

# Rozdział 1

## Lista 1.

### 1.1.

Napisz rekurencyjne funkcje, które dla danego drzewa binarnego  $T$  obliczają:

- liczbę wierzchołków  $T$

```
def nodes tree
    return 0 if tree.nil?
    return 1 if tree.leaf?
    return 1 + nodes(tree.left) + nodes(tree.right)
```

- maksymalną odległość między wierzchołkami w  $T$  Jest to **średnica** drzewa. Zauważmy, że:
  - Puste drzewo ma średnicę 0.
  - Jeśli drzewo jest niepuste, to przez  $t_1$  i  $t_2$  oznaczmy dwa poddrzewa zakorzenione w lewym i prawym synie korzenia. Odpowiednio przez  $d_1$  i  $d_2$  oznaczmy średnice tych poddrzew, a przez  $h_1$  i  $h_2$  ich wysokości. Wówczas średnica całego drzewa wynosi  $\max(d_1, d_2, h_1 + h_2 + 2)$ .

```
def diameter(node)
    return (-1,0) if node.nil?
    (lheight, ldiameter) = diameter(node.left)
    (rheight, rdiameter) = diameter(node.right)

    height = max(lheight, rheight) + 1
    diameter = max(lheight + rheight + 2, ldiameter, rdiameter)

    return [height, diameter]
```

### 1.2.

Dla kopca minimaxowego. Przyjmij, że elementy pamiętane są w jednej tablicy (określ w jakiej kolejności). Napisz w pseudokodzie procedury:

- przywracania porządku

```
def level(i)
    floor(log2(i))
```

```

def min_level?(i)
  level(i)\%2 == 1

def trickle_down(K,i)
  if min_level?(i)
    trickle_down_min(K,i)
  else
    trickle_down_max(K,i)
  end

def trickle_down_min(K,i)
  return if K[i] has no children
  m = index of smallest of the children\
    and grandchildren (if any) of K[i]

  if K[m] is a child of K[i]
    if K[m] < K[i]
      swap(K[m], K[i])
    else # it grandchild from next min level
      if K[m] < K[i]
        swap(K[m], K[i])
      if K[m] < K[parent(m)]
        swap(K[m], K[parent(m)])
      trickle_down_min(K,m) //swap with parent doesn't change our min/max level

```

W **trickle\_down** sprawdzamy czy element przesuwany w dół należy zamienić z którymś z dzieci oraz wnuków. Dodatkowo, element podmieniamy z wnukiem, sprawdzamy czy nie popsuliśmy ojca wnuka.

Jeśli podmieniliśmy element z bezpośrednim dzieckiem, to znaczy, że porządek jest zachowany i nie musimy schodzić niżej.

```

def bubble_up(K,i)
  if min_level?(i)
    if K[i] has a parent and
      K[i] > K[parent(i)]
      swap(K[i], K[parent(i)])
      bubble_up_max(K,parent(i))
    else
      bubble_up_min(K,i)
  else #max level
    if K[i] has a parent and
      K[i] < K[parent(i)]
      swap(K[i], K[parent(i)])
      bubble_up_min(K,parent(i))
    else
      bubble_up_max(K,i)

def bubble_up_min(K,i)
  if K[i] has a grandparent
    if K[i] < K[grandparent(i)]
      swap(K[i], K[grandparent(i)])
      bubble_up_min(K,grandparent(i))

```

W **bubble\_up** sprawdzamy najpierw czy świeżo wstawiony element pasuje bardziej do poziomu, na który trafił przy wstawieniu, czy do poziomu wyżej (sprawdzamy czy jest pretendentem do maksimum czy minimum). Następnie przesuwamy go wyżej skacząc bo dziadkach, nie zaburzając przy tym porządku kopca.

```
def change(K,i,v)
    if min_level?(i)
        if K[i] < v
            K[i] = v
            trickle_down(K,i)
        else
            K[i] = v
            bubble_up(K,i)
    else
        if K[i] > v
            K[i] = v
            trickle_down(K,i)
        else
            K[i] = v
            bubble_up(K,i)
```

Jeżeli element pasuje lepiej na swoim poziomie niż poprzednik, to próbujemy upchnąć go wyżej (relacja z poniższymi elementami jest niezmienną). Jeżeli element nie pasuje na swoim poziomie, musimy upchać go w dół.

