Legalább egy azonos (1 pont)
Definiáljuk a contains függvényt, amely egy számot és egy rendezett számhármast kap paraméterül, és megadja, hogy a szám rendezett hármas valamelyik tagja-e!

```
contains :: Integral a => a -> (a,a,a) -> Bool
contains 2 (2,3,49)
contains 8 (8,5,9)
contains 5 (2,0,5)
not $ contains 0 (1,2,3)
not $ contains 2 (88,65,7)
```

Megszerkeszthető derékszögű háromszög (1 pont)
Definiáljuk a triangleArea függvényt, amely egy rendezett hármast kap paraméterül, melynek első két komponense egy háromszög befogóinak hossza, a harmadik komponens az átfogó hossza. Amennyiben megszerkeszthető és derékszögű ez a háromszög, adja meg a függvény a területét Just-ba csomagolva, különben pedig az eredmény legyen Nothing. Feltehetjük, hogy mindhárom érték pozitív!

Segítség:

a háromszög megszerkeszthető, ha bármely két oldal hosszainak összege nagyobb a harmadik oldal hosszánál
a háromszög derékszögű, ha a befogói hosszainak négyzetösszege megegyezik az átfogó hosszának négyzetével
a derékszögű háromszög területe = a befogók hosszának szorzata / 2

```
triangleArea :: (Double, Double, Double) -> Maybe Double
triangleArea (4,0.9,4.1) == Just 1.8
triangleArea (9,40,41) == Just 180
triangleArea (4,3,5) == Just 6
triangleArea (5.04,6.72,8.4) == Just 16.9344
triangleArea (3.36,2.52,4.2) == Just 4.2336
triangleArea (8.0,6,10.0) == Just 24.0
triangleArea (8,6,10) == triangleArea (6,8,10)
triangleArea (21,21,21) == Nothing
triangleArea (5,3,2) == Nothing
```

Relatív prímek (1 pont)
Matematikában a relatív prímek olyan számpárok, amelyeknek az 1-en kívül nincs más közös pozitív osztójuk.

Definiálj egy függvényt, amely megadja egy listában az összes, a paraméterként kapott pozitív egész számnál kisebb, vele relatív prímeket. Csak 0-nál nagyobb egész számok legyenek az eredményt tartalmazó listában!

Segítség: A gcd :: Integral a => a -> a -> a függvény meg tudja határozni két szám legnagyobb közös osztóját.

```
relativePrimes :: Integral a => a -> [a]
relativePrimes 10 == [1,3,7,9]
relativePrimes 100 ==
[1,3,7,9,11,13,17,19,21,23,27,29,31,33,37,39,41,43,47,49,51,53,57,59,61,63,67,69
,71,73,77,79,81,83,87,89,91,93,97,99]
null (relativePrimes 1)
relativePrimes 101 == [1..100]
```

Komplementer DNS szálak (2 pont)
A DNS az információt négy bázis segítségével kódolja: A, T, C, G. A DNS két

szálában egy adott bázissal szemben mindig csak a neki megfelelő komplementer állhat. Ez azt jelenti, hogy az A bázissal szemben T, a C bázissal szemben csak G állhat és fordítva. A bázisokat a nekik megfelelő nagybetűkkel reprezentáljuk. Definiáld a függvényt, amely az előbb megállapított szabályok alapján eldönti két DNS szálról, hogy egymás komplementerei-e! Feltehető, hogy a két sorozat csak a megfelelő nagybetűkből áll.

```
isComplementary :: String -> String -> Bool
isComplementary [] []
isComplementary "ATCG" "TAGC"
not $ isComplementary "" "TAGC"
not $ isComplementary "TAGC" ""
not $ isComplementary "ATCG" "TAGG"
not $ isComplementary "AA" "TTT"
not $ isComplementary "AAA" "TT"
not $ isComplementary "AA" "TTT"
isComplementary "TTT" "AAA"
isComplementary "TGAACAGCGAATTGCCCG" "ACTTGTCGCTTAACGGGC"
not $ isComplementary (repeat 'A') (repeat 'C')
not $ isComplementary (repeat 'G') ("CCCCCCCCT" ++ repeat 'C')
not $ isComplementary ("CCCCCCCCT" ++ repeat 'C') (repeat 'G')
not $ isComplementary ("TTTTTTG" ++ repeat 'T') (repeat 'A')
```

Szerepel legalább n-szer egy elem? (2 pont)

