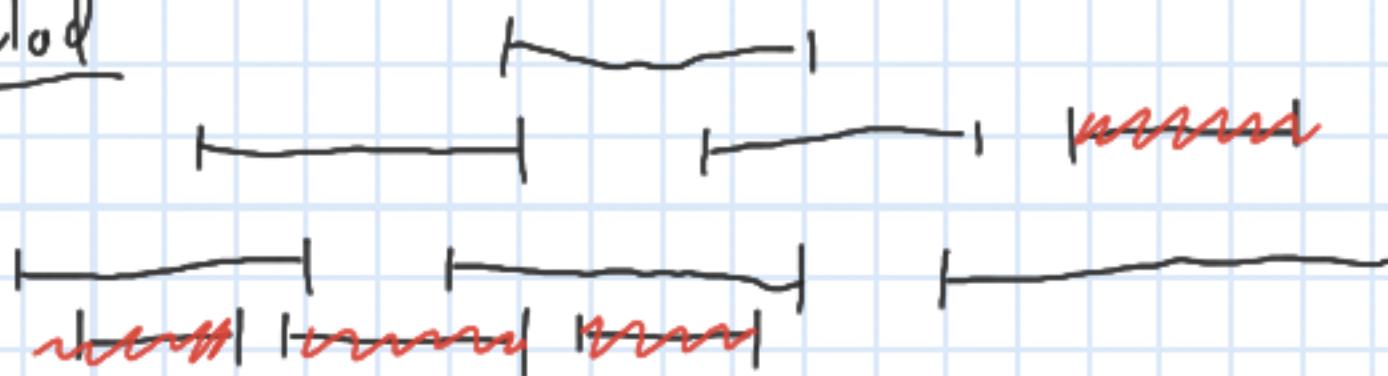
- często szybkie

Problem szeregowania zadań

Dane: Zbiór przedziałów (każdy przedział to czas trwania pewnych zajęć)

Zadanie: Wybrać jak najszybszy zbiór  
nienachodzących przedziałów

## Punktuł



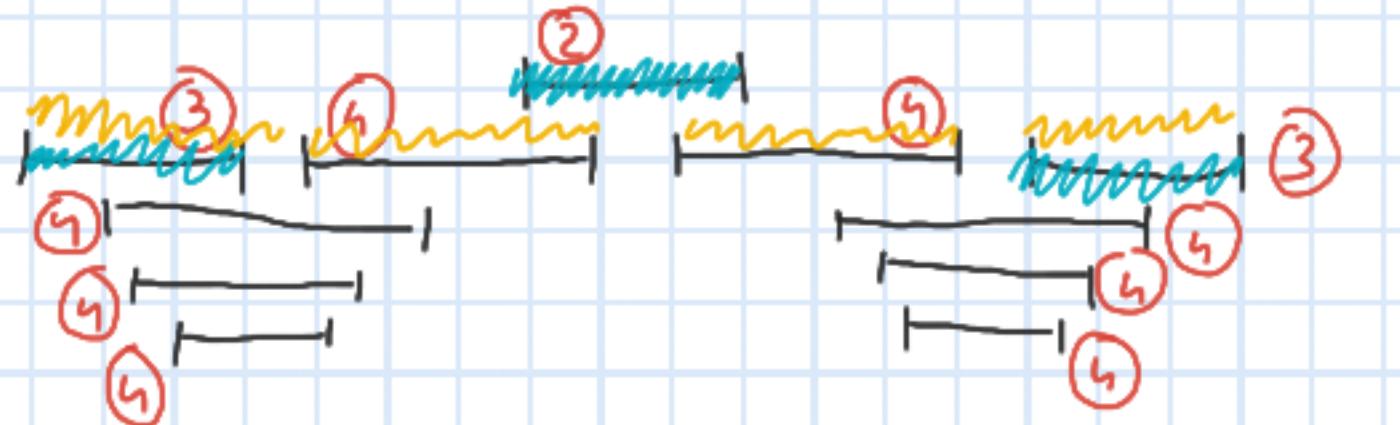
Jakie algorytmy zatrzymane wydają się naturalne?

a) najkrótszy przedział wybieramy  
pierwszy

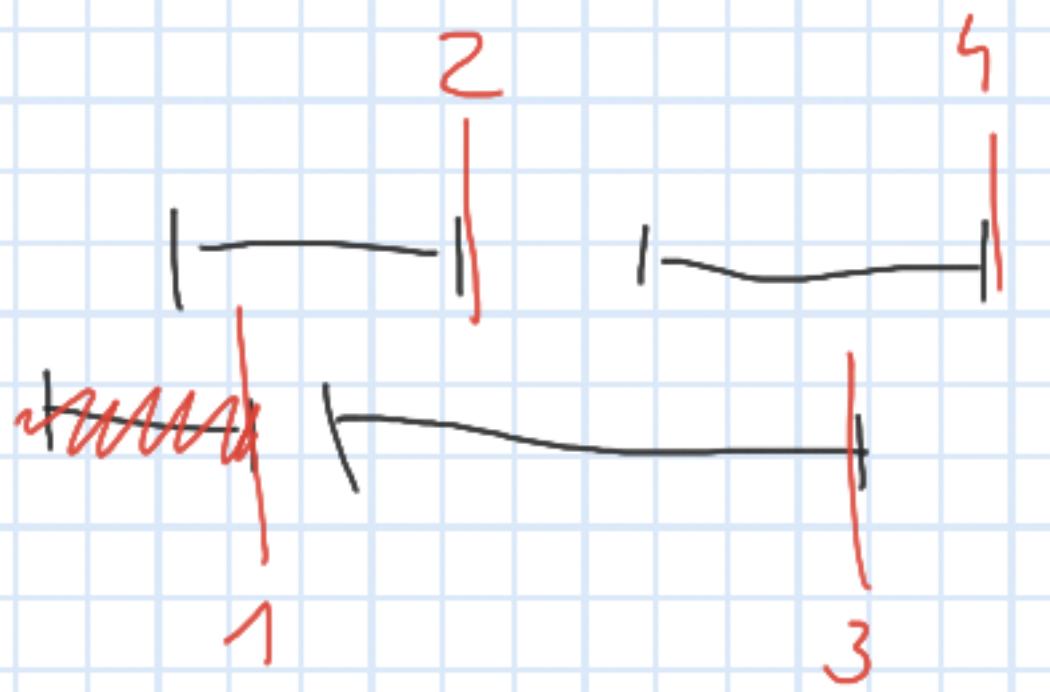
b) najwcześniej pojawiły się wybór pierwszy



c) wybieramy medial powinajacy najmniej dostepnych

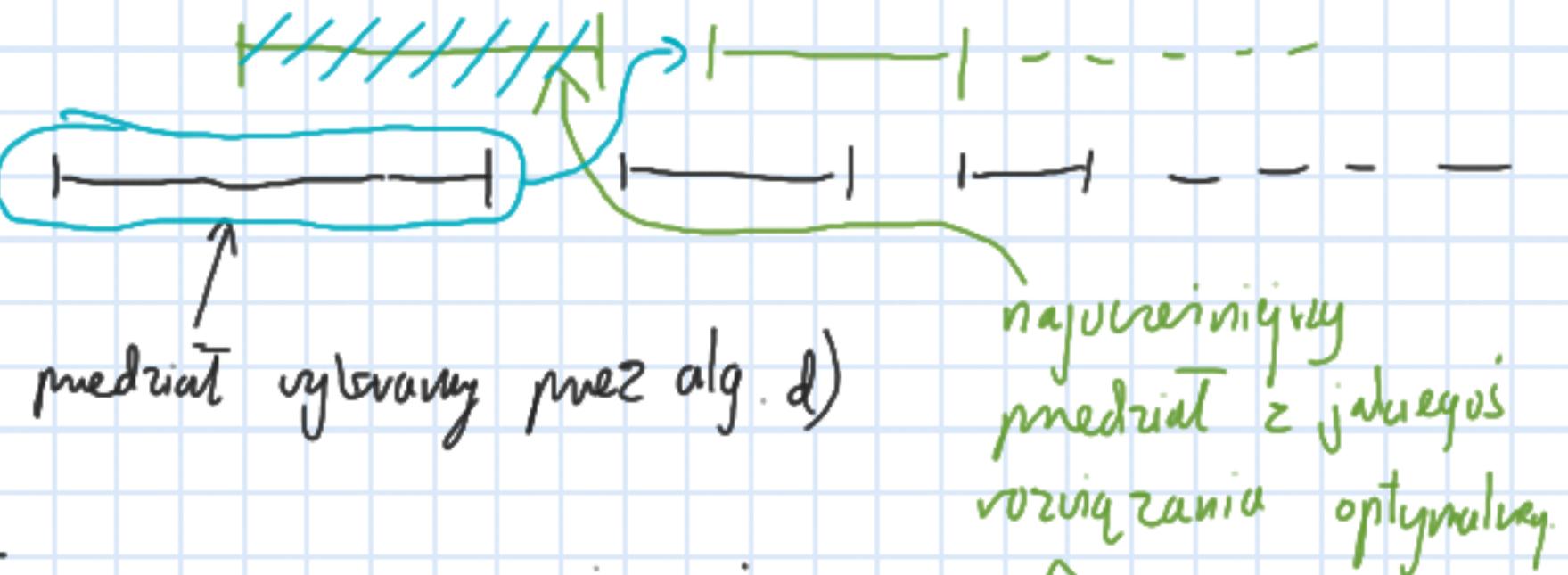


d) wybieramy medial, który się konie najmniej } Poprawne!