- usuwania minimum

```
def delete_min(K)
    min = K[1]
    swap(K[1],K[n])
    K = K[1..n-1]
    trickle_down(K,1)
    return min
```

- usuwania maksimum

```
def delete_max(K)
    m = index of largest element
        amongst K[1], K[2], K[3]
    max = K[m]
    swap(K[1], K[n])
    K = K[1..n-1]
    trickle_down(K,1)
    return max
```

### 1.3.

Porządkiem topologicznym wierzchołków acyklicznego digrafu  $G = (V, E)$  nazywamy taki liniowy porządek jego wierzchołków, w którym początek każdej krawędzi występuje przed jej końcem. Jeżeli wierzchołki z  $V$  utożsamimy z początkowymi liczbami naturalnymi to każdy ich przodek liniowy można opisać permutacją liczb  $1, 2, 3, \dots, |V|$ ; w szczególności pozwala to na porównywanie leksykograficzne porządków. Ułóż algorytm, który dla danego digrafu znajduje pierwszy leksykograficznie porządek.

$Q$  - Kolejka priorytetowa z wierzchołkami o stopniu wchodzącym równym 0.

```

dopóki Q jest niepusta rób
    usuń wierzchołek n z przodu kolejki Q
    wypisz n
    dla każdego wierzchołka m o krawędzi e od n do m rób
        usuń krawędź e z grafu
        jeżeli do m nie prowadzi żadna krawędź to
            wstaw m do Q
jeżeli graf ma wierzchołki to
    wypisz komunikat o błędzie (graf zawiera cykl)

```

Jedyną zmianą w algorytmie sortowania topologicznego, jest podmienienie kolejki na kolejkę priorytetową. W ten sposób zawsze otrzymujemy wierzchołek spełniający warunek porządku topologicznego, który ma najmniejszy możliwy indeks. Złożoność sortowania topologicznego wynosi  $O(|E| + |V|)$ . Ale w naszym przypadku, każde wstawienie elementu do kolejki trwa  $O(\log|V|)$  zamiast  $O(1)$ , zatem złożoność wynosi:  $O(|E| + |V|\log|V|)$ .

### 1.3.1. dowód poprawności

Nasz algorytm jest zachłanny, stopniowo buduje rozwiązanie używając lokalnego kryterium optymalności.

Założmy, że nasz algorytm wybrał porządek topologiczny  $T = t_1, t_2, \dots, t_n$ , podczas gdy istnieje porządek topologiczny optymalny leksykograficznie  $L = l_1, l_2, \dots, l_n$ , taki że  $T \neq L$ . Rozważmy pierwszą, najbardziej znaczącą dla porządku leksykograficznego, pozycję  $i$  na której rozwiązania się różnią. Zauważmy, że:

Wszystkie elementy  $l_1, \dots, l_{i-1}$  muszą:

- zawierać wszystkie wierzchołki będące początkami krawędzi do  $l_i$ ;
- wszystkie wierzchołki nie posiadające z  $l_i$  żadnej krawędzi, ale będące optymalnym wyborem względem kryterium leksykograficznego;
- po usunięciu z grafu  $G$  wszystkich krawędzi wychodzących z wierzchołków  $l_1, l_2, \dots, l_{i-1}$ , stopień wchodzący wierzchołka  $l_i$  wynosi 0

Zatem  $l_1, l_2, \dots, l_{i-1} = t_1, t_2, \dots, t_{i-1}$ , oraz  $t_i > l_i$ . Oznacza to, że w naszym rozwiązaniu  $T$ , po usunięciu wszystkich krawędzi prowadzących z wierzchołków  $t_1, t_2, \dots, t_{i-1}$ , istniał wierzchołek o stopniu wchodzącym 0, którego indeks był mniejszy niż  $t_i$ . A to daje nam sprzeczność, ponieważ na tym etapie wybieramy do rozwiązania wierzchołek o najmniejszym indeksie.

