Zadanie 2. (2pkt)

Danych jest n prostych $l_1, l_2, \ldots, l_n$ na płaszczyźnie $(l_i = a_i x + b_i)$, takich że żadne trzy proste nie przecinają się w jednym punkcie. Mówimy, że prosta l_i jest widoczna z punktu p jeśli istnieje punkt q na prostej l_i taki, że odcinek pq nie ma wspólnych punktów z żadną inną prostą $l_j (j \neq i)$ poza (być może) punktami p i q. Ułóż algorytm znajdujący wszystkie proste widoczne z punktu (0,+∞).

// todo (rozdział 35 cormen)

Zadanie 3. (1,5pkt)

Otoczką wypukłą zbioru P, punktów na płaszczyźnie, nazywamy najmniejszy wielokąt wypukły zawierający (w swoim wnętrzu lub na brzegu) wszystkie punkty z P. Naturalny, oparty na zasadzie dziel i zwyciężaj, algorytm znajdowania otoczki wypukłej dla zbioru P, dzieli P na dwa (prawie) równoliczne podzbiory (np. pionową prostą), znajduje rekurencyjnie otoczki wypukłe dla tych podzbiorów, a następnie scala te otoczki. Podaj algorytm wykonujący tę ostatnią fazę algorytmu, tj. algorytm scalania dwóch otoczek wypukłych.

// wylicza iloczyn wektorowy wektorów ab, bc
Product(a, b, c)
p1 = c - a
p2 = b – a
product = p1.x * p2.y – p1.y * p2.x
return product

// określa, czy wektor bc skręca w prawo względem wektora ab
IsTurnRight(a, b, c)
return Product(a, b, c) > 0

// określa, czy wektor bc skręca w lewo względem wektora ab
IsTurnLeft(a, b, c)
return Product(a, b, c) < 0

// łączy 2 otoczki wypukłe
MergeHulls(h1, h2)
// znajduje górne i dolne krawędzie łączące h1,h2
<LU, RU> = FindUpperEdges(h1, h2)
<LL, RL> = FindLowerEdges(h1, h2)
// usuwa te punkty, które wcześniej były częścią którejś z otoczek h1,h2, a teraz
// są wewnątrz otoczki h (czyli nie należą do niej)
h1 = RemoveInsideLeft(h1, LU, LL)
h2 = RemoveInsideRight(h2, RU, RL)
return $h_1 \cup h_2$

FindUpperEdges(h1, h2) // szuka 2 wierzchołków, które od góry łączą otoczki
// wyznaczamy kandydatów na punkty łączące otoczki
LU = $\{p \in h_1 : p.x\ jest\ największe\ w\ h_1\ (p.y\ jest\ największe\ dla\ remisów\}$
RU = $\{p \in h_2 : p.x\ jest\ najmniejsze\ w\ h_1\ (p.y\ jest\ najw.dla\ remisów\}$
while true: // szukanie górnej krawędzi łączącej
    anyChange = false // czy zmienił się jakiś z kandydatów górnej krawędzi
    LU2 = nextH(LU) // „wyższy” z sąsiadów LU z lewej otoczki
    RU2 = nextH(RU) // „wyższy” z sąsiadów RU z prawej otoczki
    // „podnoszenie” prawego górnego kandydata
    if (IsTurnLeft(LU, RU, RU2)
        RU = RU2
        anyChange = true
    // „podnoszenie” lewego górnego kandydata
    if (IsTurnRight(RU, LU, LU2)
        LU = LU2
        anyChange = true
    // znaleźliśmy 2 punkty łączące otoczki z góry
    if (!anyChange) return <LU,RU>

FindLowerEdges(h1, h2) // szuka 2 wierzchołków, które od dołu łączą otoczki
// wyznaczamy kandydatów na punkty łączące otoczki
LL = $\{p \in h_1 : p.x\ jest\ największe\ w\ h_1\ (p.y\ jest\ najmniejsze\ dla\ remisów\}$
RL = $\{p \in h_2 : p.x\ jest\ najmniejsze\ w\ h_1\ (p.y\ jest\ najmn.dla\ remisów\}$
while true: // szukanie dolnej krawędzi łączącej
    anyChange = false // czy zmienił się jakiś z kandydatów dolnej krawędzi
    LL2 = nextL(LL) // „niższy” z sąsiadów LL z lewej otoczki
    RL2 = nextL(RL) // „niższy” z sąsiadów RL z prawej otoczki
    // „obniżanie” prawego dolnego kandydata
    if (IsTurnRight(LL, RL, RL2)
        RL = RL2
        anyChange = true
    // „obniżanie” lewego dolnego kandydata
    if (IsTurnLeft(RU, LU, LU2)
        LU = LU2
        anyChange = true
    // znaleźliśmy 2 punkty łączące otoczki z góry
    if (!anyChange) return <LL,RL>

```
// usuwa te punkty, które były częścią otoczki h2, a teraz są wewnątrz otoczki h
RemoveInsideLeft(h2, RU, RL)
for-each p in h2:
        if (IsTurnLeft(RL, RU, p) // jest poza nową otoczką
                remove p from h2
return h2

// usuwa te punkty, które były częścią otoczki h1, a teraz są wewnątrz otoczki h
RemoveInsideRight(h1, LU, LL)
for-each p in h1:
        if (IsTurnRight(RL, RU, p) // jest poza nową otoczką
                remove p from h1
return h1
```