Dowód poprawności algorytmu d)

Rozważmy pierwszy moment, w którym nasz algorytm dokonał wyboru uniemożliwiającego skonstruowanie optymalnego rozwiązańia



Tworzymy nowe rozwiązańie przez ↑  
wykorzystanie pierwotnego medialu z rozw. opt  
i zastąpienie go medialatem wybranym  
przez algorytm d)

Spójrzcie! Czyli algorytm jest poprawny!

## Kody Huffmmana

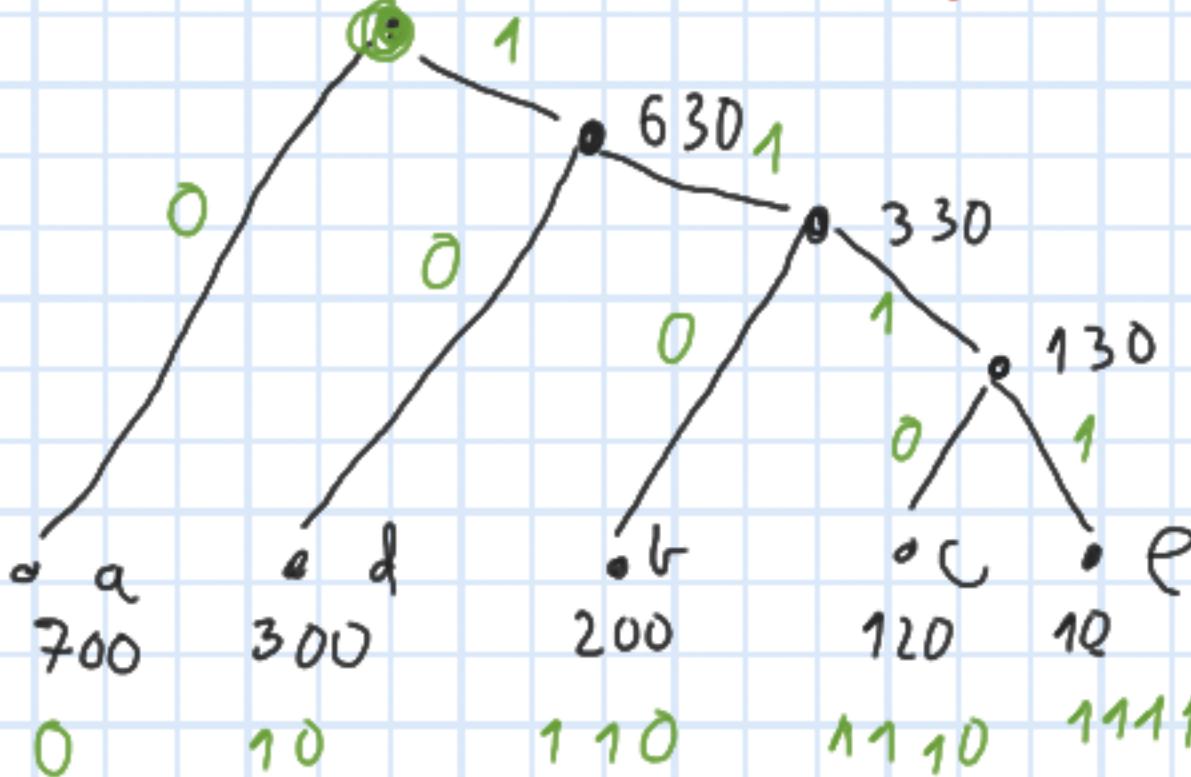
Kod binarny z symbolami różnych długosów

nie musi być optymalny

symbole	a	b	c	d	e
---------	---	---	---	---	---

częstotliwość (f)	700	200	120	300	10
	000	001	010	011	100

$$3 \cdot 700 + 3 \cdot 200 + 3 \cdot 120 + 3 \cdot 300 + 3 \cdot 10 = 3990$$



2220

## Naturalny algorytm zachłanny

- uziem dwa symbole o najmniejszej częstotliwości ( $x, y$ )
- połącz je w jeden nowy symbol ( $xy$ ) o częstotliwości  $f(xy) = f(x) + f(y)$
- usun symbole  $x, y$



- powtarzaj powyższe aż nie zostanie tylko jeden symbol (koniec tworzonego drzewa)