Állapítsd meg, hogy egy elem szerepel-e egy tetszőleges listában legalább n-szer!

```
atLeastNtimes :: Eq a => Int -> a -> [a] -> Bool
atLeastNtimes 2 'b' "abba" == True
atLeastNtimes 0 3 [] == True
atLeastNtimes 1 3 [] == False
atLeastNtimes 0 3 [1..5] == True
atLeastNtimes 0 3 [1..5] == True
atLeastNtimes 1 4 [1..5] == True
atLeastNtimes 2 3 [1..5] == False
atLeastNtimes (-1) 3 [1..5] == True
atLeastNtimes (-1) 3 [] == True
atLeastNtimes 100 'a' (repeat 'a') == True
atLeastNtimes 1000 'b' (cycle "abba") == True
```

Kiemelt vásárlók (2 pont)

Definiáljuk a valuableCustomer függvényt, mely egy üzlet vásárlóinak nevét és az általuk vásárolt termékek értékeit tartalmazó listából visszaadja azon vásárlók neveit, akik vásároltak a megadott határt elérő értékű terméket. Amennyiben egy vásárló többször vásárolt az adott határ felett, akkor többször szerepeljen a neve a listában.

```
valuableCustomer :: Int -> [(String, Int)] -> [String]
valuableCustomer 2000 [("A", 500), ("B", 2000)]  == ["B"]
valuableCustomer 1000 [("A", 1000), ("B", 500), ("B", 1000)]  == ["A", "B"]
valuableCustomer 1000 [("A", 1000), ("A", 2000), ("B", 1000)]  == ["A", "A",
"B"]
valuableCustomer 1000 [("A", 500)]  == []
valuableCustomer 1000 []  == []
valuableCustomer 1 (replicate 20 ("A", 0)  ++ replicate 20 ("B", 1)) ==
replicate 20 "B"
```

```
take 10 (valuableCustomer 100 (zip (cycle (map (:[])['A'..'Z'])) [1..])) ==
["V","W","X","Y","Z","A","B","C","D","E"]
```

Feltételes, részleges forgatás (2 pont)

Definiáld a partialReverse függvényt, amely kiválogatja egy paraméterül kapott listából az adott hosszúságú szavakat és az eredményül adott listában minden második elemet megfordít. A szavak maguk végesek, de a lista lehet végtelen.

```
partialReverse :: Int -> [String] -> [String]
partialReverse 10 [] == []
partialReverse 2 ["alma"] == []
partialReverse (-4) ["alma"] == []
partialReverse 5 ["teszt"] == ["teszt"]
partialReverse 4 ["alma", "fa", "cica", "házi", "csak"] == ["alma", "acic",
"házi", "kasc"]
take 5 (partialReverse 2 (repeat "ab")) == ["ab", "ba", "ab", "ba", "ab"]
```

Karakterláncok összehasonlítása (2 pont)

Definiáljuk a match függvényt, mely összehasonlítja két karakterlánc azonos pozíción található karaktereit, különbözőség esetén Just adatkonstruktorban visszaadja az első karakterlánc eltérő karakterét. Amennyiben nincs különbség, vagy a bal oldali lista prefixe a jobb oldali listának, Nothing adatkonstruktort ad vissza.

```
match :: String -> String -> Maybe Char
match "alma" "alma" == Nothing
match "alma" "asztal" == Just 'l'
match "asztal" "alma" == Just 's'
match "" "" == Nothing
match "alma" "" == Just 'a'
match "" "alma" == Nothing
match "alma" "almafa" == Nothing
match "abrakadabra" "abbrakadabra" == Just 'r'
match (reverse ('b':(replicate 20 'a'))) (replicate 21 'a') == Just 'b'
match ((replicate 10 'a') ++ ('b':(replicate 10 'a'))) (replicate 21 'a') ==
Just 'b'
match (replicate 21 'a') (reverse ('b':(replicate 20 'a')))  == Just 'a'
match (replicate 20 'a') (reverse ('b':(replicate 20 'a')))  == Nothing
match (replicate 21 'a') (repeat 'a')  == Nothing
match (repeat 'a') (replicate 21 'a') == Just 'a'
match (replicate 100 'a') (cycle "aa")  == Nothing
match (cycle "ab") ("abababababab" ++ repeat 'a') == Just 'b'
```

Alapértelmezett érték (2 pont)

