Lista 1

tags: `AiSD`

Zadanie 1

Liczba wierzchołków

```
def count(v):
    if leaf(v):
        return 1
    else:
        return count(v.left) + count(v.right) + 1

return count(root)
```

Uzasadnienie

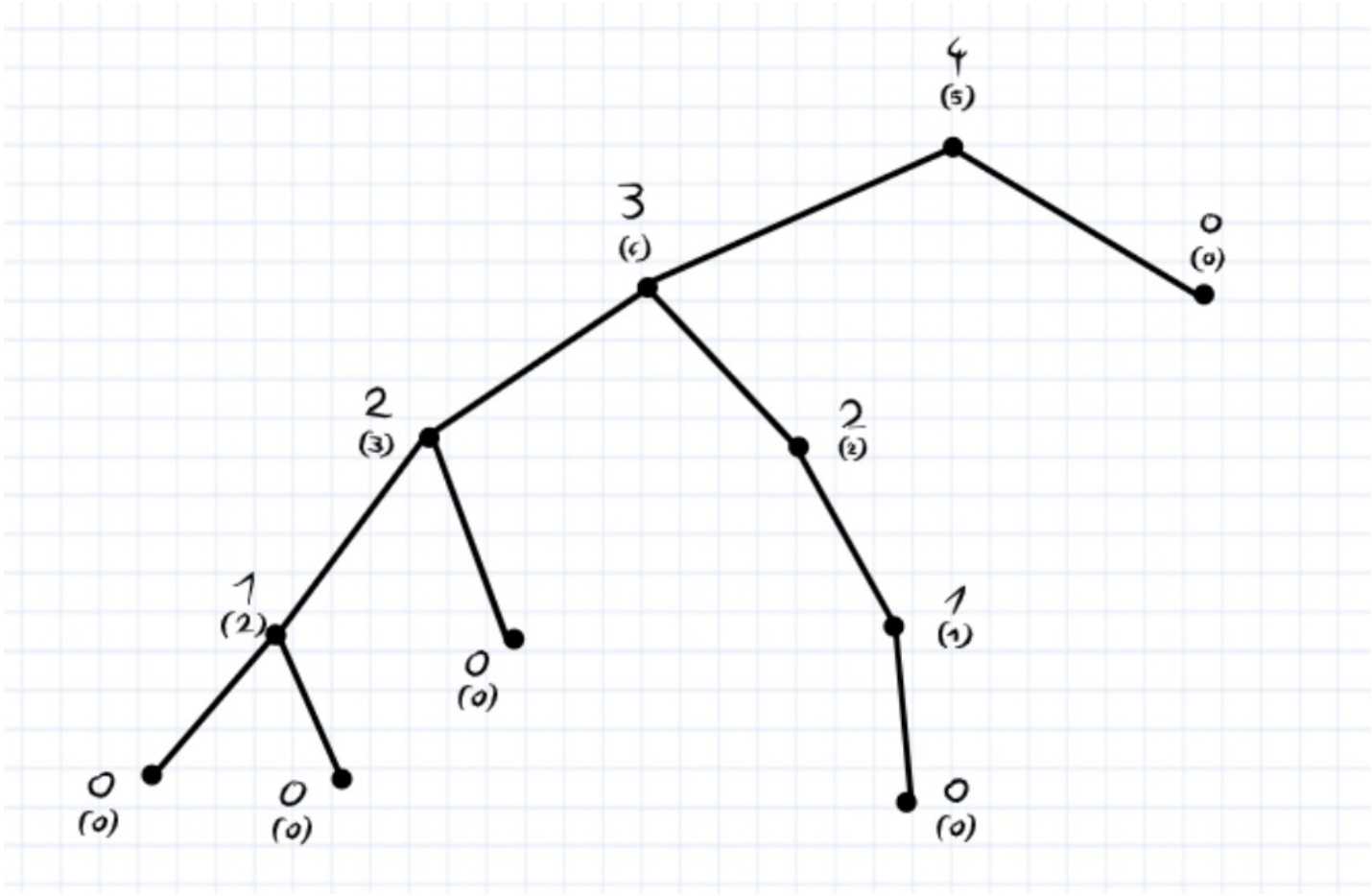
Funkcja zwraca sumę wierzchołków w lewym i prawym poddrzewie powiększoną o wierzchołek, z którego wychodzi. Wywołana dla korzenia zwróci sumę wszystkich wierzchołków w drzewie.