## 1.4.

Niech  $u$  i  $v$  będą dwoma wierzchołkami w grafie nieskierowanym  $G = (V, E, c)$ , gdzie  $c : E \rightarrow R_+$  jest funkcją wagową. Mówimy, że droga z  $u = u_1, u_2, \dots, u_{k-1}, u_k = v$  z  $u$  do  $v$  jest sensowna, jeżeli dla każdego  $i = 2, \dots, k$  istnieje droga z  $u_i$  do  $v$  krótsza od każdej drogi z  $u_{i-1}$  do  $v$  (przez długość drogi rozumiemy sumę wag jej krawędzi). Ułóż algorytm, który dla danego  $G$  oraz wierzchołków  $u$  i  $v$  wyznaczy liczbę sensownych dróg z  $u$  do  $v$ .

Wywołujemy algorytm **Dijkstry** dla wierzchołka  $v$ . Algorytm działa w czasie  $O(|E| + |V|\log|V|)$ . Otrzymujemy tym samym informację o długości najkrótszych dróg prowadzących z każdego wierzchołka grafu do wierzchołka  $v$ . Niech  $S : V \rightarrow R_+$  będzie funkcją zwracającą najkrótszą drogę z wierzchołka  $V$  do  $v$ . Wtedy: Niech  $T$  będzie listą wierzchołków grafu, posortowaną niemalejąco według długości najkrótszej drogi  $S$ . Niech  $x$  w tej tablicy będą początkowo 0 i oznaczają liczbę sensownych dróg.

$T = [(d_1, S(d_1), x_1), (d_2, S(d_2), x_2), \dots, (d_{n-1}, S(d_{n-1}), x_{n-1})]$

Teraz, przechodząc listę  $T$  od lewej strony do prawej:

- jeżeli nasz  $x_i$  jest incydentalny z wierzchołkiem  $v$ , to dodajemy do  $x_i$  jedynekę. Oznacza ona sensowną drogę, będącą najkrótszą drogą prowadzącą z  $d_i$  do  $v$ .
- do naszego  $x_i$  dodajemy  $x_j$  wszystkich wierzchołków leżących na lewo ( $j < i$ ), z którymi nasz wierzchołek jest incydentalny.

Drogi takie oznaczają wszystkie drogi spełniające warunek: dla każdego  $i = 2, \dots, k$  droga z  $u_i$  do  $v$  jest krótsza od każdej drogi z  $u_{i-1}$  do  $v$  (czyli w szczególności od drogi najkrótszej).

Po przejściu w ten sposób całej tablicy, znamy ilość sensownych dróg z każdego wierzchołka do wierzchołka  $v$ .

Dijkstra kosztuje  $O(|E| + |V|\log|V|)$ , sortowanie listy  $O(|V|\log|V|)$ , listę przeglądamy liniowo względem długości  $O(|V|)$ , dodawań  $x'$ ów mamy, co najwyżej,  $O(|E|)$ . Razem  $O(|E| + |V|\log|V|)$ .

## 1.5.

Ułóż algorytm, który dla zadanego acyklicznego grafu skierowanego  $G$  znajduje długość najdłuższej drogi w  $G$ . Następnie zmodyfikuj algorytm tak, by wypisywał drogę o największej długości (jeśli jest kilka takich dróg, to Twój algorytm powinien wypisać dowolną z nich).

LengthTo - tablica  $|V(G)|$  elementów początkowo równych 0.

