Zadanie 1.

Ułóż algorytm znajdujący najtańszą drogę przejścia przez tablicę, w którym oprócz ruchów dopuszczalnych w wersji problemu prezentowanej na wykładzie, dozwolone są także ruchy w górę i w dół tablicy.

```
PartialSums(A, m, n) // znajduje najtańsze koszty dotarcia do każdego pola
for j from 1 to m // dla każdej kolumny
	D[0][j] = +inf // pierwszy wiersz
	D[n+1][j] = +inf // ostatni wiersz
for i from 1 to n // dla pierwszej kolumny
	D[i][1] = A[i][1]
for j from 2 to m // dla pozostałych kolumn
	for i from 1 to n // dla każdego wiersza w kolumnie, porównaj z pop. kol.
		D[i][j] = a[i][j] + min(D[i-1][j-1], D[i][j-1], D[i+1][j-1])
	for i from 2 to n // relaksacja z poprzednimi elementami kolumny
		D[i][j] = min(D[i][j], A[i][j] + D[i-1][j])
	for i from n-1 to 1 // relaksacja z następnymi elementami kolumny
		D[i][j] = min(D[i][j], A[i][j] + D[i+1][j])
return D

FindPath(A, D, m, n) // zwraca najtańszą ścieżkę
value = D[1][n] // najmniejsza wartość w danej kolumnie
w = 1 // indeks wiersza z najmniejszym kosztem w danej kolumnie
for i from 2 to n // znajdź najmniejszą wartość w ostatniej kolumnie
	if (value < D[i][n])
		value = D[i][n]
		w = i
path = list(A[w][n]) // najtańsza ścieżka
while True // tworzenie najtańszej ścieżki
	p = FindMin(D, p) // indeks poprzedniego pola najtańszej ścieżki
	value = A[p.x][p.y] // wartość poprzedniego pola
	if (value == path.top) // doszliśmy do pierwszego pola
		return path
	path.push_front(A[p.x][p.y]) // dodaj poprz. pole do najtańszej ścieżki

CheapestPath(A, m, n) // zwraca najtańszą ścieżkę
D = PartialSums(A, m, n)
return FindPath(A, D, m, n)
```

Zadanie 2. (1.5pkt)

Ułóż algorytm, który dla danego ciągu znajduje długość najdłuższego jego podciągu, który jest palindromem.

```
LPS(seq)
int L[n][n] // długości najdłuższych palindromów w zakresie [L1,L2]
Dla każdego i od 0 do n – 1 // każdy pojedynczy znak jest palindromem
        L[i][i] = 1
Dla każdego k od 2 do n // sprawdzamy długości palindromów
        Dla każdego i od 0 do n – k
                j = i + k – 1
                if (seq[i] == seq[j] and k = 2) // palindrom długości 2
                        L[i][j] = 2
                else if (seq[i] == seq[j]) // przedłużenie palindromu o 2
                        L[i][j] = L[i+1][j-1] + 2
                else // nie wydłuża palindromu
                        L[i][j] = max(L[i][j-1], L[i+1][j])
return L[0][n-1]
```

Zadanie 3. (2pkt).

Zadanie 3 Liczba elementów > k

3. (2pkt). W n-elementowej tablicy A pamiętany jest rosnący ciąg liczb naturalnych. Nie znamy wartości jej elementów, ale możemy się o nie pytać. Pytanie o wartość $A[i]$ kosztuje nas c_i.

   Ułóż algorytm, który dla danych liczb naturalnych $c_1, c_2, \ldots, c_n$ oraz liczby k obliczy najmniejszym kosztem (liczonym jako suma kosztów zadanych pytań), ile liczb w tablicy A ma wartość większą niż k.