Maksymalna odległość

```
len = 0

def rec(v):
     if leaf(v):
          return  0
     if !v:
          return  -1

     left = rec(v.left)
     right = rec(v.right)

     max_len = left + right + 2
     if max_len > len:
          len = max_len

     return  max(left, right) + 1

return rec(root)
```

Uzasadnienie

Funkcja zwraca odległość najdalszego wierzchołka w poddrzewie licząc od `v`. Przy każdym przejściu obliczamy sumę odległości lewego i prawego poddrzewa, oraz dwóch krawędzi prowadzących do `v` -- czyli największą odległość między wierzchołkami w poddrzewie. Teraz wystarczy tylko spamiętać największą taką wartość spośród wszystkich wywołań funkcji.



Zadanie 2

Idea rozwiązania

Tworzymy kopiec minimalny(H) i maksymalny(L). Całość implementujemy jako pojedynczą tablicę, gdzie parzyste elementy są wierzchołkami kopca L, a nieparzyste kopca H. Mając dany wierzchołek i możemy odnieść się do jego potomków/rodziców w następujący sposób:

```
//K to kopiec minimaksowy
def children(K, i):
    if i % 2 == 0:
        return ( K[2*i], K[2*i + 2] )
    else:
        return ( K[2*i + 1], K[2*i + 3] )

def parent(K, i):
    if i % 2 == 0:
        return (i div 4) * 2
    else:
        return ((i+1) div 4) * 2 - 1
```

```
def jump(K, i):
    if i % 2 == 0:
        h = i-1
        l = i
    else:
        h = i
        l = i+1

    //poprawiamy, jeśli nie ma pary
    if K[l] == null:
        l = parent(K, l)
    if K[h] == null:
        h = parent(K, h)

    if l < h:
        K[l] <-> K[h]
        move_up_H(K, h)
        move_up_L(K, l)
```

Pseudokod

```
def move_up_H(K, i):
    k <- i
    repeat
        j <- k
        if j > 1 and K[parent(j)] > K[k]:
            k <- parent(j)
        K[j] <-> K[k]
    until j == k

//to samo tylko nierówności się odwracają
def move_up_L(K, i):
    k <- i
    repeat
        j <- k
        if j > 2 and K[parent(j)] < K[k]:
            k <- parent(j)
        K[j] <-> K[k]
    until j == k



def move_down_H(K, i):
    k <- i
    repeat
        j<-k
        left_child = children(K, j)[0]
        right_child = children(K, j)[1]
        if left_child <= n and K[left_child] < K[k]:
            k <- left_child
        if right_child < n and K[right_child] < K[k]:
            k <- right_child
        K[j] <-> K[k]
    until j == k
    jump(k)

//to samo tylko nierówności się odwracają
def move_down_L(K, i):
    k <- i
    repeat
        j<-k
        left_child = children(K, j)[0]
        right_child = children(K, j)[1]
        if left_child <= n and K[left_child] > K[k]:
            k <- left_child
        if right_child < n and K[right_child] > K[k]:
            k <- right_child
        K[j] <-> K[k]
    until j == k
    jump(k)



def set(K, i, u):
    x <- K[i]
    K[i] <- u

    if i % 2 == 0:
        if u < x:
            move_up_H(K, i)
        else:
            move_down_H(K, i)

    else:
        if u < x:
            move_down_L(K, i)
        else:
            move_up_L(K, i)

def removemin(K):
    set(1, n)

def removemax(K):
    set(2, n)
```

Zadanie 3

Idea rozwiązania

W rozwiązaniu wykorzystamy zmodyfikowany algorytm Kahn'a. Zaczniemy od przygotowania sobie kopca wyposażonego w operację wyciągania największego elementu. Teraz dla zadanego grafu G będziemy szukać wierzchołków bez krawędzi wychodzących. Znalezione wierzchołki będą umieszczane w kopcu i usuwane z G. Dla wierzchołków w kopcu będziemy powtarzać następującą procedruę:

- wyciągamy największy wierzchołek z kopca
- umieszczamy go na początku listy wynikowej
- przeglądamy jego rodziców
- dla każdego rodzica, którego liczba krawędzi wychodzących wynosi zero usuwamy go z grafu i wrzucamy na kopiec

Pseudokod

```
 K <- pusty kopiec maksymalny
W <- pusta lista wynikowa

for v in G:
    if liczba potomków v == 0:
        G.remove(v)
        K.add(v)

while not K.empty():
    mxv = K.removemax()
    W.push_front(mxv)

    for rodzic in mxv.rodzice:
        if liczba potomkow rodzic == 0:
            G.remove(rodzic)
            K.add(rodzic)

return W
```

Lemat 1. Każdy DAG G ma przynajmniej jeden wierzchołek ze stopniem wychodzącym 0.

