Zadanie 1.

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

• liczbę wierzchołków w T

```
Count(T)
if T is leaf
	return 0
return Count(T->Left) + Count(T->Right) + 1
```

• maksymalną odległość między wierzchołkami w T

Idea: Dla każdego wierzchołka wyliczamy głębokość jego lewego i prawego poddrzewa a następnie sprawdzamy, czy ich suma jest większa od dotychczas najdłuższej znalezionej ścieżki.

```
MaxDistance(T)
[depth, distance] = Depth(T)
return max(depth, distance)

Depth(T) // zwraca parę – głębokość drzewa i długość najdłuższej ścieżki
if T is leaf
	return [0, 0]
depth1, depth2 = 0 // głębokości lewego i prawego poddrzewa
dist1, dist2 = 0 // maksymalna odległość między wierzchołkami w poddrzewach
distance = 0
if T->Left != null
	[depth1, dist1] = Depth(T->Left)
	distance = distance + depth1 + 1
if T->Right != null
	[depth2, dist2] = Depth(T->Right)
	distance = distance + depth2 + 1
return [max(depth1, depth2) + 1, max(distance, dist1, dist2)]
```

Zadanie 2.
Napisz w pseudokodzie procedury:
• przywracania porządku
• usuwania minimum
• usuwania maksimum
z kopca minimaksowego. Przyjmij, że elementy tego kopca pamiętane są w jednej tablicy (określ w jakiej kolejności).
Użyj pseudokodu na takim samym poziomie szczegółowości, na jakim zostały napisane w Notatce nr 2 odpowiednie procedury dla zwykłego kopca.

1. Wersja z poziomami minmax na zmianę

Kopiec minimaksowy zawiera w korzeniu najmniejszą wartość, a w jednym z synów korzenia wartość maksymalną. Indeksując poziomy od 0, na poziomach parzystych zawiera elementy minimalne, a na poziomach nieparzystych maksymalne. Każde poddrzewo kopca zawiera w korzeniu albo wartość minimalną, albo maksymalną.

```
RemoveMin(K) // usuwanie najmniejszego elementu kopca
K[1] = K[n]
PushDown(K, 1)

RemoveMax(K) // usuwanie największego elementu kopca
maksimum = max(K[2], K[3])
K[maksimum] = K[n]
PushDown(K, maksimum)

PushDown(K, index) // porządkowanie kopca
depth = ⌊log_2(index)⌋ mod 2
if depth = 0 // poziom minimów
        PushDownMin(K, index)
else // poziom maksimów
        PushDownMax(K, index)
```

```
PushDownMin(K, index)
if 4 * index <= n // istnieją 2 poziomy niżej
        minimum = min(K[2*index, 2*index + 1, 4 * index, …, 4 * index + 3])
        if minimum >= 4 * index // minimum to wnuk
                if K[minimum] < K[index] // wnuk mniejszy od dziadka
                        Swap(K[index], K[minimum])
                        if K[minimum] > K[minimum div 2] // większy od rodzica
                                Swap(K[minimum], K[minimum div 2])
                        PushDownMin(K, minimum)
        else // minimum to syn
                if K[minimum] < K[index]
                        Swap(K[index], K[minimum])
else if 2 * index <= n // istnieje 1 poziom niżej
        minimum = min(K[2 * index, …, 2 * index + 1])
        if K[minimum] < K[index]
                Swap(K[index], K[minimum])
```

