7. Gaia
Haskell (Programazio Funtzionala)

7.1. Sarrera

7.1.1. Helburua

Programazio funtzionalaren ezaugarriak aztertzea da gai honetako helburua. Horretarako Haskell lengoaia erabiliko da.

Lengoaia funtzional bat ikasteak errekurtsibitatearen teknikan sakontzea dakar alde batetik. Errekurtsibitatearen barruan, hasteko, oinarrizko errekurtsibitatea berrikusiko da eta gero murgilketa bezala ezagutzen den teknika landuko da. Beste aldetik, zerrenden eraketa bezala ezagutzen den baliabidea eta sarrera/irteera ere landuko dira.

7.1.2. Haskell

Haskell programazio-lengoaia funtzionala da.

Haskell lengoaian programa bat funtzio multzo bat izan ohi da. Funtzio bakoitza era independientean exekuta daiteke eta baita beste funtzio batetik deituz ere (kasu bietan parametro formalen ordez argumentu errealak ipiniz).

Haskell-en web orri ofiziala http://haskell.org da eta bertatik sistema eragile desberdinetarako Haskell plataforma instala daiteke. Haskell erabiltzen hasteko era errazena WinHugs interpretatzailea da eta http://www.haskell.org/hugs/ helbidetik eraitsi daiteke. GHC konpilatzailea ere eskura daiteke http://www.haskell.org/ghc/ helbidean.

7.2. Oharrak

7.2.1. Maiuskula eta minuskulen erabilera izenetan

Maiuskulak eta minuskulak ondo erabiltzeko honako arau hauek jarraitu behar dira:

- Datu-moten izenak maiuskulaz hasten dira: Int, Bool, Char, …
- Datu-mota berriak definitzerakoan, eragiketa eraikitzaileen izenak maiuskulaz hasten dira: Phutsa, Pilaratu, Ahutsa, Eraiki, …
- Moten ordez ipintzen diren aldagaien izenak minuskulaz hasten dira: t.
- True eta False konstanteak maiuskulaz hasten dira.
- Eraikitzaileak ez diren beste eragile denak eta aldagaien izenak minuskulaz hasten dira (maiuskulak erdian ipintzea egon arren): bikoitia, bakoitia, leh, hond, badago, luzera, x, s, r, p, q, zatb, kenduLuz, kenduLuzBik, …

7.2.2. Azalpenak ipintzeko bi era

Haskell-ez idatzitako programetan azalpenak ipini nahi izanez gero, bi aukera daude:

- -- erabiliz:

  -- sinboloen atzetik lerroa bukatu arte datorren dena azalpena izango da.

  Adibidea:

  --Hau azalpen bat da eta lerroa bukatzen denean bukatuko da azalpena

- {- eta -} erabiliz:

  {- eta -} sinboloen artean doan dena azalpena da. Lerro batetik bestera pasatuta ere azalpenak aurrera jarraitzen du.

  Adibidea:

  {- Hau azalpena da baina
  lerroz alda gaitezke. -}

7.3. Haskell-en aurredefinituta dauden oinarrizko motak

7.3.1. Bool

- Balioak: {True, False}
- Eragiketak:
  - ✓ && (and)
  - ✓ || (or)
  - ✓ not (not)
- Propietateak (φ, ψ eta γ expresio boolearrak direla kontsideratuz):
  - ✓ φ && ψ ≡ ψ && φ (&& trukakorra da)
  - ✓ φ || ψ ≡ ψ || φ (|| trukakorra da)
  - ✓ not(φ && ψ) ≡ (not φ) || (not ψ)
  - ✓ not(φ || ψ) ≡ (not φ) && (not ψ)
  - ✓ φ && (ψ || γ) ≡ (φ && ψ) || (φ && γ)
  - ✓ False || φ ≡ φ (False || eragilearentzat elementu neutroa da)
  - ✓ False && φ ≡ False
  - ✓ True || φ ≡ True
  - ✓ True && φ ≡ φ (True && eragilearentzat elementu neutroa da)
  - ✓ …

7.3.2. Int

- Balioak: tarte mugatu bateko zenbaki osoak (minBound, maxBound)
- Eragile aritmetikoak:
  - ✓ - (zeinu negatiboa)
  - ✓ + (batuketa)
  - ✓ – (kenketa)
  - ✓ * (biderketa)
  - ✓ ^ (berredura, 2 ^ 3 = 8)
  - ✓ div (zatiketa osoa; idazteko bi era: 16 `div` 3 = 5 edo div 16 3 = 5, eta ` karakterea P letraren ondoan dagoen tildea da)
  - ✓ mod (zatiketa osoaren hondarra; idazteko bi era: 16 `mod` 3 = 1 edo mod 16 3 = 1, eta ` karakterea P letraren ondoan dagoen tildea da)
- Propietateak:
  - ✓ x + y = y + x Trukakortasuna
  - ✓ x + (y + z) = (x + y) + z Elkarkortasuna
  - ✓ x * (y + z) = (x * y) + (x * z) Banakortasuna
  - ✓ 0 + x = x 0 neutroa da batuketarentzat
  - ✓ 1 * x = x 1 neutroa da biderketarentzat
  - ✓ …

7.3.3. Integer

- Balioak: zenbaki osoak inolako mugarik gabe
- Eragile aritmetikoak: Int kasukoak
- Propietateak: Int kasukoak

7.3.4. Float, Double

- Balioak: zenbaki errealak
- Eragile aritmetikoak: Ez ditugu erabiliko
- Propietateak: Int kasukoen antzekoak

7.3.5. Char

- Balioak: karaktereak ('a', …, 'z', 'A', …, 'Z', …) 0-ren teklaren ondoan dagoen teklako apostrofoa erabiliz
- Eragileak: Ez ditugu erabiliko
- Propietateak: Ez ditugu aztertuko

7.3.6. Oinarrizko motentzako eragile erlazionalak

Jarraian erakusten diren eragile erlazionalak aipatu diren oinarrizko mota denentzat balio dute:

- ✓ == (berdin)
- ✓ /= (ezberdin)
- ✓ > (handiagoa)
- ✓ >= (handiagoa edo berdina)
- ✓ < (txikiagoa)
- ✓ <= (txikiagoa edo berdina)

7.3.7. Oinarrizko motentzat errekurtsiboak ez diren eragiketa berrien definizioa

Lehenago aipatu diren eragiketak erabiltzeaz gain eragiketa berriak ere defini eta erabil daitezke.

Eragiketak definitzerakoan mota eta ekuazioak emando ditugu. Ekuazio bakoitzari zenbaki bat egokituko diogu geroago ekuazio hori erreferentziatu ahal izateko.

Funtzio berrien motak ematerakoan, hasteko, datuen motak ipiniko dira -> geziaz bereiztuta eta bukatzeko emaitzaren mota ipiniko da, hau ere datuen motetatik -> geziaz bereiztuta. Beti emaitza bakarra dago. Beraz azkeneko mota emaitzari dagokiona izango da eta aurreko denak datuenak izango dira. Gerta daiteke daturik ez egotea, datu bakarra egotea edo datu bat baino gehiago egotea, baina beti emaitza bakarra egongo da.

Bost adibide emango dira jarraian:

- **pi2**: funtzio hau funtzio konstantea da, izan ere ez du daturik behar eta beti 3.1415 zenbaki erreala itzuliko du.

  Eragiketaren **mota**:
  ```
  pi2:: Float
  ```
  Eragiketa definitzen duen **ekuazioa**:
  ```
  pi2 = 3.1415
  ```

Oharra: funtzio honi pi2 deitu zaio Haskell-en **pi** funtzioa aurredefinituta dagoelako.

- **f**: sarrerako datuak onartu arren, beti balio bera itzultzen duten funtzioak ere defini daitezke.

  Eragiketaren **mota**:
  ```
  f:: Int -> Int
  ```
  Eragiketa definitzen duen **ekuazioa**:
  ```
  f x = 100
  ```

  Definizioaren arabera, f funtzioari edozein zenbaki oso emanda (x), funtzioak beti 100 balioa itzuliko du.

- **bikoitia**: zenbaki oso bat emanda, True itzuliko du bikoitia bada eta False bakoitia bada.

  Eragiketaren **mota**:
  ```
  bikoitia:: Int -> Bool
  ```
  Eragiketa definitzen duen **ekuazioa**:
  ```
  bikoitia x = (x `mod` 2) == 0
  ```

  bikoitia funtzioak x balioarentzat x `mod` 2 eta 0 konparatzearen balioa itzuliko duela adierazten da ekuazio horren bidez (konparazioaren emaitza True edo False izango da).

  Eragiketa bera definitzeko beste aukera bat honako hau izango litzateke:
  ```
  bikoitia x
      | (x `mod` 2) == 0     = True          (1)
      | otherwise            = False         (2)
  ```

  Definizio hori honela ulertu beharko genuke:
  x `mod` 2 eta 0 berdinak badira True itzuli eta bestela False itzuli.

- **bakoitia**: zenbaki oso bat emanda, True itzuliko du zenbaki hori bakoitia bada eta False itzuliko du zenbaki hori bikoitia bada.

  Eragiketaren **mota**:
  bikoitia:: Int -> Bool
  Eragiketa definitzen duen **ekuazioa**:
  bakoitia x = not(bikoitia x) (1)

  Kasu honetan, x zenbaki oso bat emanda, *bakoitia* funtzioak x zenbaki horrentzat itzuliko duena *bikoitia* funtzioak x zenbaki horrentzat itzuliko lukeenaren kontrakoa dela adierazten da. Beraz, hasteko *bikoitia* funtzioak x balioarentzat itzultzen duena kalkulatuko da eta gero balio horren aurkakoa hartuko da *not* eragilearen bidez.

  Adibide honetan ikusten den bezala, funtzio berri bat definitzerakoan aurretik definituta dauden funtzioak ere erabil daitezke.

- **hand3**: hiru zenbaki oso emanda handiena itzuliko du

  Eragiketaren **mota**:
  hand3:: Int -> Int -> Int -> Int
  Eragiketa definitzen duten **ekuazioak**:

```
hand3 x y z
    | x >= y && x >= z      = x      (1)
    | y > x && y >= z       = y      (2)
    | otherwise             = z      (3)
```

  Definizio hori honela ulertu beharko genuke:
  x >= y && x >= z betetzen bada, orduan x balioa itzuli, bestela y > x && y >= z betetzen bada, orduan y itzuli eta bestela z.

  Adibide honetan bezala baldintza-zerrenda bat agertzen denean, baldintzak goitik behera aztertuko dira Haskell-en eta betetzen den lehenengo baldintzaren kasuari dagokion emaitza itzuliko da.

7.3.8. Oinarrizko errekurtsioa zenbaki osoentzat

Zenbaki osoen gaineko eragiketak burutzen dituzten funtzio errekurtsibo batzuk azalduko dira atal honetan.

- **bider**: funtzio honek, x eta y zenbaki osoak emanda, bien arteko biderkadura kalkulatzen du x balioa y aldiz batuz. Kasu berezi bezala, y-ren balioa negatiboa denean errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

```
bider:: Int -> Int -> Int
```

  Eragiketa definitzen duen **ekuazioa**:

```
bider x y
  | y < 0              = error "Bigarren balioa negatiboa da."
  | x == 0 || y == 0   = 0
  | x == 1             = y
  | otherwise          = x + bider x (y – 1)
```

- **bneg**: funtzio honek, x eta y zenbaki osoak emanda, bien arteko biderkadura kalkulatzen du. Bigarren balioa negatiboa denean ere emaitza ondo kalkulatuko du. Definizioan aurretik definitu den *bider* funtzioa erabiltzen da. Funtzio hau ez da errekurtsiboa, ez baitio bere buruari deitzen. Errekurtsibitatea *bider* funtzioan dago.

  Eragiketaren **mota**:

```
bneg:: Int -> Int -> Int
```

  Eragiketa definitzen duen **ekuazioa**:

```
bneg x y
  | x == 0 || y == 0     = 0
  | (x < 0) && (y < 0)       = bider (-x) (-y)
  | (x > 0) && (y > 0)       = bider x y
  | (x < 0) && (y > 0)       = bider x y
  | (x > 0) && (y < 0)       = bider (-x) (-y)
```

- **errusiar**: funtzio honek, x eta y zenbaki osoak emanda, bien arteko biderkadura kalkulatzen du *biderketa errusiarra* bezala ezagutzen den metodoa jarraituz. Kasu berezi bezala, y-ren balioa negatiboa denean errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

```
errusiar:: Int -> Int -> Int
```

  Eragiketa definitzen duen **ekuazioa**:

```
errusiar x 0 = 0
errusiar x y
  | y < 0                  = error "El segundo valor es negativo."
  | (y `mod` 2 == 0)       = errusiar (x + x) (y `div` 2)
  | otherwise              = x + errusiar (x + x) (y `div` 2)
```

- **zatios**: funtzio honek, x eta y zenbaki osoak emanda, x eta y-ren arteko zatidura osoa kalkulatzen du, horretarako x balioari y balioa zenbat aldiz kendu ahal zaion zenbatuz. Kasu berezi bezala, y-ren balioa zero denean errore-mezua aurkeztuko du. Gainera x edota y negatiboa baldin bada ere, errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

  ```
  zatios:: Int -> Int -> Int
  ```

  Eragiketa definitzen duen **ekuazioa**:

  ```
  zatios x y
    | y == 0              = error "Zatitzailea 0"
    | (x < 0) || (y < 0)  = error "Gutxienez bietako bat negatiboa da."
    | x < y               = 0
    | otherwise           = 1 + zatios (x - y) y
  ```

- **zatihond**: funtzio honek, x eta y zenbaki osoak emanda, x eta y-ren arteko zatiketa osoaren hondarra kalkulatzen du. Horretarako x balioari eta ondoren lortzen diren kendurei y balioa kenduz joango da, y baino txikiagoa den balio bat gelditu arte. Kasu berezi bezala, y-ren balioa zero denean errore-mezua aurkeztuko du. Gainera x edota y negatiboa baldin bada ere, errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

  ```
  restodiv:: Int -> Int -> Int
  ```

  Eragiketa definitzen duen **ekuazioa**:

  ```
  restodiv x y
    | y == 0              = error "Divisor 0"
    | (x < 0) || (y < 0)  = error "Gutxienez bietako bat negatiboa da."
    | x < y               = x
    | otherwise           = restodiv (x - y) y
  ```

7.4. Zerrendak

Zerrenda bat jarraian erakusten den eran adierazten da: [2, 7, 35, -8, 0]
Zerrenda hori zenbaki osoz osatuta dago eta bere mota [Int] da. Zerrenda hutsa honako era honetan adierazten da: []

Zerrenden mota parametrodun mota da, [t], eta t parametroaren arabera balio boolearrez osatutako zerrendak, zenbaki osoz osatutako zerrendak eta abar eduki ditzakegu. Beraz aurretik definitutako edozein mota erabiliz mota horretako elementuz osatutako zerrendak defini daitezke baina zerrenda bateko elementu denak mota berekoak izan behar dute.

Adibideak:
[True, True, True, False, True] Bere mota [Bool] da (boolearrez osatutako zerrenda)
[[0, 5], [], [77, -50, 3]] Bere mota [[Int]] da (osoez osatutako zerrendez eratutako zerrenda)

7.4.1. Eragiketa eraikitzaileak zerrendentzat

Datu-mota bati dagozkion eragiketa eraikitzaileak mota horretako balioak zein diren adierazteko dira.

Edozein zerrenda jarraian aipatzen diren bi eragileak erabiliz eraiki daiteke:

- [] **Zerrenda hutsa sortu**
- : **Ezkerretik elementu bat erantsi**

Lehenengo eragiketak zerrenda hutsa sortzeko balio du eta bigarrenak ezkerretik elementuak banan-banan erantsiz edo sartuz joateko balio du.

Esate baterako [6, 10] zerrenda honako era honetan eraikiko litzateke: 6:10:[]
Hasteko zerrenda hutsa sortuko litzateke, gero 10 zenbakia gehituko litzaioke zerrenda hutsari eta bukatzeko 6 balioa gehituko litzaioke dagoeneko 10 zenbakia duen zerrendari.

[] → [10] → [6, 10]

[2, 7, 35, -8, 0] zerrenda honako era honetan eraikiko litzateke: 2:7:35:-8:0:[]

Haskell-entzat [6,10] eta 6:10:[] baliokideak dira, biak onartzen ditu, baina espezifikazio ekuazionalaren bidez eragiketa berriak definitzeko eta eragiketa berri horien propietateak frogatzeko bigarren aukera egokiagoa da.

Edozein zerrenda jarraian aipatzen diren bi eskema hauetakoren batera egokitzen da:

- ✓ [] (zerrenda hutsa da)
- ✓ x:s (ez da zerrenda hutsa eta bere lehenengo elementua x da eta beste elementu denek s azpizerrenda osatzen dute).

[2, 7, 35, -8, 0] zerrenda 2:7:35:-8:0:[] eran idatz daiteke
x: s
s azpizerrenda 7:35:-8:0:[] zerrenda da

Zerrenda batek bi elementu edo gehiago dituenean

x:z:s

erakoa dela ere esan daiteke, komeni bada. Lehenengo elementua **x** izango da, bigarrena **z** eta gainontzeko elementuek **s** azpizerrenda osatzen dute.

[2, 7, 35, -8, 0] es 2:7:35:-8:0:[]
x:z:s

Eragiketa eraikitzaileen motak honako hauek dira:

```
[] :: [t]
: :: t -> [t] -> [t]
```

Lehenengo eragiketaren motak, hau da, []-en motak, eragiketa horrek geuk nahi dugun motako (t motako) zerrenda hutsa sortzen duela adierazten du.

Bigarren eragiketaren kasuan, : eragileari t motako elementu bat eta t motako zerrenda bat emanda t motako zerrenda bat itzultzen duela adierazten da. Eragiketen motak eta elementuen motak ematerakoan **::** notazioa erabiltzen da. Funtzioetan sarrerako datuak parentesi artean joango dira komaz bereiztuta eta emaitza -> sinboloen ondoren joango da. Sarrerako daturik ez badago, emaitzaren mota bakarrik emango da, [] eragilearen kasuan bezala. Funtzioen kasuan gerta daiteke sarrerako daturik ez egotea, datu bakarra egotea edo datu bat baino gehiago egotea, baina beti emaitza bat egongo da.

7.4.2. Errekurtsiboak ez diren eragiketa batzuk zerrendentzat

Jarraian zerrendekin kalkuluak burutzeko balio duten eta errekurtsiboak ez diren eragiketa batzuk definituko dira. Funtzio bat edo eragiketa bat definitzeko, funtzioaren mota eta funtzioak zer egiten duen zehazten duten ekuazioak eman behar dira.

- **leh**: Zerrenda bat emanda, zerrendako lehenengo elementua itzuliko du. Zerrenda hutsa bada, errorea sortuko da.

  Adibideak:

  ```
  leh [4, 7, 8] = 4
  leh [False, True] = False
  leh [[0, 5], [], [77, -50, 3]] = [0, 5]
  ```

  Eragiketaren **mota**:

  ```
  leh:: [t] -> t
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  leh [] = error "Zerrenda hutsa. Ez dago lehenengo elementurik."   (1)
  leh (x:s) = x                                                     (2)
  ```

  Errore-mezu bat aurkezteko Haskell-eko *error* funtzioa erabiltzen da eta zein mezu aurkeztu nahi den zehaztu beharko da.

- **hond**: Zerrenda bat emanda, zerrendako lehenengo elementua kenduz lortzen den zerrenda itzuliko du. Zerrenda hutsa bada, errorea sortuko da.

  Adibideak:

  hond [4, 7, 8] = [7, 8]
  hond [False, True] = [True]
  hond [[0, 5], [], [77, -50, 3]] = [[], [77, -50, 3]]

  Eragiketaren **mota**:

  hond:: [t] -> [t]

  Eragiketa definitzen duten **ekuazioak**:

  hond [] = error "Zerrenda hutsa. Ez dago hondarrik." (1)
  hond (x:s) = s (2)

  Zerrenda hutsari dagokion errore-mezua aurkezteko Haskell-eko *error* funtzioa erabili da eta aurkeztu nahi den mezua zein den zehaztu da komatxoen artean.

- **hutsa_da**: Zerrenda bat emanda, True itzuliko du zerrenda hutsa bada eta False itzuliko du zerrenda hutsa ez bada.

  Adibideak:

  hutsa_da [] = True
  hutsa_da [4, 7, 8] = False
  hutsa_da [False, True] = False

  Eragiketaren **mota**:

  hutsa_da:: [t] -> Bool

  Eragiketa definitzen duten **ekuazioak**:

  hutsa_da [] = True (1)
  hutsa_da (x:s) = False (2)

  Emandako zerrenda hutsa bada (hau da, [] erakoa bada), orduan True itzuliko da. Emandako zerrenda hutsa ez bada (hau da, x:s erakoa bada, x zerrendako lehenengo elementua izanda eta s zerrendako gainontzeko elementuez osatutako azpizerrenda izanda), orduan False itzuliko da.

7.4.3. Errekurtsiboak diren eragiketa batzuk zerrendentzat

- **badago**: t motako elementu bat eta t motako zerrenda bat emanda, True balioa itzuliko du elementua zerrendan baldin badago eta False itzuliko du ez badago.

  Adibideak:

  ```
  badago 8 [] = False
  badago 8 [4, 8, 7, 8] = True
  badago False [False, True, True] = True
  badago [10, 9] [[0, 5], [], [77, -50, 3]] = False
  ```

  Eragiketaren **mota**:

  ```
  badago:: Eq t => t -> [t] -> Bool
  ```

  Oharra: Kasu honetan elementu bat zerrenda batean agertzen al den erabakitzeko, elementuak konparatu edo alderatu beharra dago. Haskell-en aurredefinituta dauden motetako elementuen artean konparaketa buru daiteke inolako arazorik gabe, baina erabiltzaileak definitutako moten kasuan konparaketa definitu gabe egon daiteke. Horrela, t motarentzat konparaketa definitu gabe baldin badago, *badago* izeneko funtzioak ez du zentzurik mota horrentzat. Horregatik, *badago* funtzioaren mota ematerakoan, konparaketa definituta duten t motentzat bakarrik balioko duela adierazten da **Eq t =>** espresioaren bidez.

  Eragiketa definitzen duten **ekuazioak**:

  ```
  badago x [] = False                    (1)
  badago x (y:s)
      | x == y    = True                 (2)
      | x /= y    = badago x s           (3)
  ```

  Ekuazio horien bitartez x elementua zerrenda batean agertzen al den ala ez erabakitzen da. Zerrenda hutsa bada (hau da, [] erakoa bada), x ez da hor egongo eta False itzultzen da zuzenean. Zerrenda hutsa ez bada, (hau da, y:s erakoa bada, zerrendako lehenengo elementua y dela eta zerrendako beste elementu denez osatutako azpizerrenda s dela kontsideratuz) x balioa y:s zerrendako lehenengo elementuaren berdina bada (hau da, x balioa y-ren berdina bada) erantzuna True da zuzenean baina x balioa y:s zerrendako lehenengo elementuaren berdina ez bada (hau da x eta y berdinak ez badira), orduan x elementua s azpizerrendan ba al dagoen begiratu beharko da.

- ++: t motako bi zerrenda emanda, bata besteari erantsiz edo biak elkartuz lortzen den zerrenda itzuliko du.

  Adibideak:

  ```
  [] ++ [] = []
  [4, 8, 7, 8] ++ [] = [4, 8, 7, 8]
  [1, 8] ++ [4, 8, 7] = [1, 8, 4, 8, 7]
  [False, True, True] ++ [False, False] = [False, True, True, False, False]
  ```

Eragiketaren **mota**:

```
(++):: [t] -> [t] -> [t]
```

Eragiketa definitzen duten **ekuazioak**:

```
[] ++ s = s                 (1)
(x:r) ++ s = x:(r ++ s)     (2)
```

Bigarren ekuazioak x:r eta s zerrendak elkartzeko lehenengo r eta s zerrendak elkartu eta gero r ++ s zerrenda berriari x elementua ezkerretik gehitu behar zaiola dio.

- **luzera**: t motako zerrenda bat emanda, elementu-kopurua (zerrenda horretan zenbat elementu dauden) itzuliko du.

  Adibideak:

  ```
  luzera [] = 0
  luzera [5, 8, 7, 8] = 4
  luzera [False, True, True] = 3
  luzera [[0, 5], [], [77, -50, 3]] = 3
  ```

  Eragiketaren **mota**:

  ```
  luzera:: [t] -> Int
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  luzera [] = 0                    (1)
  luzera (x:r) = 1 + (luzera r)    (2)
  ```

  Emandako zerrenda hutsa bada (hau da, [] erakoa bada), orduan luzera edo elementu-kopurua 0 da. Emandako zerrenda hutsa ez bada (hau da, x:r erakoa bada), zerrendak gutxienez x elementua du eta gero r azpizerrendako elementu-kopurua zenbatu beharko da.

- **bikop**: Int motako zerrenda bat emanda, zerrendan zenbat elmentu bikoiti dauden itzuliko du.

  Adibideak:

  ```
  bikop [] = 0
  bikop [5, 8, 7, 8] = 2
  bikop [7, 11, 9] = 0
  bikop [0, 3, 3, 5] = 1
  ```

  Eragiketaren **mota**:

  ```
  bikop:: [Int] -> Int
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  bikop [] = 0                              (1)
  bikop (x:r)
      | bikoitia x    = 1 + (bikop r)       (2)
      | bakoitia x    = bikop r             (3)
  ```

  Emandako zerrenda hutsa bada (hau da, [] erakoa bada), 0 elementu bikoiti daude zerrenda horretan. Emandako zerrenda hutsa ez bada, hau da, adibidez x:r erakoa bada (zerrendako horretako lehenengo elementua x dela eta zerrendako beste elementu denez osatutako azpizerrenda r dela kontsideratuz), zerrendako lehenengo elementua bikoitia bada (hau da, x bikoitia bada) bikoiti-kopurua 1 gehi "r azpizerrendako bikoiti-kopurua" izango da. Baina zerrendako lehenengo elementua bakoitia bada (hau da, x bakoitia bada) bikoiti-kopurua "r azpizerrendako bikoiti-kopurua" izango da, x ez da zenbatu behar eta, ez baita bikoitia.

bikop funtzioaren definizioan lehendik definituta ditugun *bikoitia* eta *bakoitia* funtzioak erabili dira. Funtzio berriak definitzerakoan lehendik definituta dauden funtzioak erabil daitezke laguntzaile bezala.

- **tartekatu**: t motako bi zerrenda emanda, bi zerrenda horietako elementuak tartekatuz lortzen den zerrenda itzuliko du funtzio honek. Lehenengo zerrendako lehenengo elementua zerrenda berriko lehenengo elementua izango da, bigarren zerrendako lehenengo elementua zerrenda berriko bigarren elementua izango da eta abar. Hasierako zerrenda biak luzera berekoak ez badira, errorea gertatuko da (errore-mezua aurkeztuko da).

  Adibidea:
  tartekatu [**7, 5, 4**] [8, 2, 0] = [**7**, 8, **5**, 2, **4**, 0]

  Eragiketaren **mota**:
  tartekatu:: [t] -> [t] -> [t]
  Eragiketa definitzen duten **ekuazioak**:

```
tartekatu [] s
  | (luzera s) /= 0      = error "Luzera desberdineko zerrendak."      (1)
  | otherwise            = []                                          (2)

tartekatu (x:r) s
  | (luzera (x:r)) /= (luzera s) = error "Luzera desberdineko zerrendak."  (3)
  | otherwise                    = x:((leh s): (tartekatu r (hond s)))     (4)
```

  Kasu honetan 4 ekuazio eman dira *tartekatu* funtzioa definitzeko.

  Lehenengo bi ekuazioetan lehenengo zerrenda hutsa denean ([] erakoa denean) zer egin behar den zehazten da. Bigarren zerrenda (s zerrenda) hutsa ez bada, zerrenda bien luzera desberdina izango denez, errore-mezua aurkeztuko da eta bestela, hau da, bigarren zerrendaren luzera [] zerrendaren luzeraren berdina denean, zerrenda biak hutsak direnez emaitza bezala ere zerrenda hutsa itzuliko da.