Dowód: Weźmy najdłuższą ścieżkę w G. Musi mieć skończoną długość, bo graf jest acykliczny. Niech v będzie jej końcem. Nie ma on wychodzących krawędzi do żadnego wierzchołka z tej ścieżki, bo powstałby cykl, ani do żadnych innych, bo wtedy możnaby przedłużyć ścieżkę. Czyli $outdeg(v) = 0$.

Porządek topograficzny

Ze zbioru wierzchołków o stopniu wychodzącym 0 na pewno możemy wybrać jeden i dodać do rozwiązania w dowolnej kolejności, gdyż nie ma wierzchołków, które musiałyby się znajdować po nich. Po usunięciu dostajemy mniejszy DAG, który z lematu 1 również musi mieć jakieś wierzchołki stopnia 0 i możemy powtórzyć dla niego rozumowanie.

Porządek leksykograficzny

W dowolnym momencie na kopcu znajdują się wszystkie wierzchołki, które mogą zostać dołączone do rozwiązania, my wybieramy ten największy zapewniając minimalność w porządku leksykograficznym.

Zadanie 4

tags: `AiSD`, `lista1`

Wykorzystamy algorytm Dijkstry do przypisania wszystkim wierzchołkom odległości od v. Wiedząc, że nasz algorytm znajduje jednynie najkrótsze ścieżki do v, to szukając sensownych ścieżek wystarczy wybierać takie, w których odległości wierzchołków są rosnące (licząc od końca ścieżki).

```
//0, 1 to wierzchołki odpowiednio początkowy i końcowy.


dist = [ tablica wszystkich odległości wierzchołków od v]
G = [ graf jako lista sąsiedztwa ]
sensowne = [ liczba sensownych dróg z każdego wierzchołka do v (początkowo same nulle)]

def rec(u):

    if sensowne(u) != null: //czy spamiętane
        return sensowne(u)

    if u == 0: //0, 1 to wierzchołki odpowiednio początkowy i końcowy.
        return 1

    //suma
    s = 0
    for w in G[u]:
        if dist[w] > dist[u]:
            s += rec(w)

    sensowne(u) = s //spamiętywanie
    return s

return rec(1)
```

Złożoność obliczeniowa

$O(E*log(V) + E + V)$

Zadanie 5

tags: `AiSD`, `lista1`

Najdłuższa ścieżka będzie zaczynać się w wierzchołku s o stopniu wchodzącym 0, inaczej możnaby ją przedłużyć o inny wierzchołek, który ma krawędź do niego, lub jeśli byłby on już na ścieżce, to istniałby cykl.

Działamy na acyklicznym grafie skierowanym. To oznacza, że istnieje dla niego porządek topologiczny. Sortujemy topologicznie wierzchołki, dzięki czemu konstruując tablicę najdłuszych dróg wiemy, że gdy skończymy wyznaczać drogę dla danego jej wierzchołka ta droga nie zostanie przedłużona przez dalsze wierzchołki w tablicy.

```
dist = [0, 0, 0, ..., 0] // jest ich tyle co |V|

ordered = topologicalSort(graph)
for v in ordered:
    for u in neighbours(v):
        if dist(u) <= dist(v) + 1:
            dist(u) = dist(v) + 1

return max(dist)
```

Wersja rozszerzona o wypisywanie drogi

```
dist = [[], [], [], ..., []] // jest ich tyle co |V|

ordered = topologicalSort(graph)
for v in ordered:
    for u in neighbours(v).append(v): //na potrzeby zwracania pełnej drogi v jest swoim sąsiadem
        if dist(u).length <= dist(v).length + 1:
            dist(u) = dist(v).append(u)


return max(dist)
```

Dowód

T(n) := dla każdego wierzchołka v (1 <= v <= n) dist[v] to najdłuzsza możliwa droga ze źródła do v.

Podstawa (n=1):

Odwiedzamy jeden wierzchołek, jest nim źródło. Jego odległość od źródła to 0.

Krok (T(n) => T(n+1)):

Wybieramy (n+1)-szy wierzchołek v. Spośród wszystkich wierzchołków mających krawędź do v wybieramy ten o najdłuższej drodze do źródła. Taka droga powiększona o 1 to najdłuższa droga z v.

Złożoność Czasowa

- Sortowanie topologiczne $O(n)$
- Przejście po grafie $O(n + m)$
- Przejście po tablicy dla wypisania wyniku $O(n)

Złożoność Pamięciowa

Działamy na liście sąsiedztwa oraz tablicy $O(n+m)$