PushDownMax(K, index) analogicznie

2. Wersja z 2 kopcami diamentowymi L, H (z notatki 2)
Mamy 2 kopce – kopiec maksymalny H zawierający $\left\lceil \frac{n}{2} \right\rceil$ elementów oraz kopiec minimalny L zawierający $\left\lfloor \frac{n}{2} \right\rfloor$ elementów.
Między liśćmi tych kopców istnieją krawędzie oraz każda ścieżka z korzenia L do korzenia H jest niemalejąca. Indeksujemy elementy według kolejności usuwania od końca – element o indeksie i będzie usunięty jako i-ty ostatni. Zatem korzeń H będzie miał indeks 1, korzeń L indeks 2, a kolejne 4 indeksy to lewy syn H, lewy syn L, prawy syn H, prawy syn L itd…
Dzięki takiemu indeksowaniu uzyskamy 2 efekty – zawsze usuwamy element o największym indeksie (nie tworzymy „dziur” w tablicy) oraz zawsze zachowamy równoliczność kopców L,H.

```
RemoveMax(K) // usuwanie największego elementu kopca H
K[1] = K[n] // zapisujemy w korzeniu H wartość z ostatniego liścia L/H
P – parent of K[n]
change child of P from K[n] to K[n-1]
remove K[n] // usuwamy ostatni liść L/H
PushDown(K, 1) // przesuwamy wartość z korzenia H w dół
```

```
RemoveMin(K) // usuwanie najmniejszego elementu kopca L
K[2] = K[n] // zapisujemy w korzeniu L wartość z ostatniego liścia L/H
P – parent of K[n]
change child of P from K[n] to K[n-1]
remove K[n] // usuwamy ostatni liść L/H
PushUp(K, 2) // przesuwamy wartość z korzenia L w górę

PushDown(K, index) // przesuwanie wartości z korzenia H na właściwe miejsce
if index % 2 = 1 and index * 2 + 3 ≤ n // wierzchołek z H ma obu synów w H
        maximum = max(K[2*index+1], K[2*index+3]) // większy z synów
        if K[maximum] > K[index] // przesuwamy w dół wewnątrz H
                Swap(K[index], K[maximum])
                PushDown(K, maximum)
else if index % 2 = 1 and index * 2 ≥ n // wierzch. jest liściem H i ma syna w L
        if K[index+1] > K[index] // przesuwamy w dół do L
                Swap(K[index], K[index+1])
                PushDown(K, index+1)
else if index % 2 = 1 // wierzchołek z H ma lewego syna w H oraz prawego w L
        maximum = max(K[2*index+1], K[2*index+3]) // większy z synów
        if K[maximum] > K[index] // przesuwamy w dół wewnątrz H
                Swap(K[index], K[maximum])
                PushDown(K, maximum)
else if index % 4 = 0 // wierzchołek z L będący lewym synem
        if K[index/2] > K[index] // przesuwamy w dół wewnątrz L
                Swap(K[index], K[index/2])
                PushDown(K, index/2)
else if index % 4 = 2 and index != 2 // wierzchołek z L będący prawym synem
        if K[(index-2)/2] > K[index] // przesuwamy w dół wewnątrz L
                Swap(K[index], K[(index-2)/2])
                PushDown(K, (index-2)/2)
```

PushUp zdefiniowany jest podobnie, tylko zmieniają się operatory > na < oraz indeksy:

```
PushUp(K, index) // przesuwanie wartości z korzenia L na właściwe miejsce
if index % 2 = 0 and index * 2 + 2 ≤ n // wierzchołek z L ma obu synów w L
	minimum = min(K[2*index], K[2*index+2]) // mniejszy z synów
	if K[minimum] < K[index] // przesuwamy w górę wewnątrz L
		Swap(K[index], K[minimum])
		PushUp(K, minimum)
else if index % 2 = 0 and index * 2 ≥ n − 1 // jest liściem L i ma syna w H
	if K[index-1] < K[index] // przesuwamy w górę do H
		Swap(K[index], K[index-1])
		PushUp(K, index-1)
else if index % 2 = 0 // wierzchołek z L ma lewego syna w L oraz prawego w H
	minimum = min(K[2*index], K[2*index+1]) // mniejszy z synów
	if K[minimum] < K[index] // przesuwamy w górę wewnątrz H
		Swap(K[index], K[minimum])
		PushUp(K, minimum)
else if index % 4 = 3 // wierzchołek z H będący lewym synem
	if K[index/2] < K[index] // przesuwamy w górę wewnątrz H
		Swap(K[index], K[index/2])
		PushUp(K, index/2)
else if index % 4 = 1 and index != 1 // wierzchołek z H będący prawym synem
	if K[(index-3)/2] > K[index] // przesuwamy w górę wewnątrz H
		Swap(K[index], K[(index-3)/2])
		PushDown(K, (index-3)/2)
```

Zadanie 3.

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

Najpierw zauważmy, że każdy acykliczny digraf zawiera co najmniej 1 wierzchołek o stopniu wejściowym 0 (inaczej zawierałby cykl).
Zapamiętajmy krawędzie w postaci macierzy sąsiedztwa.
Będziemy wypisywać kolejne wierzchołki o stopniu wejściowym 0, a następnie je usuwać wraz z krawędziami sąsiadującymi.

```
TopologicalSort (V, E)
sorted = []
Q = [] // kolejka wierzchołków o stopniu wejściowym 0
For-each v from V:
        if Deg_in(v) = 0:
                Q.push(v)
While Q is not empty:
        v = Q[0]
        For-each s – neighbour of v:
                remove edge v-s
                Deg_in(s) -= 1
                if Deg_in(s) = 0
                        Q.push(s)
        sorted.push(v)
return sorted
```

Zadanie 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 $u = u_1, u_2, \dots, u_{k-1}, u_k = v$ do v jest sensowna, jeśli dla każdego i = 2, …, 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.