TopOrder( $G$ ) - posortowane topologicznie wierzchołki grafu  $G$ .

```

for each  $v$  in topOrder( $G$ ) do  $|v|$ 
    for each edge  $(v, w)$  in  $E(G)$  do  $|w|$ 
        if LengthTo[ $w$ ]  $\leq$  LengthTo[ $v$ ] + weight( $v, w$ ) then
            LengthTo[ $w$ ] = LengthTo[ $v$ ] + weight( $v, w$ )

return max(LengthTo[ $v$ ] for  $v$  in  $V(G)$ )

```

Przechodzimy topologicznie wierzchołki grafu, dzięki temu, przechodząc listę wierzchołków wiemy, że rozpatrzyliśmy już wszystkie wierzchołki które prowadzą do aktualnie przeglądanej wierzchołka. Czyli, dla wierzchołka  $v$  rozpatrzyliśmy wszystkie drogi do niego prowadzące, i znamy najdłuższą.

Dla każdego wierzchołka na liście sprawdzamy, czy wierzchołek do którego możemy z niego dojść posiada dłuższą drogę. W ten sposób zachłannie wybieramy zawsze najdłuższą drogę jaką można dojść do każdego wierzchołka, a ponieważ robimy to w porządku topologicznym

Sortowanie topologiczne działa w czasie  $O(E + V)$ , więc całość działa w czasie  $O(E + V + E + V) = O(E + V)$ .

Aby wypisać drogę musimy tylko zapamiętać, dla których wierzchołków spełniony był warunek:

Prev - tablica początkowo równa *nil*.

```

def lognest G
    for each vertex  $V$  in topOrder( $G$ ) do
        for each edge  $(V, W)$  in  $E(G)$  do
            if LengthTo[ $W$ ]  $\leq$  LengthTo[ $V$ ] + weight( $G, (V, W)$ ) then
                LengthTo[ $W$ ] = LengthTo[ $V$ ] + weight( $G, (V, W)$ )
                Prev[ $W$ ] =  $V$ 

    return max(LengthTo[ $V$ ] for  $V$  in  $V(G)$ )

def longest_path G
     $x$  = lognest( $G$ )
    do
        print  $x$ 

```

```
        x = Prev[x]
until x.nil?
```

### 1.5.1. szkic dowodu poprawności kryterium optymalizacyjnego

Załóżmy że w *itym* kroku nasz algorytm wylicza nieoptymalną długość drogi do wierzchołka *a*. Ma to znaczenie tylko w momencie, kiedy rozpatrujemy krawędzie wychodzące z *a*. Ponieważ, rozpatrzyliśmy do tego momentu wszystkie wierzchołki grafu prowadzące do *a*, sprawdziliśmy długość wszystkich możliwych dróg prowadzących do *a* i zachłannie wybraliśmy największą.

## 1.6.

## Rozdział 2

### Lista 2.

#### 2.1.

Przeczytaj notatkę o algorytmach zachłannych...

#### 2.2.

Danych jest  $n$  odcinków  $I_j = \langle p_j, k_j \rangle$ , leżących na osi  $OX$ ,  $j = 1, \dots, n$ . Uałoś algorytm znajdujący zbiór  $S \subseteq \{I_1, \dots, I_n\}$ , nieprzecinających się odcinków, o największej mocy.

Sortujemy odcinki rosnąco wg.  $k_j$ . Wybieramy pierwszy, potem kolejny, który się zmieści po pierwszym.

Dowód: Niech nasz algorytm daje uporządkowanie  $U$ . Załóźmy, że istnieje lepsze, optymalne uporządkowanie  $S$ . Weźmy pierwszą parę odcinków, które różnią się w obu tych uporządkowaniach, tzn.  $U_i \neq S_i$ . Jeżeli odcinek  $S_i$  kończy się wcześniej niż  $U_i$  to zostałby on wybrany przez nasz algorytm. Jeśli kończy się na tej samej pozycji to nie ma znaczenia, który z nich został wybrany. Podobnie, indukcyjnie można zastosować to rozumowanie dla pozostałych par różnych odcinków w obu uporządkowaniach. Algorytm zachłanny daje więc rozwiązanie optymalne.

