

PROGRAMAZIOAREN METODOLOGIA

**KUDEAKETAREN ETA INFORMAZIO SISTEMEN INFORMATIKAREN INGENIARITZAKO
GRADUA**

BILBOKO INGENIARITZA ESKOLA (UPV/EHU)

LENGOAIA ETA SISTEMA INFORMATIKOAK SAILA

1. MAILA

2018-19 IKASTURTEA

6. GAIA

PROGRAMA ERREKURTSIBOEN ERALDAKETA

**BURSTALL-EN METODOA:
ERREKURTSIBOTIK ITERATIBORAKO ERALDAKETA**

José Gaintzarain Ibarmia

Bulegoa: P3I 40

Tutoretza ordutegia: GAUR-en begiratu

AURKIBIDEA

| | |
|---|----|
| 6.1. SARRERA ETA MOTIBAZIOA..... | 5 |
| 6.2. BURSTALL-EN METODOA: OINARRIZKO URRATSAK..... | 7 |
| 6.3. METODOAREN APLIKAZIOA..... | 9 |
| 6.3.1. Oinarrizko funtzio errekursiboak (kasu sinple bat eta kasu errekursibo bat)..... | 9 |
| 6.3.2. Elkarkorra ez den eragiketa duten funtzioak..... | 16 |
| 6.3.3. Kasu sinple bat baino gehiago dituzten funtzioak..... | 22 |
| 6.3.4. Kasu errekursibo bat baino gehiago dituzten funtzioak..... | 29 |
| 6.4. LABURPENA ETA ONDORIOAK..... | 36 |

6.1. SARRERA ETA MOTIBAZIOA

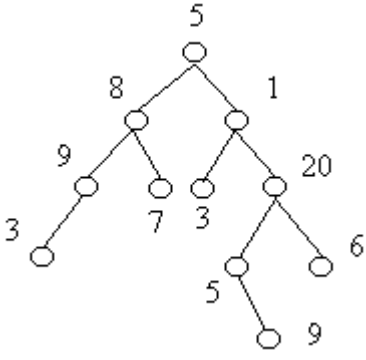
Programazio-problema batzuetan soluzio errazena eta argiena programa errekurtsibo bat izaten da. Esate baterako jarraian aipatzen diren hiru kasuetan hori gertatzen da:

- Definizio matematiko induktiboetan oinarritutako problemak. Adibidez, faktoriala edo fibonacciaren kasuan bezala:

$$\begin{aligned} \text{faktoriala}(0) &= 1 \\ \text{faktoriala}(n) &= n * \text{faktoriala}(n - 1) \end{aligned}$$

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \end{aligned}$$

- Datu-mota errekurtsiboekin zerikusia duten problemak. Ohikoenak diren datu-mota errekurtsiboak zerrendak, pilak eta zuhaitz bitarrak dira.

| | | | | | | |
|---|---|----|---|---|---|--|
| <div>[8, 5, 10, 20, 7]</div> <div>zerrendak</div> | <table><tr><td>22</td></tr><tr><td>4</td></tr><tr><td>9</td></tr><tr><td>8</td></tr></table> <div>pilak</div> | 22 | 4 | 9 | 8 | <div></div> <div>zuhaitz bitarrak</div> |
| 22 | | | | | | |
| 4 | | | | | | |
| 9 | | | | | | |
| 8 | | | | | | |

- Era bereko azpiproblematan zati edo antola daitezkeen problemak. Adibidez, Hanoiko dorreen problema.

Nola mugitu A lekuan dauden hiru kaxak B lekura, aldian kaxa bakarra mugitu daitekeela jakinda eta kaxa baten gainean bera baino handiagoa den beste kaxa bat ezin dela ipini jakinda? Laguntza bezala C lekua erabil daiteke.



Gainera era hauetako problemetan gehienetan soluzio iteratiboa nahiko zaila izaten da eta erroreak agertzeko arriskua handia izaten da. Soluzio errekurtsiboetan aldiz, soluzioa sinplea denez erroreak burutzeko arriskua askoz txikiagoa izaten da.