Na początku musimy poznać długość najkrótszej ścieżki do v z każdego wierzchołka. Wtedy sensownymi ścieżkami będą te, dla których koszt(a->c) jest większy niż koszt(a->b) + koszt(b->c).
Zatem wywołujemy Djikstrę zaczynając od końca.

```
Djikstra(G[1,…,n],c, start, end) // zwraca listę kosztów dotarcia do wierzchołka
cost = [] // koszt dotarcia do danego wierzchołka
for i in 1,…,n:
        cost[i] = ∞
cost[start] = 0
Q = [1,…,n] // kolejka priorytetowa wierzchołków do odwiedzenia
```

```
While Q is not empty:
	v = min(Q) // najbliższy wierzchołek do rozważenia
	for-each s – neighbour of v
		// jeśli znaleźliśmy krótszą ścieżkę niż dotychczasowa
		if (cost[s] > cost[v] + E(v,s))
			cost[s] = cost[v] + E(v,s)
return cost

// zwraca ilość „sensownych" ścieżek
CountValidPaths(G[1,…,n], cost[1,…,n], start, end)
if(start = end)
	return 1
count = 0
for-each s – neighbour of start:
// ścieżka jest sensowna, jeśli sąsiad ma bliżej do końca niż akt. wierzchołek
	if (cost[s] < cost[start])
		count += CountValidPaths(G, cost, s, end)
return count

Paths(G[1,…,n],c, u, v)
cost = Djikstra(G, c, v, u)
return CountValidPaths(G, cost, u, v)
```

Zadanie 5.

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

Załóżmy, że krawędzie zapamiętujemy w postaci listy sąsiedztwa.
Każdy wierzchołek zapamiętujemy jako strukturę zawierającą pole v_{path}, oznaczające najdłuższą ścieżkę zaczynającą się w v.

FindPath(G) // najdłuższa ścieżka w grafie skierowanym G
max_path = []
// sortujemy rosnąco wierzchołki względem stopnia wyjściowego
For-each verticle v from G:
    v_{path} = LongestPathFromV(G, v)
    max_path = max(max_path, v_{path})
return max_path
// najdłuższa ścieżka zaczynająca się w wierzchołku v
LongestPathFromV(G, v)
max_path = [v]
for-each s – neighbour of v:
    if $s_{path} = [\emptyset]$
        s_{path} = LongestPathFromV(G, s)
    max_path = max(max_path, $[v] + s_{path}$)
return max_path

Zadanie 6. (1.5pkt)

Dany jest niemalejący ciąg n liczb całkowitych dodatnich $a_1 \leq a_2 \leq \cdots \leq a_n$. Wolno nam modyfikować ten ciąg za pomocą następującej operacji: wybieramy dwa elementy a_i, a_j spełniające $2a_i \leq a_j$ i wykreślamy je oba z ciągu. Ułóż algorytm obliczający, ile co najwyżej elementów możemy w ten sposób usunąć.

Aby otrzymać optymalne rozwiązanie, będziemy każdego $i = 1,2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$ dobierać elementy w pary $\left(a_i, a_{i+\frac{n}{2}}\right), \left(a_{i+1}, a_{i+\frac{n}{2}+1}\right), \ldots$

Jeśli para $\left(a_i, a_{i+\frac{n}{2}}\right)$ nie spełnia warunku, to sprawdzamy parę $\left(a_i, a_{i+\frac{n}{2}+1}\right)$

CountRemoved(A) // funkcja
count = 0
k = 0
for $i = 1,2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$
    hasPair = false
    while $i + k \leq \left\lceil \frac{n}{2} \right\rceil$ and !hasPair
        if $2 * A[i] \leq A\left[\left\lfloor \frac{n}{2} \right\rfloor + i + k\right]$
            count += 2
            hasPair = true
        else   k++
return count

Zadanie 7. (1.5 pkt)

7. (1,5pkt) Dany jest nieskierowany graf ważony $G = (V, E; c)$ z $c : E \to R_+$ oraz ciąg $v_1, v_2, \ldots, v_k$ różnych wierzchołków z V. Niech D_j $(0 \le j \le k)$ będzie sumą długości najkrótszych ścieżek między wszystkimi parami wierzchołków pozostającymi w G po usunięciu wierzchołków $v_1, v_2, \ldots, v_j$ (wraz z wierzchołkiem usuwamy wszystkie incydentne z nim krawędzie).
Ułóż algorytm obliczający wartości $D_0, D_1, \ldots, D_k$.