  Beste bi ekuazioetan lehenengo zerrenda hutsa ez denean (adibidez x:r erakoa denean, zerrendako lehenengo elementua x dela eta zerrendako gainontzeko elementuez osatutako azpizerrenda r dela kontsideratuz), zer egin behar den zehazten da. Hor (3) ekuazioan x:r zerrendaren luzera eta s zerrendaren luzera desberdinak badira errore-mezua aurkeztuko dela adierazten da. Laugarren ekuazioan x:r eta s zerrendek luzera bera dutenean bi zerrenda horietako elementuak tartekatuz osatutako zerrenda nola eraikitzen den zehazten da. Zerrenda berriko lehenengo elementua x:r zerrendako lehenengo elementua izango da (x elementua). Zerrenda berriko bigarren elementua s zerrendako lehenengo elementua izango da, baina elementu horrek izen berezirik ez duenez, *leh(s)* bezala adierazi behar da. Zerrenda berriaren gainontzeko zatia kalkulatzeko, lehenengo zerrendako (x:r zerrendako) gainontzeko elementuak, hau da, r azpizerrendako elementuak eta bigarren zerrendako (s zerrendako) gainontzeko elementuak tartekatu beharko dira. Baina s zerrendako gainontzeko elementuez osatutako azpizerrendari izenik ez diogunez eman, *hond(s)* bezala adierazi beharko da azpizerrenda hori.

- **tartekatu2**: t motako bi zerrenda emanda, bi zerrenda horietako elementuak tartekatuz lortzen den zerrenda itzuliko du baina kasu honetan lehenengo zerrendako lehenengo elementua zerrenda berriko bigarren elementua izango da, bigarren zerrendako lehenengo elementua zerrenda berriko lehenengo elementua izango da eta abar. Hasierako zerrenda biak luzera berekoak ez badira, errorea gertatuko da (errore-mezua aurkeztuko da).

  Adibidea:

  tartekatu2 [**7, 5, 4**] [8, 2, 0] = [8, **7**, 2, **5**, 0, **4**]

  Eragiketaren **mota**:
  tartekatu2:: [t] -> [t] -> [t]

  Eragiketa definitzen duten **ekuazioak**:

```
tartekatu2 [] s
   | (luzera s) /= 0      = error "Luzera desberdineko zerrendak."    (1)
   | otherwise            = []                                        (2)

tartekatu2 (x:r) s
   | luzera (x:r) /= (luzera s)   = error "Luzera desberdineko zerrendak."   (3)
   | otherwise                    = (leh s):(x: (tartekatu2 r (hond s)))     (4)
```

  Ekuazio hauek ematerakoan aurreko adibideko, hau da, *tartekatu* izeneko funtzioaren kasuko ideia bera jarraitu da, baina (4) ekuazioan zerrenda berria eraikitzerakoan elementuak kontrako ordenean ipini dira lortu nahi den zerrenda lortu ahal izateko. Hori da hain zuzen ere *tartekatu* eta *tartekatu2* izeneko funtzioen arteko desberdintasuna.

- **pos_bik_kendu**: t motako zerrenda bat emanda, posizio bikoitietako elementuak kenduz lortzen den zerrenda itzuliko du funtzio honek. Datu bezala emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko du.

  pos_bik_kendu [7, **5**, 6, **8**, 2] = [7, 6, 2]

  Eragiketaren **mota**:
  pos_bik_kendu:: [t] -> [t]

  Eragiketa definitzen duten **ekuazioak**:

```
pos_bik_kendu [] = []
pos_bik_kendu (x:s)
      | hutsa_da s          = x:[]
      | otherwise           = x:(pos_bik_kendu (hond s))
```

- **pos_bak_kendu**: t motako zerrenda bat emanda, posizio bakoitietako elementuak kenduz lortzen den zerrenda itzuliko du funtzio honek. Datu bezala emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko du.

  pos_bik_kendu [**7**, 5, **6**, 8, **2**] = [5, 8]

Eragiketaren **mota**:
pos_bak_kendu:: [t] -> [t]

Eragiketa definitzen duten **ekuazioak**:

```
pos_bak_kendu [] = []
pos_bak_kendu (x:s)
        | hutsa_da s            = []
        | otherwise             = (leh s):( pos_bak_kendu (hond s))
```

7.4.4. Zerrendentzako eragile erlazionalak

Oinarrizko datu-moten atalean aipatutako eragile erlazionalak zerrendekin ere erabil daitezke.

Zerrenda bat beste bat baino txikiagoa edo handiagoa al den erabakitzeko hitzak alfabetikoki ordenatzeko erabiltzen den teknika bera jarraitzen da.

Esate baterako [3, 3, 4] zerrenda [5, 1] zerrenda baino txikiaga da 3 zenbakia 5 baino txikiagoa delako.

7.5. Errekurtsibitate gurutzatua

Bi funtziok elkar deitzen diotenean errekurtsibitate gurutzatua dutela esaten da:

- **bikoitiag eta bakoitiag**: Errekurtsibitate gurutzatuaren bidez, zero baino handiagoa edo berdina den zenbaki bat bikoitia ala bakoitia den erabakitzen duten funtzioak.

```
bikoitiag 7 = False
bikoitiag 12 = True
bakoitiag 7 = True
bakoitiag 12 = False
```

Jarraian funtzio bakoitzari dagozkion **mota** eta ekuazioak emango dira:

```
bikoitiag:: Int -> Bool

bikoitiag x
  | x < 0            = error "Balio negatiboa"
  | x == 0           = True
  | otherwise        = bakoitiag (x – 1)

bakoitiag:: Int -> Bool

bakoitiag x
  | x < 0            = error "Balio negatiboa"
  | x == 0           = False
  | otherwise        = bikoitiag (x - 1)
```

Funtzio hauek ez dira batere eraginkorrak zenbaki bat bikoitia ala bakoitia den erabakitzeko, baina errekurtsibitate gurutzatuaren adibide bezala balio dute.

- **pos_bik_kendug y pos_bak_kendug**: Errekurtsibitate gurutzatuaren bidez, datu bezala emandako zerrendako posizio bikoitietako eta posizio bakoitietako elementuak kentzen dituzten funtzioak.

```
pos_bik_kendug [7, 5, 6, 8, 2] = [7, 6, 2]
pos_bak_kendug [7, 5, 6, 8, 2] = [5, 8]
```

Jarraian funtzio bakoitzari dagozkion **mota** eta ekuazioak emango dira:

```
pos_bik_kendug:: [t] -> [t]

pos_bik_kendug [] = []
pos_bik_kendug (x:s) = x:(pos_bak_kendug s)

pos_bak_kendug:: [t] -> [t]

pos_bak_kendug [] = []
pos_bak_kendug (x:s) = pos_bik_kendug s
```

7.6. Modularitatea Haskell-en

7.6.1. Lau moduluren definizioa aurreko ataletan emandako eragiketak erabiliz

Haskell-en modulu desberdinak definitzea eta modulu batetik beste modulu batean definitutako funtzioak erabiltzea oso erraza da.

Esate baterako, gai honetan definitu ditugun funtzioak kontuan hartuz lau modulu defini ditzakegu:

a) *Ez_errek* izeneko modulua oinarrizko motentzat definitutako funtzio ez errekurtsiboekin (pi2, f, bikoitia, bakoitia, hand3).

b) *Errek_zenb* izeneko modulua Int motarentzat definitutako funtzio errekurtsiboekin (bider, bneg, errusiar, zatios, zatihond).

c) *Zerrendak* izeneko modulua zerrendentzat definitutako funtzio errekurtsibo eta ez errekurtsiboekin (leh, hond, hutsa_da, badago, luzera, bikop, tartekatu, tartekatu2, pos_bik_kendu, pos_bak_kendu).

d) *Errek_g* izeneko modulua errekurtsibitate gurutzatua duten funtzioekin (bikoitiag, bakoitiag, pos_bik_kendug, pos_bak_kendug).

Modulu bakoitza era independientean erabil daiteke baina beste modulu batetik inportatuak ere izan daitezke, hau da, modulu batean beste moduluren batean definitutako funtzioak erabil daitezke import aukerarekin. Modulu denetako funtzioak erabiltzeko erraztasuna eskainiko duen modulu nagusi bat ere defini daiteke.

Modulu bakoitza fitxategi desberdin batean joango da. Orain fitxategi bakoitza nola gelditzen den erakutsiko da.

a) "Ez_errek.hs" fitxategia

```
module Ez_errek where

-- Sarrerako daturik hartzen ez duen eta beti 3.1415 balioa itzultzen duen funtzioa
pi2:: Float
pi2 = 3.1415

-- Sarrerako datu bezala zenbaki oso bat hartu eta beti 100 balioa itzultzen duen funtzioa
f:: Int -> Int
f x = 100

-- Zenbaki oso bat emanda, zenbakia bikoitia al den ala ez erabakitzen duen funtzioa
bikoitia:: Int -> Bool
bikoitia x = (x `mod` 2) == 0

-- Zenbaki oso bat emanda, zenbakia bakoitia al den ala ez erabakitzen duen funtzioa
bakoitia:: Int -> Bool
bakoitia x = not (bikoitia(x))

-- Hiru zenbaki oso emanda, balio handiena itzultzen duen funtzioa
hand3:: Int -> Int -> Int -> Int
hand3 x y z
        | x >= y && x >= z          = x
        | y > x && y >= z           = y
        | otherwise                 = z
```

b) "Errek_zenb.hs" fitxategia

```
module Errek_zenb where

import Ez_errek

--Oinarrizko errekurtsibitatea zenbakiekin

--x balioa y aldiz batuz x * y kalkulatzen duen funtzioa.
--Kasu berezi bezala, y balioa negatiboa baldin bada, errore-mezua aurkeztuko du.

bider:: Int -> Int -> Int

bider x y
  | y < 0               = error "Bigarren balioa negatiboa da."
  | x == 0 || y == 0    = 0
  | x == 1              = y
  | otherwise           = x + bider x (y - 1)
```

```
--x * y balioa batuketaren bidez kalkulatzen duen funtzioa.
--y balioa negatiboa denean ere kalkulua ondo egiten da.
--Aurretik definitu den "bider" funtzioa erabiltzen da laguntzaile bezala.
--"bneg" funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen.
--Errekurtsibitatea "bider" funtzioan dago.

bneg:: Int -> Int -> Int

bneg x y
  | x == 0 || y == 0     = 0
  | (x < 0) && (y < 0)         = bider (-x) (-y)
  | (x > 0) && (y > 0)         = bider x y
  | (x < 0) && (y > 0)         = bider x y
  | (x > 0) && (y < 0)         = bider (-x) (-y)


--Errusiar biderkaketa bezala ezagutzen den metodoa jarraituz x * y
--kalkulatzen duen funtzioa.
--Kasu berezi bezala, y balioa negatiboa baldin bada, errore-mezua aurkeztuko du.

errusiar:: Int -> Int -> Int

errusiar x 0 = 0

errusiar x y
  | y < 0              = error "Bigarren balioa negatiboa da."
  | bikoitia y         = errusiar (x + x) (y `div` 2)
  | otherwise          = x + errusiar (x + x) (y `div` 2)


-- y balioa x balioari zenbat aldiz kendu dakiokeen zenbatuz, x eta y-ren arteko
-- zatidura osoa kalkulatzen duen funtzioa.
-- Kasu berezi bezala, zatitzailea zero denean, errore-mezua aurkeztuko da.
-- Gainera x edo y negatiboa baldin bada ere errore-mezua aurkeztuko da.

zatios:: Int -> Int -> Int

zatios x y
  | y == 0             = error "Zatitzailea 0"
  | (x < 0) || (y < 0)   = error "Gutxienez bietako bat negatiboa da."
  | x < y              = 0
  | otherwise          = 1 + zatios (x - y) y

-- y balioa baino txikiagoa den balio bat eduki arte y balioa x balioari eta horrela
--lortuz joango diren balioei kenduz, x eta y-ren arteko
-- zatidura osoaren hondarra kalkulatzen duen funtzioa.
-- Kasu berezi bezala, zatitzailea zero denean, errore-mezua aurkeztuko da.
-- Gainera x edo y negatiboa baldin bada ere errore-mezua aurkeztuko da.
```

```
zatihond:: Int -> Int -> Int

zatihond x y
   | y == 0              = error "Zatitzailea 0"
   | (x < 0) || (y < 0)  = error "Gutxienez bietako bat negatiboa da."
   | x < y               = x
   | otherwise           = zatihond (x - y) y
```

import Ez_errek ipiniz, modulu honetatik *Ez_errek* moduluko funtzioak erabilgarri daude.

c) "Zerrendak.hs" fitxategia

```
module Zerrendak where
import Ez_errek

-- Zerrenda bat emanda, zerrendako lehenengo elementua itzultzen duen funtzioa.
-- Zerrenda hutsa baldin bada, errore-mezua itzuliko du.

leh:: [t] -> t
leh [] = error "Zerrenda hutsa. Ez dago lehenengo elementurik."
leh (x:s) = x
-----------------------------------------------------
-- Zerrenda bat emanda, zerrendako lehenengo elementua kenduz lortzen den zerrenda
-- itzultzen duen funtzioa.
-- Zerrenda hutsa baldin bada, errore-mezua itzuliko du.

hond:: [t] -> [t]
hond [] = error "Zerrenda hutsa. Ez dago hondarrik."
hond (x:s) = s
-----------------------------------------------------
-- Zerrenda bat emanda, zerrenda hutsa al den ala ez erabakitzen duen funtzioa.

hutsa_da:: [t] -> Bool
hutsa_da [] = True
hutsa_da (x:s) = False
-----------------------------------------------------
-- Elementu bat eta zerrenda bat emanda, elementua zerrendan agertzen al den ala ez
-- erabakitzen duen funtzioa.
-- t motako elementuek konparagarriak izan behar dute (Eq t =>)

badago:: Eq t => t -> [t] -> Bool
badago x [] = False
badago x (y:s)
        | x == y        = True
        | x /= y = badago x s
-----------------------------------------------------
-- Zerrenda bat emanda, zerrendako elementu-kopurua kalkulatzen duen funtzioa.
```

```
luzera:: [t] -> Int
luzera [] = 0
luzera (x:r) = 1 + luzera r
-------------------------------------------------
-- Zerrenda bat emanda, zerrendako elementu bikoitien kopurua kalkulatzen duen
funtzioa.

bikop:: [Int] -> Int
bikop [] = 0
bikop (x:r)
        | bikoitia(x)    = 1 + bikop r
        | bakoitia(x)    = bikop r
-------------------------------------------------
-- Bi zerrenda emanda, zerrenda bietako elementuak tartekatuz lortzen den zerrenda
-- kalkulatzen duen funtzioa.
-- Hasteko lehenengo zerrendatik hartu behar da.
-- Zerrenda biek luzera bera ez badute, errore-mezua aurkeztuko da.

tartekatu:: [t] -> [t] -> [t]
tartekatu [] s
        | luzera s /= 0  = error "Luzera desberdineko zerrendak."
        | otherwise              = []
tartekatu (x:r) s
        | luzera (x:r) /= luzera s       = error "Luzera desberdineko zerrendak."
        | otherwise                      = x:((leh s): (tartekatu r (hond s)))
-------------------------------------------------
-- Bi zerrenda emanda, zerrenda bietako elementuak tartekatuz lortzen den zerrenda
-- kalkulatzen duen funtzioa.
-- Hasteko bigarrengo zerrendatik hartu behar da.
-- Zerrenda biek luzera bera ez badute, errore-mezua aurkeztuko da.

tartekatu2:: [t] -> [t] -> [t]
tartekatu2 [] s
        | luzera s /= 0  = error "Luzera desberdineko zerrendak."
        | otherwise              = []
tartekatu2 (x:r) s
        | luzera (x:r) /= luzera s       = error "Luzera desberdineko zerrendak."
        | otherwise                      = (leh s):(x:(tartekatu2 r (hond s)))
-------------------------------------------------
--Zerrenda bat emanda, posizio bikoitietako elementuak kentzen dituen funtzioa

pos_bik_kendu:: [t] -> [t]

pos_bik_kendu [] = []
pos_bik_kendu (x:s)
        | hutsa_da s             = x:[]
        | otherwise              = x:(pos_bik_kendu (hond s))
-------------------------------------------------
```

```
--Zerrenda bat emanda, posizio bakoitietako elementuak kentzen dituen funtzioa

pos_bak_kendu:: [t] -> [t]

pos_bak_kendu [] = []
pos_bak_kendu (x:s)
        | hutsa_da s            = []
        | otherwise             = (leh s):(pos_bak_kendu (hond s))
```

[] eta : eragile eraikitzaileak eta ++ eragilea aurredefinituta daude Haskell-en.

import Ez_errek ipiniz, modulu honetatik *Ez_errek* moduluko funtzioak erabilgarri daude.

d) "Errek_g.hs" fitxategia

```
module Errek_g where

--Errekurtsibitate gurutzatua

--Errekurtsibitate gurutzatuaren bidez, 0 baino handiagoa edo berdina
--den zenbaki oso bat bikoitia ala bakoitia den erabakitzen duten funtzioak
bikoitiag:: Int -> Bool

bikoitiag x
  | x < 0               = error "Balio negatiboa"
  | x == 0              = True
  | otherwise           = bakoitiag (x - 1)
---
---
bakoitiag:: Int -> Bool

bakoitiag x
  | x < 0               = error "Balio negatiboa"
  | x == 0              = False
  | otherwise           = bikoitiag (x - 1)

-------------------------------------------------

--Errekurtsibitate gurutzatuaren bidez, posizio bikoitietako eta
--posizio bakoitietako elementuak kentzen dituzten funtzioak
pos_bik_kendug:: [t] -> [t]

pos_bik_kendug [] = []
pos_bik_kendug (x:s) = x:(pos_bak_kendug s)
```

```
pos_bak_kendug:: [t] -> [t]

pos_bak_kendug [] = []
pos_bak_kendug (x:s) = pos_bik_kendug s
```

7.6.2. Modulu nagusiaren definizioa

Modulu bakoitza era independientean erabil daiteke baina modulu denetan definitutako funtzio denak erabiltzeko aukera eduki nahi denean modulu nagusi bat definitzea izaten da onena. Kasu honetan "Nagusia.hs" modulua definitu da helburu horrekin eta bertan beste modulu denak inportatzen dira.

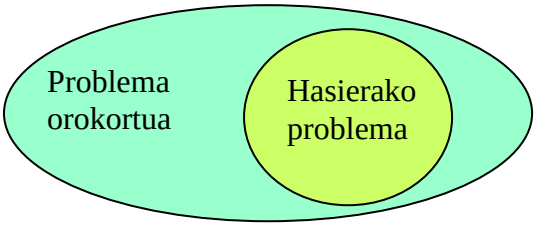
```
module Nagusia where
import Ez_errek
import Zerrendak
import Errek_zenb
import Errek_g
```

7.7. Murgilketa

Atal honetan *murgilketa* bezala ezagutzen den teknika azalduko da. Ebatzi beharreko problema zuzenean ebatzi beharrean, problema hori orokortu eta orokortze horren bidez lortutako problema berria ebaztean datza teknika hau. Orokortuz lortutako problema berria ebatzitakoan, hasierako problema ere ebatzita geratuko da, hau da, problema orokortua ebatziz hasierako problema ere ebatzita geratuko da, hasierako problema problema orokortuaren kasu partikular bat izango baita. Horregatik, hasierako problema problema orokortuan **murgilduta** geratu dela esan daiteke.

Problema bat orokortzeko, hasierako problemari parametro berriak gehitzen zaizkio murgilketaren teknikan. Parametro berri horiek helburu desberdinak izan ditzakete:

- Tarte bat zeharkatzeko indize bezala erabiliko diren parametroak izan daitezke.
- Bukaerako emaitza kalkulatzeko erabiliko diren behin-behineko emaitzak edo emaitza partzialak gordetzeko erabiliko diren parametro berriak ere izan daitezke (zenbatzaile bezala edo baturak gordetzeko edo zerrendak gordetzeko, eta abar).



Problema batzuetan, soluzio errekurtsiboa lortzea nahi baldin bada, murgilketaren teknika erabiltzea derrigorrezkoa da. Beste problema batzuetan murgilketa ez da beharrezkoa baina hala ere erabil daiteke, izan ere gehienetan eraginkortasun-maila hobetzen da murgilketa erabiliz.

7.7.1. Murgilketarik gabeko soluzio errekurtsiboak

7.3.8, 7.4.3 eta 7.5 ataletan aurkeztu diren funtzio errekurtsiboetan murgilketa ez da erabili. Funtzio horiek 7.6 atalean definitutako *Errek_zenb*, *Zerrendak* eta *Errek_g* moduluetan jaso dira. Beraz funtzio horiek murgilketarik gabeko soluzio errekurtsiboen adibide bezala har ditzakegu. Kasu horietan hasierako problema zuzenean ebazten da, orokortu gabe.

7.7.2. Tarteak zeharkatzeko indize bezala erabiliko diren parametroak gehitzean oinarritzen den murgilketa

Era honetako murgilketetan, tarte bat zeharkatzeko indize bezala erabiliko den parametro berri bat edukiko dugu.

a) Osoa den zenbaki positibo baten zatitzaileen zerrenda: *zatizer*

Osoa den zenbaki positibo baten zatitzaileen zerrenda kalkulatzeko, x zatitzen duten [1..x] tarteko zenbakiak aurkitu behar dira.

zatizer izeneko funtzioaren definizio errekurtsiboak honako egitura hau izan beharko luke:

```
-- Funtzio honek, x zenbaki oso bat emanda, [1..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer:: Int -> [Int]
zatizer x
        | x < 0 || x == 0       = error "Zenbakia ez da positiboa."
        | otherwise             = ???
```

Baina ezin dugu zuzenean x balioan oinarritutako soluzio errekurtsiborik planteatu. Faktorialaren definizioa hartuz, x zenbaki baten faktoriala x – 1 zenbakiaren faktoriala erabiliz kalkula daiteke. Zerrenden gainean definitutako funtzio errekurtsiboak aztertuz, adibidez zerrenda baten luzera, elementu bat gutxiago duen azpirrendaren luzera erabiliz kalkula daiteke. Baina zenbaki baten zatitzaileen zerrendaren kasuan, x zenbaki baten zatitzaileen zerrendak ez du zerikusirik x – 1 zenbakiaren zatitzaileen zerrendarekin. Adibide bezala, 8 eta 7 zenbakien zatitzaileen zerrendak har ditzakegu. Alde batetik, 8 zenbakiaren zatitzaileen zerrenda [1, 2, 4, 8] da. Beste aldetik, 7 zenbakiaren zatitzaileen zerrenda [1, 7] da. Beraz, ezinezkoa da 8ren zatitzaileen zerrenda 7ren zatitzaileen zerrendatik kalkulatzea.

Zatitzaileen zerrenda kalkulatzerakoan, errekurtsibitatea ez dago x zenbakian, errekurtsibitatea [1..x] tartean dago. Horrela, x zenbakiaren zatitzaileen zerrenda kalkulatzeko, tarte horretako lehenengo elementuak x zatitzen al duen ala ez aztertu behar da (beraz hasieran 1 zenbakiak x zenbakia zatitzen al duen aztertuko da). Jarraian, x zenbakiaren beste zatitzaileak aurkitzeko, [2..x] tarteko zenbakiak aztertu beharko dira. [2..x] tartea finkatu ondoren, [2..x] tartekoak diren x zenbakiaren zatitzaileen zerrenda, 2 zenbakiak x zenbakia zatitzen al duen aztertuz eta [3..x] tartean x zenbakiaren zatitzaile gehiago bilatuz lortuko da. Garapen honetan nabaria den bezala, errekurtsibitatea tarte horretan dago eta tarte hori gero eta txikiagoa da. Orokorrean, tarteko beheko muga bm baldin bada, [bm..x] tartekoak diren x zenbakiaren zatitzaileak, bm zenbakiak x zatitzen al duen aztertuz eta gero [bm + 1..x] tartean bilatzen jarraituz lortuko dira. Eraginkortasuna dela eta, x div 2 zenbakitik aurrera x zenbakiaren zatitzaile bakarra x bera izango denez, bilaketa [1..x div 2] tartera muga daiteke.

Beraz, murgilketa x zenbakiaren zatitzaileak [1..x] tartean bilatu beharrean [bm..x] tarte orokor batean bilatzean oinarrituko da. Definituko dugun funtzio berrian bi parametro edukiko ditugu, x eta bm. Hor bm balioak x zenbakiaren zatitzaileak bilatzerakoan kontsideratu beharreko tartearen beheko muga zein den adierazteko balioko digu.

Funtzio berriari zatizer_lag deituko diogu, zatizer funtzioa definitzeko laguntzaile bezala erabiliko baitugu.

```
-- Funtzio honek, x eta bm bi zenbaki oso emanda, [bm..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- bm zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer_lag:: Int -> Int -> [Int]
zatizer_lag x bm
        | x < 0 || x == 0       = error "Zenbakia ez da positiboa."
        | bm <= 0               = error "Beheko muga ez da positiboa."
        | bm > x                = []
        | bm > (x `div` 2)      = [x]
        | x `mod` bm == 0       = bm : (zatizer_lag x (bm + 1))
        | otherwise             = zatizer_lag x (bm + 1)
```

zatizer_lag funtzioa definitu ondoren, zatizer funtzioaren definizioa eman dezakegu, izan ere zatizer funtzioa zatizer_lag funtzioaren kasu partikular bat da: bm parametroaren balioa 1 denekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda, [1..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu
-- murgilketa erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer:: Int -> [Int]
zatizer x = zatizer_lag x 1
```

Definizioan ikus daitekeen bezala, zatizer funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina zatizer_lag funtzioa errekurtsiboa da.

zatizer_lag funtzioak ebazten duen problema zatizer funtzioak ebazten duena baino orokorragoa da, izan ere zatizer_lag funtzioaren bidez esate baterako, 5 baino handiagoak edo berdinak diren 28 zenbakiaren zatitzaileak kalkula baititzakegu:

```
zatizer_lag 28 5
```

erantzuna [7, 14, 28] izango litzateke.

zatizer funtzioarekin aldiz, zatitzaile denak kalkulatzen dira beti, ez dago beheko muga aukeratzerik:

```
zatizer 28
```

erantzuna [1, 2, 4, 7, 14, 28] izango litzateke.

zatizer_lag erabiliz 28ren zatitzaile denen zerrenda kalkulatzea nahi izanez gero, honako hau ipini beharko litzateke:

```
zatizer_lag 28 1
```

b) 1 edo handiagoa den zenbaki bat lehena al den erabakitzen duen funtzioa:

1 edo handiagoa den zenbaki oso bat lehena izango da justu bi zatitzaile baldin baditu. Adibidez 2, 3, 5, 7, 11, 13 eta 17 zenbakiak lehenak dira justu bi zatitzaile dituztelako (1 eta zenbakia bera). Bestalde, 1 zenbakia ez da lehena zatitzaile bakarra duelako eta, esate baterako, 4, 6, 8, 9, 10, 12, 14 eta 16 ez dira lehenak bi zatitzaile baino gehiago dituztelako.

Zenbaki bat lehena al den erabakitzeko aukera bat bere zatitzaileen zerrenda kalkulatu eta zerrenda horretan justu bi zenbaki al dauden ala ez aztertzea izango litzateke:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- lehena al den erabakiko du murgilketa erabili gabe.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

lehena_gb:: Int -> Bool
lehena_gb x
        | x <= 0                    = error "Zenbakia ez da positiboa."
        | luzera (zatizer x) == 2   = True
        | luzera (zatizer x) /= 2   = False
```

Zenbaki bat lehena al den erabakitzen duen funtzio hori ez da errekurtsiboa, ez baitio bere buruari deitzen eta gainera murgilketa erabili gabe definitu da. Hala ere gogoratu luzera eta zatizer funtzioak errekurtsibitatean oinarrituta daudela.

lehena_gb funtzioaren arazoa eraginkortasun eza da. Izan ere, 28 lehena al den erabakitzeko, hasteko bere zatitzaileen zerrenda kalkulatzen du: [1, 2, 4, 7, 14, 28]. Zatitzaileen zerrenda kalkulatu ondoren, zatitzaileak zenbatzen ditu eta justu bi ez badira False itzuliko du eta justu bi badira True itzuliko du. Zenbaki bat lehena al den erabakitzeko era hau ez da eraginkorra, 2 zenbakiak 28 zenbakia zatitzen duela ikusi ondoren bai baitakigu 28 ez dela lehena, 1 eta 28 zenbakiez gain gutxienez hirugarren zatitzaile bat ere baduelako. Beraz, zatitzaile gehiago bilatzen jarduteak ez du zentzurik. Adibide bezala 1001 zenbakia hartzen badugu, gauza bera gertatzen da. Izan ere, lehena_gb funtzioa jarraituz, hasteko 1001en zatitzaileen zerrenda kalkulatuko litzateke: [1, 7, 11, 13, 77, 91, 143, 1001]. Gero, zerrenda horretako elementuak zenbatuko lirateke eta, justu bi ez daudenez, False itzuliko litzateke. Adibide honetan ere, 1001en zatitzaileak bilatzen hasitakoan 7 zenbakiak 1001 zatitzen duela ikustean badakigu 1001 ez dela lehena eta 1001en zatitzaile gehiago bilatzen jarraitzeak ez du zentzurik.