Baina soluzio errekurtsiboen arazoa eraginkortasun eza da. Soluzio errekurtsiboak sinpleak edo errazak izaten dira baina ez dira eraginkorrak izaten. Soluzio iteratiboetan justu kontrakoa gertatzen da: soluzioak zailagoak izaten dira (bai idazteko eta baita ulertzeko ere) baina askoz eraginkorragoak izaten dira. Horregatik era hauetako problemen aurrean, hasteko soluzio errekurtsibo sinplea idazten da eta gero metodo zehatzak erabiliz soluzio errekurtsibo ez-eraginkor hori eraldatu edo transformatu egiten da eraginkorra eta iteratiboa den programa bat lortuz.

Hasierako soluzio errekurtsiboa zuzena baldin bada eta eraldatze-metodoa ondo aplikatu bada errorerik ez da agertuko.

Zuzenean soluzio iteratibo bat pentsatzea baino errazagoa eta seguruagoa da metodo zehatz bat erabiliz soluzio errekurtsibo bat iteratibo bihurtzea.

Errekurtsibotik iteratiborako eraldaketa egiteko Burstall-en metodoa erabiliko dugu. Metodo honetan hedatze eta tolestatze urratsak emanez eta funtzio errekurtsiboak betetzen duen inbariantea kontuan hartuz burutzen da eraldaketa.

Programa bat diseinatzeko hutsetik hasi eta programa osoa pentsatzen denean, "beheranzko diseinua" jarraitu dela esaten da eta eraldaketaren bidezko programen diseinua "diseinu horizontala" bezala ezagutzen da.

6.2. BURSTALL-N METODOA: OINARRIZKO URRATSAK

Programa errekurtsibo denek erreurrentzia-erlazio bat betetzen dute. Burstall-en metodoan erreurrentzia-erlazio horretatik inbariantea lortzen da eta inbariantea erabiliz programa iteratiboa osatzen duten atal desberdinak kalkulatu dira.

Burstall-en metodoan programa iteratiboak diseinatzeko jarraitzen diren urratsak 6. gaian ikusitako urratsen berdinak dira, baina Burstall-en metodoaren kasuan inbariantea edukitzeaz gain soluzio errekurtsiboa ere izango dugu laguntza bezala eta horrek asko erraztuko du prozesua.

Burstall-en metodoaren bidez jarraian agertzen den eskeman zehazten den erako eraldaketa bat burutuko da:

| Hasierako funtzio errekurtsiboa | | Burstall-en metodoa jarraituz lortutako funtzio iteratiboa |
|--|---|---|
| <u>function</u> $f_re(\bar{x}:T)$ <u>return</u> $\bar{y}:S$ <u>is</u> {Hasierako baldintza} <u>if</u> oinarritzko_kasua(\bar{x}) <u>then</u> $\bar{y} := g(\bar{x})$; <u>else</u> $\bar{y} := h(\bar{x}, f_re(t(\bar{x})))$; <u>end if</u> ; {Bukaerako baldintza} | → | <u>function</u> $f_it(\bar{x}:T)$ <u>return</u> $\bar{y}:S$ <u>is</u> {Hasierako baldintza} Hasieraketak; <u>while</u> {INB} B <u>loop</u> Aginduak <u>end loop</u> ; Bukaerako aginduak; {Bukaerako baldintza} |

{INB} inbariantea da, B iterazioan (while-ean) mantentzeko bete beharreko baldintza, \bar{x} sarrerako datuak eta \bar{y} irteerako emaitzak.

Burstall-en metodoko urratsak honako hauek dira:

- **1. urratsa:** Erreurrentzia-erlazioa planteatu.

Erreurrentzia-erlazioa funtzio errekurtsibotik lortzen da: kasu errekurtsibotik hain zuzen ere.

- **2. urratsa:** Iterazioaren inbariantea lortu.

Iterazioaren inbariantea erreurrentzia-erlazioa orokortuz lortzen da. Erreurrentzia-erlazioa orokortzeko aldagai berriak erabili beharko dira eta aldagai berri horiek gero iterazioan agertuko dira.

Errekurrentzia-erlazioa nola orokortu erabakitzeko, adibide bat garatzea da onena. Adibidea garatzerakoan espresioak hedatu eta tolestatu egin beharko dira behin eta berriz azkeneko emaitza lortu arte. Tolestatze-urratsen ondoren gelditzen diren espresioek errekurrentzia-erlazioa nola orokortu erakutsiko digute: aldatuz doazen elementuak aldagai berriez ordezkatu beharko dira.