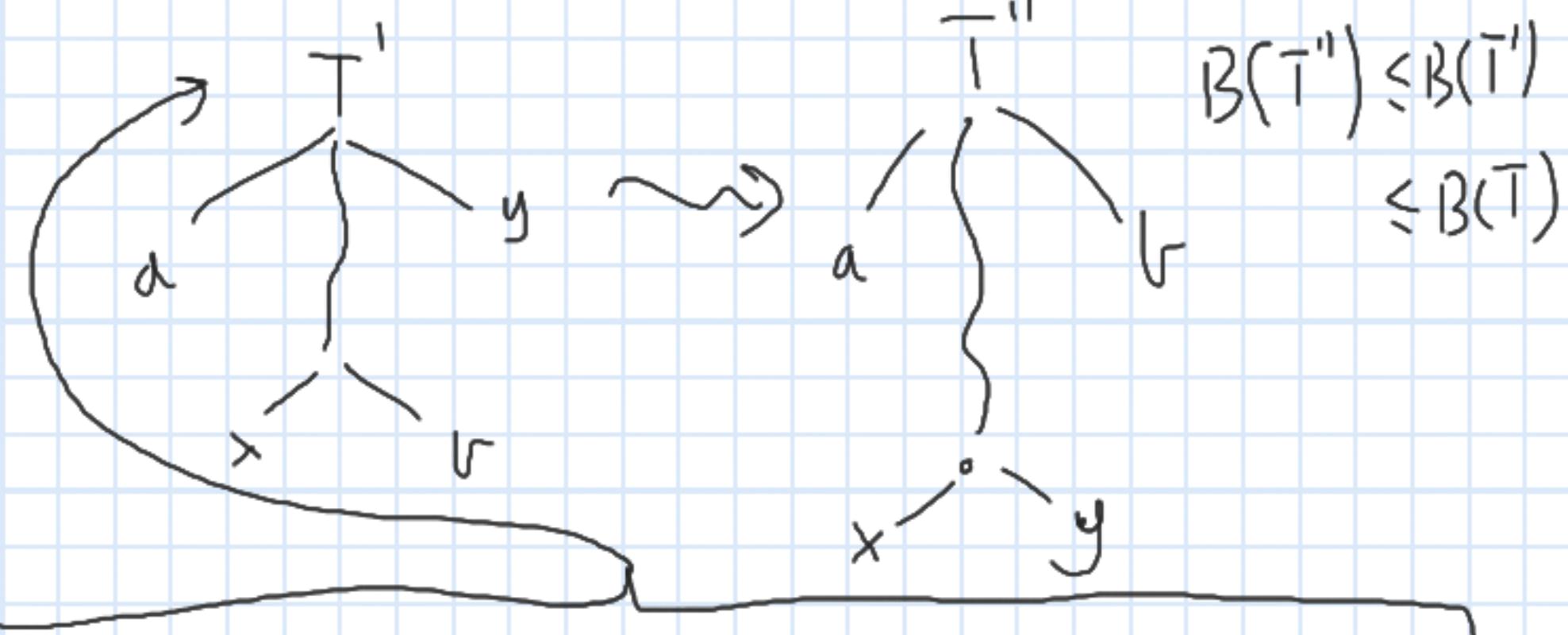
## Dowód poprawności

$T$  - dwoje kodujące

$B(T)$  - koszt dwoja częstości s

$$B(T) = \sum_{s-\text{symbol}} f(s) \cdot d_T(s)$$

dł. kodu symbolu s  
dwoje  $T$

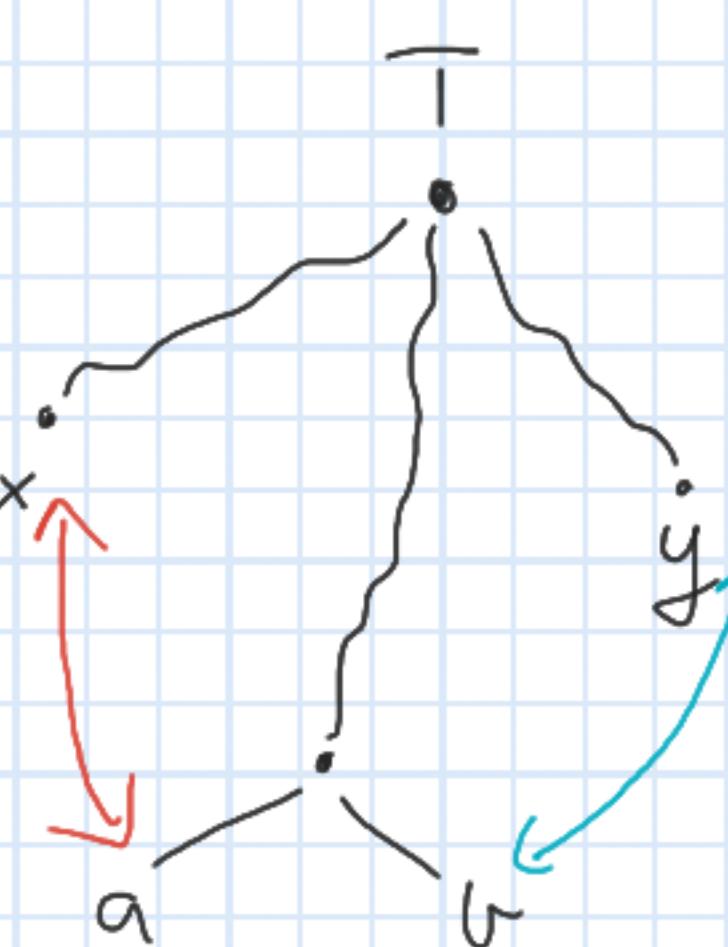


Krok 1: Dwa najbardziej symbole można umieszcać w wspólnym uziale

$x, y$  - dwa najbardziej symbole

$a, b$  - dwa symbole w najdłuższej gałęzi dwoja

$T$  - pewne optymalne dwoje



$T'$  - dwoje powstałe z  $T$  przez zamianę a oraz x

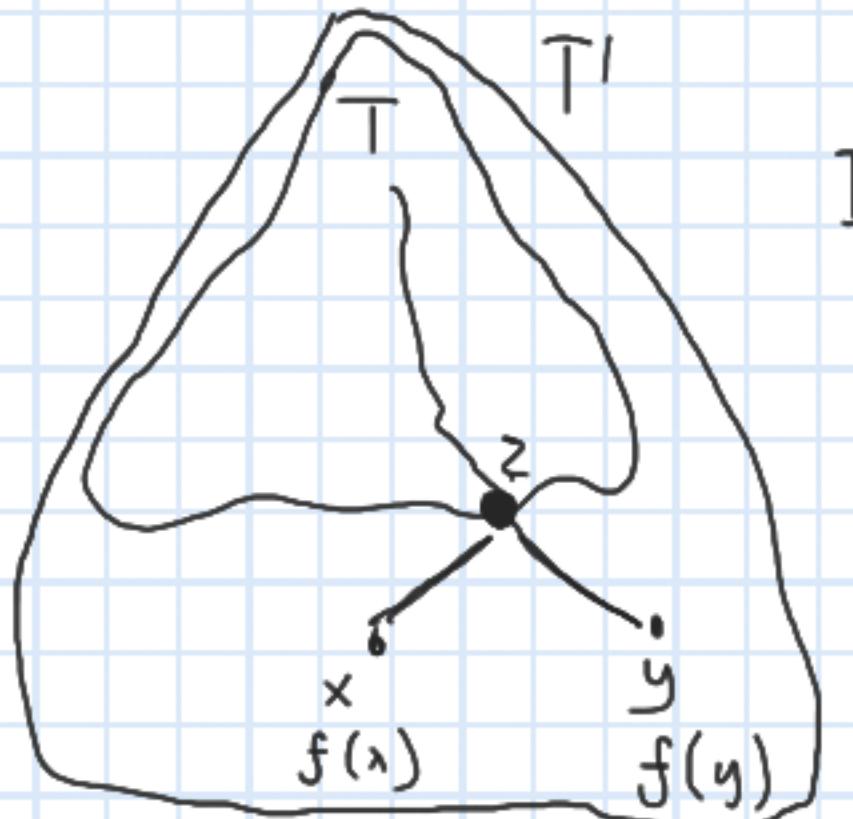
$$\begin{aligned} B(T') &= B(T) + (f(a) - f(x)) \cdot d_{T'}(x) \\ &\quad + (-f(a) + f(x)) \cdot d_{T'}(a) \end{aligned}$$

$$= B(T) + (f(a) - f(x)) (d_{T'}(x) - d_T(a))$$

$\geq 0$        $\leq 0$

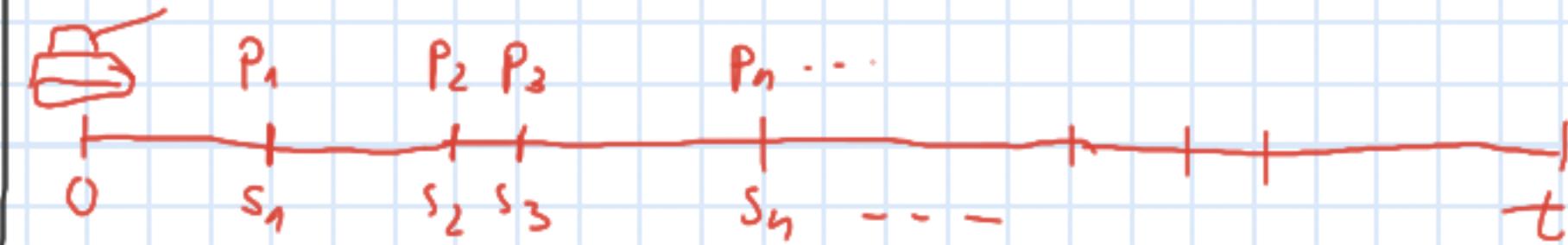
$\leq 0$

## Krok 2 Optymalna podstruktura



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

## Zadanie dwurzędowe (tankowania czołgu)



L - pojemność baku czołgu (litry)

s<sub>i</sub> - odległość stacji od punktu 0 (km)

czołg spala 1l/1km

P<sub>i</sub> - cena za litr na każdej stacji

### Tmy problemy

// wózka startuje z pełnym bakiem

a) obliczyć minimalną liczbę tankowań, żeby dotrzeć do punktu t

b) obliczyć minimalny koszt  
dotarcia do punktu t

↗ (1) na każdej stacji można tankować tyle ile się chce

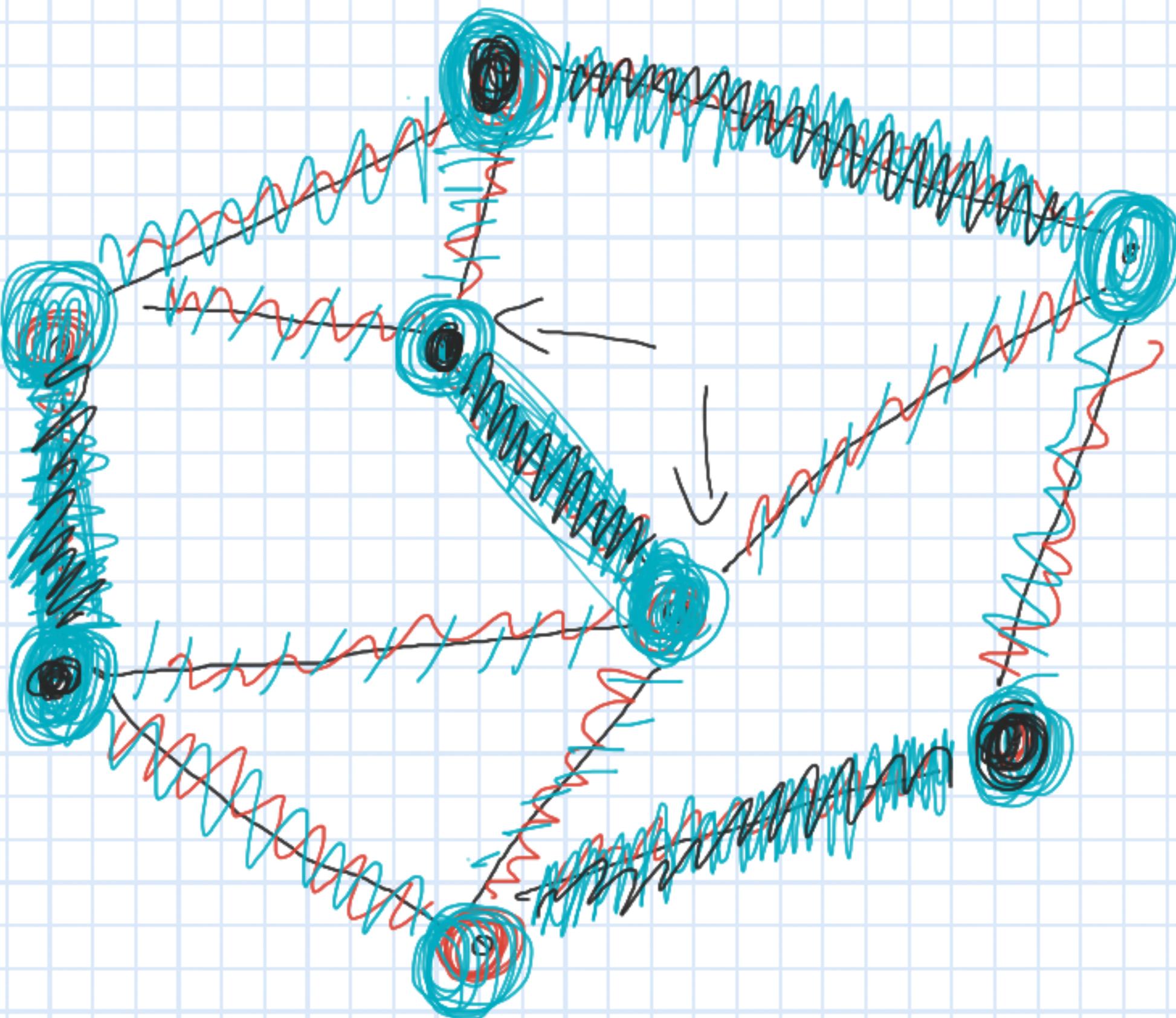
↗ (2) jeśli tankując to do pełna

Galeria sztuki

NP - zupełny

OPT · log n

2 · OPT



# Algorytmy i Struktury Danych

## Wykład 8

### Algorytmy grafowe

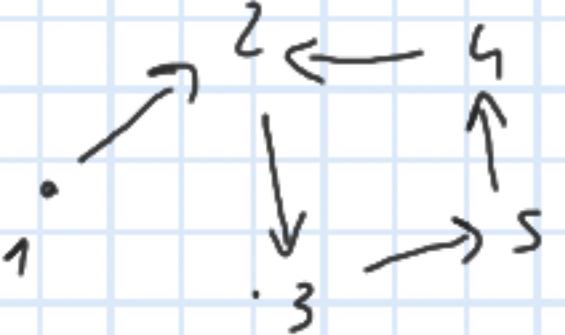
Graf skierowany to para  $G = (V, E)$  gdzie