Orain zatizer funtzioa erabili gabe zuzenean x zenbakia lehena al den erabakitzen duen funtzio errekurtsibo bat definitzea da gure helburua. Plantamendu berri honetan, lehena ez den 1 zenbakia kasu berezi bezala hartuko da eta beste edozein x zenbakirentzat, x lehena izango da [2..x – 1] tartean x zenbakiaren zatitzailerik ez badago. Adibidez 7 zenbakiarentzat [2..6] tartean ez dago zatitzailerik eta ondorioz 7 lehena da. Baina 8 zenbakiaren kasuan, [2..7] tartean gutxienez zatitzaile bat badago (2 eta 4 zenbakiak 8ren zatitzaileak dira) eta ondorioz 8 ez da lehena. Era berean 28 zenbakiarentzat [2..27] tartean gutxienez zatitzaile bat badago (2, 4, 7 eta 14 zenbakiak 28ren zatitzaileak dira)

eta ondorioz 28 ez da lehena. Bestalde, 2 zenbakiaren kasuan [2..2 – 1] tartea, hau da, [2..1] tartea hutsa da (goiko muga beheko muga baino txikiagoa delako) eta tarte hutsean 2ren zatitzailerik ezin denez egon, 2 zenbakia lehena da.

Orain badakigu zenbaki bat lehena ez dela erabakitzeko ez dagoela bere zatitzaile denak aurkitu beharrik, [2..x – 1] tartekoa den bere zatitzaile bat aurkitzearekin nahikoa da. Beraz, eraginkorragoa den funtzio bat lortzeko ideia hori hartuko dugu kontuan:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- lehena al den erabakiko du [2..x – 1] tartean zatitzailerik ba al
-- duen aztertuz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

lehena:: Int -> Bool
lehena x
        | x <= 0                    = error "Zenbakia ez da positiboa."
        | otherwise                 = ???
```

Baina x zenbakian oinarritutako soluzio errekurtsiborik ezin da aurkitu. Hemen ere aurretik definitu den zatizer funtzioarekin genuen arazo bera daukagu. Izan ere, x lehena izatea edo ez izatea ez dago x – 1 lehena izatearekin edo ez izatearekin erlazionatuta. Adibide bezala, alde batetik 3 lehena da eta 2 ere lehena da. Beste aldetik, 16 ez da lehena eta 15 ere ez da lehena. Baina 18 ez da lehena eta 17 bai.

Zenbaki baten zatitzaileak kalkulatzerakoan bezala, kasu honetan ere errekurtsibitatea ez dago x zenbakian, errekurtsibitatea [2..x – 1] tartean dago. Beraz, x zenbakiak [2..x – 1] tartean zatitzailerik ba al duen erabakitzeko, tarte horretako lehenengo zenbakiak x zatitzen al duen aztertuko da (hasieran 2 zenbakiak x zatitzen al duen erabakiko da). Zatitzen badu, badakigu x ez dela lehena eta prozesua hor bukatuko da, ez dago zatitzaile gehiago bilatu beharrik. Ez badu zatitzen, [3..x – 1] tartean x zenbakiaren zatitzailerik ba al dagoen aztertzen jarraitu beharko da. [3..x – 1] tartea finkatu ondoren, tarte horretako lehenengo zenbakiak, hau da, 3 zenbakiak x zatitzen al duen begiratu beharko da. Zatitzen badu, badakigu x ez dela lehena eta prozesua hor bukatuko da, ez dago zatitzaile gehiago bilatu beharrik. Ez badu zatitzen, [4..x – 1] tartean x zenbakiaren zatitzailerik ba al dagoen aztertzen jarraitu beharko da. [4..x – 1] tartea finkatu ondoren, tarte horretako lehenengo zenbakiak, hau da, 4 zenbakiak x zatitzen al duen begiratu beharko da. Prozesuak x zenbakiaren zatitzaile bat aurkitu arte edo x zenbakiaren zatitzailerik ezingo dela aurkitu ziur egon arte jarraitu beharko du. Eraginkortasuna dela eta, x div 2 zenbakitik aurrera x zenbakiaren zatitzaile bakarra x bera izango denez, bilaketa [2..x div 2] tartera muga daiteke.

Garapen honetan nabaria den bezala, errekurtsibitatea tarte horretan dago eta tarte hori gero eta txikiagoa da. Orokorrean, tarteko beheko muga bm baldin bada, [bm..x – 1] tartekoa den x zenbakiaren zatitzailerik ba al dagoen jakiteko, bm zenbakiak x zatitzen al duen aztertu beharko da eta, bm ez bada x zenbakiaren zatitzailea, [bm + 1..x – 1] tartean x zenbakiaren zatitzailerik ba al dagoen begiratzen jarraitu beharko da.

Beraz, murgilketa x zenbakiak [2..x – 1] tartean zatitzailerik ba al duen aztertzean oinarritu beharrean [bm..x – 1] tarte orokor batean zatitzailerik ba al duen aztertzean oinarrituko da. Definituko dugun funtzio berrian bi parametro edukiko ditugu, x eta bm. Hor bm balioak x zenbakiak zatitzailerik ba al duen erabakitzerakoan kontsideratu beharreko tartearen beheko muga zein den adierazteko balioko digu.

Funtzio berriari lehena_lag deituko diogu, lehena funtzioa definitzeko laguntzaile bezala erabiliko baitugu.

```
-- Funtzio honek, x eta bm bi zenbaki oso emanda, True itzuliko du
-- x zenbakiak [bm..x – 1] tartean zatitzailerik ez badu eta bestela
-- False itzuliko du.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- bm zenbakia 1 edo txikiagoa baldin bada, errorea.
-- Kasu berezi bezala, x zenbakiaren balioa 1 denean False itzuliko du.

lehena_lag:: Int -> Int -> Bool
lehena_lag x bm
        | x < 0 || x == 0       = error "Zenbakia ez da positiboa."
        | bm <= 1               = error "Beheko muga 2 baino txikiagoa."
        | x == 1                = False
        | bm > (x – 1)          = True
        | x `mod` bm == 0       = False
        | otherwise             = lehena_lag x (bm + 1)
```

Hor bm > (x – 1) ipini beharrean bm > (x `div` 2) ipiniz laugarren baldintza hobetu daiteke. Orain, lehena_lag funtzioa definitu ondoren, lehena funtzioaren definizioa eman dezakegu, izan ere lehena funtzioa lehena_lag funtzioaren kasu partikular bat da: bm parametroaren balioa 2 denekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- lehena al den erabakiko du [2..x - 1] tartean zatitzailerik ba al
-- duen aztertuz eta murgilketa erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

lehena:: Int -> Bool
lehena x = lehena_lag x 2
```

Definizioan ikus daitekeen bezala, lehena funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina lehena_lag funtzioa errekurtsiboa da.

lehena_lag funtzioak ebazten duen problema lehena funtzioak ebazten duena baino orokorragoa da, izan ere lehena_lag funtzioaren bidez esate baterako, 27 zenbakiak 10 edo handiagoak diren zatitzailerik ba al duen jakin baitezakegu. Baldintza hori betetzen duen zatitzailerik ez badago, True itzuliko du eta baldintza hori betetzen duen zatitzaileren bat baldin badago, orduan False itzuliko du:

```
lehena_lag 27 10
```

[10..26] tartean 27ren zatitzailerik ez dagoenez, erantzuna True izango litzateke.

lehena izeneko funtzioarekin aldiz 2 zenbakitik abiatuta aztertzen da zatitzailerik ba al dagoen ala ez:

lehena 27

[10..26] tartean 27ren zatitzaileak badaudenez, erantzuna False izango litzateke eta horrek 27 ez dela lehena esan nahi du.

lehena_lag funtzioa erabiliz 27 zenbakia [2..26] tartean zatitzailerik ez duen zenbaki bat al den erabaki nahi badugu, honako hau ipini beharko da:

lehena_lag 27 2

27 zenbakia [2..26] tartean zatitzailerik ez duen zenbaki bat ez denez, kasu honetan False balioa itzuliko luke funtzioak.

Bigarren parametroaren balioaren arabera, 27 zenbakiarentzat lehena_lag funtzioak True edo False itzuli dezakeela ikus dezakegu:

lehena_lag 27 10 → True
lehena_lag 27 2 → False
lehena_lag 27 5 → False

7.7.3. Behin-behineko emaitzak gordetzeko balio duten parametroak gehitzean oinarritutako murgilketa

Era honetako murgilketan, geroago edo bukaeran behin-betiko emaitza kalkulatzeko erabiliko den emaitza partziala (edo behin-behineko emaitza) gordetzeko balio duen parametro berri bat izango du problema orokortuak.

a) Luzeren zerrenda: *azpiluz*

Zenbaki osozko zerrenda bat emanda, azpiluz izeneko funtzioak jarraian dauden balio berdinez osatutako azpizerrenden luzerez osatutako zerrenda kalkulatu behar du:

azpiluz [5, 5, 9, 8, 8, 8, 8, 7, 7, 8, 0, 0, 7] = [2, 1, 4, 2, 1, 2, 1]

azpiluz [3, 3, 3, 3, 3, 3, 3, 3, 3] = [9]

Sarrera bezala emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz izeneko funtzioaren definizio errekurtsiboak honako egitura hau izan beharko luke:

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Sarrerako datua hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz:: [Int] -> [Int]

azpiluz [] = []
azpiluz (x:s)
        | hutsa_da s            = 1:[]
        | x == (leh s)          = ???
        | x /= (leh s)          = ???
```

Baina kontua da nola gorde azpizerrenda baten luzera. Luzera hori ez da behin-betikoa azpizerrenda bukatu arte. Gainera azpizerrenda bakoitzaren behin-betiko luzera bakarrik gorde nahi da.

Azpizerrenda bakoitzaren kasuan, azpizerrendaren behin-behineko luzera gordez joan beharko dugu behin-betiko luzera ezagutu arte. Azpizerrendaren behin-betiko luzera kalkulatutakoan, eraikitzen ari garen luzeren zerrendan gorde dezakegu luzera hori.

Murgilketarik gabe ere egin daiteke hau, baina ez zuzenean, beste funtzio batzuk laguntzaile bezala definituz:

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrendaren luzera itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_zenbatu:: [Int] -> Int

berdinak_zenbatu [] = error "Zerrenda hutsa."

berdinak_zenbatu (x:s)
        | hutsa_da s            = 1
        | x == (leh s)          = 1 + (berdinak_zenbatu s)
        | x /= (leh s)          = 1
```

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrenda kenduz geratzen den zerrenda itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_kendu:: [Int] -> [Int]

berdinak_kendu [] = error "Zerrenda hutsa."

berdinak_kendu (x:s)
        | hutsa_da s            = []
        | x == (leh s)          = berdinak_kendu s
        | x /= (leh s)          = s
```

```
-- Funtzio honek, zenbaki osozko zerrenda
-- bat emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Kalkulua murgilketarik gabe egiten du.
-- Sarrerako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz_gb:: [Int] -> [Int]

azpiluz_gb [] = []

azpiluz_gb (x:s) = (berdinak_zenbatu (x:s)):(azpiluz_gb (berdinak_kendu (x:s)))
```

azpiluz_gb funtzioa erabiliz, azpizerrenda bakoitza bi aldiz zeharkatu beharko da: lehenengo aldian elementuak zenbatuko dira eta bigarren aldian kendu egingo dira. Zerrenda luzeen kasuan azpizerrenda bakoitza bi aldiz zeharkatu beharrak eraginkortasun-maila txarragoa izatea ekarriko du berarekin. Jarraian murgilketa erabiliz, azpizerrenda bakoitza behin zeharkatuko duen soluzio bat emango da.

Murgilketan oinarritutako soluzioan, azpizerrenda bakoitzaren behin-behineko luzera gordez joateko zenbatzaile bezala balioko duen parametro berri bat behar da. Beraz kasu honetan murgilketa burutzeko, parametro bezala zenbaki osozko zerrenda bat edukitzeaz gain beste zenbaki bat ere izango duen funtzio laguntzailea kontsideratuko da. Parametro berri hori azpizerrenda bakoitzaren luzera kalkulatzeko zenbatzaile bezala erabiliko da. Azpizerrenda bakoitzarekin hasterakoan zenbatzaile hori 0 balioarekin hasieratu beharko da eta azpizerrenda bakoitza bukatzean, zenbatzaileak une horretan duen balioa eraikitzen ari garen luzeren zerrendan gorde beharko da.

Funtzio laguntzailea definitzerakoan, zenbatzaile bezala erabiliko den parametroak ezin du 0 konstantea izan, parametro orokor bat erabili behar baita definizioa ematerakoan. Hori dela eta, funtzio laguntzailearen esanahia honako hau izango da:

> Zenbaki osozko zerrenda bat eta zenbaki oso bat emanda, jarraian dauden balio berdinez osatutako azpizerrenden luzerez osatutako zerrenda kalkulatuko da. Baina lehenengo azpizerrendaren kasuan azpizerrenda horren luzera gehi bigarren parametroaren balioa gordeko da.

Funtzio berriari azpiluz_lag deituko diogu, azpiluz funtzioa definitzeko laguntzaile bezala erabiliko baitugu.

```
-- Funtzio honek, zenbaki osozko zerrenda bat eta zenbaki oso
-- bat emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Lehenengo azpizerrendaren kasuan, bere luzera gehi luz itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz_lag:: [Int] -> Int -> [Int]

azpiluz_lag [] luz = []

azpiluz_lag (x:s) luz
        | hutsa_da s            = (1 + luz):[]
        | x == (leh s)          = azpiluz_lag s (1 + luz)
        | x /= (leh s)          = (1 + luz): (azpiluz_lag s 0)
```

Zenbatzaileak 0tik hastea da ohikoena, baina kasu honetan luz izeneko zenbatzailea parametro orokor bat izango denez, funtzioari deitzean zenbatzaile hori geuk nahi dugun balioarekin hasiera dezakegu.

azpiluz_lag funtzioa definitu ondoren, azpiluz funtzioaren definizioa eman dezakegu, izan ere azpiluz funtzioa azpiluz_lag funtzioaren kasu partikular bat da: luz parametroaren balioa 0 denekoa hain zuzen ere.

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Sarrerako datua hutsa baldin bada, zerrenda hutsa itzuliko da.
-- Definizio hau murgilketaren teknikan oinarrituta dago.

azpiluz:: [Int] -> [Int]
azpiluz r = azpiluz_lag r 0
```

Definizioan ikus daitekeen bezala, azpiluz funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina azpiluz_lag funtzioa errekurtsiboa da.

azpiluz_lag funtzioak ebazten duen problema azpiluz funtzioak ebazten duena baino orokorragoa da, izan ere azpiluz_lag funtzioaren bidez lehenengo azpizerrendari dagokion zenbatzailea geuk nahi dugun balioarekin hasieratu baitezakegu:

azpiluz_lag [7, 7, 7, 20, 20, 9, 8] 2 = [5, 2, 1, 1]

azpiluz_lag [7, 7, 7, 20, 20, 9, 8] 9 = [12, 2, 1, 1]

azpiluz funtzioa erabiliz aldiz, azpizerrenda denen luzera zehatzak kalkulatzen dira, baita lehenengo azpizerrendaren kasuan ere:

azpiluz [7, 7, 7, 20, 20, 9, 8] = [3, 2, 1, 1]

Zuzenean azpiluz_lag funtzioa erabiliz azpizerrenda denen luzera zehatzak kalkulatzea nahi badugu (lehenengo azpizerrendarena ere barne), zenbatzailea 0 balioarekin hasieratu beharko dugu:

azpiluz_lag [7, 7, 7, 20, 20, 9, 8] 0

b) Azpizerrenden zerrenda: azpizer

Zenbaki osozko zerrenda bat emanda, azpizer izeneko funtzioak jarraian dauden balio berdinez osatutako azpizerrendez eratutako zerrenda kalkulatu behar du

azpizer [5, 5, 9, 8, 8, 8, 8, 7, 7, 8, 0, 0, 7] =

= [[5, 5], [9], [8, 8, 8, 8], [7, 7], [8], [0, 0], [7]]

azpizer [3, 3, 3, 3, 3, 3, 3, 3, 3] = [[3, 3, 3, 3, 3, 3, 3, 3, 3]]

Sarrera bezala emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpizer izeneko funtzioaren definizio errekurtsiboak honako egitura hau izan beharko luke:

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda, zerrendan
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Sarrerako lehenengo zerrenda hutsa baldin bada, zerrenda hutsa
-- itzuliko da.

azpizer:: [Int] -> [[Int]]
azpizer [] = []
azpizer (x:s)
        | hutsa_da s            = [[x]]
        | x == (leh s)    = ???
        | x /= (leh s)    = ???
```

Baina kontua da nola gorde azpizerrenda bakoitza. Azpizerrenda hori ez da behin-betikoa azpizerrenda bukatu arte. Gainera azpizerrenda bakoitza bere osotasunean gorde nahi da.

Azpizerrenda bakoitzaren kasuan, azpizerrendaren osagaiak gordez joan beharko dugu azpizerrenda osoa lortu arte. Azpizerrenda osoa eskuratutakoan, eraikitzen ari garen azpizerrenden zerrendan gorde dezakegu azpizerrenda hori.

Murgilketarik gabe ere egin daiteke hau, baina ez zuzenean, beste funtzio batzuk laguntzaile bezala definituz:

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrenda itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_hartu:: [Int] -> [Int]

berdinak_hartu [] = error "Zerrenda hutsa."

berdinak_hartu (x:s)
        | hutsa_da s            = [x]
        | x == (leh s)          = x:(berdinak_hartu s)
        | x /= (leh s)          = [x]
```

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrenda kenduz geratzen den zerrenda itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_kendu:: [Int] -> [Int]

berdinak_kendu [] = error "Zerrenda hutsa."

berdinak_kendu (x:s)
        | hutsa_da s            = []
        | x == (leh s)          = berdinak_kendu s
        | x /= (leh s)          = s
```

```
-- Funtzio honek, zenbaki osozko zerrenda
-- bat emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Kalkulua murgilketarik gabe egiten du.
-- Sarrerako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpizer_gb:: [Int] -> [[Int]]

azpizer_gb [] = []

azpizer_gb (x:s) = (berdinak_hartu (x:s)):(azpizer_gb (berdinak_kendu (x:s)))
```

azpizer_gb funtzioa erabiliz, azpizerrenda bakoitza bi aldiz zeharkatu beharko da: lehenengo aldian elementuak hartu egingo dira eta bigarren aldian kendu egingo dira. Zerrenda luzeen kasuan azpizerrenda bakoitza bi aldiz zeharkatu beharrak eraginkortasun-maila txarragoa izatea ekarriko du berarekin. Jarraian murgilketa erabiliz, azpizerrenda bakoitza behin zeharkatuko duen soluzio bat emango da.

Murgilketan oinarritutako soluzioan, azpizerrenda bakoitzaren osagaiak gordez joateko azpizerrenda bat gordetzeko balioko duen parametro berri bat behar da.

Beraz kasu honetan murgilketa burutzeko, parametro bezala zenbaki osozko zerrenda bat edukitzeaz gain zenbaki osozko beste zerrenda bat ere izango duen funtzio laguntzailea kontsideratuko da. Bigarren parametro hori azpizerrenda

bakoitza gordez joateko erabiliko da. Azpizerrenda bakoitzarekin hasterakoan parametro hori [] balioarekin hasieratu beharko da eta azpizerrenda bakoitza bukatzean, bigarren parametro horrek une horretan duen balioa eraikitzen ari garen azpizerrenden zerrendan gorde beharko da.

Funtzio laguntzailea definitzerakoan, azpizerrenda bat gordetzeko erabiliko den parametroak ezin du [] konstantea izan, parametro orokor bat erabili behar baita definizioa ematerakoan. Hori dela eta, funtzio laguntzailearen esanahia honako hau izango da:

> Zenbaki osozko bi zerrenda emanda, jarraian dauden balio berdinez osatutako azpizerrendez osatutako zerrenda kalkulatuko da. Baina lehenengo azpizerrendaren kasuan azpizerrenda horri bigarren parametroaren balioa elkartuko dio eskuinetik.

Funtzio berriari azpizer_lag deituko diogu, azpizer funtzioa definitzeko laguntzaile bezala erabiliko baitugu.

```
-- Funtzio honek, zenbaki osozko bi zerrenda emanda, zerrendan
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Lehenengo azpizerrendaren kasuan, azpizerrenda hori bigarren
-- parametroarekin elkartuta itzuliko du.
-- Sarrerako lehenengo zerrenda hutsa baldin bada, zerrenda hutsa
-- itzuliko da.

azpizer_lag:: [Int] -> [Int] -> [[Int]]

azpizer_lag [] azpi = []

azpizer_lag (x:s) azpi
        | hutsa_da s            = (x:azpi):[]
        | x == (leh s)          = azpizer_lag s (x:azpi)
        | x /= (leh s)          = (x:azpi):(azpizer_lag s [])
```

azpizer_lag funtzioa definitu ondoren, azpizer funtzioaren definizioa eman dezakegu, izan ere azpizer funtzioa azpizer_lag funtzioaren kasu partikular bat da: azpi parametroaren balioa [] denekoa hain zuzen ere.

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda, zerrendan
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Sarrerako lehenengo zerrenda hutsa baldin bada, zerrenda hutsa
-- itzuliko da.
-- Definizio hau murgilketaren teknikan oinarrituta dago.

azpizer:: [Int] -> [[Int]]
azpizer r = azpizer_lag r []
```

Definizioan ikus daitekeen bezala, azpizer funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina azpizer_lag funtzioa errekurtsiboa da.

azpizer_lag funtzioak ebazten duen problema azpizer funtzioak ebazten duena baino orokorragoa da, izan ere azpizer_lag funtzioaren bidez lehenengo azpizerrendari geuk nahi dugun zerrenda erantsi baitiezaiokegu:

azpizer_lag [7, 7, 7, 20, 20, 9, 8] [5, 5] = [[7, 7, 7, 5, 5], [20, 20], [9], [8]]

azpizer_lag [7, 7, 7, 20, 20, 9, 8] [3, 7, 7, 3] =

= [[7, 7, 7, 3, 7, 7, 3], [20, 20], [9], [8]]

azpizer funtzioa erabiliz aldiz, datu bezala emandako lehenengo zerrendaren azpizerrendez osatutako zerrenda itzuliko da, eta lehenengo azpizerrendaren kasuan ere ez zaio ezer erantsiko:

azpizer [7, 7, 7, 20, 20, 9, 8] = [[7, 7, 7], [20, 20], [9], [8]]

Zuzenean azpizer_lag funtzioa erabiliz datu bezala emandako lehenengo zerrendaren azpizerrendez osatutako zerrenda kalkulatzea nahi badugu (eta lehenengo azpizerrendari ezer erantsi gabe) bigarren parametroa [] balioarekin hasieratu beharko dugu:

azpizer_lag [7, 7, 7, 20, 20, 9, 8] [] = [[7, 7, 7], [20, 20], [9], [8]]

7.7.4. Tarteak zeharkatzeko indize bezala erabiliko diren parametroak eta behin-behineko emaitzak gordetzeko erabiliko diren parametroak gehitzean oinarritzen den murgilketa

Atal honetan aurkeztuko den murgilketa-adibidean bi parametro berri ipiniko dira: bata tarte bat zeharkatzeko indize bezala erabiliko da eta bestea bukaerako emaitza kalkulatzeko beharrezkoa den behin-behineko emaitza edo emaitza partziala gordetzeko erabiliko da.

a) Zenbaki beteak:

1en berdina edo handiagoa den x zenbaki bat betea dela esaten da bere zatitzaileen batura (x bera kontuan hartu gabe) x baldin bada.

Adibidez 6 betea da 1 + 2 + 3 = 6 delako eta 28 ere betea da 1 + 2 + 4 + 7 + 14 = 28 delako.

Aukera bat dagoeneko ezagutzen ditugun *zatizer* eta *batu* izeneko funtzioak erabiltzea izango litzateke. Kasu honetan soluzioa murgilketa erabili gabe lortuko genuke:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- betea al den erabakiko du zerrenda bateko elementuak batzen dituen
-- "batu" eta zenbaki oso baten zatitzaileen zerrenda kalkulatzen duen
-- "zatizer" funtzioak erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- Funtzio hau murgilketa erabili gabe definitu da.

betea_gb:: Int -> Bool
betea_gb x
        | x <= 0                          error "Zenbakia ez da positiboa."
        | ((batu (zatizer x)) - x) == x   = True
        | otherwise                       = False
```

Baina betea_gb funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen.

Zenbaki bat betea al den erabakitzen duen funtzio errekurtsibo bat definitzen saia gaitezke. Funtzio horrek honako egitura hau izango luke:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- betea al den erabakiko du errekurtsibitatea erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

betea:: Int -> Bool
betea x
        | x <= 0          = error " Zenbakia ez da positiboa."
        | otherwise       = ???
```

Baina x parametroan oinarritutako soluzio errekurtsiborik ezin da eman kasu honetan ere. Alde batetik x zenbakiaren zatitzaileak kalkulatu behar dira. Horretarako, zatizer eta lehena izeneko funtzioetan egin den bezala, [1..x – 1] tartea zeharkatzeko indize bezala erabiliko den parametro berri bat ipini beharko da. Kontuan hartu x zenbakia betea al den erabakitzeko [1..x – 1] tarteko zatitzaileak aurkitu behar direla. Baina, gainera, x zenbakiaren tarte horretako zatitzaileen batura kalkulatu behar da, nahiz eta bukaeran itzuli beharreko emaitza batura hori ez izan. Bukaeran itzuli beharreko emaitza batura hori erabiliz lortuko da. Batura hori x zenbakiaren berdina baldin bada, orduan True itzuliko da eta bestela False itzuliko da. Beraz emaitza partzial bat kalkulatu behar da (batura) gero emaitza partzial hori erabiliz bukaerako emaitza lortu ahal izateko (azpiluz eta azpizer funtzioetan egin denaren antzera).

Guztira bi parametro berri behar dira, bata [1..x – 1] tartea zeharkatuz joateko eta bestea zatitzaileen batura gordez joateko.

Beraz, murgilketa erabiliz, [1..x – 1] tarteko zatitzaileak kalkulatu beharrean, beheko muga bezala bm edozein zenbaki izan dezakeen [bm..x – 1] tarteko zatitzaileak kalkulatuko dira. Gainera, une bakoitzean ordura arte aurkitu diren zatitzaileen batura eramango duen zbat izeneko parametro bat ere edukiko da. Guztira, definituko dugun funtzio berriak hiru parametro izango ditu: x, bm eta zbat. Eraginkortasuna dela eta, x div 2 zenbakitik aurrera x zenbakiaren zatitzaile bakarra x bera izango denez, bilaketa [2..x div 2] tartera muga daiteke.