- **3. urratsa:** Aldagai berrien hasieraketa.

Iterazioa hasi aurretik inbarianteak egiazkoa izan behar du eta propietate hori hartuko da kontuan aldagai berriak nola hasieratu erabakitzeko. Eraikiko den programa berrian agertuko diren aldagai berri denak inbariantean agertzen dira eta hor ikusiko da aldagai bakoitzak zein balio izan behar duen inbariantea betetzeko.

- **4. urratsa:** Bukaera (baldintza eta bukaerako esleipena)

Urrats honetan iterazioan mantentzeko B baldintza eta while-a bukatu ondoren emaitza zein den zehazteko burutu beharreko agindua kalkulatuko dira.

- Iterazioa kasu sinplea edo oinarritzko kasua betetzen denean bukatuko da. Horregatik, oinarritzko kasuan ez gauden bitartean aurrera jarraitu beharko da while-aren barruan bueltak emanez.

$B \equiv$ kasu sinplea ez betetzea

Normalean $\neg B$ kalkulatzeko errazagoa izaten da eta B kalkulatzeko lehenengo $\neg B$ kalkulatu da.

$\neg B \equiv$ kasu sinplea betetzea

- Bukaerako esleipena funtzio iteratiboak itzuliko duen emaitza zein den zehazteko izaten da. Bukaerako emaitza zein izango den erabakitzeko inbariantea hartu eta kasu sinplea betetzen dela kontsideratu behar da. Horrela lortzen den balioa izango da funtzioak itzuli behar duen emaitza.

- **5. urratsa:** While-aren barruko aginduak.

While-aren barruan joango diren aginduak kalkulatzeko inbariantea hartu eta kasu errekursiboan gaudela kontsideratuz, hedatze-urrats bat eta tolestatze-urrats bat eman beharko dira. Tolestatze-urratsaren ondoren gelditzen den espresioak adieraziko digu nola eguneratu aldagai berriak while-aren barruan.

6.3. METODOAREN APLIKAZIOA

Jarraian Burstall-en metodoa lau kasu desberdinetan nola erabili erakutsiko da:

- A) Oinarrizko funtzio errekurtsiboak (kasu simple bat eta kasu errekurtsibo bat).
- B) Elkarkorra ez den eragiketa duten funtzio errekurtsiboak.
- C) Kasu simple bat baino gehiago dituzten funtzioak.
- D) Kasu errekurtsibo bat baino gehiago dituzten funtzioak.

Kasu desberdinetako ezaugarriak nahastuta dituzten funtzioak ere agertuko zaizkigu ariketetan:

- Elkarkorra ez den eragiketa eta kasu errekurtsibo bat baino gehiago dituzten funtzioak. (B eta D kasuak nahastuta)
- Kasu simple bat baino gehiago eta kasu errekurtsibo bat baino gehiago dituzten funtzioak. (C eta D kasuak nahastuta)

Kasu bat baino gehiagotako ezaugarriak nahastuta dituzten ariketetan kasu horietako bakoitzean esandakoa jarraitu beharko da.

6.3.1. OINARRIZKO FUNTZIO ERREKURTSIBOAK (KASU SIMPLE BAT ETA KASU ERREKURTSIBO BAT)

Kasu honetan 6.2 atalean aipatutako urrats orokorretan esandakoa jarraituko da inolako ezaugarri berezirik gabe.

Osoak diren zenbakiz osatutako zerrenda bat emanda, zerrendako elementu denen batura itzultzen duen *batu* izeneko funtzioa hartu eta metodoa aplikatuko diogu. Gogoratu adibidez $\text{batu}([5, 2, 9]) = 16$ dela.

Ariketetako enuntziatuetan funtzio errekurtsiboa Haskell-ez idatzita etorriko da:

| Funtzio errekurtsiboa Haskell-ez idatzita: | |
|--|----------------------|
| batu: $([Int]) \rightarrow Int$ | |
| <u>Hasierako baldintza</u> : {true} | |
| batu([]) = 0 | ← kasu simplea |
| batu(x:s) = x + batu(s) | ← kasu errekurtsiboa |

Hasierako baldintza 'true' izateak funtzioari sarrerako datutzat emango zaion zerrendak inolako baldintzarik ez duela bete behar adierazten du.