- $V$  to skończony zbiór wierzchołków

- $E$  to zbiór krawędzi, gdzie każda

krawędź to para  $(u, v)$ ,  $u, v \in V$   
 $u \neq v$

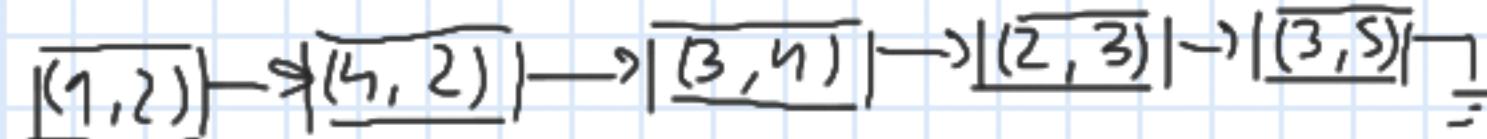
## Punkty



Graf nieskierowany - j.w., ale krawędź  
to dwielementowy podzbiór  $\{u, v\} \subseteq V$

## Reprezentacja

### ① Lista / tablica krawędzi

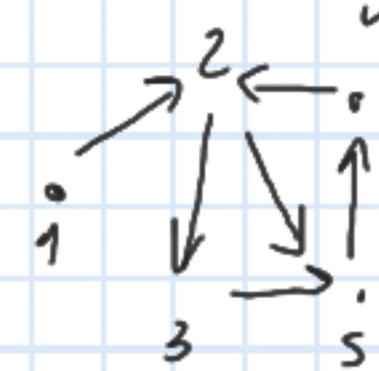


### ② Reprezentacja macierzowa

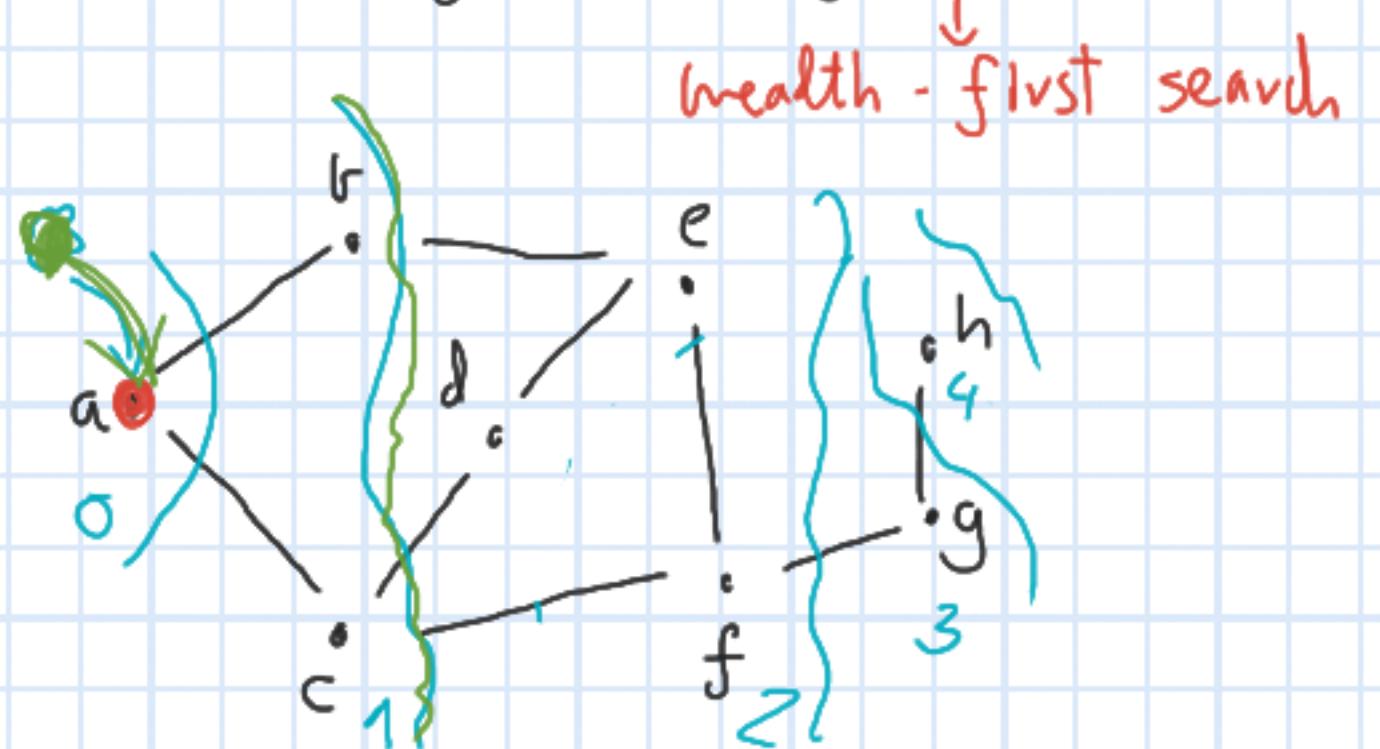
	1	2	3	4	5
1	-	1	0	0	0
2	0	-	1	0	0
3	0	0	-	1	1
4	0	1	0	-	0
5	0	0	0	0	-

### ③ Listy sąsiedztwa

1	$\rightarrow 2$
2	$\rightarrow 3, 5$
3	$\rightarrow 5$
4	$\rightarrow 2$
5	$\rightarrow 4$



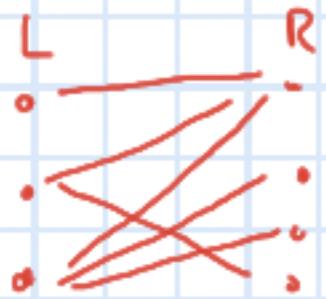
# Pnieszukiwanie grafu metodą BFS (w szen)



$Q: a | b c | e f d | g | h$

## Zadania obowiązkowe

- proszę zaimplementować algorytm sprawdzający czy graf jest dwudzielny



def BFS( $G, s$ )

#  $G = (V, E)$ ,  $s \in V$

$Q = \text{Queue}()$

for  $v \in V:$

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

$v.d = -1$

$v.parent = \text{None}$

$s.d = 0$

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

$Q.\text{put}(s)$

$n = |V|$

$\leftarrow Q = []$

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

rep. listowa

rep. macierz.

$O(V+E)$

$O(V^2)$

while not  $Q.\text{isEmpty}()$ :

$u = Q.\text{get}()$

$\leftarrow Q.\text{pop}(0)$  - Nieefektywne!

for  $v$  - sąsiad  $u$ :

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

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

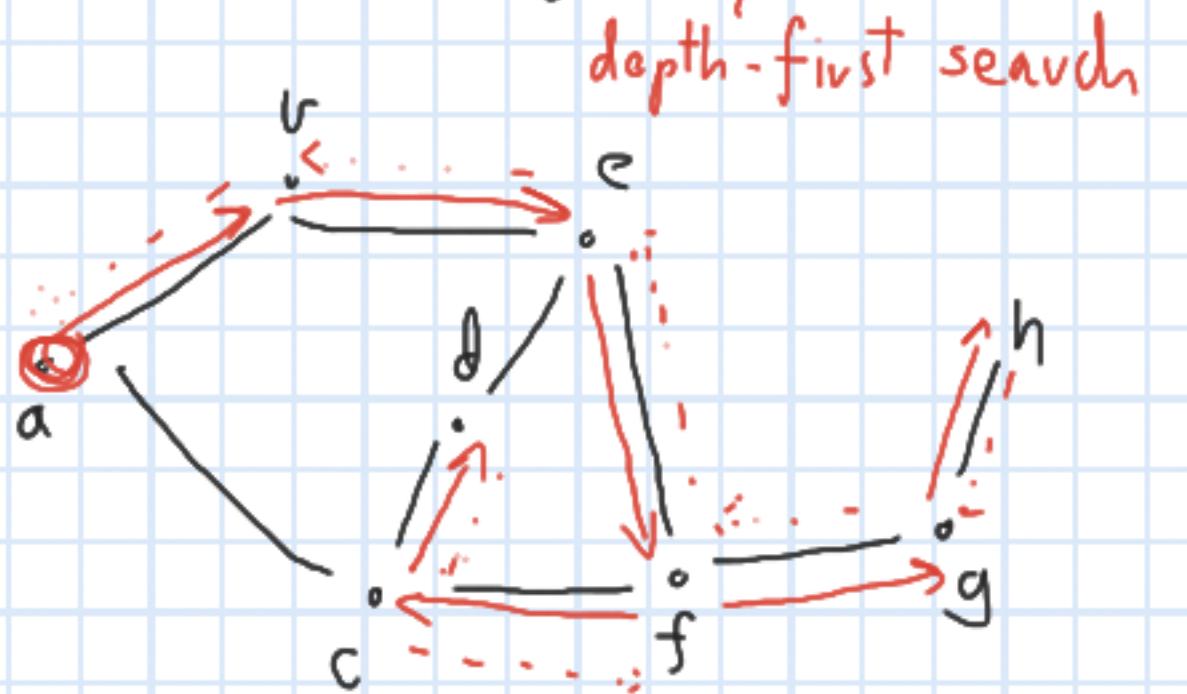
$v.d = u.d + 1$

$v.parent = u$

$Q.\text{put}(v)$

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

# Peszukiwanie w głębi (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  
u.visited = True

for v - sąsiad u:

if not v.visited:

v.parent = u

DFSVisit(G, v)

time += 1

czas odwiedzenia

czas pnenowania

# Algorytmy i Struktury Danych

## Wykład 9

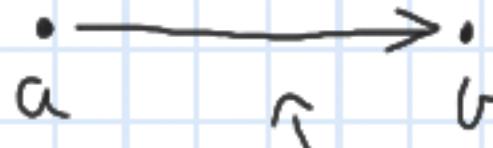
### Zastosowania algorytmu DFS

#### ① Sztuczne topologiczne

DAG - directed acyclic graph

Sztuczne topologiczne dag'u polega na ułożeniu wierzchołków w takiej kolejności, że krawędzie łączące tylko "z lewej na prawą"