Funtzio berriari betea_lag deituko diogu, betea izeneko funtzioa definitzeko laguntzaile bezala erabiliko baitugu.

```
-- Funtzio honek, x, bm eta zatibatu hiru zenbaki oso emanda, True itzuliko du
-- x zenbakiak [bm..x - 1] tartean dituen zatitzaileen batura gehi zatibatu
-- balioa x balioaren berdina al den erabakiko du. ez badu eta bestela
-- False itzuliko du.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- bm zenbakia 0 edo negatiboa baldin bada, errorea.

betea_lag:: Int -> Int -> Int -> Bool
betea_lag x bm zatibatu
        | x <= 0                                = error "Zenbakia ez da positiboa."
        | bm <= 0                               = error "Beheko muga ez da
positiboa."
        | (bm > (x `div` 2)) && (zatibatu == x) = True
        | (bm > (x `div` 2)) && (zatibatu /= x) = False
        | (x `mod` bm) == 0                     = betea_lag x (bm + 1) (bm +
zatibatu)
        | (x `mod` bm) /= 0                     = betea_lag x (bm + 1) zatibatu
```

betea_lag funtzioa definitu ondoren, betea izeneko funtzioaren definizioa eman dezakegu, izan ere betea funtzioa betea_lag funtzioaren kasu partikular bat da: bm parametroaren balioa 1 eta zbat parametroaren balioa 0 denekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- betea al den erabakiko du murgilketa erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

betea:: Int -> Bool
betea x = betea_lag x 1 0
```

Definizioan ikus daitekeen bezala, betea izeneko funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina betea_lag funtzioa errekurtsiboa da.

betea_lag funtzioak ebazten duen problema betea funtzioak ebazten duena baino orokorragoa da, izan ere betea_lag funtzioaren bidez 25 zenbakiaren [3..24] tarteko zatitzaileen batura gehi 20 balioa 25 al den jakin baitezakegu:

```
betea_lag 25 3 20
```

Erantzuna True izango litzateke, izan ere [3..24] tartean 25 zenbakiaren zatitzaile bakarra daukagu (5) eta zatitzaile horri 20 batuz 25 balioa lortzen da.

Esate baterako, betea_lag 25 3 15 deia eginez gero, erantzun bezala False jasoko genuke.

Era berean, betea_lag 77 5 30 deia eginez ere erantzun bezala False jaso genuke, [5..76] tartekoak diren 77ren zatitzaileen batura (7 + 11) gehi 30 ez baita 77.

Baina betea_lag 77 5 59 deiak True balioa itzuliko luke 7 + 11 + 59 baturaren balioa 77 baita.

Bestalde betea_lag 28 4 3 deiak ere True itzuliko luke 4 + 7 + 14 + 3 = 28 baita, baina betea_lag 28 4 0 deiaren erantzuna False izango litzateke 4 + 7 + 14 + 0 baturaren emaitza ez baita 28. Beste kasu bat aztertuz, lehena_lag 28 6 3 deiaren erantzuna ere False izango litzateke 7 + 14 + 3 baturaren balioa ez baita 28.

Gogoratu betea izeneko funtzioarekin zatitzaileak beti 1etik hasita bilatuko direla eta baturaren hasierako balioa 0 izango dela:

betea 77 = False
betea 28 = True
betea 25 = False

Zuzenean betea_lag erabiliz 28 betea al den erabakitzea nahi badugu, honako hau idatzi beharko dugu:

betea_lag 28 1 0

7.7.5. Murgilketaren beste aplikazio batzuk: bukaerako errekurtsibitatea eta eraginkortasuna

Aurreko ataleko adibide batzuetan ikusi den bezala, batzutan ez da beharrezkoa murgilketa erabiltzea baina hala ere erabil daiteke, murgilketarik gabe definitutako funtzioak baino eraginkorragoak diren funtzioak lortuz. Atal honetan oraindik eraginkortasun handiagoa lortzeko beste urrats bat emango dugu eta *bukaerako errekurtsibitatea* duten soluzioak murgilketa erabiliz nola lortu ditzakegun azalduko da. Bukaerako errekurtsibitatea edukitzeak eraginkortasun-maila hobetzen du orokorrean.

a) Zerrenda bateko elementuen batura:

Zenbaki osozko zerrenda bateko elementuen batura era errekurtsiboan kalkulatzen duen funtzioa honela defini daiteke:

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- zerrendako elementuen batura kalkulatuko du
-- murgilketaren teknika erabili gabe.

batu_gb :: [Int] -> Int

batu_gb [] = 0

batu_gb (x:s) = x +  (batu_gb s)
```

Definizio horretan ez da murgilketa erabili. Zerrenda bateko elementuen batura kalkulatzen duen funtzio horrek ez du *bukaerako errekurtsibitaterik*, dei errekurtsiboei dagokien sekuentzia edo zuhaitza garatu ondoren, garapen

horretan sortuz joan diren elementuak batzeko zuhaitza berriro zeharkatu behar baita kontrako eran:

	Bukaerako emaitza kalkulatzeko behar diren balioak sortuz doan deien sekuentzia edo zuhaitza	Balio denak sortu ondoren, planteatu diren baturak kalkulatuz joan beharko da eta horretarako zuhaitza berriro zeharkatu beharko da	
Hasiera (erroa)	batu_gb [3, 5, 10, 6]	24	Bukaerako emaitza
	↓	↑	
	3 + batu_gb [5, 10, 6]	3 + 21 = 24	
	↓	↑	
	5 + batu_gb [10, 6]	5 + 16 = 21	
	↓	↑	
	10 + batu_gb [6]	10 + 6 = 16	
	↓	↑	
	6 + batu_gb []	0 + 6 = 6	
	↓	↑	
hostoa	0 →	0	

batu_gb [3, 5, 10, 6] deiak adar bakarreko zuhaitz bat sortzen du eta adabegi bakoitzean emaitza partzial bat lagatzen da (zenbaki bat). Azkeneko adabegira, hau da, 0 balioa duen adabegira iritsitakoan (adabegi hori hostoa da), zuhaitza berriro hostotik erroraino zeharkatu beharko da, adabegi bakoitzean lagatako zenbakia jasoz eta batura kalkulatuz. Errora iritsitakoan behin-betiko emaitza edukiko da.

Beraz, zuhaitza bi aldiz zeharkatu behar da. Zuhaitza oso sakona baldin bada, bi aldiz zeharkatu beharrak eraginkortasun eza sortzen du.

Hostora iritsitakoan bukaerako emaitza edukitzea izango litzateke onena. Horrela zuhaitza bigarren aldiz (oraingoan hostotik errora) zeharkatu beharrik ez genuke izango eta funtzioa eraginkorragoa izango litzateke.

Hori murgilketa erabiliz lor daiteke. Errotik hostora joaterakoan adabegietan balioak lagaz joan eta gero bukaerako emaitza eraikitzeko zuhaitza berriro (oraingoan hostotik errora) zeharkatu beharrean, zuhaitza eraikitzerakoan, hau da, errotik hostora joaterakoan bukaerako emaitza eraikiz joan gaitezke. Eraikiz ari garen emaitza hori beheruntz pasatuz joango ginateke batura gordez doan parametro berri baten bidez.

Parametro bezala batu beharreko elementuez osatutako zerrenda edukitzeaz gain, batura kalkulatuz joateko erabiliko den beste parametro bat ere baduen batu_lag izeneko funtzioa definitiko dugu orain:

```
-- Funtzio honek, zenbaki osozko zerrenda bat eta b zenbaki oso bat emanda,
-- zerrendako elementuen batura gehi b kalkulatuko du.

batu_lag :: [Int] -> Int -> Int

batu_lag [] b = b

batu_lag (x:s) b = batu_lag s (x + b)
```

batu_lag funtzioa definitu ondoren, batu_mr izeneko funtzioaren definizioa eman dezakegu, izan ere batu_mr funtzioa batu_lag funtzioaren kasu partikular bat da: b parametroaren balioa 0 denekoa hain zuzen ere.

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- zerrendako elementuen batura kalkulatuko du
-- murgilketaren teknika erabiliz.

batu_mr :: [Int] -> Int

batu_mr r = batu_lag r 0
```

Definizioan ikus daitekeen bezala, batu_mr izeneko funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina batu_lag funtzioa errekurtsiboa da.

batu_lag funtzioak ebazten duen problema batu_mr funtzioak ebazten duena baino orokorragoa da, izan ere batu_lag funtzioaren bidez zerrenda bateko elementuen batura kalkulatzeaz gain batura horri beste balio bat ere batu diezaiokegu:

```
batu_lag [10, 4, 5] 6
```

deiaren kasuan erantzuna 25 izango litzateke (19 + 6).

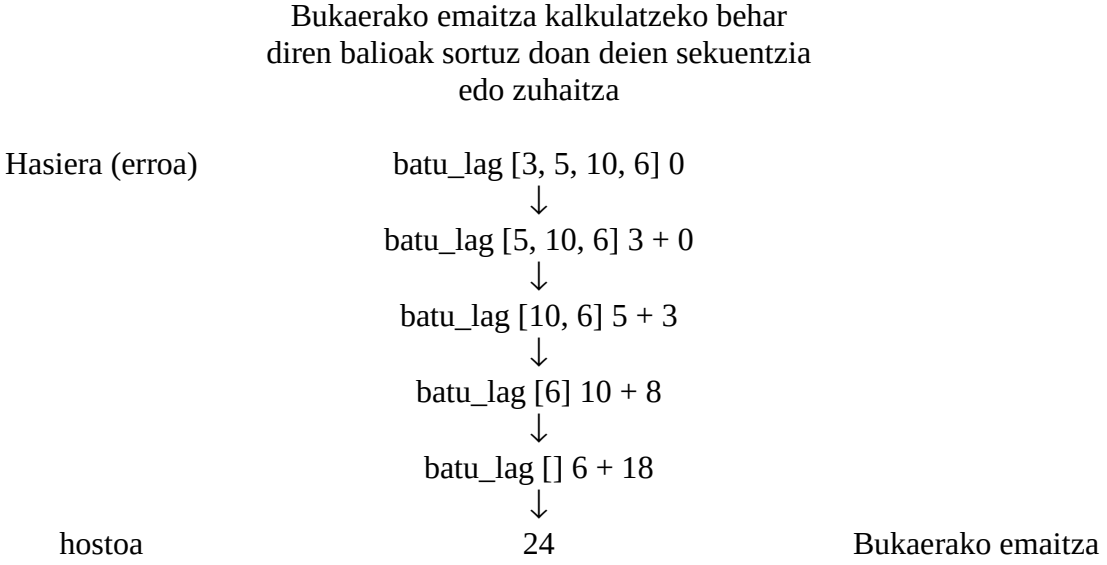
Bestalde, batu_mr funtzioak zerrendako elementuen batura kalkulatzen du, beste ezer gehitu gabe:

```
batu_mr [10, 4, 5] = 19
```

Zuzenean batu_lag funtzioa erabiliz [10, 4, 5] zerrendako elementuen batura kalkulatu nahi badugu, honako hau ipini beharko dugu:

```
batu_lag [10, 4, 5] 0
```

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratzen denez (zuhaitza berriro zeharkatu beharrik gabe), batu_lag funtzioak bukaerako errekurtsibitatea du:



batu_lag [3, 5, 10, 6] 0 deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzerakoan ordura arte tratatu diren elementuen batura edukiko da. Azkeneko adabegira iritsitakoan (27 balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro goruntz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu.

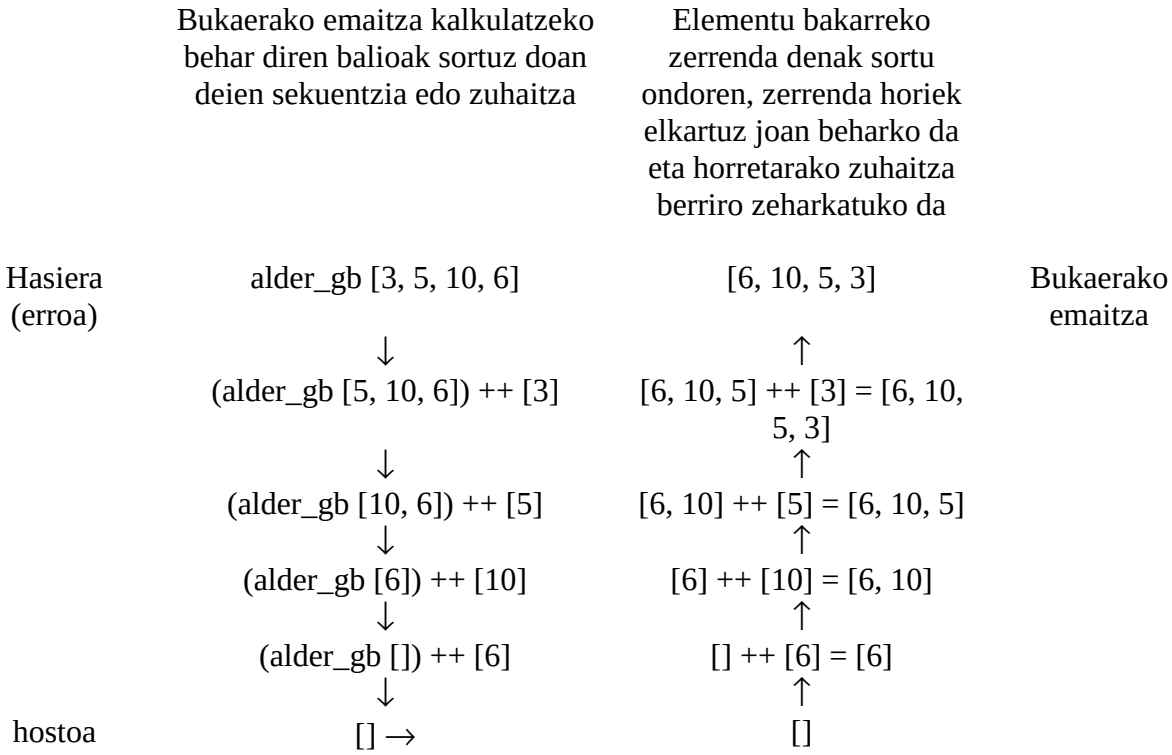
Beraz batu_lag funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den batu_gb funtzioa baino eraginkorragoa da.

b) Zerrenda baten alderantzizkoa.

Zerrenda baten alderantzizkoa era errekurtsiboan honela kalkula daiteke:

```
-- Funtzio honek, zerrenda bat emanda, bere alderantzizkoa itzuliko
-- du ++ erabiliz eta murgilketa erabili gabe.

alder_gb :: Show t => [t] -> [t]

alder_gb [] = []

alder_gb (x:s) = (alder_gb s) ++ [x]
```

Definizio horretan ez da murgilketa erabili. Zerrenda baten alderantzizkoa kalkulatzen duen funtzio horrek ez du bukaerako errekurtsibitatea, beharrezkoak diren dei errekurtsiboz osatutako sekuentzia (edo zuhaitza) eraiki ondoren, behin-betiko emaitza kalkulatzeko adabegi bakoitzean sortu diren elementu bakarreko zerrendak elkartu behar baitira eta horretarako zuhaitza berriro zeharkatu behar da behetik gora:



alder_gb [3, 5, 10, 6] deia adar bakarreko zuhaitz bat eraikiz joango da eta adabegi bakoitzean emaitza partzial bat lagako da. Azkeneko adabegira iritsitakoan, hau da, [] balioa duen eta hostoa den adabegira iritsitakoan, adabegi bakoitzean lagatako elementu bakarreko zerrenda jasotzeko eta behin-betiko zerrenda eraikiz joateko zuhaitz dena berriro zeharkatu beharko da. Errora iritsitakoan behin-betiko zerrenda edukiko da.

Beraz zuhaitza bi aldiz zeharkatu beharko da. Zuhaitza oso sakona baldin bada, bi aldiz zeharkatu beharrak eraginkortasun-maila txarragoa izatea ekarriko du berarekin.

Hostora iritsitakoan bukaerako emaitza edukitzea izango litzateke onena. Horrela zuhaitza bigarren aldiz (oraingoan hostotik errora) zeharkatu beharrik ez genuke izango eta funtzioa eraginkorragoa izango litzateke.

Hori murgilketa erabiliz lor daiteke. Errotik hostora joaterakoan adabegietan balioak lagaz joan eta gero bukaerako emaitza eraikitzeko zuhaitza berriro (oraingoan hostotik errora) zeharkatu beharrean, zuhaitza eraikitzerakoan, hau da, errotik hostora joaterakoan bukaerako emaitza eraikiz joan gaitezke. Eraikiz ari garen emaitza hori beheruntz pasatuz joango ginateke alderantzizko zerrenda gordez doan parametro berri baten bidez.

Parametro bezala alderantziz ipini beharreko zerrenda edukitzeaz gain alderantzizko zerrenda kalkulatuz joateko erabiliko den beste parametro bat ere baduen alder_lag izeneko funtzioa definitiko dugu orain:

```
-- Funtzio honek, bi zerrenda emanda, lehenengo zerrendaren
-- alderantzizkoa kalkulatu eta
-- bigarren zerrendari elkartuta itzuliko  du.

alder_lag :: Show t => [t] -> [t] -> [t]

alder_lag [] q = q

alder_lag (x:s) q = alder_lag s (x:q)
```

alder_lag funtzioa definitu ondoren, alder_mr izeneko funtzioaren definizioa eman dezakegu, izan ere alder_mr funtzioa alder_lag funtzioaren kasu partikular bat da: q parametroaren balioa [] denekoa hain zuzen ere.

```
-- Funtzio honek, zerrenda bat emanda, bere alderantzizkoa itzuliko
-- du murgilketa erabiliz.

alder_mr :: Show t => [t] -> [t]

alder_mr r = alder_lag r []
```

Definizioan ikus daitekeen bezala, alder_mr izeneko funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina alder_lag funtzioa errekurtsiboa da.

alder_lag funtzioak ebazten duen problema alder_mr funtzioak ebazten duena baino orokorragoa da, izan ere alder_lag funtzioaren bidez lehenengo zerrendaren alderantzizkoari beste zerrenda bat erantsi diezaiokegu:

alder_lag [2, 8, 9] [4, 5]

Erantzuna [9, 8, 2, 4, 5] izango litzateke.

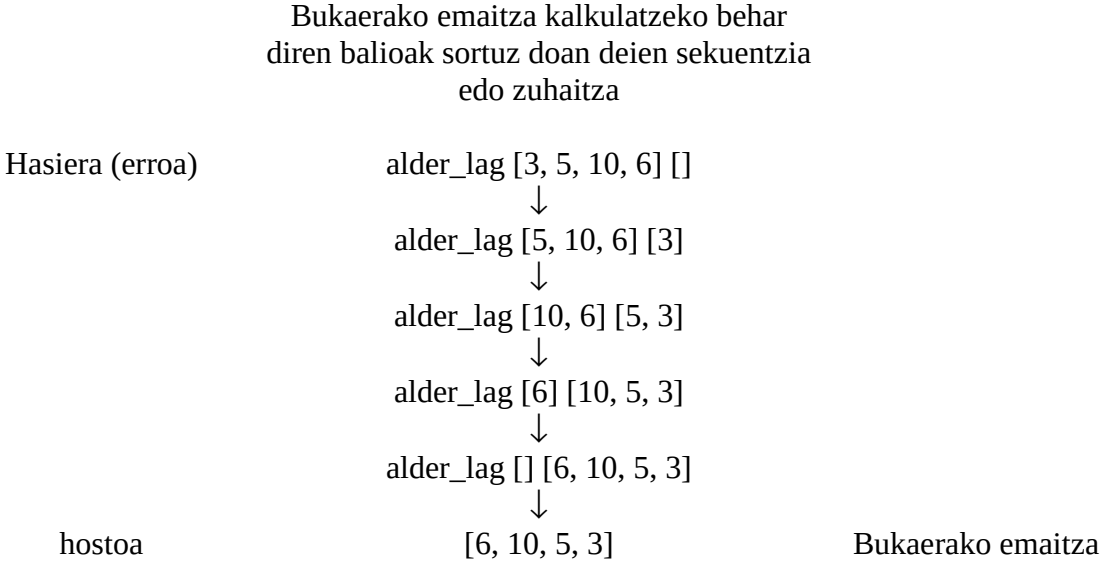
Baina alder_mr funtzioarekin zerrendaren alderantzizkoa kalkulatzen da, beste ezer erantsi gabe:

alder_mr [2, 8, 9] = [9, 8, 2]

alder_lag erabiliz [2, 8, 9] zerrendaren alderantzizkoa kalkulatzea nahi izanez gero, honako hau ipini beharko genuke:

alder_lag [2, 8, 9] []

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratzen denez (zuhaitza berriro zeharkatu beharrik gabe), alder_lag funtzioak bukaerako errekurtsibitatea du:



alder_lag [3, 5, 10, 6] [] deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzerakoan ordura arte tratatu den zerrenda zatiaren alderantzizkoa edukiko da. Azkeneko adabegira iritsitakoan ([6, 10, 5, 3] balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro goruntz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu.

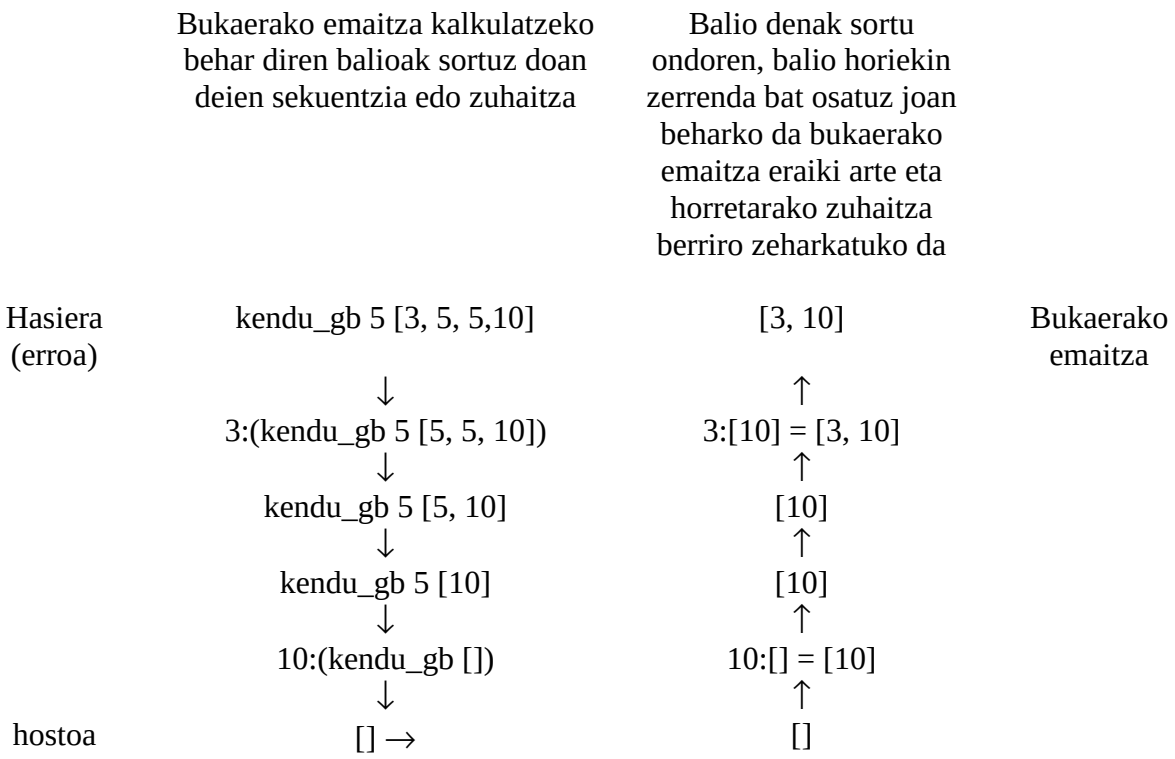
Beraz alder_lag funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den alder_gb funtzioa baino eraginkorragoa da.

c) Elementu baten agerpen denak zerrendatik kentzen dituen funtzioa.

Elementu bat eta zerrenda bat emanda, zerrendatik elementu horren agerpen denak ezabatzen dituen funtzioa honela defini daiteke errekurtsiboki:

```
-- Funtzio honek, elementu bat eta zerrenda bat emanda, elementu horren
-- agerpen denak kenduz gelditzen den zerrenda itzuliko du
-- murgilketa erabili gabe.

kendu_gb :: (Show t, Eq t) => t -> [t] -> [t]

kendu_gb x [] = []

kendu_gb x (y:s)
        | x == y        = kendu_gb x s
        | x /= y        = y:(kendu_gb x s)
```

Definizio horretan ez da murgilketa erabili. Zerrendatik elementu baten agerpen denak ezabatzen dituen funtzio horrek ez du *bukaerako errekurtsibitaterik*, dei errekurtsiboei dagokien sekuentzia edo zuhaitza garatu ondoren, garapen horretan geratuz joan diren elementuekin emaitza osatzeko zuhaitza berriro zeharkatu behar baita kontrako eran:



kendu_gb 5 [3, 5, 5, 10] deia adar bakarreko zuhaitz bat eraikiz joango da eta adabegi bakoitzean emaitza partzial bat lagako da. Azkeneko adabegira iritsitakoan, hau da, [] balioa duen eta hostoa den adabegira iritsitakoan, adabegi bakoitzean lagatako elementua jasotzeko eta behin-betiko zerrenda eraikiz joateko zuhaitz dena berriro zeharkatu beharko da. Errora iritsitakoan behin-betiko zerrenda edukiko da.

Beraz zuhaitza bi aldiz zeharkatu beharko da. Zuhaitza oso sakona baldin bada, bi aldiz zeharkatu beharrak eraginkortasun-maila txarragoa izatea ekarriko du berarekin.

Hostora iritsitakoan bukaerako emaitza edukitzea izango litzateke onena. Horrela zuhaitza bigarren aldiz (oraingoan hostotik errora) zeharkatu beharrik ez genuke izango eta funtzioa eraginkorragoa izango litzateke.

Hori murgilketa erabiliz lor daiteke. Errotik hostora joaterakoan adabegietan balioak lagaz joan eta gero bukaerako emaitza eraikitzeko zuhaitza berriro (oraingoan hostotik errora) zeharkatu beharrean, zuhaitza eraikitzerakoan, hau da, errotik hostora joaterakoan bukaerako emaitza eraikiz joan gaitezke. Eraikiz ari garen emaitza hori beheruntz pasatuz joango ginateke zerrenda gordez doan parametro berri baten bidez.

Parametro bezala kendu beharreko elementua eta zerrenda bat edukitzeaz gain elementu horren agerpenak zerrendatik kenduz lortuko den zerrenda kalkulatuz joateko erabiliko den beste parametro bat ere baduen kendu_lag_ald izeneko funtzioa definitiko dugu orain:

```
-- Funtzio honek, elementu bat eta bi zerrenda emanda, elementu horren
-- agerpen denak lehenengo zerrendatik kenduz gelditzen den zerrenda
-- eta aurretik (ezkerretik) bigarren zerrendaren alderantzizkoa
-- elkartuz lortzen den zerrenda itzuliko du.
-- Aurretik definitutako alder_mr funtzioa erabiltzen da.

kendu_lag_ald :: (Show t, Eq t) => t -> [t] -> [t] -> [t]

kendu_lag_ald x [] q = alder_mr q

kendu_lag_ald x (y:s) q
        | x == y        = kendu_lag_ald x s q
        | x /= y        = kendu_lag_ald x s (y:q)
```

kendu_lag_ald funtzioa definitu ondoren, kendu_mr_ald izeneko funtzioaren definizioa eman dezakegu, izan ere kendu_mr_ald funtzioa kendu_lag_ald funtzioaren kasu partikular bat da: q parametroaren balioa [] denekoa hain zuzen ere

```
-- Funtzio honek, elementu bat eta zerrenda bat emanda, elementu horren
-- agerpen denak kenduz gelditzen den zerrenda itzuliko du
-- zerrenda baten alderantzizko kalkulatzen duen funtzioan oinarritutako
-- murgilketa erabiliz.