Jarraian funtzio errekursiboa ADA* lengoiaz idatziko da (ADA* lengoia ADA eta Haskell-en arteko nahasketa bat da)

Funtzio errekursiboa ADA*-z idatzita:

```
function batu_re(h: [Int]) return r: Int is
Hasierako baldintza: {true}
if hutsa_da(h) then r := 0;
else r := leh(h) + batu_re(hond(h));
end if;
Bukaerako baldintza: {r = batu(h)}
```

Gogoratu *leh* funtzioak zerrenda bateko lehenengo elementua itzultzen duela eta *hond* funtzioak zerrendako lehenengo elementua kenduz gelditzen den zerrenda itzultzen duela

Haskell-en parametroak patroien bidez adierazi daitezke, hau da, zerrenda hutsa daukagunean zer egin behar den adierazteko parametro gisa [] ipini dezakegu eta hutsa ez den zerrenda daukagunean zer egin behar den adierazteko x:s (edo antzeko zerbait) ipini dezakegu. ADA* lengoian aldiz parametroek beti aldagaiak izan behar dute, ezin dira [] eta x:s erako espresioak ipini parametroen lekuan. Horregatik Haskell-etik ADA*-rako itzulpena egin denean [] eta x:s espresioen ordez h aldagaia ipini da. Hor, h parametroa zerrenda bat da baina ez dakigu hutsa al den ala ez eta horregatik *batu_re* funtzioan *hutsa_da* funtzioa erabili da eta h hutsa ez denean bere elementuak maneiatzeko *leh* eta *hond* funtzioak erabili dira.

Bukaerako baldintzaren bidez *batu_re* funtzioak itzuliko duen r emaitza *batu* funtzioak h zerrendarentzat itzuliko lukeenaren berdina dela adierazten da. Bukaerako baldintzan hori horrela ipini behar da *batu_re* izeneko funtzioa *batu* funtzioaren bertsio errekursiboa delako, ez da funtzio bera.

Burstall-en metodoko bost urratsak aplikatzerakoan ADA* lengoiaz idatzitako bertsioa hartuko dugu kontuan, hau da, *batu_re* funtzioa:

- **1. urratsa:** Errekurrentzia-erlazioa.

Errekurrentzia-erlazioa kasu errekursiboari dagokion funtzioaren definizioa da.

$$\text{batu_re}(h) = \text{leh}(h) + \text{batu_re}(\text{hond}(h))$$

- **2. urratsa:** Inbariantea kalkulatu.

Inbariantea kalkulatzeko errekurrentzia-erlazioa orokortu behar da. Horretarako onena adibide bat garatzea izaten da hedatze eta tolestatze urratsak emanez. Tolestatze-urratsen ondoren gelditzen diren espresioek errekurrentzia-erlazioa nola orokortu erakutsiko digute.

| | |
|----------------------|---|
| batu_re([5, 2, 9]) = | |
| hedatu → | = leh([5, 2, 9]) + batu_re(hond([5, 2, 9])) = |
| tolestatu → | = 5 + batu_re([2, 9]) = |
| hedatu → | = 5 + (leh([2, 9]) + batu_re(hond([2, 9]))) = |
| tolestatu → | = 7 + batu_re([9]) = |
| hedatu → | = 7 + (leh([9]) + batu_re(hond([9]))) = |
| tolestatu → | = 16 + batu_re([]) = |
| oinarrizko kasua → | = 16 + 0 = |
| | = 16 |

Hedatze-urratsean *batu_re* funtzioa bere definizio errekurtsiboa kontuan hartuz ordeztu behar da.

Tolestatze-urratsean espresioa sinplifikatu egin behar da ahal diren eragiketak burutuz.

Garatu dugun adibidean tolestatze-urrats bakoitzaren ondoren zenbaki oso bat eta argumentu bezala zerrenda bat duen *batu_re* funtzioaren arteko batura bat daukagu.