Zadanie 4. (1,5pkt)

Dane jest drzewo binarne (możesz założyć dla prostoty, że jest to pełne drzewo binarne), którego każdy wierzchołek vi skrywa pewną liczbę rzeczywistą xi. Zakładamy, że wartości skrywane w wierzchołkach są różne.
Mówimy, że wierzchołek v jest minimum lokalnym, jeśli wartość skrywana w nim jest mniejsza od wartości skrywanych w jego sąsiadach. Ułóż algorytm znajdujący lokalne minimum odkrywając jak najmniej skrywanych wartości.

```
Find-Local-Min(T) // T to korzeń drzewa
If T is leaf
        return T.value
else if T.left is leaf // tylko prawe poddrzewo
        if T.right.value < T.value
                return Find-Local-Min(T.right)
        else    return T.value
else if T.right is leaf // tylko lewe poddrzewo
        if T.left.value < T.value
                return Find-Local-Min(T.left)
        else    return T.value
else // oba poddrzewa
        minV = min(T.value, T.left.value, T.right.value)
        if minV == T.right.value
                return Find-Local-Min(T.right)
        else if minV == T.left.value
                return Find-Local-Min(T.left)
        else    return T.value
```

Dowód:
Najpierw wykażmy, co następuje:

Lemat 1:
W dowolnym momencie wykonywania algorytmu rodzic (jeśli istnieje) T ma większą wartość niż sam T.

Dowód lematu:
Rozważmy najpierw przypadek, w którym T jest korzeniem całego drzewa. W tym przypadku T nie ma rodzica i twierdzenie jest ok.
W takim razie rozważmy wykonanie algorytmu dla rodzica T. Jedynym sposobem, aby algorytm mógł kontynuować rekurencyjnie w dół do T, było spełnienie jednego z wyróżnionych warunków. Jeśli T było lewym dzieckiem swojego rodzica, to pierwszy warunek musiał mieć wartość true, która mówi, że wartość T jest mniejsza niż wartość rodzica.
Podobnie jest w przypadku, gdy T jest prawym dzieckiem.

Biorąc pod uwagę ten fakt, rozważmy teraz warunki, w których T jest zwracany przez algorytm. Pierwszym z nich jest sytuacja, gdy oba jego dzieci mają większą wartość od niego samego. T jest tutaj z pewnością lokalnym minimum, ponieważ pokazano, że rodzic również ma większą wartość. Innym sposobem zwrócenia T jest sytuacja, gdy jest on liściem (nie ma dzieci). W tym przypadku jedynym węzłem, do którego T może być podłączony, jest jego rodzic i ponownie wiemy, że wartość rodzica jest większa niż wartość T, a zatem T jest lokalnym minimum.

Złożoność:
Wiemy, że liczba wierzchołków w kompletnym drzewie binarnym wynosi $n=2^d-1$, a zatem jego głębokość to d, które można przedstawić jako lg(n+1) poprzez rozwiązanie równania dla d
Zauważmy teraz, że za każdym razem, gdy w algorytmie wykonywane jest wywołanie rekurencyjne, jest ono wywoływane z węzłem na poziomie o 1 niższym niż aktualnie rozpatrywany węzeł. Tak więc algorytm będzie rekurencyjnie przemierzał ścieżkę w dół drzewa, co zajmie co najwyżej d kroków. W każdym punkcie wykonywane są trzy podejrzenia, stąd całkowita liczba sond będzie wynosić $O(3d) = O(3\lg(n+1)) = O(\lg(n))$.

Zadanie 5. (2 pkt)

Dane jest nieukorzenione drzewo z naturalnymi wagami na krawędziach oraz liczba naturalna C.
Ułóż algorytm obliczający, ile jest par wierzchołków odległych od siebie o C.

```
CountPairs(T,C)
counter = 0
for-each vertice v from T:
        counter += DFS(T, v, C, 0)
return counter / 2

DFS(T,v,C,sum)
counter = 0
for-each v2 – neighbour of v:
        new_sum = sum + cost(v,v2)
        if (new_sum == C)
                counter += 1
        else if(new_sum < C)
                counter += DFS(T,v2,C,new_sum)
return counter
```

Zadanie 6. (1,5pkt)