Definicja stanu:

$T_{i,j}$ = najmniejszy koszt znalezienia rozwiązania na przedziale $[i, j]$ w pesymistycznym przypadku

Przejście:

$$T_{i,j} = \min_{i \le l \le j} \left(c_l + \max(T_{i,l-1}, T_{l+1,j}) \right)$$

Początkowe T:

ij	1	2	3	4	5
1	c_1	∞	∞	∞	$\cdots$
2	0	c_2	∞	∞	$\cdots$
3	0	0	c_3	∞	∞
4	0	0	0	c_4	∞
5	0	0	0	0	c_5

$P_{i,j}$ - pamiętamy pierwsze sprawdzenie dla przedziału, żeby móc potem odtworzyć sekwencję

Zadanie 4. (2pkt) ŹLE

Zmodyfikuj algorytm znajdujący najdłuższy wspólny podciąg dwóch ciągów n elementowych, tak by działał w czasie O(n^2) i używał O(n) pamięci.

```
// najpierw znajduje długość NWP, a potem zwraca któryś z tych NWP
// (czasami może być kilka optymalnych rozwiązań)
LCS-Linear(X, Y, n)
c – tablica rozmiaru 2 x (n+1), długości najdłuższych podciągów
c[1, 0] = 0
for j from 0 to n // pierwszy wiersz
        c[0][j] = 0
for i from 1 to n // kolejne wiersze (elementy z X)
        for j from 1 to n // kolejne kolumny (elementy z Y)
                if (X[i-1] == Y[j-1]) // dane podciągi X,Y kończą się tą samą wartością
                        c[i % 2][j] = c[(i-1) % 2][j-1] + 1
                else
                        c[i % 2][j] = max(c[(i-1) % 2][j], c[i % 2][j-1])
i = n % 2
subseq = „”
for j from 1 to n // dla każdej litery Y sprawdź, czy wydłuża sekwencję z X
        if C[i][j] > C[i][j-1] // wydłużenie najdłuższej sekwencji
                subseq += Y[j-1]
return subseq
```

Zadanie 5. (2pkt)
Ułóż algorytmy, które dla danych podciągów x i y rozwiązują następujące wersje problemu znajdowania najdłuższego wspólnego podciągu:
• znajdowanie najdłuższego wspólnego podciągu zawierającego podciąg „matma”,
• znajdowanie najdłuższego wspólnego podciągu nie zawierającego podciągu „matma”,
• znajdowanie najdłuższego wspólnego podciągu zawierającego podsłowo „matma”,
• znajdowanie najdłuższego wspólnego podciągu nie zawierającego podsłowa „matma”

LCS z warstwą na szukanie każdej literki + jedna.
Każda z poniższych wersji działa w czasie/pamięci $O(n^2)$.

• znajdowanie najdłuższego wspólnego podciągu zawierającego podciąg „matma”:

T_0 – najdłuższy wspólny podciąg
T_1 – najdłuższy wspólny podciąg zawierający literę „m”

$$T_1[i][j] = \begin{cases} \max(T_1[i-1][j], T_1[j-1][i]) \ jeśli\ x[i] \neq y[j] \\ T_1[i-1][j-1] + 1\ jeśli\ x[i] = y[j] \neq \mathrm{m} \wedge T_1[i-1][j-1] > 0 \\ T_1[i-1][j-1]\ jeśli\ x[i] = y[j] \neq "m" \\ T_0[i-1][j-1] + 1\ jeśli\ x[i] = y[j] = "m" \end{cases}$$

T_2 – najdłuższy wspólny podciąg zawierający podciąg „ma”

$$T_2[i][j] = \begin{cases} \max(T_2[i-1][j], T_2[j-1][i]) \ jeśli\ x[i] \neq y[j] \\ T_2[i-1][j-1] + 1\ jeśli\ x[i] = y[j] \neq \mathrm{a} \wedge T_2[i-1][j-1] > 0 \\ T_2[i-1][j-1]\ jeśli\ x[i] = y[j] \neq "a" \\ T_1[i-1][j-1] + 1\ jeśli\ x[i] = y[j] = "\mathrm{a}" \wedge T_1[i][j] > 0 \\ 0\ w\ \ p.p. \end{cases}$$

