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)
```

Złożoność wynosi $O(|E| + |V|log|V|)$.

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.

Djikstra i lecimy wyniki na lewo. TBC

1.5.

Wejście: Skierowany acykliczny graf.

LengthTo - tablica $|V(G)|$ elementów początkowo równych 0
TopOrder(G) - posortowane topologicznie wierzchołki.

```
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))

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

Sortowanie topologiczne działa w czasie $O(E + V)$, więc całość działa w czasie $O(E + V + E + V) = O(E + V)$. a żeby wypisać drogę musimy tylko zapamiętać, dla których wierzchołków spełniony był IF.

1.6.

1.7. Druga wersja zadania 1.5.

w wersji O(n+m) Q - kolejka wierzchołków o indeg = 0 P - kolejka wierzchołków posortowanych topologicznie T - wyzerowana tablica o dlugosci —V—

```
while !Q.empty:
v = Q.pop();
   P.;push_back(v);
      foreach e=(v,u) in G:
            delete(e);
                  if(in_deg(u) == 0):
```

```
Q.push(u);

foreach v in P:
    foreach (v,u) in G:
        if T[u] < T[v] + w(v,u):
            T[u] = T[v] + w(v,u);

return max(T[v] for v in G)
```

żeby była rekonstrukcja ścieżki to tak: wierzchołek końcowy będziemy mieli po wyznaczeniu z tego maxa więc potrzebna nam tylko tabela poprzedników PREV[v] dla v w G i tą tablicę wystarczy wypełniać przy podstawianiu nowej wartości pod T[u]