### Zastosowania



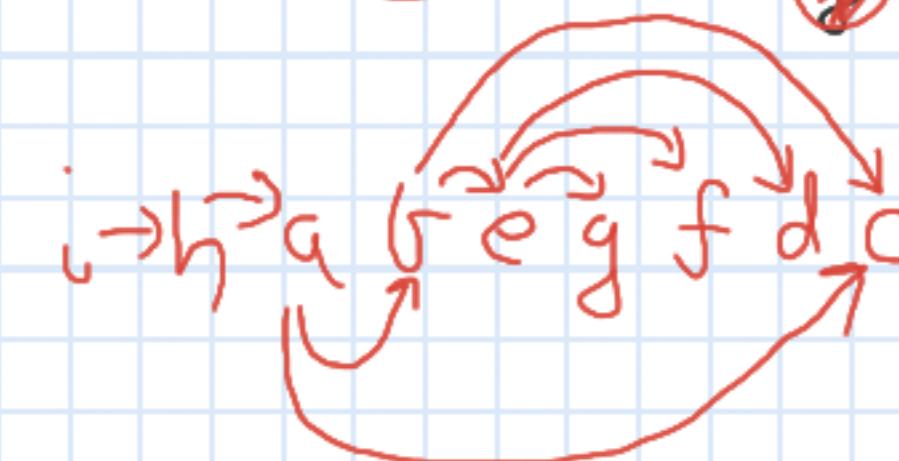
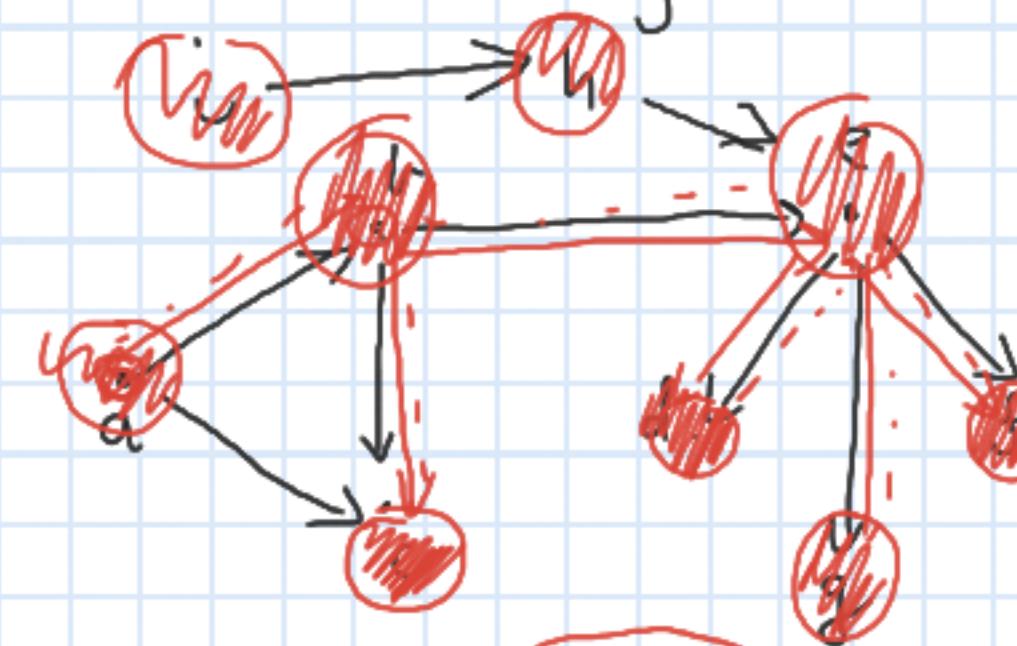
zadanie b musi być wykonane przed zadaniem a