kendu_mr_ald :: (Show t, Eq t) => t -> [t] -> [t]

kendu_mr_ald x r = kendu_lag_ald x r []
```

Definizioan ikus daitekeen bezala, kendu_mr_ald izeneko funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina kendu_lag_ald funtzioa errekurtsiboa da.

kendu_lag_ald funtzioak ebazten duen problema kendu_mr_ald funtzioak ebazten duena baino orokorragoa da, izan ere kendu_lag_ald funtzioaren bidez lehenengo zerrendatik elementuaren agerpen denak kentzeaz gain, beste zerrenda baten alderantzizkoa erantsi diezaiokegu ezkerretik:

kendu_lag_ald 5 [3, 5, 5, 10] [0, 5, 2]

erantzuna [2, 5, 0, 3, 10] izango litzateke.

Baina kendu_mr_ald funtzioarekin, emandako elementuaren agerpen denak zerrendatik kenduz geratzen den zerrenda kalkulatzen da, beste ezer erantsi gabe:

kendu_mr_ald 5 [3, 5, 5, 10] = [3, 10]

Zuzenean kendu_lag_ald erabiliz [3, 5, 5, 10] zerrendatik 5 zenbakiaren agerpen denak kenduz geratzen den zerrenda lortu nahi bada, honako hau ipini beharko da:

kendu_lag_ald 5 [3, 5, 5, 10] []

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratzen denez (zuhaitza berriro zeharkatu beharrik gabe), kendu_lag_ald funtzioak bukaerako errekurtsibitatea du:

	Bukaerako emaitza kalkulatzeko behar diren balioak sortuz doan deien sekuentzia edo zuhaitza	
Hasiera (erroa)	kendu_lag_ald 5 [3, 5, 5, 10] []	
	↓	
	kendu_lag_ald 5 [5, 5, 10] (3:[])	
	↓	
	kendu_lag_ald 5 [5, 10] (3:[])	
	↓	
	kendu_lag_ald 5 [10] (3:[])	
	↓	
	kendu_lag_ald 5 [] (10:3:[])	
	↓	
hostoa	alder_mr [10, 3] = [3, 10]	Bukaerako emaitza

kendu_lag_ald 5 [3, 5, 5, 10] [] deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzerakoan ordura arte tratatu den zerrenda zatitik hartu beharreko elementuz osatutako zerrendaren alderantzizkoa edukiko da. Azkeneko adabegira iritsitakoan ([3, 10] balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro goruntz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu. Adibidean erakusten den bezala, prozesuan eraikiz joan garen zerrendaren alderantzizkoa itzuli behar da bukaerako emaitza bezala.

Beraz kendu_lag_ald funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den kendu_gb funtzioa baino eraginkorragoa da.

BESTE AUKERA BAT:

Murgilketa aplikatuz baina alderantzizkoa kalkulatzen duen funtzioa erabili gabe ere definitu daiteke elementu baten agerpen denak zerrendatik kentzen dituen funtzioa. Kasu honetan bi zerrenda elkartzeko balio duen (++) funtzioa erabiliko da.

Parametro bezala kendu beharreko elementua eta zerrenda bat edukitzeaz gain elementu horren agerpenak zerrendatik kenduz lortuko den zerrenda kalkulatuz joateko erabiliko den beste parametro bat ere baduen kendu_lag_elk izeneko funtzioa definitiko dugu orain:

```
-- Funtzio honek, elementu bat eta bi zerrenda emanda, elementu horren
-- agerpen denak lehenengo zerrendatik kenduz gelditzen den zerrenda
-- eta aurretik (ezkerretik) bigarren zerrenda
-- elkartuz lortzen den zerrenda itzuliko du.
-- ++ funtzioa erabiliko da.

kendu_lag_elk :: (Show t, Eq t) => t -> [t] -> [t] -> [t]

kendu_lag_elk x [] q = q

kendu_lag_elk x (y:s) q
        | x == y        = kendu_lag_elk x s q
        | x /= y        = kendu_lag_elk x s (q ++ [y])
```

kendu_lag_elk funtzioa definitu ondoren, kendu_mr_elk izeneko funtzioaren definizioa eman dezakegu, izan ere kendu_mr_elk funtzioa kendu_lag_elk funtzioaren kasu partikular bat da: q parametroaren balioa [] denekoa hain zuzen ere

```
-- Funtzio honek, elementu bat eta zerrenda bat emanda, elementu horren
-- agerpen denak kenduz gelditzen den zerrenda itzuliko du
-- bi zerrenda elkartzen dituen ++ eragiketan oinarritutako murgilketa
-- erabiliz.

kendu_mr_elk :: (Show t, Eq t) => t -> [t] -> [t]

kendu_mr_elk x r = kendu_lag_elk x r []
```

Definizioan ikus daitekeen bezala, kendu_mr_elk izeneko funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina kendu_lag_elk funtzioa errekurtsiboa da.

kendu_lag_elk funtzioak ebazten duen problema kendu_mr_elk funtzioak ebazten duena baino orokorragoa da, izan ere kendu_lag_elk funtzioaren bidez lehenengo zerrendatik elementuaren agerpen denak kentzeaz gain, beste zerrenda bat erantsi diezaiokegu ezkerretik:

kendu_lag_elk 5 [3, 5, 5, 10] [0, 5, 2]

Erantzuna [0, 5, 2, 3, 10] izango litzateke.

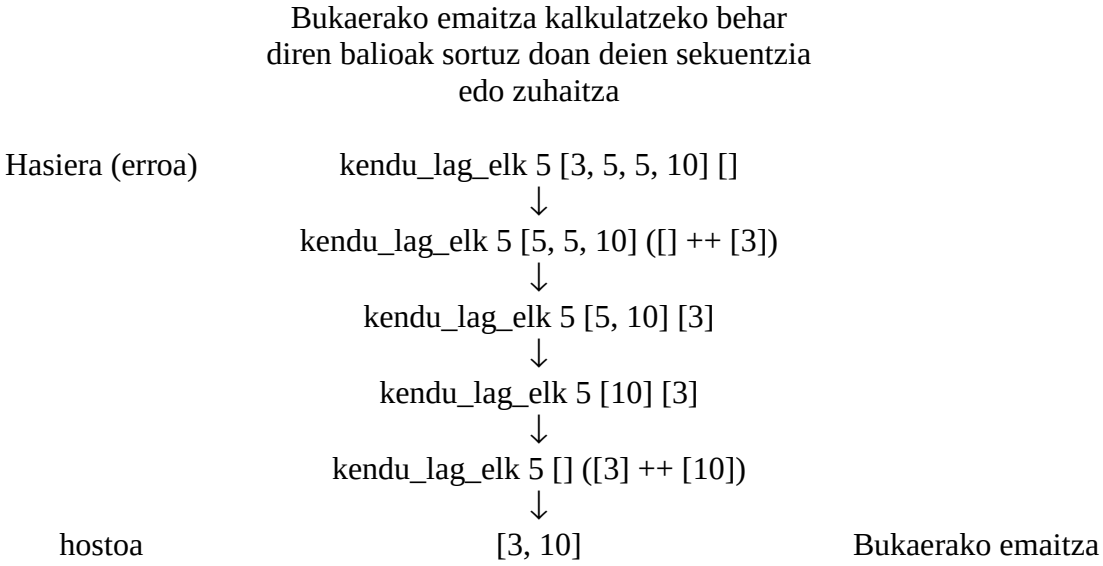
Baina kendu_mr_elk funtzioarekin, emandako elementuaren agerpen denak zerrendatik kenduz geratzen den zerrenda kalkulatzen da, beste ezer erantsi gabe:

kendu_mr_elk 5 [3, 5, 5, 10] = [3, 10]

Zuzenean kendu_lag_elk erabiliz [3, 5, 5, 10] zerrendatik 5 zenbakiaren agerpen denak kenduz geratzen den zerrenda lortu nahi bada, honako hau ipini beharko da:

kendu_lag_elk 5 [3, 5, 5, 10] []

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratzen denez (zuhaitza berriro zeharkatu beharrik gabe), kendu_lag_elk funtzioak bukaerako errekurtsibitatea du:



kendu_lag_elk 5 [3, 5, 5, 10] [] deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzerakoan ordura arte tratatu den zerrenda zatitik hartu beharreko elementuz osatutako zerrenda edukiko da. Azkeneko adabegira iritsitakoan ([3, 10] balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro goruntz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu.

Beraz kendu_lag_elk funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den kendu_gb funtzioa baino eraginkorragoa da.

d) Fibonacci-a: *fib*

0ren fibonaccia 0 da, 1en fibonaccia 1 da eta 2 edo handiagoa den x zenbaki baten fibonaccia x – 1 eta x – 2 zenbakien fibonacciak batuz lortzen da.

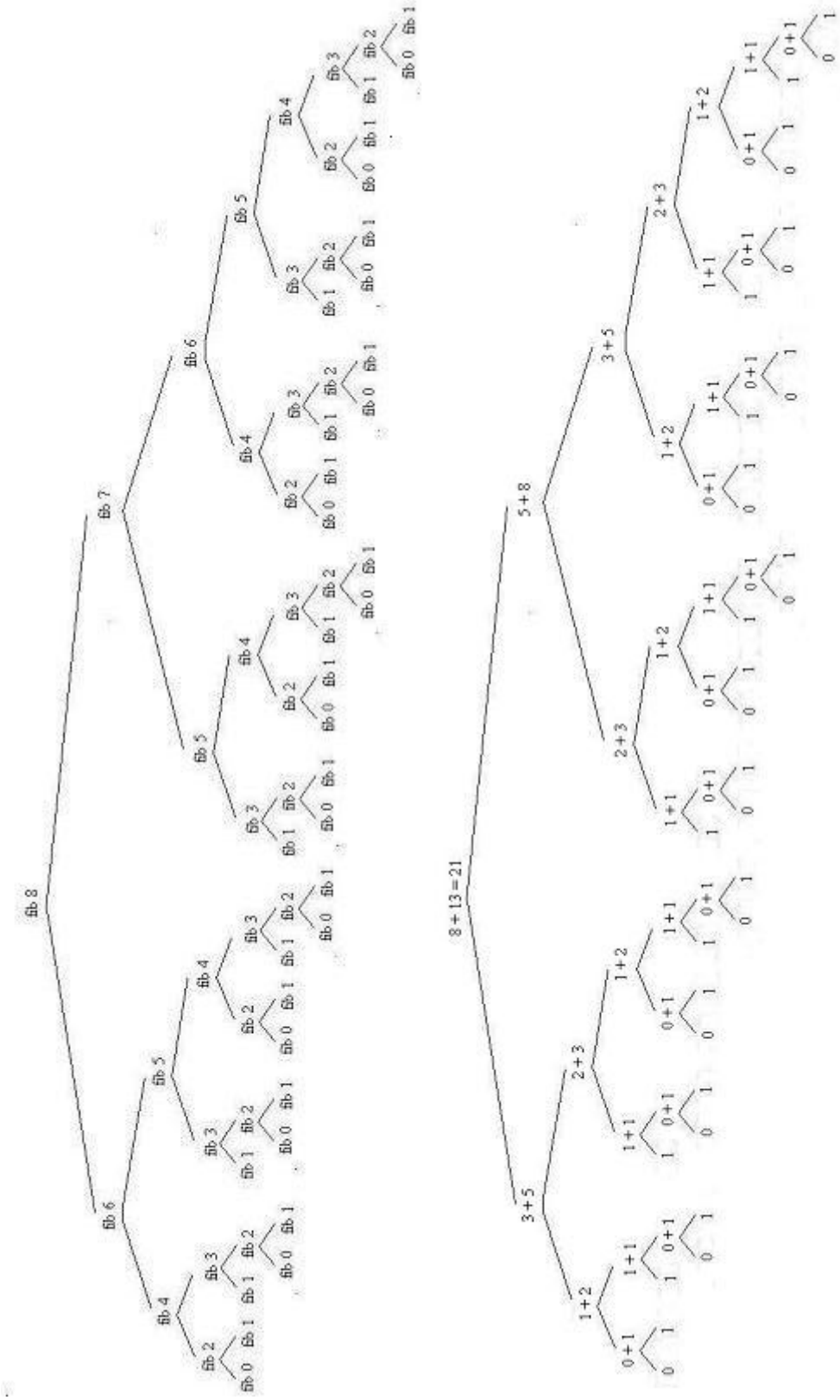
Adibideak:

fib 2 = fib 0 + fib 1 = 0 + 1 = 1
fib 3 = fib 1 + fib 2 = 1 + 1 = 2
fib 4 = fib 2 + fib 3 = 1 + 2 = 3
fib 5 = fib 3 + fib 4 = 2 + 3 = 5
fib 6 = fib 4 + fib 5 = 3 + 5 = 8
fib 7 = fib 5 + fib 6 = 5 + 8 = 13
fib 8 = fib 6 + fib 7 = 8 + 13 = 21

Honaho funtzio honek x zenbakiaren fibonaccia kalkulatzen du murgilketa erabili gabe.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabili gabe.
-- x zenbakia negatiboa baldin bada, errorea.

fib:: Int -> Int
fib x
        | x <= (-1)             = error "Zenbaki negatiboa."
        | x == 0        = 0
        | x == 1        = 1
        | otherwise             = (fib (x - 1)) + (fib (x - 2))
```

Definizio horretan ez da murgilketarik erabili. Fibonaccia kalkulatzen duen funtzio horrek ez du *bukaerako errekurtsibitaterik,* dei errekurtsiboei dagokien zuhaitza garatu ondoren, garapen horretan agertuz joan diren elementuekin emaitza osatzeko zuhaitza berriro zeharkatu behar baita kontrako eran, hostoetatik erroraino. Beraz zuhaitza bigarren aldiz zeharkatu behar da.


fib 8
fib 6
fib 7
fib 4
fib 5
fib 5
fib 6
fib 2
fib 3
fib 3
fib 4
fib 4
fib 5
fib 0 fib 1
fib 1 fib 2
fib 2 fib 3
fib 3 fib 4
fib 3 fib 4
fib 4 fib 5
8 = 13 = 21
3 + 5
5 + 8
1 + 2
2 + 3
2 + 3
3 + 5
0 + 1
1 + 1
1 + 1
1 + 2
1 + 2
2 + 3
0 1
1 0 + 1
1 0 + 1
1 1 + 1
1 1 + 1
1 2 + 3

Aurreko orrialdeko irudian, 8ren fibonaccia kalkulatzeko behar diren deiez osatutako zuhaitza ikus daiteke. Zuhaitz hori errotik hasita eraikitzen da (fib 8 ipintzen duen lekutik) eta beheruntz joan behar da hostoetara iritsi arte (fib 0 eta fib 1 duten adabegietaraino). Fibonaccia kalkulatzerakoan oinarrizko balioak fib 0 eta fib 1 direnez (0 eta 1 hurrenez hurren), hostoetatik abiatuz batura kalkulatu beharko da. Beraz, baturen kalkulua burutzerakoan, zuhaitza hostoetatik errora zeharkatuko da. Errora iritsitakoan bukaerako batura edukiko da, hau da, 8ren fibonaccia.

Zuhaitza bi aldiz zeharkatu behar izateaz gain, kalkulu batzuk errepikatuta daudela ere ikus dezakegu:

- 6ren fibonaccia 2 aldiz kalkulatzen da.
- 5en fibonaccia 3 aldiz kalkulatzen da.
- 4ren fibonaccia 5 aldiz kalkulatzen da.
- 3ren fibonaccia 8 aldiz kalkulatzen da.
- 2ren fibonaccia 13 aldiz kalkulatzen da.
- 1en fibonaccia 21 aldiz kalkulatzen da.
- 0ren fibonaccia 13 aldiz kalkulatzen da.

Kalkuluak hainbeste aldiz errepikatu behar izateak, eraginkortasun eza dakar. Murgilketa erabiliz kalkuluak hainbeste aldiz ez errepikatzea eta zuhaitza behin bakarrik zeharkatzea lortuko dugu.

2 edo handiagoa den x zenbakiaren fibonaccia murgilketa erabiliz honela kalkula daiteke:

- 2 zenbakitik abiatu eta [2..x] tarteko zenbakiak zeharkatuz joan.
- Tartea zeharkatu ahala, zenbaki bakoitzaren fibonaccia kalkulatuz joan (2rena, 3rena, 4rena, …)
- Une bakoitzean zein zenbakitan gauden jakiteko, hau da, beheko muga une bakoitzean zein den jakiteko, bm izeneko parametro berri bat erabiliko da.
- Zenbaki baten fibonaccia kalkulatzeko aurreko bi zenbakien fibonaccia ezagutu behar denez, bm zenbakiaren aurreko bi zenbakien (hau da, bm – 1 eta bm – 2 zenbakien) fibonaccia gordeta edukiko dugu beste bi parametro berriren bidez, aa eta aur (aurrekoaren aurrekoa eta aurrekoa)

Funtzio berriari fib_lag deituko diogu, izan ere fibonaccia kalkulatzen duen funtzioa murgilketaren bidez kalkulatzeko laguntzaile bezala erabiliko baitugu.

```
-- Funtzio honek, x, bm, aa eta aur lau zenbaki oso emanda,
-- [bm..x] tartea zeharkatuko du.
-- x zenbakia negatiboa baldin bada, errorea.
-- bm-ren balioa 1 edo txikiagoa baldin bada, errorea.

fib_lag:: Int -> Int -> Int -> Int -> Int
fib_lag x bm aa aur
        | x <= (-1)             = error "Zenbaki negatiboa."
        | bm <= 1               = error "Beheko muga 2 baino txikiagoa."
        | x == 0 || x == 1      = x
        | bm > x                = error "Eremu hutsa."
        | bm == x               = aa + aur
        | otherwise             = fib_lag x (bm + 1) aur (aa + aur)
```

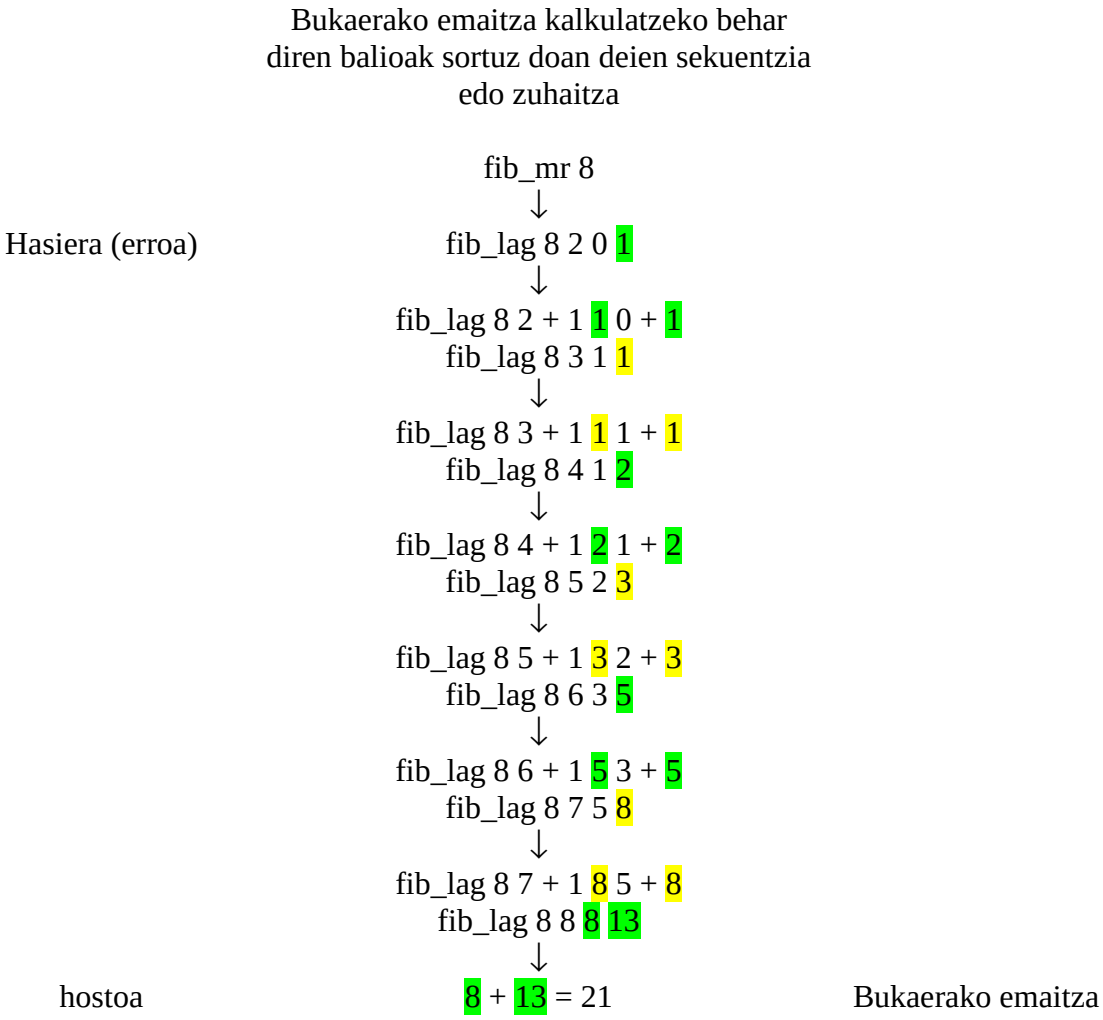
fib_lag funtzioa definitu ondoren, fib_mr izeneko funtzioaren definizioa eman dezakegu, izan ere fib_mr funtzioa fib_lag funtzioaren kasu partikular bat da: bm, aa eta aur parametroen balioak hurrenez hurren 2, 0 eta 1 direnekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabiliz.
-- x zenbakia negatiboa baldin bada, errorea.

fib_mr:: Int -> Int

fib_mr x = fib_lag x 2 0 1
```

Definizioan ikus daitekeen bezala, fib_mr izeneko funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina fib_lag funtzioa errekurtsiboa da.

fib_lag funtzioak ebazten duen problema fib_mr funtzioak ebazten duena baino orokorragoa da.

fib_mr erabiliz 8ren fibonaccia nola kalkulatuko genukeen azalduko da jarraian:



Adibide honekin, murgilketaren bidez lortutako soluzioaren bidez askoz adabegi gutxiago dituen eta adar bakarra duen zuhaitza eraikitzen dela ikus dezakegu. Gainera ez du kalkulurik errepikatzen. Ondorioz, askoz eraginkorragoa da.

Int mota mugatua da. Int motaren barruan adieraz daitekeen balio txikiena zein den jakiteko honako hau idatzi beharko dugu:

```
minBound :: Int
```

Eta balio handiena zein den jakiteko honako hau idatzi beharko dugu:

```
maxBound :: Int
```

Int mota mugatua denez, fib, fib_lag eta fib_mr funtzioak, nahiko txikiak diren x parametroaren balioentzat erantzun ezinda gelditzen dira edo zentzurik ez duen erantzuna itzultzen dute (adibidez balio negatibo bat). Hori dela eta, Integer mota erabiltzen duten bertsioak erabil ditzakegu. Integer motak ez du mugarik eta zenbaki oso denak adieraz daitezke mota horretan. Integer mota erabiliz idatzi diren bertsio hauei izenean 2 erantsi zaie:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabili gabe.
-- x zenbakia negatiboa baldin bada, errorea.
-- Int mota erabili beharrean Integer mota erabili da.

fib2:: Integer -> Integer
fib2 x
        | x <= (-1)             = error "Zenbaki negatiboa."
        | x == 0                = 0
        | x == 1                = 1
        | otherwise             = (fib2 (x - 1)) + (fib2 (x - 2))
```

```
-- Funtzio honek, x, bm, aa eta aur lau zenbaki oso emanda,
-- [bm..x] tartea zeharkatuko du.
-- x zenbakia negatiboa baldin bada, errorea.
-- bm-ren balioa 1 edo txikiagoa baldin bada, errorea.
-- Int mota erabili beharrean Integer mota erabili da.

fib2_lag:: Integer -> Integer -> Integer -> Integer -> Integer
fib2_lag x bm aa aur
        | x <= (-1)             = error "Zenbaki negatiboa."
        | bm <= 1               = error "Beheko muga 2 baino txikiagoa."
        | x == 0 || x == 1      = x
        | bm > x                = error "Eremu hutsa."
        | bm == x               = aa + aur
        | otherwise             = fib2_lag x (bm + 1) aur (aa + aur)
```

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabiliz.
-- x zenbakia negatiboa baldin bada, errorea.
-- Int mota erabili beharrean Integer mota erabili da.

fib2_mr:: Integer -> Integer

fib2_mr x = fib2_lag x 2 0 1
```

Integer mota erabiliz idatzitako bertsio hauen bidez, fib2_mr funtzioa fib2 funtzioa baino eraginkorragoa dela ikus daiteke. Horretarako nahikoa da, esate baterako, honako kasu hauekin frogatzea:

```
fib2 40
fib2_mr 40

fib2 30
fib2_mr 30
```

eta abar.

7.8. Aurredefinitutako funtzio batzuk

Aurreko ataletan, errekurtsibitatea eta murgilketa erabiliz funtzio asko definitu ditugu. Funtzio horietako batzuk dagoeneko Haskell-en definituta daude (izen desberdin batekin). Funtzio horiek aurredefinitutako funtzioak bezala ezagutzen dira. Atal honetan Haskell-en aurredefinituta dauden funtzio batzuk aipatuko ditugu. Gauza bera egiten duten funtzioak defini ditzakegun arren, ondo dator aurredefinitutako funtzioak ezagutzea, horrela ez baitaukagu geuk definitutako funtzioak dituen fitxategia eskura eduki beharrik.

Aurredefinitutako funtzio batzuk erabili ahal izateko honako hau ipini beharko dugu fitxategiaren goiburukoan:

```
import Data.List
```

Data.List Haskell-en aurredefinituta dagoen modulu bat da eta aipatutako lerroa ipintzearekin nahikoa da, ez da beste ezer egin behar.

head :: [t] -> t Zerrendako lehenengo elementua itzuliko du

head [4, 6, 8] → 4 head [] → error

tail :: [t] -> [t] Zerrendako lehenengo elementua kenduz geratzen den zerrenda itzuliko du

tail [4, 6, 8] → [6, 8] tail [] → error

last :: [t] -> t Zerrendako azkeneko elementua itzuliko du

last [4, 6, 8] → 8 last [] → error

init :: [t] -> [t] Zerrendako azkeneko elementua kenduz geratzen den zerrenda itzuliko du

init [4, 6, 8] → [4, 6] init [] → error

null :: [t] -> Bool Zerrenda hutsa al den erabakitzen du

null [4, 6, 8] → False null [] → True

length :: [t] -> Int Zerrendaren luzera (elementu-kopurua) itzultzen du

length [4, 6, 8] → 3 length [] → 0

sum :: Num t => [t] -> t Zerrendako elementuen batura itzuliko du. Elementuek zenbakizkoak izan beharko dute.

sum [4, 6, 8] → 18 sum [] → 0

product :: Num t => [t] -> t Zerrendako elementuen biderkadura itzuliko du. Elementuek zenbakizkoak izan beharko dute.

product [4, 6, 10] → 240 product [] → 1

reverse :: [t] -> [t] Alderantzizko zerrenda itzuliko du.

reverse [4, 6, 8] → [8, 6, 4] reverse [] → []

`elem` :: Eq t => t -> [t] -> Bool Elementu bat zerrenda batean al dagoen erabakiko du. Elementuentzat berdina izatea zer den definituta egon beharko du.

9 `elem` [4, 9, 9, 8] → True
7 `elem` [4, 9, 9, 8] → False
9 `elem` [] → False

`notElem` :: Eq t => t -> [t] -> Bool Elementu bat zerrenda batean ez al dagoen erabakiko du. Elementuentzat berdina izatea zer den definituta egon beharko du.

9 `notElem` [4, 9, 9, 8] → False
7 `notElem` [4, 9, 9, 8] → True
9 `notElem` [] → True

9 `notElem` [4, 9, 9, 8] eta
not (9 `elem` [4, 9, 9, 8]) baliokideak dira

take :: Int -> [t] -> [t] Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu hartuz osatutako zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa bada, zerrenda osoa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda hutsa itzuliko da. Zenbakiak Int motakoa izan behar du.

take 2 [4, 6, 8] → [4, 6]
take 5 [4, 6, 8] → [4, 6, 8]
take (-2) [4, 6, 8] → []
take 0 [4, 6, 8] → []

genericTake :: Integral t1 => t1 -> [t2] -> [t2] Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu hartuz osatutako zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa bada, zerrenda osoa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda hutsa itzuliko da. Zenbakiak Int edo Integer motakoa izan beharko du, hau da Integral erakoa.

genericTake 2 [4, 6, 8] → [4, 6]
genericTake 5 [4, 6, 8] → [4, 6, 8]
genericTake (-2) [4, 6, 8] → []
genericTake 0 [4, 6, 8] → []

takeWhile :: (t -> Bool) -> [t] -> [t] Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak False balioa ematen dion zerrendako lehenengo elementura arteko elementuez osatutako zerrenda itzuliko du. Beraz funtzioak False ematen dion elementu bat aurkitu arte edo zerrenda bukatu arte jarraituko du.

takeWhile (< 3) [1, 2, 3, 4, 1, 2, 3, 4] == [1, 2]
takeWhile (< 9) [1, 2, 3] == [1, 2, 3]
takeWhile (< 0) [1, 2, 3] == []

drop :: Int -> [t] -> [t] Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu kendu ondoren geratzen den zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa bada, zerrenda hutsa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda osoa itzuliko da. Zenbakiak Int motakoa izan beharko du.

drop 2 [4, 6, 8] → [8]
drop 5 [4, 6, 8] → []
drop (-2) [4, 6, 8] → [4, 6, 8]
drop 0 [4, 6, 8] → [4, 6, 8]

genericDrop :: Integral t1 => t1 -> [t2] -> [t2] Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu kendu ondoren geratzen den zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa bada, zerrenda hutsa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda osoa itzuliko da. Zenbakiak Int edo Integer motakoa izan behar du, hau da Integral erakoa.

genericDrop 2 [4, 6, 8] → [8]
genericDrop 5 [4, 6, 8] → []
genericDrop (-2) [4, 6, 8] → [4, 6, 8]
genericDrop 0 [4, 6, 8] → [4, 6, 8]

dropWhile :: (t -> Bool) -> [t] -> [t] Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak False balioa ematen dion zerrendako lehenengo elementura arteko elementuak kenduz osatutako zerrenda itzuliko du. Beraz funtzioak False ematen dion elementu bat aurkitu arte edo zerrenda bukatu arte jarraituko du.

dropWhile (< 3) [1, 2, 3, 4, 1, 2, 3, 4] == [3, 4, 5, 1, 2, 3]
dropWhile (< 9) [1, 2, 3] == []
dropWhile (< 0) [1, 2, 3] == [1, 2, 3]

maximum :: Ord t => [t] -> t Zerrendako balio handiena itzuliko du. Zerrendako elementuentzat ordenak definituta egon beharko du (txikiagoa izatea, handiagoa, eta abar).

maximum [4, 9, 9, 8] → 9 maximum [] → error

minimum :: Ord t => [t] -> t Zerrendako balio txikiena itzuliko du. Zerrendako elementuentzat ordenak definituta egon beharko du (txikiagoa izatea, handiagoa, eta abar).

minimum [4, 9, 9, 8] → 4 minimum [] → error

concat :: [[t]] -> [t] Zerrendez osatutako zerrenda bat emanda, zerrenda denak elkartuz zerrenda bakarra itzuliko du.

concat [[1, 2, 3], [4], [], [5, 6]] → [1, 2, 3, 4, 5, 6]

and :: [Bool] -> Bool Balio Boolearrez osatutako zerrenda bat emanda, denak True badira True eta bestela False itzuliko du.

and [4 > 1, True, 5 == 5] → True
and [4 < 1, True, 5 == 5] → False

or :: [Bool] -> Bool Balio Boolearrez osatutako zerrenda bat emanda, gutxienez bat True bada True eta bestela False itzuliko du.

or [4 > 1, True, 5 == 5] → True
or [4 < 1, True, 5 == 5] → True

map :: (t1 -> t2) -> [t1] -> [t2] Funtzio bat eta zerrenda bat emanda, funtzioa zerrendako elementu bakoitzari aplikatuz osatzen den zerrenda berria itzuliko du.

map (>5) [3, 8, 5, 9] → [False, True, False, True]
map (+2) [3, 8, 5, 9] → [5, 10, 7, 11]

any :: (t -> Bool) -> [t] -> [Bool] Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak zerrendako elementuren batentzat True itzultzen badu, orduan any-k ere True itzuliko du eta bestela False itzuliko du.

any (>5) [3, 8, 5, 9] → True
any (>10) [3, 8, 5, 9] → False

all :: (t -> Bool) -> [t] -> [Bool] Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak zerrendako elementu denentzat True itzultzen badu, orduan all-ek ere True itzuliko du eta bestela False itzuliko du.

all (>5) [3, 8, 5, 9] → False
all (>2) [3, 8, 5, 9] → True

zip :: [t1] -> [t2] -> [(t1, t2)] Bi zerrenda emanda, posizio berean dauden elementuekin eratutako bikoteez osatutako zerrenda itzuliko du. Zerrenda bat bestea baino laburragoa bada, bikote-eraketa zerrenda laburrena bukatzean bukatuko da.

zip [1, 2, 3] [4, 5, 6] → [(1, 4), (2, 5), (3, 6)]
zip [1, 2] [4, 5, 6] → [(1, 4), (2, 5)]

Azpimarratzekoa da zip [1, 2, 3] [4, 5, 6] eta
[(x, y) | x <- [1, 2, 3], y <- [4, 5, 6]] zerrenda desberdinak direla, izan ere azken hau honako zerrenda hau da:
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]

zip3 :: [t1] -> [t2] -> [t3] -> [(t1, t2, t3)] Hiru zerrenda emanda, posizio bereko elementuez eratutako hirukoteen zerrenda itzuliko du.

zip3 [1, 2, 3] [4, 5, 6] [7, 8, 9]→ [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
zip3 [1, 2] [4, 5, 6] [7, 8, 9]→ [(1, 4, 7), (2, 5, 8)]

7.9. Moten gaineko baldintzak

Aurreko ataletako funtzio batzuen motak ematerakoan motek bete beharreko baldintzak ipini ditugu (Eq t, Show t, eta abar). Atal honetan moten gainean ezarri daitezkeen baldintza batzuk aipatuko ditugu:

- **Eq** t: t motako elementuak bedinak al diren ala ez erabakitzeko bideak definituta egon behar duela adierazteko.
- **Ord** t: t motako elementuen artean ordenak (handiagoa, txikiagoa, eta abar) definituta egon behar duela adierazteko.
- **Show** t: t motako elementuak pantailan aurkezteko erak definituta egon behar duela adierazteko. Mota berri bat definitzen denean (adibidez zuhaitz bitarra, pila, multzoa, pertsona, bezeroa, produktua, eta abar), mota horretako elementuak pantailan nola aurkeztuko diren finkatu behar da. Aurkezteko era ez bada finkatzen, Haskell-ek ez du jakingo nola aurkeztu eta errorea sortuko du.
- **Read** t: t motako elementuak karaktere-kate bezala irakurri ahal izango direla adierazteko. Mota berri bat definitzen denean, mota horretako elementuak teklatu bidez irakurtzeko era bat zehaztu beharko da.
- **Bounded** t: t motak beheko muga eta goiko muga izan behar dituela adierazteko. Esate baterako, Int mota beheko eta goiko mugak dituen mota bat da.
- **Num** t: t motak zenbakizkoa izan beharko duela adierazteko. Esate baterako, Int, Integer, Float, Double eta Rational (Integer motako elementuekin osatutako zatikiak) zenbakizko motak dira.
- **Integral** t: t motak zenbaki osozkoa izan beharko duela adierazteko, hau da, Int edo Integer.
- **Fractional** t: adibidez t motak Rational izan beharko duela adierazteko.
- **Floating** t: t motak Float edo Double izan behar duela adierazteko.
- **Enum** t: t motak zenbagarria izan behar duela adierazteko.

Mota berri bat definitzerakoan, goian aipatu diren baldintza batzuk bete beharko dituela adierazi dezakegu. Adibidez:

```
data Pila t = Phutsa | Pilaratu (t, Pila t)
        deriving (Eq, Ord, Show, Read)

data Urtaroa = Negua | Udaberria | Uda | Udazkena
        deriving (Eq, Ord, Enum, Show, Read)
```

Bigarren adibide honetan, Eq eta Ord-en bidez urtaroen izenak konpara daitezkeela (Negua Udaberria baino txikiagoa da, Udaberria Uda baino txikiagoa da eta abar) adierazten da; Enum-en bidez zenbatu daitezkeela; Show-ren bidez pantailan hor agertzen diren bezala aurkeztuko direla eta Read-en bidez izen horiek teklatutik ere irakur daitezkeela esaten da.

Funtzio bat definitzerakoan, baldintza bakarra baldin badago, esate baterako Eq, honela ipiniko genuke:

```
f :: Eq t => …
```

Baldintza bat baino gehiago badaude, esate baterako Eq, Show eta Read, honela ipiniko lirateke:

```
f :: (Eq t, Show t, Read t) => …
```

7.10. Zerrenda-eraketa

Haskell-en erabili daitekeen zerrenda-eraketarako teknika landuko dugu atal honetan. Teknika hori era isolatuan edo gai honetako aurreko ataletan landutako errekurtsibitatearekin nahasian erabil daiteke.

7.10.1. Notazioa

Haskell-eko zerrenda-eraketaren teknika matematikan multzoak definitzeko erabili ohi den teknikaren antzekoa da. Teknika matematiko hori dagoeneko irakasgai honetako 2. gaian erabili dugu lengoaiak hitzez osatutako multzo bezala definitzeko.

Matematikan multzoak honako era honetan ohi dira askotan:

$$\{ x \mid x \in D \land P(x)\}$$

que se leería como "D eremukoak diren eta P propietatea (edo baldintza) betetzen duten x elementuez osatutako multzoa". Hona hemen adibide bat:

$$\{ x \mid x \in N \land x \bmod 10 = 0\}$$

Hor zenbaki arrunten eremukoak diren eta 10 zenbakiaren anizkoitzak izatearen propietatea (edo baldintza) betetzen duten zenbakiez osatutako multzoa daukagu.

Hona hemen beste adibide bat:

$$\{ x \mid 20 \leq x \leq 100 \land \neg\exists y(2 \leq y \leq x - 1 \land x \bmod y = 0)\}$$

Hor 20 eta 100 zenbakiek mugatzen duten tartekoak edo eremukoak diren eta lehena izatearen propietatea (edo baldintza) betetzen duten zenbakiez osatutako multzoa daukagu. Beraz, $20 \leq x \leq 100$ eremua izango litzateke eta $\neg\exists y(2 \leq y \leq x - 1 \land x \bmod y = 0)$ propietatea izango litzateke.

2. gaian landutako multzoen definizioetara itzuliz, egitura bera dutela ikus dezakegu:

$$\{ w \mid w \in A^* \land |w| \bmod 2 = 0\}$$

Hor eremua A^* izango litzateke (hau da, A alfabetoaren gainean definitutako lengoaia unibertsala) eta propietatea luzera bikoitia edukitzea izango litzateke. Beraz, A alfabetoaren gainean definitutakoak diren eta luzera bikoitia duten hitzez osatutako lengoaia definitu da.

Multzoak definitzeko egitura hori jarraituz, zerrendak definitzeko aukera eskaintzen du Haskell-ek. Zerrendak definitzeko honako egitura hau izango genuke:

```
[ x | x <- D, P(x)]
```

Beraz matematikako { eta } sinboloen ordez [eta] sinboloak erabili behar dira, ∈ sinboloaren ordez <- erabili behar da eta ∧ sinboloaren ordez koma erabili behar da.

7.10.2. Oinarrizko adibideak

Jarraian, zerrenda-eraketaren teknika darabilten oinarrizko adibide batzul azalduko dira.

1. adibidea:
Osoa den n zenbakia emanda, 0tik n-ra arteko zenbaki denak dituen zerrenda itzuliko du *zenbakiak* izeneko honako funtzio honek.

```
zenbakiak:: Int -> [Int]

zenbakiak n = [ x | x <- [0..n]]
```

zenbakiak 5 deiak [0, 1, 2, 3, 4, 5] zerrenda itzuliko du.

Beste aukera bat:

```
zenbakiak2:: Int -> [Int]

zenbakiak2 n = [0..n]
```

2. adibidea:
Osoa den n zenbakia emanda, 0tik n-ra arteko zenbaki bikoiti denak dituen zerrenda itzuliko du *bikoitiak* izeneko honako funtzio honek.

```
bikoitiak:: Int -> [Int]

bikoitiak n = [ x | x <- [0..n], x `mod` 2 == 0]
```

bikoitiak 5 deiak [0, 2, 4] zerrenda itzuliko du.

3. adibidea:
Osoa den n zenbakia emanda, [0..n] tartekoak diren zenbakiz osatutako (x, y) erako bikote denez eratutako zerrenda itzuliko du *bikoteak* izeneko honako funtzio honek.

```
bikoteak:: Int -> [(Int, Int)]

bikoteak n = [ (x, y) | x <- [0..n], y <- [0..n]]
```

bikoteak 3 deiak honako zerrenda hau itzuliko du [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3, 1), (3, 2), (3, 3)]

4. adibidea:

Negatiboak ez diren zenbaki osoz osatutako (x, y) erako bikote denez eratutako zerrenda lortzeko *bikoteak_infinitua* izeneko honako funtzio hau defini dezakegu.

```
bikoteak_infinitua:: [(Integer, Integer)]

bikoteak_infinitua = [ (x, y) | x <- [0..], y <- [0..]]
```

[0..] zerrendaren bidez negatiboak ez diren zenbaki osoen {0, 1, 2, 3, …} multzo infinitua adieraz dezakegu.

bikoteak_infinitua deiak honako zerrenda infinitu hau itzuliko du:
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), …]

Zerrenda horretan (1, 0), (1, 1), (1, 2), …(2, 0), (2, 1), (2, 2), (2, 3), …erako bikoteak ere agertu beharko lukete baina bikoteak_infinitua funtzioak ez ditu inoiz sortuko. Funtzio horrek lehenengo osagai bezala 0 balioa duten bikoteak sortuz joango da amaierarik gabe. 2. gaian, N × N multzoaren zenbagarritasuna aztertu genuenean, N × N multzoko bikoteak ordenatzeko era egokia honako hau zela esan genuen:

[(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), (0, 3), (1, 2), (2, 1), (3, 0), …]

Baina Haskell-ek ez du sortuko automatikoki orden egoki hori. Momentuz honela lagako dugu problema hau baina aurrerago ariketa bezala planteatuko dugu.

5. adibidea:

Osoa den n zenbaki bat emanda, *bikoteak_finitua* izeneko funtzioak bikoteak_infinitua funtzioa sortuz joango den zerrenda infinituko lehenengo n bikoteez osatutako zerrenda itzuliko du:

```
bikoteak_finitua:: Integer -> [(Integer, Integer)]

bikoteak_finitua n = genericTake n bikoteak_infinitua
```

Osoa den n zenbaki bat eta zerrenda bat emanda, *genericTake* funtzioak zerrendako lehenengo n elementuez osatutako zerrenda itzuliko du.

bikoteak_finitua 5 deiak honako zerrenda finitua itzuliko du:
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)]

6. adibidea:

Osoa den n zenbakia emanda, lehenengo osagaia bigarrena baino txikiagoa duten [0..n] tartekoak diren zenbakiz osatutako (x, y) erako bikote denez eratutako zerrenda lortzeko *bikoteak_hand* izeneko honako funtzio hau defini dezakegu.

```
bikoteak_hand:: Integer -> [(Integer, Integer)]

bikoteak_hand n = [ (x, y) | x <- [0..n], y <- [0..n], x < y]
```

bikoteak_hand 3 deiak [(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)] zerrenda itzuliko du.

7. adibidea:

Osoa den n zenbakia emanda, [0..n] tartekoak diren zenbakiz osatutako (x, y) erako bikoteei dagozkien baturez eratutako zerrenda lortzeko *bikoteak_batu* izeneko honako funtzio hau defini dezakegu.

```
bikoteak_batu:: Int -> [Int]

bikoteak_batu n = [ x + y | x <- [0..n], y <- [0..n]]
```

bikoteak_batu 3 deiak [0 + 0, 0 + 1, 0 + 2, 0 + 3, 1 + 0, 1 + 1, 1 + 2, 1 + 3, 2 + 0, 2 + 1, 2 + 2, 2 + 3, 3 + 0, 3 + 1, 3 + 2, 3 + 3] zerrenda itzuliko luke, baina eragiketak eginda

[0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6]

8. adibidea:

Osoa den n zenbakia emanda, [], [1], [1, 2], [1, 2, 3], …, [1, 2, 3, …, n] zerrendez osatutako zerrenda lortzeko *zerrendak* izeneko honako funtzio hau defini dezakegu.

```
zerrendak:: Int -> [[Int]]

zerrendak n = [ [1..x] | x <- [0..n]]
```

zerrendak 3 deiak [[], [1], [1, 2], [1, 2, 3]] zerrenda itzuliko luke.

9. adibidea:

Osoa den n zenbakia emanda, n-ren zatitzaile denez osatutako zerrenda kalkulatuko duen *zatizer_ze* funtzioa honela defini daiteke zerrenda-eraketaren teknika erabiliz.

```
zatizer_ze:: Integer -> [Integer]

zatizer_ze n = [ x | x <- [1..n], n `mod` x == 0]
```

zatizer_ze 12 deiak [1, 2, 3, 4, 6, 12] zerrenda itzuliko du.
zatizer_ze 7 deiak [1, 7] zerrenda itzuliko du.

Kasu berezi bezala, n-ren balioa 0 bada edo negatiboa bada, [1..n] tartea hutsa izango da eta ondorioz zatizer_ze funtzioak zerrenda hutsa itzuliko du.

zatizer_ze (-5) deiak [] zerrenda itzuliko du.

Bestalde, n-ren balioa 0 edo negatiboa denean zerrenda hutsa itzuli beharrean errore-mezu bat aurkeztea nahi izanez gero, honako funtzio hau defini genezake:

```
zatizer2_ze:: Integer -> [Integer]

zatizer2_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = [ x | x <- [1..n], n `mod` x == 0]
```

Eraginkorragoa den funtzio bat ere defini genezake [2..(n `div` 2)] tartea bakarrik aztertuz:

```
zatizer3_ze:: Integer -> [Integer]

zatizer3_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | n == 1 = [1]
        | otherwise     = [1] ++ [ x | x <- [2..( n `div` 2)], n `mod` x == 0] ++ [n]
```

Hirugarren bertsio hau jarraituz, 2 edo handiagoa den zenbaki bat daukagun bakoitzean, 1 eta n zuzenean sartuko dira zatitzaileen zerrendan eta gero [2..(n `div` 2)] tartean bakarrik bilatu beharko da.

zatizer_ze eta zatizer3_ze funtzioen arteko desberdintasuna nabaria da 1000000 bezalako zenbaki handientzat.

10. adibidea:

Osoa den n zenbakia emanda, n lehena al den erabakiko duen *lehena_ze* funtzioa honela defini daiteke zerrenda-eraketaren teknika erabiliz.

```
lehena_ze:: Integer -> Bool

lehena_ze n
        | n <= 0          = error "Zenbakia ez da positiboa"
        | n == 1 = False
        | otherwise       = (length [ x | x <- [2..( n `div` 2)], n `mod` x == 0]) == 0
```

lehena_ze 12 deiak False balioa itzuliko du
lehena_ze 7 deiak True balioa itzuliko du

11. adibidea:

Osoa den n zenbakia emanda, n zenbakiaren faktoriala itzuliko duen *fakt_ze* funtzioa honela defini dezakegu zerrenda-eraketaren teknika eta aurredefinitutako funtzioak erabiliz.

```
fakt_ze:: Integer -> Integer

fakt_ze n = product [1..n]
```

Funtzio horrek 1 balioa itzuliko du n negatiboa denean. Izan ere, n-ren balioa 1 baino txikiagoa denean, [1..n] tartea hutsa izango da eta aurredefinitutako product funtzioak 1 itzuliko du. Zenbaki negatiboentzat emaitza bezala 1 itzuli beharrean errore-mezua aurkeztea nahi izanez gero, honako funtzio hau defini dezakegu:

```
fakt2_ze:: Integer -> Integer

fakt2_ze n
        | n <= (-1)       = error "Zenbakia negatiboa da"
        | otherwise       = product [1..n]
```

fakt_ze 4 eta fakt2_ze 4 deiek 24 balioa itzultzen dute.

12. adibidea:

Osoa den n zenbakia emanda, n zenbakia betea al den erabakiko duen *betea_ze* funtzioa definituko dugu orain zerrenda-eraketaren teknika eta aurredefinitutako funtzioak erabiliz.
Positiboa den n zenbaki bat betea da bere zatitzaileen batura (n bera kontuan hartzeke) n denean.

```
betea_ze:: Integer -> Bool

betea_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = (sum [ x | x <- [1..n - 1], n `mod` x == 0]) == n
```

betea_ze 6 deiak True itzuliko du 6 = 1 + 2 + 3 baita
betea_ze 28 deiak True itzuliko du 28 = 1 + 2 + 4 + 7 + 14 betezen baita
betea_ze 12 deiak False itzuliko du 12 ≠ 1 + 2 + 3 + 4 + 6 baita

28ren ondoren hurrengo zenbaki betea 496 da.

Zatitzaileak [1..(n `div` 2)] tartean bakarrik bilatuz funtzioaren eraginkortasuna hobetu daiteke.

```
betea2_ze:: Integer -> Bool

betea2_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = (sum [ x | x <- [1..(n `div` 2)], n `mod` x == 0]) == n
```

13. adibidea:

Osoa den n zenbakia emanda, n zenbakiaren faktoreen zerrenda itzuliko duen *faktoreak_ze* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz.

Gogoratu n-ren faktoreak n-ren zatitzaile lehenak direla:

```
faktoreak_ze:: Integer -> [Integer]

faktoreak_ze n
        | n <= 1        = error "Zenbakia 2 baino txikiagoa da"
        | otherwise     = [ x | x <- [2..n], n `mod` x == 0, lehena_ze x]
```

faktoreak_ze 6 deiak [2, 3] itzuliko du
faktoreak_ze 12 deiak [2, 3] itzuliko du
faktoreak_ze 8 deiak [2] itzuliko du
faktoreak_ze 7 deiak [7] itzuliko du

Funtzio hori hobetzeko, alde batetik n lehena al den aztertu daiteke eta n lehena ez denean zatitzaile lehenen bilaketa [2..(n `div` 2)] tartera muga dezakegu.

```
faktoreak2_ze:: Integer -> [Integer]

faktoreak2_ze n
        | n <= 1          = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n     = [n]
        | otherwise       = [ x | x <- [2..(n `div` 2)], n `mod` x == 0, lehena_ze x]
```

14. adibidea:

Osoa den n zenbakia emanda, n zenbakia faktore lehenetan deskonposatu eta deskonposaketa horri dagokion zerrenda itzuliko duen *desk_ze* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
desk_ze:: Integer -> [Integer]

desk_ze n
        | n <= 1          = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n     = [n]
        | otherwise       = qs((faktoreak2_ze n) ++
                                (desk_ze (n `div` (product (faktoreak2_ze n)))))
```

Hor qs funtzioa zenbaki osozko zerrenda bat ordenatzen duen funtzioa da. Funtzio hori 13.adibidean definitu da.

desk_ze 6 deiak [2, 3] itzuliko du
desk_ze 12 deiak [2, 2, 3] itzuliko du
desk_ze 8 deiak [2, 2, 2] itzuliko du
desk_ze 7 deiak [7] itzuliko du

Definizio batean behin baino gehiagotan kalkulatzen den zati bat agertzen denean, **where** erabiliz izen bat eman diezaiokegu.

```
desk2_ze:: Integer -> [Integer]

desk2_ze n
        | n <= 1          = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n     = [n]
        | otherwise       = qs(w ++ (desk2_ze (n `div` (product w))))
                                where w = faktoreak2_ze n
```

15. adibidea:

2tik hasita, zenbaki denen faktoreen zerrendak itzuliko dituen *denen_faktoreak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
denen_faktoreak :: [[Integer]]

denen_faktoreak = [faktoreak2_ze y | y <- [2..]]
```

denen_faktoreak funtzioa honako zerrenda infinitua aurkeztuz joango litzateke:
[[2], [3], [2], [5], [2, 3], [7], [2], [3], [2, 5], [11], [2, 3], [13], [2, 7], [3, 5], [2], …

16. adibidea:

Osoa den n zenbakia emanda, 2tik hasita n-ra arteko zenbaki denen faktoreen zerrendak itzuliko dituen *faktoreak_finitua* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
faktoreak_finitua :: Integer -> [[Integer]]

faktoreak_finitua n = [faktoreak2_ze x | x <- [2..(n + 1)]]
```

faktoreak_finitua 5 deiak honako zerrenda finitu hau itzuliko luke:
[[2], [3], [2], [5], [2, 3]]
Hor 2, 3, 4, 5 eta 6ren faktoreen zerrendak ditugu.

Definizio hori jarraituz, n-ren balioa 0 edo txikiagoa denean zerrenda hutsa itzuliko da. Beste aukera bat n negatiboa denean errore-mezua aurkeztea izango litzateke. Gainera genericTake eta denen_faktoreak funtzioak ere erabil ditzakegu:

```
faktoreak_finitua2 :: Integer -> [[Integer]]

faktoreak_finitua2 n
        | n < 0         = error "Negatiboa"
        | otherwise     = genericTake n denen_faktoreak
```

17. adibidea:

2tik hasita eta amaierarik gabe, zenbaki bakoitza eta zenbakiari dagokion faktore-zerrendaz eratutako bikoteak dituen zerrenda aurkeztuz joango den *faktorizatuak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
faktorizatuak :: [(Integer,[Integer])]

faktorizatuak = zip [2..] denen_faktoreak
```

Faktorizatuak funtzioa honako zerrenda infinitua aurkeztuz joango litzateke:
[(2, [2]), (3, [3]), (4, [2]), (5, [5]), (6, [2, 3]), (7, [7]), (8, [2]), (9, [3]), (10, [2, 5]), (11, [11]), (12, [2, 3]), (13, [13]), (14, [2, 7]), (15, [3, 5]), (16, [2]), …

18. adibidea:

Lehenengo osagai bezala lehena ez den eta faktore bakarra duen zenbakia duten *faktorizatuak* zerrendako bikoteez osatutako zerrenda itzuliko duen *faktore_bakarrekoak* funtzioa honela defini dezakegu:

```
faktore_bakarrekoak :: [(Integer,[Integer])]

faktore_bakarrekoak = [(x,y) | (x,y) <- faktorizatuak, not (lehena_ze x), length y == 1]
```

faktore_bakarrekoak funtzioa honako zerrenda infinitu hau aurkeztuz joango litzateke:
[(4, [2]), (8, [2]), (9, [3]), (16, [2]), (25, [5]), (27, [3]), …

faktore_bakarrekoak funtzioa definitzeko beste aukera bat honako hau da:

```
faktore_bakarrekoak2 :: [(Integer,[Integer])]

faktore_bakarrekoak2 = [(x,y:s) | (x,y:s) <- faktorizatuak, not (lehena_ze x), s == []]
```

faktore_bakarrekoak funtzioa definitzeko hirugarren aukera:

```
faktore_bakarrekoak3 :: [(Integer,[Integer])]

faktore_bakarrekoak3 = [ (x,[y]) | (x,[y]) <- faktorizatuak, not (lehena_ze x)]
```

Bigarren osagaia elementu bakarreko zerrenda izatea nahi dugunez eta gainera, lehenengo osagaia lehena ez izateagatik lehenengo osagai hori eta dagokion faktore-zerrendako elementu bakarrak berdinak ezin dutenez izan, honako laugarren aukera hau ere eman dezakegu:

```
faktore_bakarrekoak4 :: [(Integer,[Integer])]

faktore_bakarrekoak4 = [ (x, [y]) | (x, [y]) <- faktorizatuak, x /= y]
```

19. adibidea:

Lehenak ez diren eta faktore bakarra duten zenbakiz osatutako zerrenda infinitua aurkeztuz joango den *erabateko_berredurak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
erabateko_berredurak :: [Integer]

erabateko_berredurak = [ x | (x,y) <- faktore_bakarrekoak]
```

erabateko_berredurak funtzioa honako zerrenda aurkeztuz joango litzateke:
[4, 8, 9, 16, 25, 27, …

20. adibidea:

Osoa den n zenbaki bat emanda, lehenak ez diren eta faktore bakarra duten lehenengo n zenbakiz osatutako zerrenda itzuliko duen *erabateko_lehenengoak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
erabateko_lehenengoak :: Integer -> [Integer]

erabateko_lehenengoak n

  | n < 0          = error "Balio negatiboa"
  | otherwise      = genericTake n erabateko_berredurak
```

erabateko_lehenengoak 7 deiak honako zerrenda hau itzuliko luke:
[4, 8, 9, 16, 25, 27, 32]

21. adibidea:

Osoak diren n eta m bi zenbaki emanda, m baino handiagoak izanda, lehenak ez diren eta faktore bakarra duten lehenengo n zenbakiz osatutako zerrenda itzuliko duen *erabateko_handiagoak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
erabateko_handiagoak :: Integer -> Integer -> [Integer]

erabateko_handiagoak n m

  | (n < 0) || (m < 0)      = error "Balio negatiboa"
  | otherwise               = genericTake n [ y | y <- erabateko_berredurak, y > m]
```

erabateko_handiagoak 5 12 deiak honako zerrenda hau itzuliko luke:
[16, 25, 27, 32, 64]

7.11. Adibide gehiago: Murgilketa eta Zerrenda-eraketa

Atal honetan programazioan ezagunak diren algoritmoak Haskell-ez emango dira. Aurkeztutako soluzioetan murgilketaren teknika eta zerrenda-eraketaren teknika erabiltzen dira.