Orain errekurrentzia-erlazioa orokortu behar da. Errekurrentzia-erlazioa orokortzeko tolestatze-urratsen ondoren lortutako espresioak hartu behar dira kontuan. Tolestatze-urratsen ondoren lortutako espresio horietan **aldatuz doazen elementuak aldagai berriez ordezkatu** behar dira.

$$= \underbrace{7}_u + \underbrace{\text{batu_re}([9])}_v =$$

Iterazioaren **inbariantea** honako hau izango da:

$$\text{batu_re}(h) = u + \text{batu_re}(v)$$

Garrantzitsua da inbariantea errekurrentzia-erlazioaren orokorpen bat dela ikustea:

$$\text{batu_re}(h) = \underbrace{\text{leh}(h)}_u + \underbrace{\text{batu_re}(\text{hond}(h))}_v$$

leh(h) espresioaren ordeztu **u** aldagaia ipini da eta **hond(h)** espresioaren ordeztu **v** aldagaia ipini da.

- **3. urratsa:** Aldagai berrien hasieraketa.

Inbariantea lortzerakoan agertu diren aldagai berriak nola hasieratu erabaki behar da urrats honetan.

Hasieran, while-ean lehenengo aldiz sartu aurretik inbarianteak egiazkoa izan behar du eta hori da aldagaiak nola hasieratu erabakitzeko kontuan hartuko duguna. Hasieratu ondoren u eta v aldagaiek honako berdintza hau (inbariantea) bete behar dute:

$$\mathbf{batu_re(h) = u + batu_re(v)}$$

Berdintza hori bete dadin u eta v aldagaiak berdintzaren ezkerreko aldean beraien posizio bera okupatzen duten balioekin hasieratu beharko dira. Kasu honetan ezkerreko aldean u aldagaiaren leku berean ezer ez dagoenez eta batuketarekin ari garenez, 0 balioa dagoela suposatu behar da:

$$\mathbf{0 + batu_re(h) = u + batu_re(v)}$$

Eta hortik u aldagaia 0 balioarekin eta v aldagaia h balioarekin hasieratu beharko ditugula ondoriozta dezakegu:

u := 0;
v := h;

Hasieraketaren ondoren inbariantea bete egiten dela egiazta daiteke:

$$\begin{aligned} \mathbf{batu_re(h)} &= \mathbf{u + batu_re(v)} = \\ &= \mathbf{0 + batu_re(h)} = \\ &= \mathbf{batu_re(h)} \end{aligned}$$

- **4. urratsa:** Bukaera (while-aren baldintza eta bukaerako esleipena).

Urrats honetan while-ean mantentzeko bete behar den B baldintza eta while-a bukatu ondoren burutu beharko den esleipena zein diren erabaki behar da.

- Iterazioan mantentzeko bete beharreko B baldintza kalkulatzeko bide errazena $\neg B$ kalkulatzea da: $\neg B \equiv \text{"kasu sinplea"}$. Baina kasu sinplea formulatzerakoan aldagai berriak erabili behar dira.

batu_re funtzioan kasu sinplea *hutsa_da(h)* da eta 3. urratsean ikusi dugu *h*-ri lotuta doan aldagai berria *v* dela, beraz $\neg B$ idazterakoan *h* erabili beharrean *v* erabili beharko da:

$$\neg B \equiv \text{hutsa_da}(v) \text{ eta } B \equiv \neg \text{hutsa_da}(v)$$

- While-a bukatu ondoren zein esleipen burutu behar den erabakitzeko, hau da, funtzioak emaitza bezala zein balio itzuliko duen erabakitzeko, inbariantea hartu eta kasu sinplean gaudela suposatu beharko da (*hutsa_da(v)*):

$$\begin{aligned} \text{batu_re}(h) &= u + \text{batu_re}(v) = \\ &= u + 0 = \\ &= u \end{aligned}$$

Beraz bukaerako esleipena honako hau izango da:

$$r := u;$$

Gogoratu *r* aldagaia *batu_re* funtzioak itzultzen duen emaitza dela.

- **5. urratsa:** While-aren barruko aginduak

Urrats honetan aldagai berriak while-aren barruan nola eguneratu behar diren kalkulatu behar da.