### Algorytm

- wykonać algorytm DFS

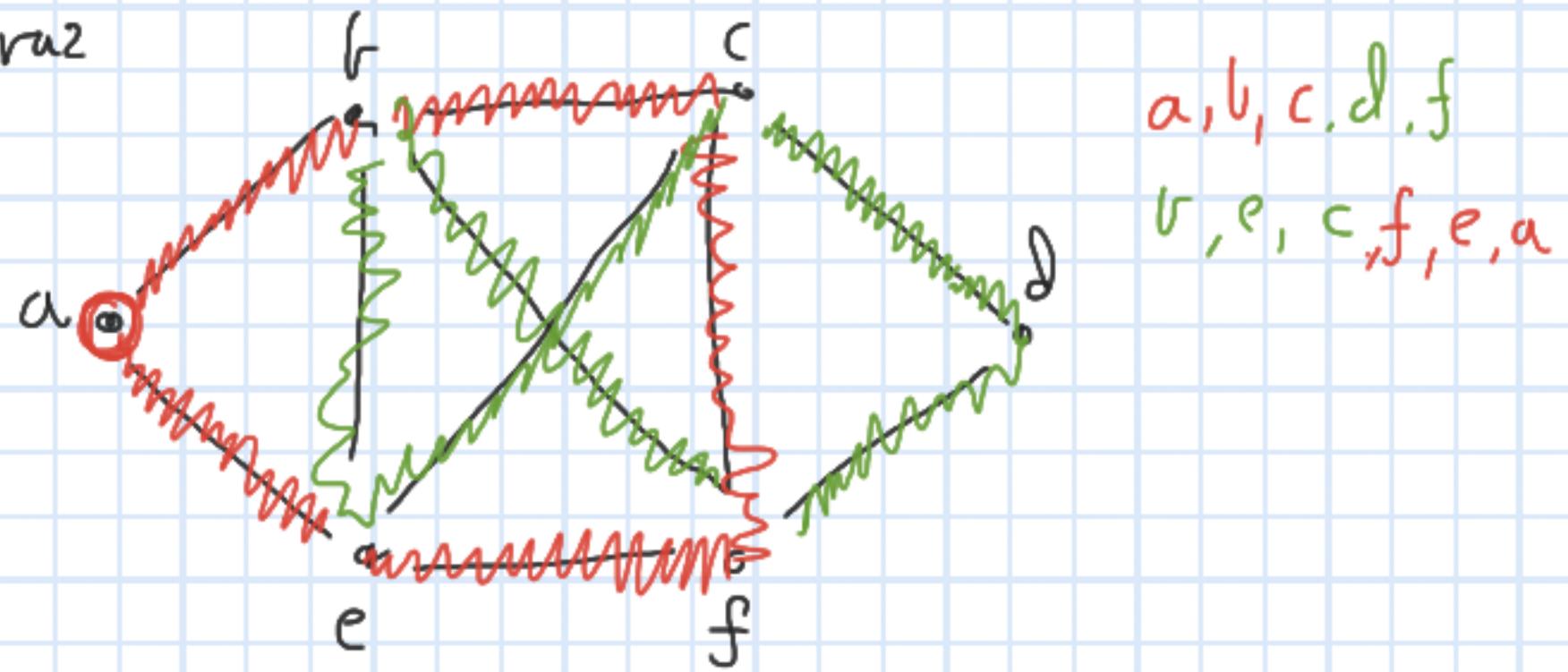
↳ po zakończeniu danego wierzchołka

dopisujemy go na koniec listy zadań do wykonania



## ② Cykl Eulera

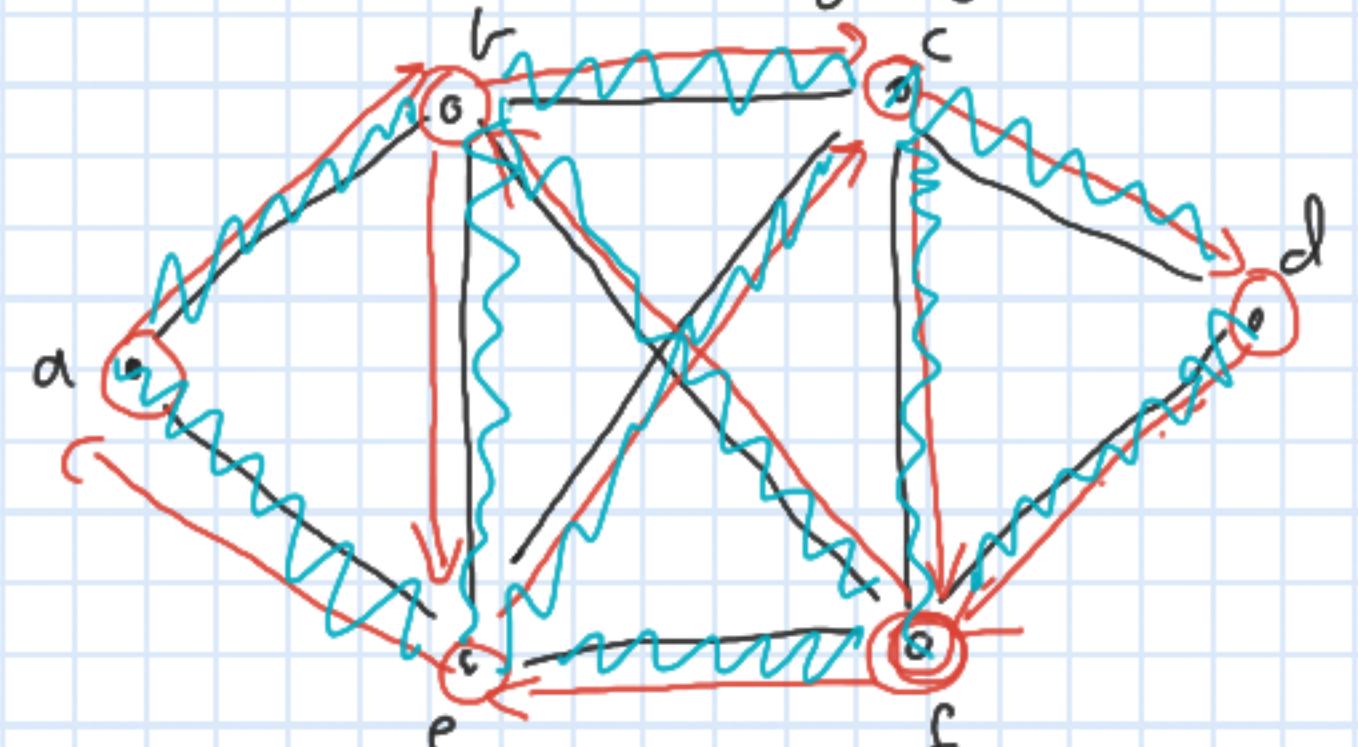
def Cykl Eulera to taki cykl, który przechodzi przez każdy krawędź dokładnie raz



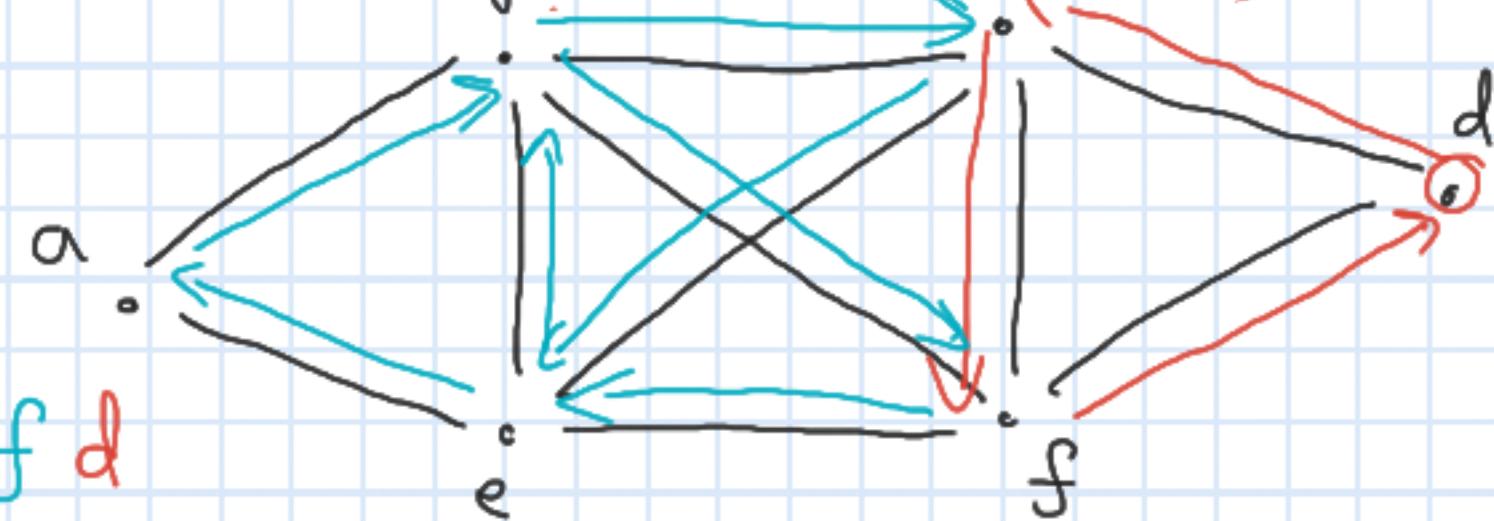
tw Graf nieskierowany posiada cykl Eulera jeśli jest spójny i każdy jego wierzchołek ma parzysty stopień

## Algorytm

- wykonujemy DFS, usuwając na bieżąco krawędzie po których wędrujemy
- po znalezieniu wierzchołka dodajemy go na koniec tworzonego cyklu