Liczymy Floyda warshalla bez tych wierzchołków, potem dorzucamy kolejne i dodajemy wyniki.

```
Calculate(G, v[1,…,k])
Sum = 0
M = FloydWarshall(G.V-v, Sum)
for j from k downto 1: // dla każdego v_j
        for-each a - neighbour of v[j] in G.V-v[1,…,j-1]:
                M[a][j] = M[j][a] = cost(G.V[a], v[j])
                Sum += cost(G.V[a], v[j])
        for a from j+1 to n: //relaksacja {v_1, … , v_{j-1}} dzięki v_j
                for b from a+1 to n:
                        if M[a][b] > M[a][j] + M[j][b]:
                                Sum = Sum – M[a][b] + M[a][j] + M[j][b]
                                M[a][b] = M[b][a] = M[a][j] + M[j][b]
        for a from j+1 to n: //relaksacja v_j dzięki {v_1, … , v_{j-1}}
                for b from j+1 to n:
                        if M[a][j] > M[a][b] + M[b][j]:
                                Sum = Sum – M[a][j] + M[a][b] + M[b][j]
                                M[a][j] = M[j][a] = M[a][b] + M[b][j]
        D[j-1] = Sum
return D
```

Zadanie 8.

Ułóż algorytm, który dla danych k uporządkowanych niemalejąco list $L_1, \ldots, L_k$ liczb całkowitych znajduje najmniejszą liczbę r, taką że w przedziale [a, a+r] znajduje się co najmniej jedna wartość z każdej z list Li, dla pewnej liczby a. Twój algorytm nie może modyfikować list Li i powinien być pamięciowo oszczędny (no i oczywiście jak najszybszy).

Idea:

Tworzymy kolejkę priorytetową Q, która zawiera pary (xs, i), gdzie

xs to indeks listy z L, natomiast i to indeks, na którym aktualnie porównywana wartość jest w liście xs. Na początku w Q będą pierwsze elementy z poszczególnych list, a potem w każdym kroku obliczamy wartość $Q.last.xs[i] - Q.first.xs[i]$, przyrównujemy ją z aktualnym r oraz zwiększamy indeks najmniejszej wartości na kolejny.
Program zakończy działanie, gdy wartość minimalna jest ostatnim elementem na swojej liście i zwraca poprawny wynik (patrz Lemat).

FindR(L[1,…,k])
$Q = < (i_1, 0), \ldots, (i_k, 0) >$ // kolejka priorytetowa
$r = +\infty$ // rozwiązanie zadania
While True
    $r' = Q.last.xs[Q.last.i] - Q.first.xs[Q.first.i]$
    $r = \min(r, r')$
    $temp = Q.first$
    remove Q.first
    $temp.i += 1$ // przejście do następnego elementu
    $if(temp.i \geq temp.xs.length)$ // jeśli on nie istnieje, to zwróć r
        return r
    $InsertSort(Q, temp)$ // wstawienie zaktualizowanej pary do Q

Lemat (o zwiększaniu wartości minimalnej Q)
Mając kolejkę priorytetową Q oraz wartość r tej kolejki, jedyną możliwością, aby r’ kolejki Q’ (powstałej z Q poprzez podmianę jednego elementu na kolejny z tej samej listy) było mniejsze od r jest podmiana najmniejszego elementu Q.

Dowód:
Niech Q będzie postaci $(q_1, \ldots, q_i, \ldots, q_k)$
Rozważmy przypadki gdy podmieniamy:
1) q_k, wtedy $q_k' \geq q_k$, skoro q_1 nie zmienia się, to $r' = q_k' - q_1 \geq q_k - q_1 = r$,
2) q_i, wtedy $q_i' \geq q_i$, skoro q_1 nie zmienia się, to albo $r' = r$, gdy $q_i' \leq q_k$, albo $r' > r$ w przeciwnym przypadku,
3) q_1, wtedy rozważmy 3 przypadki wartości q_1':
 a) $q_1' = q_1$ nic się nie zmienia
 b) $q_1 < q_1' \leq q_k$, wtedy q_k nie zmienia się, wtedy

$r' = q_k - \min(q_1', q_2) \leq q_k - q_1 = r$ (równe tylko gdy $q_1 = q_2$)
c) $q_1' > q_k$, wtedy $r' = q_1' - q_2$, co może być zarówno większe od r, gdy $q_1' - q_k > q_2 - q_1$, równe, jak i mniejsze, gdy $q_1' - q_k < q_2 - q_1$
Zatem jak widać, tylko w niektórych przypadkach zmieniania pierwszej wartości możemy poprawić r, co kończy dowód.