Pakiety matematyczne MAT1349 - 2021

Lista ℵ: 99 problemów w Prologu

Lista ta jest w całości **opcjonalna**. Podpunkty możemy robić w dowolnej kolejności i o ile ktoś jest chętny.

Lista 99 problemów w Prologu (*Ninenty-Nine Prolog Problems*) jest znaną listą problemów algorytmicznych, którą później uogólniono na kilka innych języków (Haskell, Ruby, Python, itd.). O problemach z tej listy można myśleć jako o zagadkach logiczno-programistyczno-matematycznych. Na tej opcjonalnej liście jest wybór kilku z problemów niewymagających znajomości zaawansowanych struktur danych.

Rozwiązanie to funkcja, która wykonuje zadane operacje. Każdy z podpunktów możemy rozwiązać na 2 sposoby:

a) **Programowanie funkcyjne.** Staramy się rozwiązać problem operując jedynie funkcjami, nie używamy nigdzie pętli `for` czy `while`. Nie alokujemy tablic, o ile nie jest to niezbędne. Przydadzą się szczególnie z `reduce,foldl,foldr` oraz metody z `Base.Iterators`. Możemy dopuścić generatory `(f(x) for x in xs)` jako że to iteratorowa wersja broadcastu `f.(xs)`, nie jest to klasyczna pętla.

   Przykład: problem P01 z listy, w którym mamy znaleźć ostatni element iteratora. Jedno z możliwych rozwiązań:

   ```
   f(iter) = foldl((a,b)->b, iter)
   ```

b) **Code golf.** Jest to programistyczne wyzwanie i rozrywka praktykowana w większości języków. Kryterium jest proste: rozwiązujemy problem kodem mającym jak najmniejszą iloś znaków. Nie liczy się nic więcej

   Przykład: ponownie P01. Jedno z możliwych rozwiązań:

   ```
   i->[i;][end]
   ```

   (Nie jest to dobre rozwiązanie do a), gdyż niepotrzebnie tworzy tablicę.)

Wybrane problemy, które mogą być ciekawe (można też przejrzeć pełną listę i posiłować się z innymi):

P02 Znajdź przedostatni element iteratora.

P04 Znajdź ilość elementów iteratora.

P05 Mając iterator zwróć jego elementy wypisane od tyłu.

P08 Mając iterator, który może zwracać grupy powtarzających się elementów, zwrócić je bez duplikatów. Np. jeżeli iterator zwraca $1, 1, 1, 2, 2, 5, 5, 5, 5$ wynik to $1, 2, 5$.

P10 Mając iterator, który może zwracać grupy powtarzających się elementów, zwrócić je w postaci krotek z elementem oraz jego krotnością (jest to forma kompresji) Np. jeżeli iterator zwraca ‘a‘, ‘a‘, ‘b‘, ‘c‘, ‘c‘, ‘c‘ wynik to $(2, \text{‘}a\text{‘}), (1, \text{‘}b\text{‘}), (3, \text{‘}c\text{‘})$.

P15 Mając iterator oraz `n::Int` zwróć jego elementy zduplikowane n razy. Np. $n = 3$ i $1, 4, 3$ powinno dać $1, 1, 1, 4, 4, 4, 3, 3, 3$.

P16 Mając dany iterator i `n::Int` zwróć jego elementy ignorując co $n-$ty.

P19 Mając dany iterator oraz `n::Int` zwróć jego elementy obrócone o n pozycji. Np. 3 oraz $1, 1, 2, 2, 0, 0, 0, 0$ powinno zwrócić $0, 0, 0, 1, 1, 2, 2, 0$.

P20 Mając iterator i `n::Int`, zwróć jego elementy bez k-tego.

P49 Mając `n::Int` zwróć elementy *kodu Graya* rzędu `n`. Definicja jest tutaj:

https://pl.wikipedia.org/wiki/Kod_Graya

Pamiętaj, że mając iterator w ogólności nie mamy dostępu do elementów `[]` ani nie znamy ich ilości. Możemy je tylko zwracać po kolei. Kiedy zwracamy wiele liczb, możemy to zrobić w formie tablicy lub iteratora.

Dodatkowe opcje: Algorytmy możemy pisać jeszcze krócej i bardziej elegancko korzystając z `mapreduce,mapfoldr,mapfoldr` oraz pakietu `Underscore`.