Lucas-sorozat még jobban

0. Modul

Definiálj egy modult `FastLucas` néven!

A html fájlban jobban látszódnak a képletek!

1. Saját típus

Készíts egy saját típust, amellyel még gyorsabban lehet kiszámolni a Lucas-sorozat elemeit.

Az ötlet, hogy a közvetlen képletet gyorsabb használni, mint lineárisan egyesével ellépkedni az `n.` elemig.
Az `sqrt` függvény azonban nagyon pontatlan, egyébként is a sorozat elemei a rekurzív definíció alapján kitalálható, hogy csak egészek lehetnek. A közvetlen képlet a következő, hasonló a Fibonacci-sorozatéhoz:

$\phi = \frac{1+\sqrt{5}}{2}$

$L_n = \phi^n + (1-\phi)^n$

A számolások során használd a lehető legpontosabb számtípust, az `Integer`-t vagy a `Rational`-t, amely típus a `Data.Ratio` modulban található meg. A `Rational` típus két egész szám hányadosát reprezentálja.

Segítség: A komplex számokat például a `data Complex = Complex Rational Rational`-lal el lehet kódolni, ahol az első paraméter a valós rész, a második paraméter a képzetes rész. Most is valami hasonlót kell csinálni. A képletből látható, hogy csak $\sqrt{5}$ -tel kell foglalkozni, más irracionális számmal nem, ezt érdemes kihasználni. Az is tudható, hogy az eredmény mindig egész lesz, ezt is ki lehet használni majd a `lucas` definiálásában.

Rational használata: Három fontosabb függvénye van:

- `(%) :: Integer -> Integer -> Rational`: Lehet vele definiálni `Rational` típusú értéket két `Integer` megadásával.
- `numerator :: Rational -> Integer`: Visszaadja a tört számlálóját.
- `denominator :: Rational -> Integer`: Visszaadja a tört nevezőjét.

2. Példányok a típusra

- A fordító segítségét kérve legyen egy `Show` példányunk.
- Készíts egy saját `Num` példányt az előzőleg megírt saját típusra a matematikának megfelelően!
  A `signum` függvényt elég megadni a hagyományos számolással, teljesen pontos eredményt nehéz összerakni belőle.

Emlékeztető: Saját példányt az `instance ... where` konstrukcióval csináltunk.

3. Aranymetszés / golden ratio

Definiáld a `phi` konstanst, amely a saját típusodban az aranymetszés értékét, azaz $\phi = \frac{1+\sqrt{5}}{2}$ értékét kódolja el.

4. Lucas-sorozat n. eleme

A feladat ugyanaz, mint a korábbiakban volt, a Lucas-sorozat `n.` elemét kell visszaadni az értelmes legáltalánosabb típussal. A függvény megint legyen a régi formának megfelelően egyparaméteres.
A függvény neve ugyanúgy `lucas :: (Integral a, Num b) => a -> b`.
Ha minden jól lett definiálva, akkor a `10 000 000.` elemig semmilyen probléma nem lesz, így ez az elvárás, hogy a `10 000 000.` elemet is 1 másodpercen belül ki tudja értékelni a tesztelő.

5. Lucas-sorozat felhasználása

Definiáld a már ismert `isNotPrime :: Integral a => a -> Bool` függvényt, amely a Lucas-sorozat segítségével meghatározza egy számról, hogy biztosan **nem** prímszám-e.

A felhasználási módja a következő:

Legyen a vizsgált számunk `n`, erre vagyunk kíváncsiak, hogy prímszám-e. Vegyük a Lucas-sorozat `n.` elemét (a sorozat elemeit 0-tól számozva), az ott szereplő számból vonjunk ki `1`-et. Ha az úgy kapott szám `n`-nek nem a többszöröse, akkor `n` teljesen biztosan nem prímszám (ekkor adjon vissza a függvény `True` értéket). Ha az úgy kapott szám `n`-nek többszöröse, akkor nem tudunk a számról semmi biztosat állítani, ekkor a függvény adjon vissza `False` értéket.

Tesztesetek

```
lucas 30 == 1860498
lucas 50 == 28143753123
lucas 100 == 792070839848372253127
lucas 1000 == 9719417773590817520798198207932647373779787915534568508272808108477251881844481526908061914
lucas 50000 == 240997478643825988804980698063344994982641163681531996803773375195393376825136487164660018
lucas 500000 == lucas 500000
isNotPrime 4
isNotPrime 6
not (isNotPrime 7)
not (isNotPrime 11)
not (isNotPrime 17)
not (isNotPrime 23)
isNotPrime 230
not (isNotPrime 705) -- ez az első olyan szám, ami ezen a teszten átmegy, mint összetett szám, eddig a po
isNotPrime 1000
```