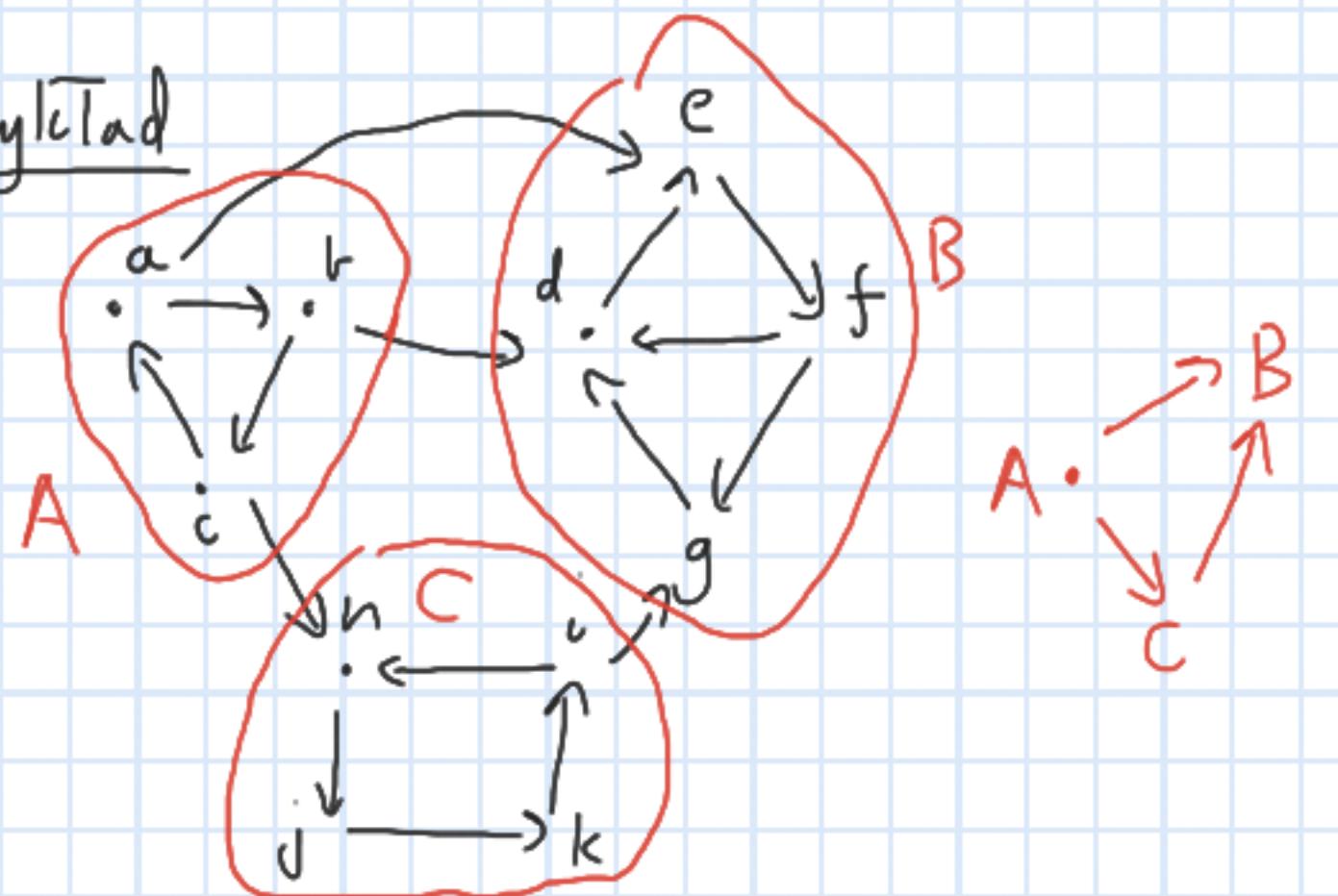
c f e a b c e r f d



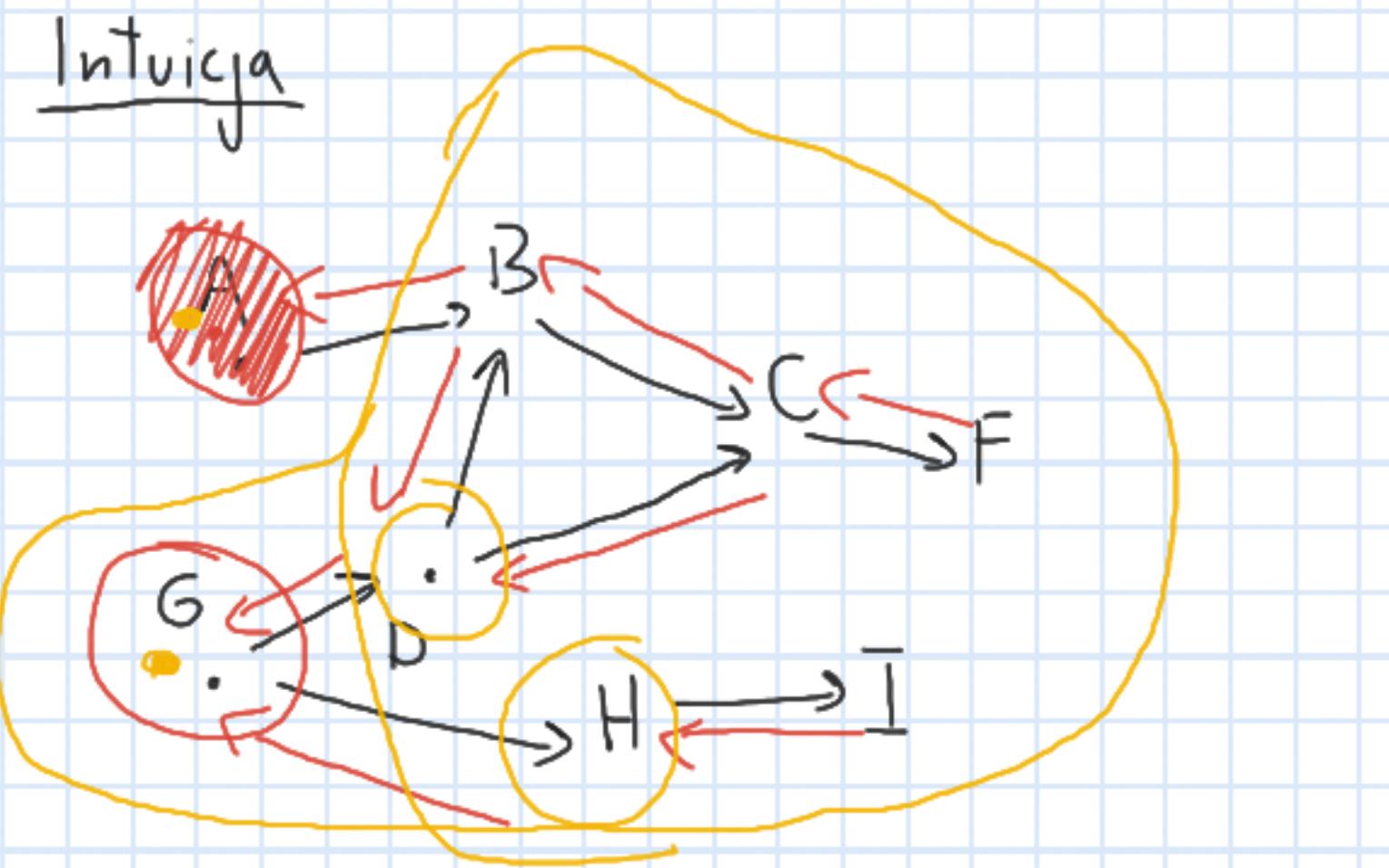
### ③ Siłnice spójne składowe (grafu skierowanego)

def Niech  $G = (V, E)$  będzie grafem skierowanym. Mówimy, że  $u, v \in V$  należą do tyl samej siłnice spójnej składowej jeśli istnieje szlak skierowany z  $u$  do  $v$  oraz z  $v$  do  $u$ .

#### Ponkładać



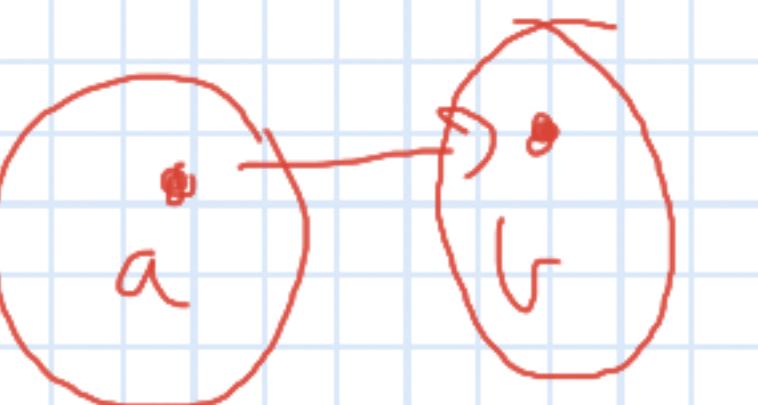
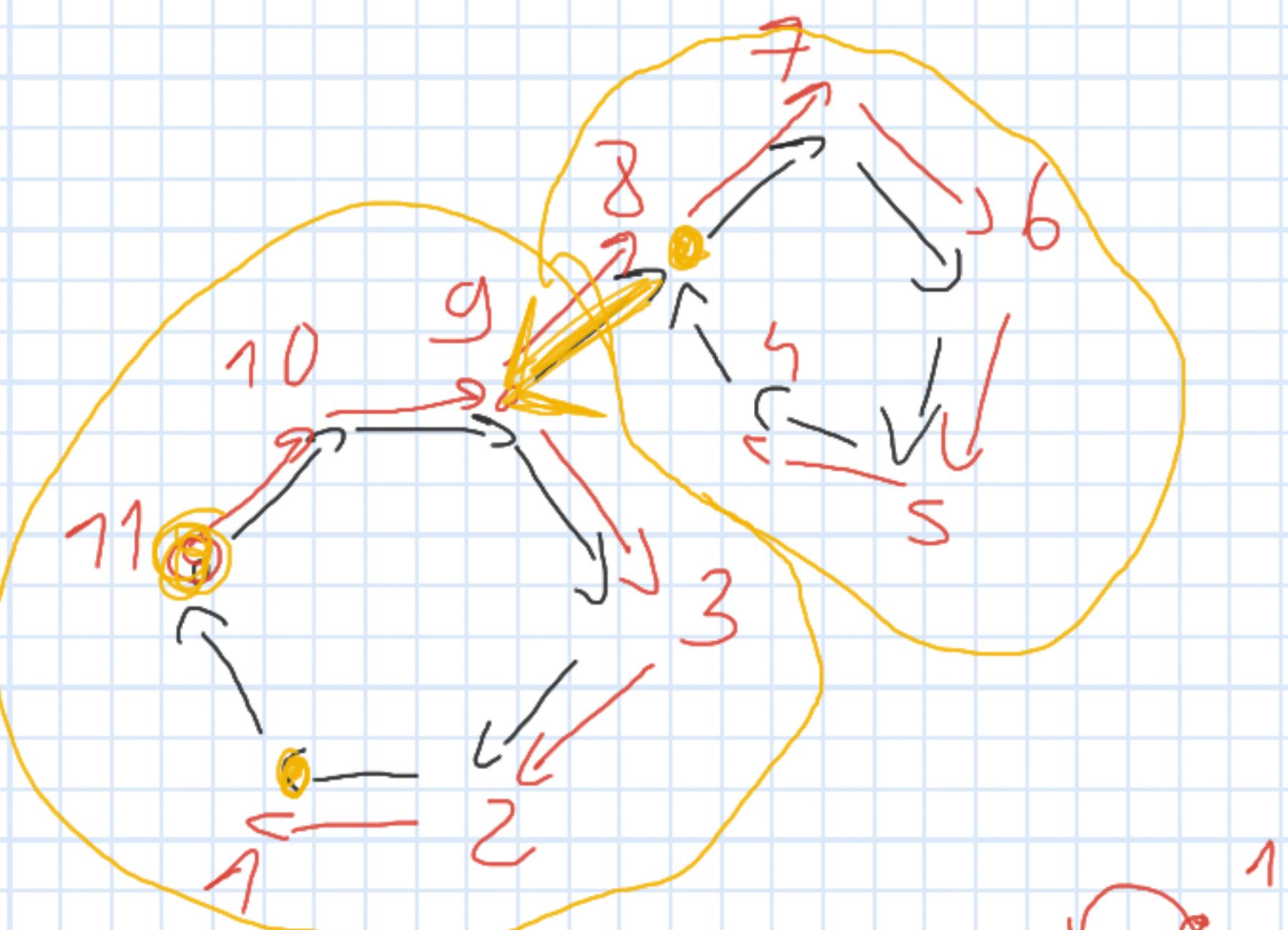
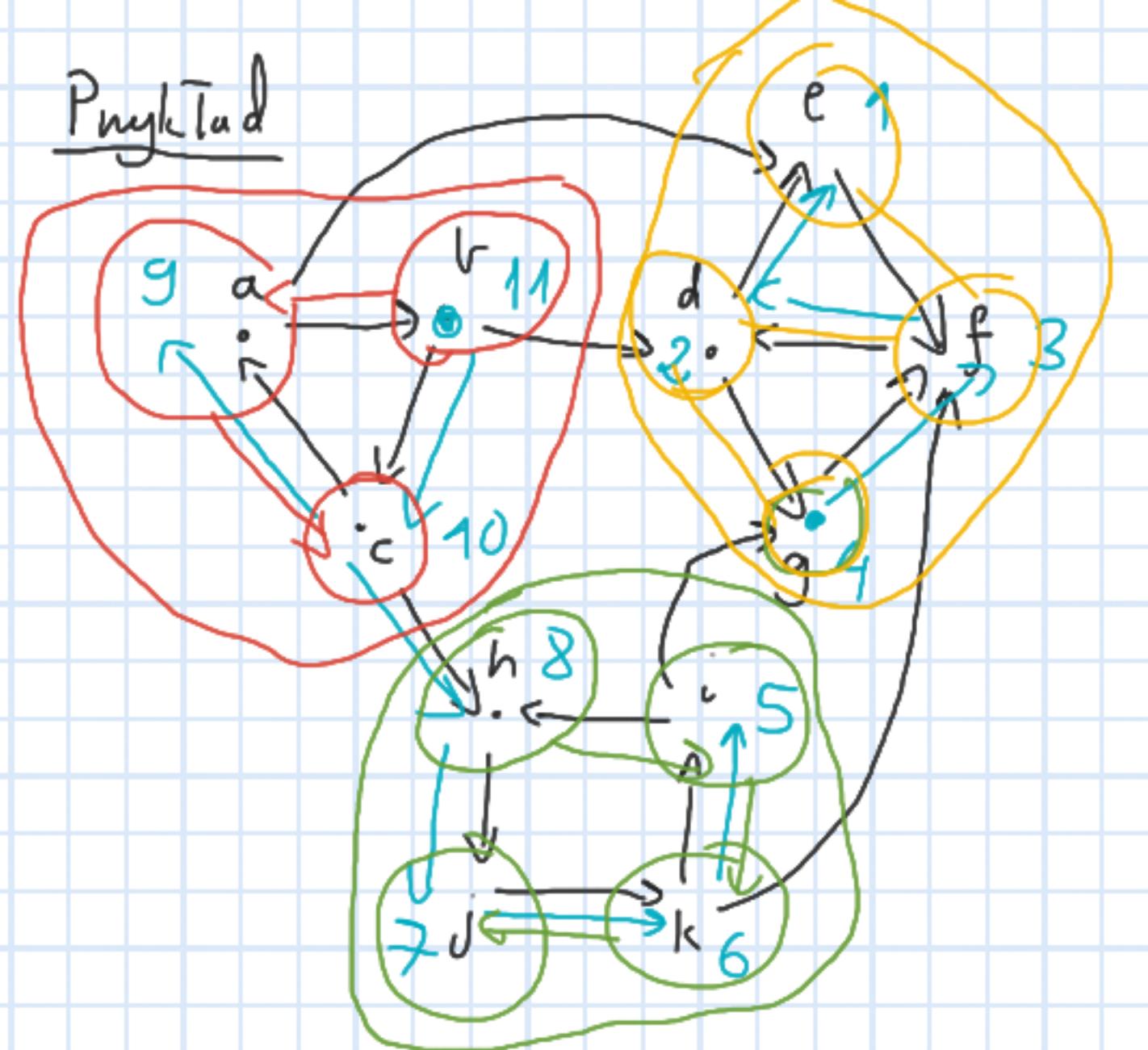
petla uwy.  
DFSVisit