7.11.1. Ordenatze azkarra (quick sort) zerrenda-eraketa erabiliz

Zerrenda-eraketaren teknikaren abantailak erakusteko, *ordenatze azkarra* edo *quick sort* metodoa jarraituz zerrendak ordenatzen dituen funtzioa garatuko dugu.

Ordenatze azkarraren metodoan x deituko diogun zerrendako lehenengo elemetua hartu, eta zerrendako beste elementuak kontsideratuz, ezkerrean x baino txikiagoak edo berdinak diren elementuez osatutako zerrenda, erdian [x] zerrenda eta eskuinaldean x baino handiagoak diren elementuez osatutako zerrenda ipiniko dira. Urrats horren ondoren x ordenatuta dago eta ezkerreko eta eskuineko zerrendak ordenatzea falta da. Bi zerrenda horiek metodo bera erabiliz ordenatuko dira (errekurtsiboki). Ordenatze-metodo hau "zatitu eta garaile izango zara" teknikaren kasu bat da. Jarraian ordenatze azkarra edo quick sort metodoa erabiliz [5, 8, 3, 2, 9, 7] zerrenda nola ordenatuko litzatekeen azalduko da.

	Hasierako zerrenda [5, 8, 3, 2, 9, 7]	
berdinak edo txikiagoak direnak	Lehenengo elementua	handiagoak direnak
[3, 2]	++ [5] ++	[8, 9, 7]

Orain [3, 2] ordenatu behar da:

	Hasierako zerrenda [3, 2]	
berdinak edo txikiagoak direnak	Lehenengo elementua	handiagoak direnak
[2] Elementu bakarra duenez ordenatuta dago	++ [3] ++	[] Hutsa denez ordenatuta dago

Beraz [3, 2] ordenatuta geratu da eta [2] ++ [3] ++ [] zerrenda lortu da, hau da, [2, 3].

Orain [8, 9, 7] ordenatu behar da:

	Hasierako zerrenda [8, 9, 7]	
berdinak edo txikiagoak	Lehenengo elementua	handiagoak direnak

direnak [7] Elementu bakarra duenez ordenatuta dago	++ [8] ++	[9] Elementu bakarra duenez ordenatuta dago

[8, 9, 7] ordenatzea bukatu da eta [7] ++ [8] ++ [9] zerrenda eraiki da, hau da, [7, 8, 9].

Bukaerako emaitza [2, 3] ++ [5] ++ [7, 8, 9] da eta eragiketak burutuz [2, 3, 5, 7, 8, 9] geratzen da.

Zerrenda bat ordenatzeko era hau burutzen duen funtzioa honela definituko genuke:

```
qs :: [Integer] -> [Integer]

qs [] = []
qs (x:s)
        | s == []           = [x]
        | otherwise         = (qs [y | y <- s, y <= x]) ++ [x] ++ (qs [y | y <- s, y > x])
```

qs [5, 8, 3, 2, 9, 7] deiak [2, 3, 5, 7, 8, 9] zerrenda itzuliko du.

Definizio horretan erakutsi den bezala, zerrenda hutsa eta elementu bakarreko zerrendak oinarrizko kasutzat har daitezke eta horrela dei errekurtsibo gutxiago egin beharko dira, baina berez oinarrizko kasu bezala zerrenda hutsa hartzearekin nahikoa da. Hori dela eta, honako definizio hau ere zuzena da:

```
qs2 :: [Integer] -> [Integer]

qs2 [] = []
qs2 (x:s) = (qs2 [y | y <- s, y <= x]) ++ [x] ++ (qs2 [y | y <- s, y > x])
```

7.11.2. Ordenatze azkarra (quick sort) murgilketa erabiliz eraginkortasuna lortzeko (bukaerako errekurtsibitatea)

Ordenatze azkarra edo quick sort metodoa jarraituz zenbaki osozko zerrenda bat ordenatzen duen qs funtzioa definitu da aurreko atalean:

```
qs :: [Integer] -> [Integer]

qs [] = []
qs (x:s)
        | s == []       = [x]
        | otherwise     = (qs [y | y <- s, y <= x]) ++ [x] ++ (qs [y | y <- s, y > x])
```

Oinarrizkoa ez den kasuan, hau da, hirugarren kasuan, funtzio horrek bi dei errekurtsibo sortzen ditu eta, ondorioz, adar ugariko zuhaitz bitarra eratuko da funtzioa exekutatzerakoan. Bukaerako errekurtsibitatea azaltzerakoan esan genuen bezala, adar ugari eratzen dituzten exekuzioak ez dira eraginkorrak izan ohi. Kasu horietan murgilketaren teknika erabiliz, adar bakarra sortzen duen eta gainera bukaerako errekurtsibitatea duen funtzioa definitu ahal izaten da. Bukaerako errekurtsibitateari esker, hostoa sortutakoan emaitza ere kalkulatuta egongo da.

Bukaerako errekurtsibitatea duen adar bakar bat eratuz ordenatze azkarraren metodoa garatzen duen qs_lag funtzioa definituko dugu jarraian murgilketaren teknika erabiliz.

Ordenatu beharreko azpizerrendak gordez joateko, pilaren kontzeptuan oinarrituko gara, nahiz eta gero pila hori zenbaki osozko zerrendez osatutako zerrenda bezala definitu. Gainera, beste parametro bat ere ipini beharko dugu ordenatutako azpizerrendak gordez joateko eta bukaerako zerrenda eraikiz joateko.

[5, 8, 3, 2, 9, 7] zerrenda ordenatzeko, pila zerrenda horrekin hasieratuko genuke:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua ([] balioarekin hasieratuko da)
[5, 8, 3, 2, 9, 7]	[]

Jarraian zerrenda hiru zatitan banatuko da: lehenengo elementua (5) erdian geratuko da, 5 baino handiagoak azpian eta 5 baino txikiagoak edo berdinak direnak gainean.

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[3, 2]	
[5]	
[8, 9, 7]	[]

Beraz orain hiru zerrenda horiek ordenatu behar ditugu.

Gailurrean dagoen zerrenda hartu eta hirutan banatuko dugu: lehenengo elementua (3) erdian geratuko da, 3 baino handiagoak azpian eta 3 baino txikiagoak edo berdinak direnak gainean.

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[2]	
[3]	
[]	
[5]	
[8, 9, 7]	[]

Orain bost zerrenda horiek ordenatu behar dira.

Gailurrean dagoena hartu behar da eta elementu bakarra duenez, ordenatuta dago. Beraz, emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[3]	
[]	
[5]	
[8, 9, 7]	[] ++ [2]

Orain lau zerrenda ditugu ordenatzeko.

Gailurrean dagoena hartu eta elementu bakarra izateagatik badakigu ordenatuta dagoela eta, ondorioz, emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[]	
[5]	
[8, 9, 7]	[] ++ [2] ++ [3]

Ordenatu beharreko hiru zerrenda geratzen zaizkigu.

Gailurrean dagoena hutsa da eta hutsa denez ordenatuta dago eta emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[5] [8, 9, 7]	[] ++ [2] ++ [3] ++ []

Bi zerrenda geratzen zaizkigu ordenatzeko.

Gailurrean dagoena hartu eta elementu bakarra duenez, badakigu ordenatuta dagoela eta zuzenean emaitza eraikitzen ari garen parametroan gorde dezakegu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[8, 9, 7]	[] ++ [2] ++ [3] ++ [] ++ [5]

Orain ordenatu beharreko zerrenda bakarra geratzen zaigu pilan.

Zerrenda bakar hori hartu eta hirutan banatuko dugu: lehenengo elementua (8) erdian geratuko da, 8 baino handiagoak azpian eta 8 baino txikiagoak edo berdinak direnak gainean.

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[7] [8] [9]	[] ++ [2] ++ [3] ++ [] ++ [5]

Berriro hiru zerrenda ditugu pilan. Gailurrean dagoena hartu eta, elementu bakarrekoa denez, ordenatuta dago eta zuzenean emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[8] [9]	[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7]

Orain pilan bi zerrenda daude ordenatzeko.

Gailurrekoa ordenatuta dago elementu bakarra izateagatik eta zuzenean emaitza eraikitzen ari garen parametroan gorde dezakegu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[9]	[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7] ++ [8]

Orain pilan zerrenda bakarra geratzen da.

Zerrenda horrek elementu bakarra du eta ordenatuta dagoenez emaitza eraikitzen ari garen parametroan gorde dezakegu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
	[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7] ++ [8] ++ [9]

Orain pila hutsik dagoenez, [5, 8, 3, 2, 9, 7] ordenatu dugu eta emaitza honako hau da

[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7] ++ [8] ++ [9]

Eragiketak burutu ondoren zerrenda hau geratzen da:

[2, 3, 5, 7, 8, 9]

Pila horretan oinarrituz (baina pilaren ordez zerrendez osatutako zerrenda erabiliz) ordenatze azkarra burutzen duen funtzioa honako hau izango litzateke:

```
qs_lag :: [[Integer]] -> [Integer] -> [Integer]

qs_lag [] ord = ord
qs_lag (x:s) ord
        | (length x == 0) || (length x == 1)          = qs_lag s (ord ++ x)
        | otherwise       = qs_lag ([q1, [head x], q2] ++ s) ord
                                where   q1 = [y | y <- (tail x), y <= (head x)]
                                        q2 = [y | y <- (tail x), y > (head x)]
```

Ordenatze azkarra bukaerako errekurtsibitatearekin burutzen duen funtzioa honako hau izango litzateke:

```
qs_be :: [Integer] -> [Integer]

qs_be r = qs_lag [r] []
```

7.11.3. Nahasketa bidezko ordenazioa (merge sort): murgilketa eraginkortasuna lortzeko (bukaerako errekurtsibitatea) eta zerrenda-eraketa

Adibide honetan definitzen diren funtzioetan murgilketaren teknika eta zerrenda-eraketaren teknika aurki ditzakegu.

Nahasketaren bidezko ordenatze-metodoa honako era honetara azal daiteke:

- Zerrendaren luzera bikoitia baldin bada, luzera bereko bi zatitan banatu zerrenda. Zerrendaren luzera bakoitia baldin bada, zati batean bestean baino elementu bat gehiago ipiniz, bitan banatu zerrenda.
- Nahasketaren metodoa erabiliz bi azpizerrenda horiek ordenatu, bakoitza bere aldetik.
- Bi zerrenda horiek ordenatu ondoren, lortutako bi zerrenda ordenatu horiek nahastuz, ordenatutako zerrenda bakarra eraiki. Ordenatuta dauden bi zerrenda nahasteko, bi zerrendatako lehenengo elementuetatik txikiena hartu beharko da urrats bakoitzean, prozesua ordenatutako zerrenda bakarra lortu arte errepikatuz. Ordenatze-metodo hau ere "zatitu eta garaile izango zara" teknikaren beste kasu bat da.

	Hasierako zerrenda [5, 8, 3, 9, 7, 2]	
Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[5, 8, 3]		[9, 7, 2]

Orain [5, 8, 3] ordenatu behar da:

	Hasierako zerrenda [5, 8, 3]	
Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[5] Elementu bakarra duenez, ordenatuta dago		[8, 3]

Orain [8, 3] ordenatu behar da:

	Hasierako zerrenda [8, 3]	
Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[8] Elementu bakarra duenez, ordenatuta dago		[3] Elementu bakarra duenez, ordenatuta dago
	Nahastu [3, 8]	

Beraz [8, 3] ordenatu da eta [3, 8] geratu da. Horretarako [8] eta [3] zerrendak nahastu dira. Zerrenda horiek nahasteko, 8 eta 3 balioetatik txikiena hartu da eta zerrenda berri bat eraikitzen hasi gara, [3] zerrenda berria geratu zaigularik. Jarraian [8] eta [] zerrendak nahastu dira. Bigarrena hutsa denez, emaitza [8] da eta orain arte eraikitako zerrenda [3] ++ [8] = [3, 8] da.

Jarraian [5] eta [3, 8] zerrendak nahastu behar dira. Bi zerrenda horiek nahasteko 5 eta 3ren artetik txikiena aukeratu behar da eta zerrenda berria eraikitzen hasi beharko da. Kasu honetan zerrenda berri bezala momentuz [3] edukiko dugu. Hurrengo urrats bezala, [5] eta [8] zerrendak nahastu behar dira. Bi zerrenda horiek nahasteko 5 eta 8ren artetik txikiena aukeratu behar da eta aurreko urratsean eraiki dugun [3] zerrendari eskuinetik erantsi beharko diogu. Beraz [3, 5] zerrenda daukagu orain. Bukatzeko, [] eta [8] zerrendak nahastu behar dira. Lehenengoa hutsa denez, [8] zerrenda lortuko da emaitza bezala eta guztira [3, 5] ++ [8] = [3, 5, 8] zerrenda geratuko zaigu.

Jarraian [9, 7, 2] zerrenda ordenatu behar da:

	Hasierako zerrenda [9, 7, 2]	
Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[9] Elementu bakarra duenez, ordenatuta dago		[7, 2]

Orain [7, 2] ordenatuko da:

	Hasierako zerrenda [7, 2]	
Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[7] Elementu bakarra duenez, ordenatuta dago		[2] Elementu bakarra duenez, ordenatuta dago
	Nahastu [2, 7]	

Beraz [7, 2] ordenatu dugu eta [2, 7] geratu zaigu. Horretarako [7] eta [2] zerrendak nahastu behar izan ditugu. Bi zerrenda horiek nahasteko, 7 eta 2 balioetatik txikiena hartu dugu eta zerrenda berri bat eraikitzen hasi gara, [2] zerrenda berria geratu zaigularik. Jarraian [7] eta [] zerrendak nahastu dira. Bigarrena hutsa denez, emaitza [7] da eta orain arte eraikitako zerrenda [2] ++ [7] = [2, 7] da.

Jarraian [9] eta [2, 7] zerrendak nahastu behar ditugu. Bi zerrenda horiek nahasteko 9 eta 2 zenbakietatik txikiena aukeratu behar dugu eta zerrenda berria eraikitzen hasi beharko dugu. Kasu honetan zerrenda berri bezala [2] edukiko dugu lehenengo urrats honen ondoren. Hurrengo urrats bezala, [9] eta [7] zerrendak nahastu behar dira. Bi zerrenda horiek nahasteko 9 eta 7ren artetik txikiena aukeratu behar da eta aurreko urratsean eraiki dugun [2] zerrendari eskuinetik erantsi beharko diogu. Beraz [2, 7]

zerrenda daukagu orain. Bukatzeko, [9] eta [] zerrendak nahastu behar dira. Bigarrena hutsa denez, [9] zerrenda lortuko da emaitza bezala eta guztira [2, 7] ++ [9] = [2, 7, 9] zerrenda geratuko zaigu.

[3, 5, 8] eta [2, 7, 9] zerrenda ordenatuak lortu ondoren, nahastu egin behar dira. Bi zerrenda horiek nahasteko 3 eta 2 balioak hartu eta txikiena aukeratu behar da zerrenda berri bat eraikitzen hasteko. Lehenengo urratsaren ondoren [2] zerrenda geratuko zaigu. Jarraian [3, 5, 8] eta [7, 9] zerrendak nahastuz joan behar dugu. Bi zerrenda horiek nahasteko, 3 eta 7 balioetatik txikiena hartu eta eraikitzen ari garen zerrenda berriari eskuinetik erantsi behar zaio. Beraz [2, 3] lortuko dugu. Hurrengo urrats bezala, [5, 8] eta [7, 9] zerrendak nahastu behar dira. Horretarako 5 eta 7 zenbakietatik txikiena aukeratu eta orain arte eraikita daukagun [2, 3] zerrendari erantsi beharko diogu eskuinetik. Beraz, [2, 3, 5] zerrenda izango dugu orain. Nahasketarekin aurrera jarraituz, [8] eta [7, 9] zerrendak ditugu nahasteko. Urrats honetan [2, 3, 5] zerrendari 8 eta 7ren artetik txikiena erantsi beharko diogu eta, ondorioz, [2, 3, 5, 7] lortuko dugu. Hurrengo urratsean [8] eta [9] nahastea da helburua. Beraz 8 eta 9 zenbakiak hartuz, txikiena [2, 3, 5, 7] zerrendari erantsiko diogu, [2, 3, 5, 7, 8] zerrenda lortuz. Azkeneko urrats bezala, [] eta [9] nahastu behar dira. Lehenengo zerrenda hutsa denez, nahasketaren emaitza [9] da eta guztira eraikitako zerrenda [2, 3, 5, 7, 8] ++ [9] = [2, 3, 5, 7, 8, 9] da.

Haskell erabiliz honela garatu dezakegu metodoa:

- Dagoeneko ordenatuta dauden bi zerrenda nahasten dituen funtzioa definitu: nahastu
- Ordenatu beharreko zerrenda bi zatitan banatu (zati bakoitzak luzera bera izanda edo gehienez zati batek besteak baino elementu bat gehiago izanda), era errekurtsiboan ordenatu zati biak eta, bukatzeko, ordenatutako bi zatiak nahasten dituen funtzioa definitu.

```
nahastu :: [Integer] -> [Integer] -> [Integer]

nahastu [] r = r
nahastu (x:s) r
        | r == [] = (x:s)
        | x <= (head r)   = x : (nahastu s r)
        | otherwise       = (head r) : (nahastu (x:s) (tail r))
```

Dagoeneko ordenatuta dauden bi zerrenda nahastuz ordenatuta dagoen zerrenda berria eraikitzen duen funtzio horrek ez du bukaerako errekurtsibitaterik. Bukaerako errekurtsibitatea izatea nahi izanez gero, emaitza bezala lortuz joanggo den zerrenda gordetzeko balio duen parametro berri bat ipini beharko genuke. Hasteko funtzio laguntzailea definituko dugu eta gero bukaerako errekurtsibitatea duen eta bi zerrenda nahasteko balio duen funtzioa definituko da:

```
nahastu_lag:: [Integer] -> [Integer] -> [Integer] -> [Integer]

nahastu_lag [] r q = q ++ r
nahastu_lag (x:s) r q
        | r == []           = q ++ (x:s)
        | x <= (head r)     = nahastu_lag s r (q ++ [x])
        | otherwise         = nahastu_lag (x:s) (tail r) (q ++ [head r])
```

```
nahastu_be:: [Integer] -> [Integer] -> [Integer]

nahastu_be r w = nahastu_lag r w []
```

Dagoeneko ordenatuta dauden bi zerrenda nahasten dituen funtzioa definitu ondoren (*nahastu* eta *nahastu_be* funtzioek gauza bera egiten dute baina nahastu_be erabiliko dugu bukaerako errekurtsibitatea duelako), nahasketa bidezko ordenazioa edo merge sort metodoa jarraituz zerrenda bat ordenatzen duen funtzioa defini dezakegu:

```
ms :: [Integer] -> [Integer]

ms [] = []
ms (x:s)
        | s == [] = [x]
        | otherwise       = nahastu_be (ms q1) (ms q2)
                                where   q1 = genericTake ((length (x:s)) `div` 2) (x:s)
                                        q2 = genericDrop ((length (x:s)) `div` 2) (x:s)
```

Dei errekurtsiboen bidez, ms funtzioak adar ugari dituen zuhaitz bitar bat eraikiko du. Prozesua xehetasun handiagoarekin aztertuz gero, azkenean elementu bakarreko zerrendak lortzen direla ikus dezakegu. Elementu bakarreko zerrendetara iritsitakoan zerrenda horiek binaka nahasten hasiko da, hasierako zerrendari dagokion zerrenda ordenatua lortu arte. Hori kontuan hartuz, adarkatzea saihesten duen eta gainera bukaerako errekurtsibitatea duen funtzioa defini dezakegu Haskell lengoaiak zerrendekin lana egiteko eskaintzen dizkigun erraztasunak erabiliz:

- Dagoeneko ordenatuta dauden bi zerrenda nahasten dituen funtzioa definitu.
- Dagoeneko ordenatuta dauden zerrendez osatutako zerrenda bat emanda, zerrenda horiek nahastuz joango den ms_lag funtzio laguntzailea definitu. Hasteko, lehenengo eta bigarren zerrendak nahastuko ditu. Nahasketa horretan lortutako zerrenda berria hirugarrenarekin nahastuko du eta abar. Zerrenda bakar batez osatutako zerrenda geratzen denean amaituko da prozesua.
- Ordenatu beharreko zerrenda bat emanda, zerrenda horretako elementuak osagai bakarreko zerrendatan ipiniz eta zerrenda horiek denak zerrenda batean sartuz lortzen den zerrenda ms_lag funtzioari pasatzeaz arduratuko den ms_be funtzioa definitu.

```
ms_lag :: [[Integer]] -> [Integer]

ms_lag [] = []
ms_lag (x:s)
        | s == [] = x
        | otherwise       = ms_lag (q : (tail s))
                                  where   q = nahastu_be x (head s)
```

Nahasketa bidezko ordenatzea bukaerako errekurtsibitatearekin burutzen duen funtzioa honako hau izango litzateke:

```
ms_be :: [Integer] -> [Integer]

ms_be r = ms_lag [[y] | y <- r]
```

7.11.4. Herbeheretar banderaren problema: zerrenda-eraketa erabiliz

(The problem of the Dutch National Flag, "A Discipline of Programming.", Edsger W. Dijkstra, 1975)

Herbeheretar bandera:

Agindu bidezko lengoaietan problema hau honela planteatzen da:

Har dezagun n osagaiko A(1..n) bektore bat eta demagun bektore horretako elementuak hiru multzotan sailka daitezkeela: gorriak, zuriak eta urdinak. Helburua, bektorearen ezkerraldean gorriak diren elementu denak, erdian zuriak eta eskuinaldean urdinak lagatzea da.

Zenbaki osozko bektore bat kontsideratzen badugu, esate baterako gorria izatea zenbaki lehena izatearekin identifika dezakegu, zuria izatea bikoiti ez lehena izatearekin eta urdina izatea bakoiti ez lehena izatearekin.

Beste aukera batzuk ere badaude, adibidez, gorria izatea positiboa ez izatearekin identifika dezakegu (≤ 0), zuria izatea positiboa eta lehena izatearekin eta urdina izatea positiboa bai baina lehena ez izatearekin. Aurreko paragrafoan esandakoa jarraituko dugu guk hemen.

Ada edo Java erako lengoaietan problema honela ebatziko genuke:

```
sgleh := 1; /* Sailkatu gabeko lehenengoaren posizioa */
lehzur := 0; /* Lehenengo zuriaren posizioa */
lehurd := n + 1; /* Lehenengo urdinaren posizioa */

while (sgleh < lehurd) loop
        if gorria(A(sgleh))     /* lehena (A(sgleh)) */
        then    lag := A(sgleh);
                A(sgleh) := A(lehzur);
                A(lehzur) := lag;
                lehzur := lehzur + 1;
                sgleh := sgleh + 1;

        elsif zuria(A(sgleh))   /* bikoitia(A(sgleh)) */
        then    sgleh:= sgleh + 1;

        else    /* urdina(A(sgleh)), hau da, bakoitia(A(sgleh)) */
                lag := A(sgleh);
                A(sgleh) := A(lehurd - 1);
                A(lehurd - 1) := lag;
                lehurd:= lehurd - 1;
        end if;
end loop;
```

Algoritmo horrek orokorrean kolore bereko elementuen ordena alda dezake. Adibidez, (6, 8, 5, 15, 11, 3, 10) bektorea hartzen badugu, algoritmoak (5, 11, 3, 10, 8, 6, 15) bektorea itzuliko luke. Beraz, 6, 8 eta 10en arteko ordena aldatu egin da.

Algoritmo horretan, while-eko edozein bueltatan bektorearen egoera honako hau da:

Gorriak	**Zuriak**	**Sailkatu gabeak**	**Urdinak**
	↑ lehzur	↑ sgleh	↑ lehurd

Kolore bereko elementuen ordena mantentzeko, beste bi while txertatu beharko genituzke algoritmo horretan:

```
sgleh := 1; /* Sailkatu gabeko lehenengoaren posizioa */
lehzur := 0; /* Lehenengo zuriaren posizioa */
lehurd := n + 1; /* Lehenengo urdinaren posizioa */

while (sgleh < lehurd) loop
        if gorria(A(sgleh))     /* lehena (A(sgleh)) */
        then    lag := A(sgleh);
                i := sgleh;
                while i > lehzur loop
                        A(i) := A(i - 1);
```

```
                    i := i - 1;
                end loop;
                A(lehzur) := lag;
                lehzur := lehzur + 1;
                sgleh := sgleh + 1;

        elsif zuria(A(sgleh))   /* bikoitia(A(sgleh)) */
        then    sgleh:= sgleh + 1;

        else    /* urdina(A(sgleh)), hau da, bakoitia(A(sgleh)) */
                lag := A(sgleh);
                i := sgleh
                while i < n loop
                        A(i) := A(i + 1);
                        i := i + 1;
                end loop;
                A(n) := lag;
                lehurd:= lehurd - 1;
        end if;
end loop;
```

Algoritmo berri honek (6, 8, 5, 15, 11, 3, 10) bektorearentzat (5, 11, 3, 6, 8, 10, 15) itzuliko luke, kolore edo talde bereko elementuen arteko ordena errespetatuz.

Zerrenda-eraketaren teknika erabiliz, sailkapen hori burutzen duen funtzioa definitzea errazagoa da Haskell lengoaian. Funtzioari *hbp* deituko diogu. Funtzio horrek kolore edo talde bereko elementuen arteko ordena mantenduko du:

```
hbp :: [Integer] -> [Integer]

hbp [] = []
hbp (x:s) = [y | y <- (x:s), lehena_ze y] ++ [y | y <- (x:s), not (lehena_ze y), y `mod` 2 == 0] ++
                [y | y <- (x:s), not (lehena_ze y), y `mod` 2 /= 0]
```

hbp [6, 8, 5, 15, 11, 3, 10] → [5, 11, 3, 6, 8, 10, 15]

Definizioan ikusten den bezala, hbp funtzioa ez da errekurtsiboa eta horregatik, definizioa hobetu daiteke zerrenda hutsaren eta hutsa ez den zerrendaren kasuak bereiztu gabe. Gainera espresio luzeak daudenean, hobe da where erabiltzea, jarraian azalduko den bezala:

```
hbp2 :: [Integer] -> [Integer]

hbp2 r = gorriak ++ zuriak ++ urdinak
                where   gorriak = [y | y <- r, lehena_ze y]
                        zuriak = [y | y <- r, not (lehena_ze y), y `mod` 2 == 0]
                        urdinak = [y | y <- r, not (lehena_ze y), y `mod` 2 /= 0]
```

Hor r zerrenda hutsa baldin bada, *gorriak*, *zuriak* eta *urdinak* izeneko hiru zerrendak ere hutsak izango dira eta bukaerako emaitza ere zerrenda hutsa izango da.

7.11.5. Herbeheretar banderaren problema: murgilketa erabiliz

Atal honetan, herbeheretar banderaren problema ebazteko beste era bat azalduko da.