Definiáljuk az applyWithDefault függvényt, amely egy lista azon elemeit melyek teljesítik a megadott feltételt, leképezi a megadott függvény szerinti értékre, azokat pedig amelyek nem teljesítik azt, a megadott alapértelmezett értékre képezi le!

```
applyWithDefault :: (a -> Bool) -> (a -> b) -> b -> [a] -> [b]
applyWithDefault (0<) pred 0 [4,3..(-1)] == [3,2,1,0,0,0]
applyWithDefault even (`div` 2) (-1) [1,2,3,4] == [-1,1,-1,2]
applyWithDefault (not . null) head '-' ["alma", "", "", "barack"] == "a--b"
applyWithDefault (const True) id 0 [1..100] == [1..100]
applyWithDefault (const False) id 0 [1..100] == replicate 100 0
applyWithDefault (const True) id 0 [] == []
take 11 (applyWithDefault odd Just Nothing [1..]) == [Just 1,Nothing,Just
```

3,Nothing,Just 5,Nothing,Just 7,Nothing,Just 9,Nothing,Just 11]
Monoton növekvő prefix (2 pont)
Definiáljuk a longestAscendingPrefix függvényt, amely megadja egy lista leghosszabb olyan prefixét, amely prefix elemeire alkalmazva a függvényt, az monoton növekvő sorozatot ad!

```
longestAscendingPrefix :: Ord b => (a -> b) -> [a] -> [a]
longestAscendingPrefix (*2) [] == []
longestAscendingPrefix (*0) [1..10] == [1,2,3,4,5,6,7,8,9,10]
longestAscendingPrefix id ['a'..'z'] == "abcdefghijklmnopqrstuvwxyz"
longestAscendingPrefix (`mod` 5) [1..10] == [1,2,3,4]
longestAscendingPrefix even [1,4,2,6,8,9,3,2,1] == [1,4,2,6,8]
longestAscendingPrefix odd [1,4,2,6,8,9,3,2,1] == [1]
longestAscendingPrefix (`mod` 5) [1..] == [1,2,3,4]
take 20 (longestAscendingPrefix (> 10) [1..]) ==
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Állatkert
Adattípus definiálása (2 pont)
Szeretnénk reprezentálni egy állatkert állatait, illetve azt hogy az állatok milyen eleséget fogyasztanak.

Az állatok három különböző eleségeket (Food) fogyasztanak, ezek lehetnek: széna (Hay), banán (Banana) és lárva (Larva).

Az állatkertben háromféle állatfaj (Animal) él, ez lehet: majom (Monkey), flamingó (Flamingo) és kecske (Goat). Mindegyik konstruktor tartalmazza az állat által fogyasztott ételt (Food), és hogy naponta hány egységet kell elfogyasztania belőle (Int).

Definiáld az algebrai adatszerkezeteket és a fordítótól kérd az Eq és Show típusosztályok automatikus példányosítását mindkettőnél!

Etetés (2 pont)
Adott az állatok listája és a rendelkezésre álló eledelek mennyisége. Szeretnénk megvizsgálni, hogy egy adott ételkészlettel meg tudjuk-e etetni az állatkert összes állatát. Az ételkészletet egy rendezett párokból álló lista reprezentálja (amely véges), melynek első komponense az étel neve, második komponense pedig az ételből készleten lévő mennyiség. Az ételkészletről feltehető, hogy egy étel neve a listában csak egyszer szerepel. Az állatkertet egy állatokat tartalmazó lista reprezentálja, amely szintén véges.

Definiáld a függvényt, amely megnézi, hogy minden állatnak tudunk-e adni a napi adagjának megfelelő mennyiségű, az állat által fogyasztott ételt! Feltehető, hogy egy állatfajta csak egyféle ételt fogyaszt.

```
canFeedAll :: [Animal] -> [(Food, Int)] -> Bool
canFeedAll [] [] == True
canFeedAll [Flamingo Larva 5] [] == False
canFeedAll [] [(Banana, 3)] == True
canFeedAll [Monkey Banana  6, Goat Hay 12, Flamingo Larva 8, Flamingo Larva 4,
Monkey Banana 9] [(Banana, 20), (Hay, 15), (Larva, 12)] == True
canFeedAll [Flamingo Larva 8, Monkey Banana  6, Goat Hay 12, Flamingo Larva 4,
Monkey Banana 9] [(Hay, 15), (Banana, 20), (Larva, 12)] == True
canFeedAll [Monkey Banana 13, Goat Hay 12, Flamingo Larva 8, Flamingo Larva 4,
```

Monkey Banana 9] [(Banana, 20), (Hay, 15), (Larva, 12)] == False
canFeedAll [Goat Hay 12, Monkey Banana 13, Flamingo Larva 8, Flamingo Larva 4, Monkey Banana 9] [(Larva, 12), (Banana, 20), (Hay, 15)] == False
Lista feldarabolása (3 pont)
Definiálj függvényt, amely feldarabol egy pozitív egész számokat tartalmazó listát a lehető leghosszabb részekre úgy, hogy a részlistákban a számok összege ne haladja meg a megadott értéket! Ha az eredeti lista tartalmaz a megadott értéknél nagyobb számokat, azok ne szerepeljenek az eredmény listákban. Feltehetjük, hogy a felső korlátként megadott érték nem negatív!