#### Algorytm

1. Wykonaj DFS dla grafu wychodzącego, zapisując w wieniach czas przetwarzania
2. Odwróć kolejność krańców
3. Wykonaj DFS drugi raz  
(w kolejności malejącej czasów przetwarzania)

Pnyktad



## ⑤ Mosty w grafach nieskierowanych

def Krawędź w spójnym grafie nieskierowanym nazywamy mostem jeśli jej usunięcie rozspojni graf



tu Krawędź  $e$  jest mostem utw. gdy nie leży na żadnym cyklu prostym w grafie

Dowód

$e$  jest mostem  $\Rightarrow$  nie leży na cyklu



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

## Algorytm

① Wykonaj DFS, dla każdego  $v \in V$  zapisując jego czas odwiedzenia  $d(v)$

② Dla każdego  $v \in V$  obliczamy

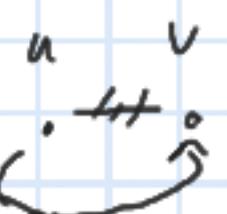
$$\text{low}(v) = \min \left( d(v), \min \text{d}(u) \right)$$

mamy krawędź  
ustoczącą z  $v$  do  $u$

$$\min \text{low}(w)$$

$w$  jest dzieciem  $v$  w  
dużym DFS

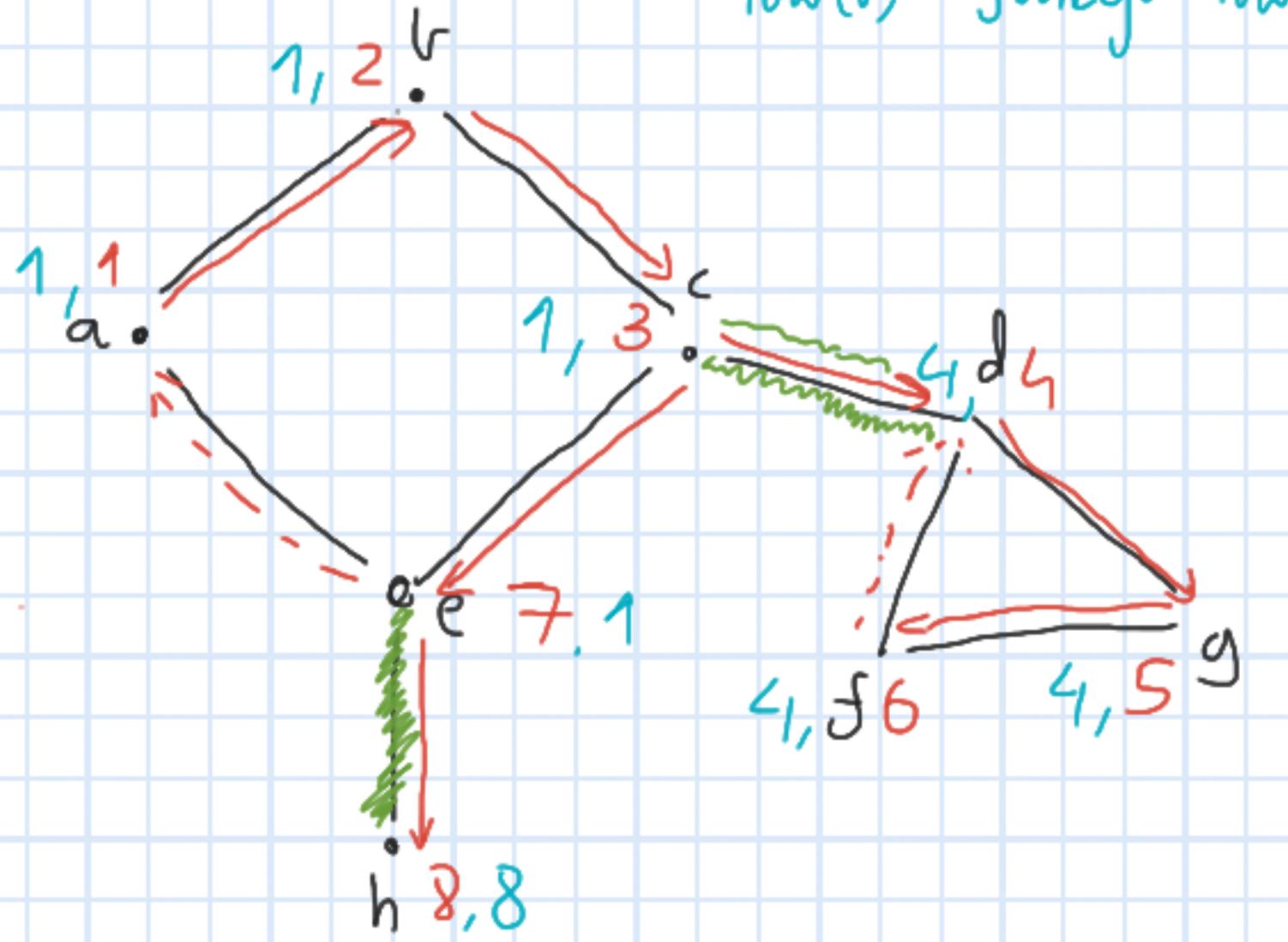
ojciec  $v$   
w DFS



③ Mosty to krańcze  $\{v, p(v)\}$

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

Punktad

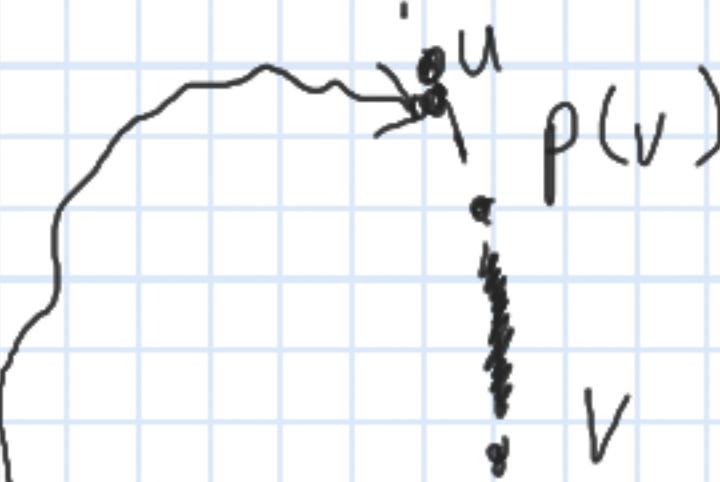


$d(v)$  - czas odwiedzenia

$low(v)$  - funkcja low

jeśli  $d(v) = low(v)$  to

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



$$V \quad d(v) = low(v)$$

<- gdyby taki cykl istniał  
to  $low(v) = d(u) < d(v)$