#### 2.3.

Rozważ następującą wersję problemu wydawania reszty: dla danych liczb naturalnych  $a, b$  ( $a \leq b$ ) chcemy przedstawić ułamek  $\frac{a}{b}$  jako sumę różnych ułamków o licznikach równych 1. Udowodnij, że algorytm zachłanny zawsze daje rozwiązanie. Czy zawsze jest to rozwiązanie optymalne (tj. o najmniejszej liczbie składników)?

Wytwarzanie ułamka egipskiego mniejszego od  $\frac{x}{y}$  o największym mianowniku:

$$\frac{x}{y} \rightarrow \frac{1}{\lceil \frac{y}{x} \rceil}$$

Licznik wyrażenia  $\frac{x}{y} - \frac{1}{\lceil \frac{y}{x} \rceil} = \frac{x\lceil \frac{y}{x} \rceil - y}{x\lceil \frac{y}{x} \rceil}$  będzie mała, lecz zawsze będzie  $\geq 0$ :

Sprawdźmy, czy faktycznie  $0 \leq x \lceil \frac{y}{x} \rceil - y < x$

$$x \lceil \frac{y}{x} \rceil - y < x \Leftrightarrow x \lceil \frac{y}{x} \rceil - x < y \Leftrightarrow \lceil \frac{y}{x} \rceil - 1 < \frac{y}{x}.$$

Co jest oczywiście prawdą. Dodatkowo:  $x \lceil \frac{y}{x} \rceil - y \geq 0$ . Więc ułamek zawsze się zmniejsza, i zatrzyma się na 0.

Kontrprzykład na optymalność. Nasz da jakieś gówno, optymalny da:

$$\frac{5}{121} = \frac{1}{33} + \frac{1}{121} + \frac{1}{363}$$

## 2.4.

Udowodnij poprawność algorytmu Boruvki Solina.

### 2.4.1. Lemat 1

W każdym momencie działania algorytmu, oraz po jego zakończeniu w  $E'$  nie będzie cyklu.

Dowód: Załóżmy nie wprost, że podczas działania algorytmu w którymś etapie pojawiła się spójna składowa, w której jest cykl. Oznaczmy ją jako  $S$ . Rozważmy następujące sytuacje:

- $S$  powstała przez połączenie dwóch superwierzchołków  $v_1$  i  $v_2$ . Oznacza to, że do zbioru  $E'$  zostały dołączone krawędzie  $e_i$  i  $e_j$ . Ponieważ  $e_i$  została dołączona jako najbliższa krawędź incydentna do  $v_1$  więc  $C(e_i) < C(e_j)$ . Ale skoro  $e_j$  została dołączona jako najbliższa krawędź incydentna do  $v_2$  to musi zachodzić (Pamiętajmy, że w grafie nie ma krawędzi o takiej samej wadze) - sprzeczność.
- $S$  powstała przez połączenie się trzech lub więcej superwierzchołków. Podzielmy powstały cykl  $C$  na następujące części: niech  $v_1, \dots, v_l$  będą kolejnymi superwierzchołkami należącymi do  $C$  a  $e_1, \dots, e_l$  będą kolejnymi krawędziami należącymi do  $C$ , które zostały dodane w zakończonym właśnie etapie algorytmu. W  $C$  krawędzie  $e_i$  oraz superwierzchołki  $v_i$  występują na przemian. Z zasady działania algorytmu możemy stwierdzić, że aby powstał taki cykl, musi zachodzić  $C(e_1) < C(e_2) \dots < C(e_l) < C(e_1)$  - Sprzeczność.

### 2.4.2. Lemat 2

W każdym etapie działania algorytmu otrzymujemy dla każdego superwierzchołka minimalne drzewo rozpinające.