```
chop2N :: Integral a => a -> [a] -> [[a]]

chop2N 13 [] == []
chop2N 1 [6,7,1,6,7] == [[1]]
chop2N 13 [1..10] == [[1,2,3,4],[5,6],[7],[8],[9],[10]]
chop2N 13 [6,7,1,6,7] == [[6,7],[1,6],[7]]
chop2N 12 [6,7,1,6,7] == [[6],[7,1],[6],[7]]
chop2N 13 [13,1,12,2,12,1] == [[13],[1,12],[2],[12,1]]
chop2N 13 [6,15,7,1,15,6,7] == [[6,7],[1,6],[7]]
(chop2N 11 $ take 14 $ repeat 1) == [[1,1,1,1,1,1,1,1,1,1,1],[1,1,1]]
(chop2N 13 $ take 14 $ repeat 1) == [[1,1,1,1,1,1,1,1,1,1,1,1,1],[1]]
(take 3 $ chop2N 7 $ repeat 2) == [[2,2,2],[2,2,2],[2,2,2]]
```

Hárombetűs soroló (3 pont)
Készítsd el a threeChain függvényt, amely egy szavakat tartalmazó listájáról eldönti, hogy a szomszédos elemek egy karakterben különböznek-e! A függvény igazat ad vissza, ha az alábbiak teljesülnek:

A sorozatban csak hárombetűs szavak vannak.
A szomszédos elemek csak egy betűben különböznek.
Egy szó csak egyszer szerepel a listában.

```
threeChain :: [String] -> Bool
threeChain [] == True
threeChain ["lel"] == True
threeChain ["fej","fejt"] == False
threeChain ["lel","jaj"] == False
threeChain ["lel","fel","kel"] == True
threeChain ["tol","tel","fel","fej","tej"] == True
threeChain ["tol","tel","fel","fej","tej","tel"] == False
threeChain ["eee",['e','e'..],"eef"] == False
threeChain [[a,b,c] | a <- cycle "ab", b <- "cd", c <- "ef"] == False
```

Mindent vagy semmit (3 pont)
Írjunk egy fragileUpdate nevű függvényt, amely egy lista adott indexén lévő elem értékét megpróbálja frissíteni egy a -> Maybe a függvény segítségével, amennyiben a függvény Nothing-ba képez, vagy az index túl nagy, vagy túl kicsi, akkor térjen vissza a függvényünk Nothing-al. Amennyiben sikeres volt a frissítés, adjuk vissza az új listánkat, Just-ba csomagolva.

A lista indexelése kezdődjön 0-tól.

```
fragileUpdate :: Int -> (a -> Maybe a) -> [a] -> Maybe [a]
```

Megjegyzés: A tesztesetek futtatásához szükséges a Data.Maybe modul importálása.

```
fragileUpdate 5 (\ x -> Just 5) [1..10] == Just [1,2,3,4,5,5,7,8,9,10]
fragileUpdate 0 (\ x -> Just 5) [1..10] == Just [5,2,3,4,5,6,7,8,9,10]
```

```
fragileUpdate 9 (\ x -> Just 5) [1..10] == Just [1,2,3,4,5,6,7,8,9,5]
isNothing $ fragileUpdate 10 (\ x -> Just 5) [1..10]
isNothing $ fragileUpdate (-5) (\ x -> Just 5) [1..10]
isNothing $ fragileUpdate 5 (const Nothing) [1..10]
isNothing $ fragileUpdate 0 (const Nothing) [1..10]
isNothing $ fragileUpdate 10 (const Nothing) [1..10]
isNothing $ fragileUpdate (-5) (const Nothing) [1..10]
isNothing $ fragileUpdate 5 (const Nothing) [1..]
take 10 (fromJust (fragileUpdate 5 (\ x -> Just 5) [1..])) ==
[1,2,3,4,5,5,7,8,9,10]
```