T_3 – najdłuższy wspólny podciąg zawierający podciąg „mat”
T_4 – najdłuższy wspólny podciąg zawierający podciąg „matm”
T_5 – najdłuższy wspólny podciąg zawierający podciąg „matma”

• znajdowanie najdłuższego wspólnego podciągu nie zawierającego podciągu „matma”:

T_0 - najdłuższy wspólny podciąg
T_1 - najdłuższy wspólny podciąg bez litery “m”

$$T_1[i][j] = \begin{cases} \max(T_1[i-1][j], T_1[j-1][i]) \ if\ x[i] \neq y[j] \vee x[i] = y[i] = "m" \\ T_1[i-1][j-1] + 1\ jeśli\ x[i] = y[j] \neq "m" \end{cases}$$

T_2 - najdłuższy wspólny podciąg bez podciągu “ma”

$$T_2[i][j] = \begin{cases} \max(T_2[i-1][j], T_2[i][j-1]) \ if\ x[i] \neq y[j] \\ T_2[i-1][j-1] + 1\ jeśli\ x[i] = y[j] \neq "a" \\ \max(T_1[i-1][j-1] + 1, T_2[i-1][j-1])\ if\ x[i] = y[j] = "a" \end{cases}$$

T_3 - najdłuższy wspólny podciąg bez podciągu "mat"
T_4 - najdłuższy wspólny podciąg bez podciągu "matm"
T_5 - najdłuższy wspólny podciąg bez podciągu "matma"

- znajdowanie najdłuższego wspólnego podciągu zawierającego podsłowo „matma",
- znajdowanie najdłuższego wspólnego podciągu nie zawierającego podsłowa „matma"

Zadanie 6.

Rozważmy następujący problem 3-podziału. Dla danych liczb całkowitych $< a_1, \dots, a_n \in < -C, C >$ chcemy stwierdzić, czy można podzielić zbiór $\{1, 2, \dots, n\}$ na trzy rozłączne podzbiory I, J, K, takie, że

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k$$

```
ThreeSets(A, n)
S = A[0] // suma elementów w A
min = A[0] // najmniejszy element w A
max = A[0] // największy element w A
for i from 1 to n-1:
        S += A[i]
        if(A[i] < min):        min = A[i]
        else if(A[i] > max): max = A[i]
if(S mod 3 != 0) // warunek konieczny
        return false
begin = minimum(0, min, S/3)
end = maximum(0, max, S/3)
size = end – begin // rozmiar tablicy Table
Table[size][size] // zawiera wszystkie liczby całkowite z przedziału <begin, end>
WhichRound[size][size] // w której iteracji komórka została zmieniona na true
sort(A) // dla lepszego wypełniania tablicy
// wypełnianie tablicy – wynik jest w Table[S/3][S/3]
Table[-begin, -begin] = true // zawsze można znaleźć 2 podzbiory puste
for i from 0 to n-1: // dla każdej liczby z A
        for row from 0 to size-1:
                for col from row to size-1:
                        if (Table[row-A[i]][col] and WhichRound[row-A[i]][col] != i)
                                if(row == S/3 and col == S/3) // istnieje trójpodział
                                        return true
                                else    Table[row][col] = true
```

```
            elif (Table[row][col-A[i]] and WhichRound[row][col-A[i]] != i)
                if(row == S/3 and col == S/3) // istnieje trójpodział
                    return true
                else    Table[row][col] = true
return false
```

Zadanie 7. (2 pkt)

Dwie proste równoległe l’ i l’’ przecięto n prostymi $p_1, \dots, p_n$.
Punkty przecięcia prostej p_i z prostymi l’ i l’’ wyznaczają na niej odcinek.
Niech Odc będzie zbiorem tych odcinków.
(a) Ułóż algorytm, wyznaczający w Odc podzbiór nieprzecinających się odcinków, o największej mocy.
(b) Ułóż algorytm, wyznaczający liczbę podzbiorów, o których mowa w poprzednim punkcie.

// todo