Aurreko atalean bezala, har dezagun n osagaiko A(1..n) bektore bat (Haskell-en bektorea eduki beharrean zerrenda bat edukiko dugu) eta demagun bektore horretako elementuak hiru multzotan sailka daitezkeela: gorriak, zuriak eta urdinak. Helburua, bektorearen ezkerraldean gorriak diren elementu denak, erdian zuriak eta eskuinaldean urdinak lagatzea da. Zenbaki osozko bektore bat kontsideratuko dugu eta gorria izatea zenbaki lehena izatearekin identifikatuko dugu, zuria izatea bikoiti ez lehena izatearekin eta urdina izatea bakoiti ez lehena izatearekin.

Intuitiboki, tarteko urrats batean gaudenean egoera honako hau izango da:

Gorriak	**Zuriak**	**Sailkatu gabeak**	**Urdinak**
	↑ lehzur	↑ sgleh	↑ lehurd

Murgilketaren teknika erabiliz, sailkapen hori burutzen duen funtzioa definituko da Haskell lengoaia erabiliz. Funtzioari *hbp_mg* deituko diogu. Funtzio horrek kolore edo talde bereko elementuen arteko ordena mantenduko du. Horretarako une bakoitzean lau zerrenda edukiko dira: oraindik sailkatu gabe dauden elementuez osatutako zerrenda, dagoeneko sailkatuak izan diren elementu gorriez osatutako zerrenda, dagoeneko sailkatuak izan diren elementu zuriez osatutako zerrenda eta dagoeneko sailkatuak izan diren elementu urdinez osatutako zerrenda. Beraz, zerrenda bat edukitzetik lau zerrenda edukitzera pasatuko gara, hau da, parametro bat izatetik lau parametro izatera pasatuko gara. Lau parametro izango dituen funtzio laguntzaileari hbp_lag deituko diogu:

```
hbp_lag :: [Integer] -> [Integer] -> [Integer] -> [Integer] -> [Integer]

hbp_lag [] g z u= g ++ z ++ u
hbp_lag (x:s) g z u
        | lehena_ze x                          = hbp_lag s (g ++ [x]) z u
        | (not (lehena_ze x)) && (x `mod` 2 == 0)  = hbp_lag s g (z ++ [x]) u
        | (not (lehena_ze x)) && (x `mod` 2 /= 0)  = hbp_lag s g z (u ++ [x])
```

Lehenengo parametroa oraindik sailkatu gabe dauden elementuez osatutako zerrenda da eta beste hiru parametroak, hurrenez hurren, dagoeneko sailkatuak izan diren elementu gorriez, zuriez eta urdinez osatutako zerrendak dira.

Bukatzeko, hbp_mg funtzioa hbp_lag funtzioari parametro egokiekin deitzeaz arduratuko da:

```
hbp_mg :: [Integer] -> [Integer]

hbp_mg q = hbp_lag q [] [] []
```

Hor hasieran dagoeneko sailkatuak izan diren elementu gorriez, zuriez eta urdinez osatutako zerrendak hutsak izango dira, oraindik ez delako elementurik sailkatu. Hasieran elementu denak lehenengo parametroan daude, hau da, q zerrendan.

7.12. Sarrera/Irteera

En este apartado vamos a presentar la técnica de generación de listas disponible en Haskell. Dicha técnica puede ser utilizada de manera aislada o en combinación con la recursividad estudiada en los anteriores puntos de este tema.

7.12.1. String datu-mota

Aurredefinitutako String mota [Char] motaren baliokidea edo sinonimoa da. Beraz String motako elementuak Char motako osagaiak dituzten zerrendak dira. Char motako elementuak 'a', '?' edo '7' bezala idazten dira. Char motako zerrenda bat edo String motako elementu bat hiru eratara idatz daiteke:

'A':'z':'a':'r':'o':'a':[]

['A', 'z', 'a', 'r', 'o', 'a']

edo

"Azaroa"

7.12.2. Irteera: putStr

Mezuak pantailan aurkezteko aurredefinitutako putStr funtzioa daukagu. Bere mota honako hau da:

putStr:: String -> IO ()

Beraz, String motako elementu bat edo karaktere-kate bat emanda, IO () motako zerbait itzuliko du. IO () motako zerbait lortuko dela esaterakoan, sarrera/irteerako ekintzaren bat burutuko dela baina emaitzarik ez dela gordeko adierazten da.

Erabiltzeko era:

```
putStr "348"
putStr "Azaroa"
```

1. adibidea:

putStr funtzioa erabiltzen duen kaixo izeneko funtzioaren definizioa dator jarraian. *kaixo* funtzioak beti mezu bera aurkeztuko du.: *Kaixo mundua!!*

```
kaixo:: IO ()

kaixo = putStr "Kaixo mundua!!"
```

kaixo funtzioa erabiltzeko era:

```
kaixo
```

2. adibidea:

Argumentu bezala emandako karaktere-katea aurkeztuko duen *aurkeztu* izeneko funtzioa definituko da jarraian. Funtzio honek putStr funtzioak egiten duen gauza bera egingo du. Beraz ez du ezer berririk egiten eta putStr funtzioari izena aldatzeko bakarrik balio du.

```
aurkeztu:: String -> IO ()

aurkeztu kat = putStr kat
```

Funtzio hau erabiltzeko era:

```
aurkeztu "Azaroa"
aurkeztu "348"
aurkeztu ""
aurkeztu "2014ko azaroa"
```

7.12.3. Irteera lerroz aldatuz: putStrLn

Argumentu bezala emandako karaktere-kate bat aurkeztu eta gero lerroz aldatzeko aurredefinitutako putStrLn funtzioa daukagu. Bere mota honako hau da:

```
putStrLn:: String -> IO ()
```

IO () motako zerbait lortuko dela esaterakoan, sarrera/irteerako ekintzaren bat burutuko dela baina emaitzarik ez dela gordeko adierazten da.

Erabiltzeko era:

```
putStrLn "Azaroa"
putStrLn "348"
putStrLn ""
putStrLn "2014ko azaroa"
```

3. adibidea:

Jarraian, argumentu bezala emandako karaktere-katea aurkeztu eta hurrengo lerrora jauzia egiten duen *aurkeztu_eta_jauzi* izeneko funtzioa definituko da. Funtzio honek putStrLn funtzioak egiten duen gauza bera egingo du. Beraz ez du ezer berririk egiten eta putStrLn funtzioari izena aldatzeko bakarrik balio du.

```
aurkeztu_eta_jauzi:: String -> IO ()

aurkeztu_eta_jauzi kat = putStrLn kat
```

Funtzio hau erabiltzeko era:

```
aurkeztu_eta_jauzi "Azaroa"
```

```
aurkeztu_eta_jauzi "348"
aurkeztu_eta_jauzi ""
aurkeztu_eta_jauzi "2014ko azaroa"
```

7.12.4. Irteera lerroz aldatuz: "\n"

Lerro aldaketa edo lerro jauzia "\n" bezala adieraz daiteke. "2014ko azaroa"++"\n" ++ "Bilbao" katea aurkezten badugu, "2014ko azaroa" eta "Bilbao" karaktere-kateak lerro desberdinetan agertuko dira.

```
aurkeztu "2014ko azaroa"++"\n" ++ "Bilbao"
```

7.12.5. do egitura: agindu-segidak idazteko

Orain "2014ko azaroa" eta "Bilbao" karaktere-kateak bi lerrotan aurkezteko *do* notazioa eta putStrLn funtzioa erabiliko ditugu. Horretarako, *ab* izeneko funtzioa definituko dugu. Funtzio horrek beti "2014ko azaroa" eta "Bilbao" karaktere-kateak bi lerrotan aurkeztuko ditu.

```
ab:: IO ()
ab = do
        putStrLn "2014ko azaroa"
        putStr "Bilbao"
```

IO () motako zerbait lortuko dela esaterakoan, sarrera/irteerako ekintzaren bat burutuko dela baina emaitzarik ez dela gordeko adierazten da.

Sarrera/irteerarekin zerikusia duten ekintzak daudenean, ekintza horiek segida eran ipintzeko do egitura erabiltzen da.

7.12.6. Irteera: show (String motakoak ez diren datuak aurkezteko)

putStr eta putstrLn funtzioek karaktere-kateak aurkezteko balio dute bakarrik. Ondorioz, putStr "348" idazten badugu, 348 aurkeztuko da pantailan, baina putStr 348 idazten badugu, errorea sortuko da. Zer egin dezakegu orduan zenbakizko balio bat aurkezteko? Aurkeztu daitezkeen motetako elementuak karaktere-kate bihurtzeko balio duen *show* funtzio aurredefinitua dauka Haskell-ek. show funtzioaren mota honako hau da:

```
show:: Show t => t -> String
```

Beraz, 348 balioa aurkezteko bi aukera ditugu orain:

```
putStr "348"
```

edo

```
putStr (show 348)
```

Beraz, show funtzioak 348 zenbakia "348" karaktere-katera bihurtzen du.

4. adibidea:

Adibide honetan show erabiltzea nahitaezkoa da:

```
batura_aurkeztu :: Int -> Int -> IO ()
batura_aurkeztu x y = putStr ((show x) ++ " + " ++ (show y) ++ " = " ++ (show (x + y)))
```

Erabiltzeko era:

batura_aurkeztu 10 5

5. adibidea:

Adibide honetan ere show beharrezkoa da. Aurreko funtzioarekin alderatuz, mezu desberdina aurkeztuko da.

```
batura_aurkeztu2 :: Int -> Int -> IO ()
batura_aurkeztu2 x y = putStr ((show x) ++ " gehi " ++ (show y) ++ " " ++ (show (x + y)) ++ " da")
```

Erabiltzeko era:

batura_aurkeztu2 10 5

6. adibidea:

Orain *batura_aurkeztu2* funtzioak egiten duen gauza bera *do* notazioa erabiliz egingo da.

```
batura_aurkeztu3 :: Int -> Int -> IO ()
batura_aurkeztu3 x y = do
                       putStr (show x)
                       putStr " gehi "
                       putStr (show y)
                       putStr " "
                       putStr (show (x + y))
                       putStr " da"
```

Erabiltzeko era:

batura_aurkeztu3 10 5

Sarrera/irteera-ko ekintzak daudenean, ekintza horiek zein ordenetan burutu behar diren adierazteko *do* notazioa erabiltzen da.

7. adibidea:

Hotel bati buruzko informazioa emanda (eraikitze-data, gela-kopurua eta jatetxerik ba al duen ala ez), informazio hori aurkeztuko du jarraian datorren funtzioak do notazioa erabiliz.

```
hotela :: Int -> Int -> Bool -> IO ()
hotela e_data gela_kop jatetxea = do
                                  putStr "Eraikitze-data: "
                                  putStrLn (show e_data)
                                  putStr "Gela-kopurua: "
                                  putStrLn (show gela_kop)
                                  putStr "Jatetxea badu: "
                                  putStrLn (show jatetxea)
```

Erabiltzeko era:

```
hotela 1970 108 True
```

8. adibidea:

hotela2 izeneko funtzioak hotela funtzioak egiten duen gauza bera egiten du, hau da, hotel bati buruzko informazioa emanda (eraikitze-data, gela-kopurua eta jatetxerik ba al duen ala ez), informazio hori aurkeztuko du pantailan do notazioa erabiliz. Baina hirugarren parametroaren kasuan "Bai" edo "Ez" aurkeztuko da True edo False aurkeztu beharrean. Horretarako **if-then-else** egitura erabiliko da. Orain arte funtzioak definitzerakoan | notazioa erabili dugu kasu desberdinak bereizteko, baina batzutan if-then-else egitura hobea da, batez ere do notazioaren barruan.

```
hotela2 :: Int -> Int -> Bool -> IO ()
hotela2 e_data gela_kop jatetxea = do
                                   putStr "Eraikitze-data: "
                                   putStrLn (show e_data)
                                   putStr "Gela-kopurua: "
                                   putStrLn (show gela_kop)
                                   putStr "Jatetxea badu: "
                                   if jatetxea
                                        then putStrLn "Bai"
                                        else putStrLn "Ez"
```

Erabiltzeko era:

```
hotela2 1970 108 True
```

7.12.7. Irteera: print (putStrLn + show)

Aurredefinituta dagoen print funtzioak putStrLn eta show funtzioak batera erabiliz lortzen dena burutzeko balio du. Bere mota honako hau da:

```
print:: Show t => t -> IO ()
```

Esan bezala, putStrLn eta show elkarrekin erabiltzearen baliokidea da. Beraz,

```
putStrLn (show 108)
```

eta

```
print 108
```

gauza bera dira.

9. adibidea:

hotela3 izeneko funtzioak hotela eta hotela2 funtzioek egiten duten gauza bera egiten du, hau da, hotel bati buruzko informazioa emanda (eraikitze-data, gela-kopurua eta jatetxerik ba al duen ala ez), informazio hori aurkeztuko du pantailan do notazioa erabiliz. Baina kasu honetan datu batzuk aurkezteko print funtzioa erabiliko da.
Esta función realiza lo mismo que las dos funciones anteriores, es decir, dados los datos de un hotel (la fecha de construcción, el número de habitaciones y la disponibilidad de restaurante propio) se muestra esa información por pantalla utilizando la notación do.

```
hotela3 :: Int -> Int -> Bool -> IO ()
hotela3 e_data gela_kop jatetxea = do
                                   putStr "Eraikitze-data: "
                                   print e_data
                                   putStr "Gela-kopurua: "
                                   print gela_kop
                                   putStr "Jatetxea badu: "
                                   if jatetxea
                                        then putStrLn "Bai"
                                        else putStrLn "Ez"
```

Erabiltzeko era:

```
hotela3 1970 108 True
```

7.12.8. Sarrera: getLine, <- eragilea eta read eta return funtzioak

Aurredefinitutako getLine funtzioak pantailako lerro oso bat irakurtzen du karaktere-kate bat balitz bezala gordez. getLine funtzioaren mota honako hau da:

```
getLine:: IO String
```

IO String motaren bidez, sarrera/irteerako ekintzaren bat burutu dela eta gero beste funtzioren batek erabili ahal izango duen String motako datu bat gorde dela adierazten da.

10. adibidea:

jaso_eta_aurkeztu funtzioak lerro batean sartzen den testua eskatuko du, jarraian testu hori jasoko du *lerroa* bezala izendatuz eta, bukatzeko, berriro pantailan aurkeztuko du testu hori.

```
jaso_eta_aurkeztu :: IO ()
jaso_eta_aurkeztu = do putStrLn "Lerro batean testua idatzi: "
                       lerroa <- getLine
                       putStrLn ("Hau da jasotako testua: " ++ lerroa)
```

Definizio horretan **<- eragilea** agertzen da. Eragile hori ez da esleipena. Eragile horrek IO t motako elementu bat hartu eta t motako osagaia ateratzen du. Beraz, <- eragileak IO t erako elementu bati IO zatia edo bilgarria kentzen dio.

Hori dela eta, jarraian datorren definizioa ez da zuzena, izan ere bertan <- eragilea esleipena balitz bezala erabiltzen da, baina <- ez da esleipena, IO bilgarria kentzeko balio du.

```
fff :: IO ()
fff = do putStrLn "AAA"
         lerroa <- "asd"
         putStrLn ("Testua hau da: " ++ lerroa)
```

Honako funtzio hau ere ez da zuzena, 4 balioak ez duelako IO bilgarria.

```
fff2 :: IO ()
fff2 = do putStrLn "AAA"
          lerroa <- 4
          putStr ("Testua hau da: ")
          print lerroa
```

Bestalde, **return** funtzioak elementu bat IO bilgarriarekin biltzeko balio du, hau da, t motako elementu bat hartuta, IO t motako elementu bat lortzeko balio du.

return funtzioaren mota honako hau da:

```
return :: t -> IO t
```

11. adibidea:

Adibide honetako ggg funtzioak return funtzioa zertarako den erakusten du.

```
ggg :: IO ()
ggg = do putStrLn "AAA"
         lerroa <- return "asd"
         putStrLn ("Testua hau da: " ++ lerroa)
```

Hor return "asd" zatiaren bidez **IO "asd"** elementua sortzen da. Elementu horri <- eragilea aplikatuz berriro **IO** zatia ezabatzen da. Beraz, lerroa <- return "asd" exekutatu ondoren "asd" karaktere-kateari lerroa izena ematea lortu da. Ondorioz

```
lerroa <- return "asd"
putStrLn ("Testua hau da: " ++ lerroa)
```

eta

```
putStrLn ("Testua hau da: " ++ "asd")
```

gauza bera dira.

12. adibidea:

Adibide honetako ggg2 funtzioan return eta Int motako datu bat agertzen dira.

```
ggg2 :: IO ()
ggg2 = do putStrLn "AAA"
          lerroa <- return 4
          putStrLn ("Testua hau da: ")
          print lerroa
```

Hor return 4 espresioaren bidez **IO 4** elementua sortzen da. Jarraian, elementu horri <- eragilea aplikatuz **IO** zatia edo bilgarria ezabatzen da. Beraz lerroa <- return 4 espresioa exekutatu ondoren, 4 zenbakiari lerroa izena ematea lortu dugu. Ondorioz,

```
lerroa <- return 4
putStrLn ("Testua hau da: ")
print linea
```

eta

```
putStrLn ("Testua hau da: ")
print 4
```

gauza bera dira.

Aurredefinitutako **read** funtzioak karaktere kate bat beste mota bateko elementu bihurtzen du.

13. adibidea:

Adibide honetako lortu_zenb funtzioak zenbaki oso bat eskuratuko du eta zenbaki hori itzuliko du emaitza bezala beste funtzioren batek erabili ahal dezan.

```
lortu_zenb :: IO Int
lortu_zenb = do z <- getLine
                return (read z :: Int)
```

Hor *z <- getLine* espresioaren bidez zenbaki bat lortuko da baina karaktere-kate bat balitz bezala (esate baterako "12") eta karaktere-kate horri z izena emango zaio. Bestalde *read n :: Int* espresioaren bidez z karaktere-kate hori zenbaki oso bihurtuko da (z-ren balio "12" karaktere-katea bada, 12 zenbakia lortuko da). Bukatzeko, *return (read n :: Int)* espresioaren bidez **IO 12** elementua itzuliko da. Beraz, ez da 12 itzultzen, IO 12 itzultzen da.

14. adibidea:

Adibide honetako batu_si funtzioak bi zenbaki oso lortuko ditu lortu_zenb funtzioaren bidez eta bi zenbaki horien batura aurkeztuko du pantailan.

```
batu_si :: IO ()
batu_si = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
             z1 <- lortu_zenb
             z2 <- lortu_zenb
             print (z1 + z2)
```

Hor lortu_zenb funtzioak IO Int erako elementuak lortuko ditu (esate baterako, IO 12). Bestalde *z1 <- lortu_zenb* espresioaren bidez zenbakizko zatiari z1 izena emango zaio, izan ere <- eragileak IO zatia ezabatzeko balio baitu.

15. adibidea:

Adibide honetako batu_si2 funtzioak bi zenbaki oso lortuko ditu lortu_zenb funtzioa erabili gabe eta zenbaki horien batura aurkeztuko du pantailan. Hor lortu_zenb funtzioa ez denez erabiltzen, <- eta getLine erabiliz zenbaki bakoitza karaktere-kate bezala lortuko da eta gero, read erabiliz, kate horiek zenbaki bihurtuko dira.

```
batu_si2 :: IO ()
batu_si2 = do putStrLn " Bi zenbaki idatzi, bakoitza lerro batean: "
              z1 <- getLine
              z2 <- getLine
              let y1 = (read z1 :: Int)
              let y2 = (read z2 :: Int)
              print (y1 + y2)
```

Beraz, *z1 <- getLine* espresioaren bidez lehenengo zenbakia karaktere-kate bat balitz bezala irakurriko da (esate baterako "12") eta kate horri z1 izena emango zaio. Jarraian, *read z1 :: Int* espresioa exekutatutakoan karaktere-kate hori zenbaki oso bihurtuko da (z1-en balioa "12" bada, 12 zenbakia lortuko da). Gainera **let** erabiltzen da zenbaki oso horri y1 izena emateko.

16. adibidea:

Adibide honetako lortu_zerrenda funtzioak zenbaki osozko zerrenda bat eskuratuko du eta zerrenda hori itzuliko du emaitza bezala beste funtzioren batek erabili ahal dezan.

```
lortu_zerrenda :: IO [Int]
lortu_zerrenda = do z <- getLine
                    return (read z :: [Int])
```

Hor *z <- getLine* espresioaren bidez zenbaki osozko zerrenda bat lortuko da baina karaktere-kate bat balitz bezala (esate baterako "[4, 12, 6]") eta karaktere-kate horri z izena emango zaio. Bestalde *read z :: [Int]* espresioaren bidez z karaktere-kate hori zenbaki osozko zerrenda bihurtuko da (z-ren balioa "[4, 12, 6]" karaktere-katea bada, [4, 12, 6] zenbaki osozko zerrenda lortuko da). Bukatzeko, *return (read n :: Int)* espresioaren bidez **IO [4, 12, 6]** elementua itzuliko da. Beraz, ez da [4, 12, 6] itzuliko, IO [4, 12, 6] itzuliko da.

17. adibidea:

Funtzio honek Int motako zerrenda bat lortu eta bertako zenbakien batura aurkeztuko du pantailan:

```
batu_zerrenda :: IO ()
batu_zerrenda = do putStrLn "Zerrenda bat idatzi:"
                   zer <- lortu_zerrenda
                   print (sum zer)
```

7.12.9. Sarrera/irteerako prozesuen errepikapena: errekurtsibitatea

Haskell lengoaia erabiliz datu-eskatzea bere baitan duen prosezu bat errepikatu nahi denean, errekurtsibitatea behar da. Horren erakusle, jarraian datozen adibideak dira.

18. adibidea:

Funtzio honek bi zenbaki oso eskatu, jaso, batura kalkulatu eta aurkeztu eta jarraian prozesua behin eta berriz errepikatuko du amaierarik gabe. Programa bukatzeko konpiladoretik edo Haskell menutik gelditu beharko da. Beraz, alde horretatik ez da oso egokia.

```
batu_si2_sek :: IO ()
batu_si2_sek = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                  z1 <- getLine
                  z2 <- getLine
                  let y1 = (read z1 :: Int)
                  let y2 = (read z2 :: Int)
                  print (y1 + y2)
                 batu_si2_sek
```

19. adibidea:

Adibide honetako funtzioak aurreko adibideko batu_si2_sek funtzioak egiten duen gauza bera egiten du baina emaitza aurkezterakoan mezu desberdina erabiltzen da. Beraz, funtzio honek bi zenbaki oso eskatu, jaso, batura kalkulatu eta aurkeztu eta jarraian prozesua behin eta berriz errepikatuko du amaierarik gabe. Programa bukatzeko konpiladoretik edo Haskell menutik gelditu behgarko da. Beraz, alde horretatik ez da oso egokia.

```
batu_si3_sek :: IO ()
batu_si3_sek = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                  z1 <- getLine
                  z2 <- getLine
                  let y1 = (read z1 :: Int)
                  let y2 = (read z2 :: Int)
                 putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                 print (y1 + y2)
                 batu_si3_sek
```

20. adibidea:

Funtzio honek bi zenbaki oso eskatu, jaso, batura kalkulatu eta aurkeztu eta jarraian prozesua behin eta berriz errepikatuko du erabiltzaileak bukatzea nahi duela esan arte. Beraz programa bukatzeko era egokiagoa da. Hala ere, erabiltzaileari jarraitzea nahi al duen galdetutakoan, erabiltzaileak b (bai) edo e (ez) ez badu erantzuten ere e erantzun duela ulertuko da. Beraz, erantzuna ez da kontrolatzen eta alde horretatik ez da guztiz egokia.

Funtzio honetan erabiltzailearen erantzuna String motako elementu bezala mantentzen da eta horregatik "b" (komatxo bikoitzekin) erantzun al duen aztertuko da.

```
batu_si_sek_buk :: IO ()
batu_si_sek_buk = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                     z1 <- getLine
                     z2 <- getLine
                     let y1 = (read z1 :: Int)
                     let y2 = (read z2 :: Int)
                     putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                     print (y1 + y2)
                     putStrLn "Beste batuketarik? (b/e): "
                     besterik <- getLine
                     if besterik == "b"
                       then batu_si_sek_buk
                       else putStrLn "Bukaera."
```

21. adibidea:

Funtzio honek batu_si_sek_buk funtzioak egiten duen gauza bera egiten du eta erabiltzailearen b edo e erantzunarekin arazo bera dauka. Aldaketa bakarra honako hau da: funtzio honetan erabiltzailearen erantzuna String motako elementu bezala mantendu beharrean, Char motako bezala kontsideratzen da da eta horregatik 'b' (komatxo sinpleekin edo apostrofoarekin) erantzun al duen aztertuko da.

```
batu_si_sek_buk2 :: IO ()
batu_si_sek_buk2 = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                      z1 <- getLine
                      z2 <- getLine
                      let y1 = (read z1 :: Int)
                      let y2 = (read z2 :: Int)
                      putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                      print (y1 + y2)
                      putStrLn "Beste batuketarik? (b/e): "
                      besterik <- getLine
                      let erantzuna = head besterik
                      if erantzuna == 'b'
                        then batu_si_sek_buk2
                        else putStrLn "Bukera."
```

22. adibidea:

Funtzio honek batu_si_sek_buk eta batu_si_sek_buk2 funtzioek egiten dutenaren antzekoa egiten du, baina hemen erabiltzaileak derrigorrez b (bai) edo e (ez) erantzun beharko du, ez da beste erantzunik onartuko.

Funtzio honen beste ezaugarri garrantzitsu bat honako hau da: Erabiltzaileak gutxienez batuketa bat burutu beharko duela.

```
batu_si_sek_buk3 :: IO ()
batu_si_sek_buk3 = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                      z1 <- getLine
                      z2 <- getLine
                      let y1 = (read z1 :: Int)
                      let y2 = (read z2 :: Int)
                      putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                      print (y1 + y2)
                      erantzuna <- erantzuna_eskatu
                      if erantzuna == 'b'
                        then batu_si_sek_buk3
                        else putStrLn "Bukaera."
```

Hor erantzuna_eskatu funtzioak erabiltzaileari b (bai) edo e (ez) erantzuteko eskatzen dio eta erabiltzaileak b edo e erantzun arte errepikatuko du eskatze-prozesua. Ez du beste erantzunik onartuko.

```
erantzuna_eskatu:: IO Char
erantzuna_eskatu = do  putStrLn "beste batuketarik? (b/e): "
                       besterik <- getLine
                       let er = head besterik
                       if er `elem` "be"
                         then (return er)
                         else do
                                 putStrLn "Erantzuna ez da egokia..."
                                 erantzuna_eskatu
```

erantzuna_eskatu funtzioan do bi aldiz erabiltzen da, aginduz osatutako azpisegida bat nola defini daitekeen erakutsiz.

23. adibidea:

Funtzio honek batu_si_sek_buk3 funtzioak egiten duenaren antzekoa egiten du, baina hemen erabiltzaileak bukatzea aukera dezake batuketa bat bera ere egin gabe. Horretarako, batuketarik egin nahi al duen galdetzen zaio erabiltzaileari hasieran.

```
batu_si_sek_buk4 :: IO ()
batu_si_sek_buk4 = do erantzuna <- erantzuna_eskatu2
                      if erantzuna == 'b'
                        then do
                                putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                                z1 <- getLine
                                z2 <- getLine
                                let y1 = (read z1 :: Int)
                                let y2 = (read z2 :: Int)
                                putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                                print (y1 + y2)
                                batu_si_sek_buk4
                        else putStrLn "Bukaera."
```

Hor erantzuna_eskatu2 funtzioak erabiltzaileari b (bai) edo e (ez) erantzuteko eskatzen dio eta erabiltzaileak b edo e erantzun arte errepikatuko du eskatze-prozesua. Funtzio honek erantzuna_eskatu funtzioak egiten duen gauza bera egiten du baina mezu desberdina aurkezten du hasieran, funtzio desberdin batean erabiliko delako.

```
erantzuna_eskatu2:: IO Char
erantzuna_eskatu2 = do putStrLn "Bi zenbaki batzerik nahi al du? (b/e): "
                       besterik <- getLine
                       let er = head besterik
                       if er `elem` "be"
                         then (return er)
                         else do
                                 putStrLn "Erantzuna ez da egokia..."
                                 erantzuna_eskatu2
```

erantzuna_eskatu2 funtzioan ere do bi aldiz erabiltzen da.