# Kolokwium poprawkowe ze Wstępu do Informatyki

17.1.2015

Czas trwania kolokwium: 180 minut. Zadania oddajemy na osobnych kartkach, podpisane imieniem, nazwiskiem, numerem indeksu. Oddanie zadania należącego do kolokwium oznacza rezygnację z poprzednio uzyskanego wyniku i zaakceptowanie wyniku poprawkowego. Można oddać zadania z jednego z kolokwiów, bez rezygnacji z wyniku drugiego. Zadania można rozwiązywać w dowolnej kolejności. Proszę zwrócić uwagę na różną punktację zadań.

**Kolokwium poprawkowe 1**

1. (5p.) Poziomem zagnieżdżenia listy nazywamy liczbę list, w których ta lista jest zawarta (również pośrednio).

   Np. poziom zagnieżdżenia listy `[1,2]` w liście `[1, [1,2], [[3,4]]]` jest równy 1, a listy `[3,4]` – 2.

   Głębokością listy nazywamy maksymalny poziom zagnieżdżenia list w niej zawartych.

   Napisz funkcję `glebokosc(l)`, która dla listy `l`, zawierającej listy złożone z zer i list, zwróci głębokość listy `l`.

   ```
   glebokosc([[0, [], 0], [0,0], [[[0]]]]) = 3
   glebokosc([0, 0]) = 0
   ```

2. (5p.) *Permutacją* (n-elementową) $P$ nazwiemy listę zawierającą wszystkie liczby od 0 do $n-1$, np. $P_1 = [0,4,2,3,1]$, $P_2 = [0,1]$, $P_3 = [0,1,3,2]$. Listę indeksów $[i_0, i_1, \ldots, i_k]$ nazwiemy *cyklem prostym* w permutacji $P$, jeśli składa się on z parami różnych elementów, a ponadto spełnia własność:

   $$P[i_0] = i_1, P[i_1] = i_2, \ldots, P[i_{k-1}] = i_k, P[i_k] = i_0$$

   Przykłady cykli prostych:

   - permutacja $P_4 = [3,0,2,1]$ zawiera dwa cykle proste: $c_1 = [3,1,0], c_2 = [2]$
   - permutacja $P_5 = [3,1,4,0,2,6,5]$ zawiera cztery cykle proste: $c_1 = [3,0], c_2 = [1], c_3 = [4,2], c_4 = [6,5]$
   - permutacja $P_6 = [3,4,5,1,2,0]$ sama w sobie jest cyklem prostym.

   Korzystając z faktu, że dla każdej permutacji istnieje jej rozkład na cykle proste (tj. każdy element permutacji należy do dokładnie jednego cyklu prostego) napisz funkcję *dlugosc_cykli*$(P)$, który dla zadanej permutacji $P$ zwróci listę, która na $i$-tej pozycji zawiera długość cyklu do którego należy. Przykłady:

   - *dlugosc_cykli*$([3,0,2,1]) = [3,3,1,3]$
   - *dlugosc_cykli*$([0,1,2,4,3]) = [1,1,1,2,2]$
   - *dlugosc_cykli*$([3,4,5,1,2,0]) = [6,6,6,6,6,6]$

3. (5p.) Napisz funkcję para(L1,L2), która dla zadanych dwóch list L1 i L2, stwierdzi, czy istnieją dwa elementy wspólne dla tych list, które występują na nich w tej samej kolejności.

   Np.
   para([1,2,3,4,5],[5,4,3,2,1]) → False,
   para([**1**,3,**2**],[5,4,**1**,**2**]→ True.

**Kolokwium poprawkowe 2**

1. (4p.) W pliku `ruchy.txt` znajduje się lista dozwolonych ruchów pionków na planszy do pewnej gry planszowej. Każda linia w pliku ma formę:

   `x1 y1 x2 y2`

   gdzie `x y` to pozycje na planszy, i oznacza ruch z pozycji `x1 y1` do `x2 y2`.

   Napisz funkcję `is_loop2()`, która sprawdzi, czy jest możliwe wykonanie pętli długości 2, czyli ruchu z pewnego pola `x1 y1` na pewne pole `x2 y2` i z powrotem jeśli dozwolone są tylko ruchy wypisane w pliku `ruchy.txt`.

   Przykład:

   ```
   ruchy.txt:
   1 1 1 2
   1 1 2 2
   1 2 1 1
   is_loop2() -> True
   ```

   ```
   ruchy.txt:
   1 1 2 2
   2 2 3 3
   3 3 1 1
   is_loop2() -> False
   ```

2. (3p.) Dla każdego z poniższych programów odpowiedz co zostanie wypisane na ekranie po jego wykonaniu.

   - a.py

     ```
     d = {'a' : 3}
     def add(x):
         d = {'b' : 6}
         d['c'] = x
         return sum(d.values())
     print add(9), d
     ```

   - b.py

     ```
     d = {'li' : [1, 2, 3]}
     def update(x, i, y):
         global d
         d[x] = d['li']
         d[x][i] += y
     update('la', 0, 1)
     print d
     update('li', 1, 2)
     print d
     ```

   - c.py

     ```
     lu = [-3, -2, -1]
     def liextend(li, la):
         if len(la) > 0:
             li += la
             return liextend(li, la[: -1])
         else:
             return li
     print liextend([1,2,3], lu), lu
     ```

3. (3p.) Mamy dany plik z sekwencjami DNA w kolejnych linijkach. Napisz program, który używając wyrażeń regularnych wypisze tylko te linijki, które zawierają ten sam symbol powtórzony co najmniej 5 razy bezpośrednio po sobie, a następnie (po niezerowej liczbie innych symboli) dwukrotnie więcej symboli komplementarnych (nb. uznajemy tylko komplementarność A-T, G-C).

   Np.
   ATGTAGT**AAAAAA**TGTAGTGTCGATGC**TTTTTTTTTTTT**CGTGATGAT → wypisujemy
   ATGTGTAGATG**TTTTT**ATGTGATAGTGATAGTAAAAA → nie wypisujemy, bo nie ma wystarczająco dużo A
   ACGTGTCAC**GGGGGCCCCCCCCCC**ATGCGTAG → nie wypisujemy, bo nie ma nic pomiędzy G i C