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 (0,0) 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 minimaksowego. 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[k] 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(A[m], A[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 niezmieniona). 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 $1, 2, 3, ..., |V|$; 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 $O(log|V|)$ zamaist $O(1)$, zatem złożoność wynosi: $O(|E| + |V|log|V|)$.

1.3.1. dowód poprawności

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

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

Wszystkie elementy $l_1, \ldots, l_{i-1}$ 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 $l_1, l_2, \ldots, l_{i-1}$, stopień wchodzący wierzchołka l_i wynosi 0

Zatem $l_1, l_2, \ldots, l_{i-1} = t_1, t_2, \ldots, t_{i-1}$, oraz $t_i > l_i$. Oznacza to, że w naszym rozwiązaniu T, po usunięciu wszystkich krawędzi prowadzących z wierzchołków $t_1, t_2, \ldots, t_{i-1}$, 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 $c : E \to R_+$ jest funkcją wagową. Mówimy, że droga z $u = u_1, u_2, \ldots, u_{k-1}, u_k = v$ z u do v jest sensowna, jeżeli dla każdego $i = 2, \ldots, k$ 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 $S : V \to R_+$ 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.

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

Teraz, przechodząc listę T od lewej strony do prawej:

- jeżeli nasz x_i jest incydentalny z wierzchołkiem v, to dodajemy do x_i jedynkę. 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 $i = 2, \ldots, k$ 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.

Djikstra kosztuje $O(|E| + |V|log|V|)$, sortowanie listy $O(|V|log|V|)$, listę przeglądamy liniowo względem długości $O(|V|)$, dodawań $x'w$ 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] <= 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 przeglądanego 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] <= 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 = longest(G)
        do
              print x
```

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

1.5.1. szkic dowodu poprawności kryterium optymalizacyjnego

Załóżmy że w $i$tym 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.