Dowód:

- Gdy zostanie zakończony etap 1:

Założmy, że istnieje taki superwierzchołek  $v_i$ , który nie jest minimalnym drzewem rozpinającym poddrzewa złożonego z wierzchołków należących do  $v_i$ . Weźmy więc takie minimalne drzewo rozpinające  $T$ . Istnieje krawędź  $e_i$  taka, że  $e_i \in E(v_i)$  oraz  $e_i \notin E(T)$ . Dodajmy  $e_i$ . W  $T$  powstał cykl. Ponieważ  $e_i$  jest incydentna do pewnego wierzchołka z tego cyklu, istnieje więc inna krawędź  $e'_i$  incydentna do tego samego wierzchołka. Jednak z tego, że  $e_i \in E(v_i)$  wynika, że  $C(e_i) < C(e'_i)$ . Jeśli usuniemy krawędź  $e'_i$  z  $T$  otrzymamy mniejsze drzewo rozpinające, co jest sprzeczne z założeniem o minimalności  $T$ .



- Gdy zostanie zakończony etap 2:

Z poprawności etapu 1 wiemy, że istnieje takie wywołanie etapu 2, w którym każdy z superwierzchołków jest minimalnym drzewem rozpinającym. Jest to choćby pierwsze wywołanie. Załóżmy zatem, że dla pewnego wywołania tego etapu otrzymano superwierzchołki będące minimalnymi drzewami rozpinającymi, jednak scalało przynajmniej dwa z nich w taki sposób, że dało się otrzymać mniejsze drzewo rozpinające. Niech etap  $k$ -ty będzie pierwszym takim etapem, w którym coś się popsło. Niech  $E'_1$  będzie zbiorem krawędzi przed wywołaniem etapu  $k$ , a  $E'_2$  będzie zbiorem krawędzi po jego wywołaniu. Niech  $T$  będzie minimalnym drzewem rozpinającym takim, że  $V(T) = V(v_i)$ , ale że  $E(T) \neq E(v_i)$ . Istnieje więc krawędź  $e_i \in E(v_i)$  oraz  $e_i \notin E(T)$ .

Fakt: Krawędź dodana podczas  $k$ -tego wywołania. (Nie może należeć do  $E'_1$  gdyż inaczej superwierzchołek do którego by należała nie byłby minimalnym drzewem rozpinającym, co jest sprzeczne z dowodem dla pierwszego etapu i założeniem, że wywołanie  $k$ -te jest najmniejszym wywołaniem, które zwróciło nieoptymalne drzewa) Dodajmy krawędź  $e_i$  do  $E(T)$ . W  $T$  powstał cykl. Ponieważ  $e_i$  jest najmniejszą krawędzią incydentną do pewnego superwierzchołka z tego cyklu, istnieje więc inna krawędź incydenta do tego samego superwierzchołka. Jednak jej waga jest większa niż waga krawędzi  $e_i$ , zatem zastąpienie jej krawędzią  $e_i$  da nam mniejsze drzewo rozpinające co jest sprzeczne z założeniem o optymalności  $T$ .

## 2.5.

Ułóż algorytm, który dla danego spójnego grafu  $G$  oraz krawędzi  $e$  sprawdza w czasie  $O(n + m)$ , czy krawędź  $e$  należy do jakiegoś minimalnego drzewa spinającego grafu  $G$ . Możesz założyć, że wszystkie wagi krawędzi są różne.

## 2.6.

System złożony z dwóch maszyn  $A$  i  $B$  wykonuje  $n$  zadań. Każde z zadań wykonywane jest na obydwu maszynach, przy czym wykonanie zadania na maszynie  $B$  można rozpocząć dopiero po zakończeniu wykonywania go na maszynie  $A$ . Dla każdego zadania określone są dwie liczby naturalne  $a_i$  i  $b_i$  określające czas wykonania  $i$ -tego zadania na maszynie  $A$  oraz  $B$  (odpowiednio). Ułóż algorytm ustawiający zadania w kolejności minimalizującej czas zakończenia wykonania ostatniego zadania przez maszynę  $B$ .