Macierz A rozmiaru n × n nazywamy macierzą Toeplitza, jeśli jej elementy spełniają równanie A[i, j] = A[i − 1, j − 1] dla 2 ≤ i, j ≤ n.
(a) Podaj reprezentację macierzy Toeplitza, pozwalającą dodawać dwie takie macierze w czasie O(n).
(b) Podaj algorytm, oparty na metodzie „dziel i zwyciężaj”, mnożenia macierzy Toeplitza przez wektor. Ile operacji arytmetycznych wymaga takie mnożenie?

a)

Będziemy reprezentować tą macierz w tablicy B o rozmiarze 2n-1 z indeksami:

$$B[i] = A[\max(1, n-i+1), \max(1, i-n+1)]$$
$$A[i,j] = B[?,?]$$

Wtedy dodawanie 2 macierzy będzie odbywać się w czasie 2n-1, czyli O(n).

b) Algorytm:

$$\begin{bmatrix} A & B \\ C & A \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} W - L \\ U + L \end{bmatrix}$$

$$V = (C + A)X = CX + AX$$

$$L = A(Y - X) = AY - AX$$

$$W = (B + A)Y = BY + AY$$

$$M(n) = \begin{cases} 1 & n = 1 \\ M(n-1) + 2n & 2 \nmid n \\ 3M\left(\frac{n}{2}\right) & 2 \mid n \end{cases}$$

$$M(n) = O\left(n^{\log_2 3}\right) \approx O\left(n^{1,58}\right)$$

Zadanie 7. (1pkt)

Inwersją w ciągu $A = a_1, \ldots, a_n$ nazywamy parę indeksów $1 \leq i < j \leq n$, taką że $a_i > a_j$. Pokaż jak można obliczyć liczbę inwersji w A podczas sortowania.

Najłatwiej byłoby to zrobić w bubble sort, wtedy ilość inwersji to ilość zamian, jednak to rozwiązanie jest nieoptymalne czasowo.
Zamiast tego, obliczymy liczbę inwersji sortując przez scalanie.

Idea:
Jeśli tablica ma długość większą niż 1, wykonujemy rekurencję.
Każde wywołanie zwraca posortowaną część tablicy oraz ilość inwersji w tym fragmencie. Następnie scalamy obie połowy tak, że za każdym razem, gdy element z prawej tablicy dodajemy do rozwiązania optymalnego, dodajemy do liczby inwersji ilość elementów pozostałą w lewej tablicy.

Algorytm:

```
FindCount(A[1,…,n])
return MergeSort(A, 1, n)

MergeSort(A[1,…,n], l, pmax)
count = 0
if(l < pmax)
        p = (l+pmax)/2 + 1 // pierwszy indeks prawej części tablicy
        c1 = MergeSort(A, l, p-1)
        c2 = MergeSort(A, p, pmax)
        count = c1 + c2
        while (l <= (l+pmax)/2 and p <= pmax)
                if(A[l] > A[p]) // inwersja
                        swap(A[l], A[p])
                        count = count + p - l
                        p += 1
                else
                        l += 1
return count
```

Zadanie 8. (1,5pkt)
Przeanalizuj sieć permutacyjną omawianą na wykładzie (tzw. sieć Benesa-Waksmana)
• Pokaż, że ostatnią warstwę przełączników sieci Benesa-Waksmana można zastąpić inną warstwą, która zawiera n/2−1 przełączników (a więc o jeden mniej niż w sieci oryginalnej) a otrzymana sieć nadal będzie umożliwiać otrzymanie wszystkich permutacji.

Każda permutacja ma co najmniej 1 cykl, a w każdym cyklu można usunąć 1 przełącznik.

• Uogólnij sieć na dowolne n (niekoniecznie będące potęgą liczby 2).

Dla dowolnego n rozwiązujemy problem rekurencyjnie:

$$T(n) = \begin{cases} zwykły\ przełącznik\ dla\ n = 2 \\ specjalna\ sieć\ dla\ n = 3 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right)\ dla\ n > 3 \end{cases}$$

Gdzie specjalna sieć dla n=3 składa się z 3 głębokości:
1. Przełącznik między 1 i 2 wejściem, 3 wejście osobno,
2. 1 wyjście tego przełącznika osobno, a pozostałe 2 są wejściami do nowego przełącznika.
3. 1 i 2 wyjście do nowego przełącznika, a 3 wyjście osobno.

Zadanie 9. (2pkt)
Niech P_n będzie zbiorem przesunięć cyklicznych ciągu n-elementowego o potęgi liczby 2 nie większe od n. Pokaż konstrukcję sieci przełączników realizujących przesunięcia ze zbioru P_n.
Uwagi:
• Możesz założyć, że n jest potęgą dwójki albo szczególną potęgą dwójki,…
• Sieć Benesa-Waksmana jest dobrym rozwiązaniem wartym 0pkt
(tzn. nic niewartym).

Zakładamy szczególną potęgę dwójki. Podział na pół, i-te wejście z m-tej połowy będzie m-tym wejściem i-tego przełącznika w następnej warstwie, w której wykonujemy problem rekurencyjnie dla 2*T(n/2).