Zadanie 1. (1pkt)
Niech σ będzie ciągiem instrukcji Union i F ind, w którym wszystkie instrukcje Union występują przed instrukcjami Find. Udowodnij, że algorytm oparty na strukturach drzewiastych wykonuje σ w czasie proporcjonalnym do długości σ.

// todo Rozdział 4.7 AHO

Zadanie 2. (2pkt)
Rozważamy ciągi operacji Insert(i), DeleteMin oraz Min(i) wykonywanych na S - podzbiorze zbioru {1, …, n}. Obliczenia rozpoczynamy z S = ∅.
Instrukcja Insert(i) wstawia liczbę i do S.
Instrukcja DeleteMin wyznacza najmniejszy element w S i usuwa go z S.
Natomiast wykonanie Min(i) polega na usunięciu z S wszystkich liczb mniejszych od i. Niech σ będzie ciągiem instrukcji Insert(i), DeleteMin oraz Min(i) takim, że dla każdego i, $1 \le i \le n$, instrukcja Insert(i) występuje co najwyżej jeden raz.
Mając dany ciąg σ naszym zadaniem jest znaleźć ciąg liczb usuwanych kolejno przez instrukcje DeleteMin. Podaj algorytm rozwiązujący to zadanie.
Uwaga: Zakładamy, że cały ciąg σ jest znany na początku, czyli interesuje nas wykonanie go on-line.
WSKAZÓWKA: Rozdział 4.8 z książki Aho,.

// todo

Zadanie 3. (2pkt)
Rozważamy ciągi instrukcji: Link(r, v) oraz Depth(v) wykonywanych na lesie rozłącznych drzew o wierzchołkach z etykietami ze zbioru {0, …, n − 1} (różne wierzchołki mają różne etykiety).
Operacja Link(r, v) czyni r, korzeń jednego z drzew, synem v, wierzchołka innego drzewa. Depth(v) oblicza głębokość wierzchołka v.
Naszym celem jest napisanie algorytmu, który dla danego ciągu σ wypisze w sposób on-line wyniki instrukcji Depth (tzn. wynik każdej instrukcji Depth ma być obliczony przed wczytaniem kolejnej instrukcji z ciągu σ).
Pokaż jak zastosować drzewiastą strukturę danych dla problemu Union − Find do rozwiązania tego problemu.
WSKAZÓWKA: Rozdział 4.8 z książki Aho.

// todo

Zadanie 4. (1pkt)

Rozważ taką wersję wykonywania kompresji ścieżek, w której wierzchołki wizytowane podczas wykonywania operacji Find podwieszane są pod własnego dziadka. Czy analiza złożoności przeprowadzona na wykładzie da się zastosować w tym przypadku?

// todo

Zadanie 5. (2pkt)

Załóżmy, że wzorzec P może zawierać znak ♦ (tzw. gap character).
Znak ten jest zgodny z dowolnym podsłowem (także z podsłowem pustym).
Na przykład, wzorzec ab♦ba♦c występuje w słowie cabccbacbacab jako

$$c\,\underbrace{ab}_{ab}\,\underbrace{cc}_{\diamondsuit}\,\underbrace{ba}_{ba}\,\underbrace{cba}_{\diamondsuit}\,\underbrace{c}_{c}\,ab$$

$$c\,\underbrace{ab}_{ab}\,\underbrace{ccbac}_{\diamondsuit}\,\underbrace{ba}_{ba}\,\underbrace{\quad}_{\diamondsuit}\,\underbrace{c}_{c}\,ab$$

Podaj algorytm znajdujący wystąpienie takiego wzorca w danym tekście T (oczywiście zakładamy, że ♦ nie występuje w T).

```
// znajduje pierwsze wystąpienie wzorca P w tekście
KMP_First(pat, txt)
M = len(pat)
N = len(txt)
// lps[] przechowuje długość najdłuższego prefixu równego suffixowi
lps = [0]*M
j = 0 // indeks pat[]
CalculateLPS (pat, M, lps) // oblicza wartości lps[]
i = 0 // indeks txt[]
while (N - i) >= (M - j)
        if (pat[j] == txt[i]) // znak zgadza się, przesuń oba indeksy do przodu
        i += 1
        j += 1
// znaleziono pierwsze wystąpienie wzorca, zwróć jego początkowy indeks
        if (j == M)
                return <i – j, i> // zwróć indeksy początku i końca wzorca
```

```
        elif (i < N and pat[j] != txt[i]) // znak nie zgadza się po j zgodnościach
// nie porównuj znaków lps[0..lps[j-1]], one będą się zgadzały
                if (j != 0)
                        j = lps[j-1]
// żaden znak się nie zgadzał dotychczas, przesuń o 1 jak w algorytmie naiwnym
                else
                        i += 1
return None // nie znaleziono dopasowania

// obliczanie tablicy LPS
CalculateLPS (pat, M, lps)
len = 0 // dlugość najdłuższego prefixu równego suffixowi
lps[0] = 0 // lps[0] zawsze wynosi 0
i = 1
while (i < M) // obliczamy lps[i] dla i = 1 do M-1
        if (pat[i] == pat[len]) // znaki się zgadzają
                len += 1
                lps[i] = len
                i += 1
        else // znaki się nie zgadzają
// w poprzednim porównaniu istniał jakiś prefiks równy sufiksowi
// nie zwiększamy i, bo może krótszy pattern się zgadza
                if (len != 0)
                         len = lps[len-1]
                else // nie istnieje prefiks równy sufiksowi
                        lps[i] = 0
                        i += 1

// znajduje przedział, na którym występuje wzorzec
FindPattern(Text, P)
patterns = Split(P) // rozbij wzorzec postaci „A♦BC♦D” na [„A”, „BC”, „D”]
for-each pat in patterns:
        indexes[i] = KMP_First(Text, pat)
        Text = Text[indexes[i].second+1,…] // usuń prefix tekstu ze wzorcem
// zwróć przedział, na którym występuje wzorzec
return <indexes[0].first, indexes[x].second>
```

Zadanie 6. (1pkt)
Podaj algorytm, który w czasie liniowym określa, czy tekst T powstał przez przesunięcie cykliczne tekstu T'.

Pomysł: będziemy porównywać T,T' zaczynając od indeksów i+k,j+k.
Dopóki T,T' są podobne na aktualnych pozycjach zwiększamy k, jak przestaną być podobne to zwiększamy i lub j o wartość k+1.
Zatem T jest przesunięciem cyklicznym T' wtw gdy istnieją indeksy i,j że n razy będzie równość, czyli k = n.

Złożoność: czasowa O(n), pamięciowa O(1)

```
CyclicShiftInPlace(T,T')
i = 0; j = 0; k = 0; n = len(T)
if (n != len(T')) // oczywiście teksty muszą być tej samej długości
        return False
while (i < n and j < n and k < n) // sprawdzamy każdą literę jako początek
        k = 0 // ilość wspólnych liter
        while (k < n and T[(i+k) % n] == T'[(j+k) % n])
                k += 1
        if (k < n) // nie jest to przesunięcie cykliczne (dla tych początków)
                if (T[(i+k) % n] > T'[(j+k) % n])
                        i = i + k + 1 // przesuń początek T
                else
                        j = j + k + 1 // przesuń początek T'
return k >= n // czy T jest przesunięciem cyklicznym T'
```