While-aren barruko aginduak kalkulatzeko *batu_re* funtzioaren kasu errekursiboa hartu behar da kontuan.

Iterazioaren barruan joango diren aginduak kalkulatzeko inbariantea hartu, kasu errekursiboan gaudela kontsideratu eta hedatze-urrats bat eta tolestatze-urrats bat eman beharko dira. Tolestatze-urratsaren ondoren lortzen den espresioak *u* eta *v* aldagaiak nola eguneratu adieraziko digu:

Kasu errekursiboa: $\neg \text{hutsa_da}(v)$

$$\begin{array}{ll}
 \text{batu_re}(h) & = u + \text{batu_re}(v) = \\
 \text{hedatu} \rightarrow & = u + (\text{leh}(v) + \text{batu_re}(\text{hond}(v))) = \\
 \text{tolestatu} \rightarrow & = \underbrace{(u + \text{leh}(v))}_u + \underbrace{\text{batu_re}(\text{hond}(v))}_v
 \end{array}$$

Beraz eguneraketa honako esleipen hauen bidez lortuko da:

```

u := u + leh(v);
v := hond(v);

```

Esleipenen ordena garrantzitsua da hemen. Izan ere *u* eguneratzerakoan *v* ere agertzen da eta ondorioz *u* aldagaia *v* baino lehenago eguneratu beharko da. Honako eguneraketa hau ez legoke ondo:

```


v := hond(v);
u := u + leh(v);


```

Eraiki dugun funtzio iteratibo berria honako hau da:

Funtzio iteratiboa ADA* lengoaiaz idatzita:

function batu_it(h: [Int]) **return** r: Int **is**

Hasierako baldintza $\equiv \{true\}$

u: Int;

v: [Int];

u := 0;

v := h;

while {INV} \neg hutsa_da(v) **loop**

 u := u + leh(v);

 v := hond(v);

end loop;

r = u;

Bukaerako baldintza $\equiv \{r = batu(h)\}$

Bukaerako baldintza horren bidez *batu_it* funtzio iteratiboak lortzen duen r emaitza Haskell-ez idatzitako *batu* funtzioak itzuliko lukeenaren berdina dela adierazten dugu.

6.3.2. ELKARKORRA EZ DEN ERAGIKETA DUTEN FUNTZIOAK

Aurreko adibidean hedatze-urrats bakoitzean zenbaki bat zerrendatik kanpora irtetzen zen eta tolestatze-urratsean zenbaki hori lehendik kanpoan zegoenari batuz zenbaki berri bakar bat gelditzen zitzaigun kanpoan. Hori horrela izan da batuketa elkarkorra delako, hau da, $a + (b + c) = (a + b) + c$ delako:

| | |
|----------------------|---|
| batu_re([5, 2, 9]) = | |
| hedatu → | = leh([5, 2, 9]) + batu_re(hond([5, 2, 9])) = |
| tolestatu → | = 5 + batu_re([2, 9]) = |
| hedatu → | = 5 + (leh([2, 9]) + batu_re(hond([2, 9]))) = |
| | = 5 + (2 + batu_re(hond([2, 9]))) = |
| + elkarkorra delako | = (5 + 2) + batu_re(hond([2, 9])) = |
| tolestatu → | = 7 + batu_re([9]) = |
| hedatu → | = 7 + (leh([9]) + batu_re(hond([9]))) = |
| | = 7 + (9 + batu_re(hond([9]))) = |
| + elkarkorra delako | = (7 + 9) + batu_re(hond([9])) = |
| tolestatu → | = 16 + batu_re([]) = |
| kasu simplea → | = 16 + 0 = |
| | = 16 |

7.3.2 atalean garatuko dugun adibidean ikusiko dugun bezala, espresioan agertzen den eragiketa elkarkorra ez bada, elkarkorra den eragiketa batez ordezkatu beharko da.

Guk egingo ditugun ariketetan elkarkorra ez den eragile bakarra agertuko da, zerrenda batean elementu berri bat ezkerretik sartzeko balio duen ':' eragilea hain zuzen ere. Eragile hori agertzen denean bi zerrenda elkartzeko balio duen '++' eragileaz ordezkatu beharko da. '++' eragilea elkarkorra da.

$$([5, 7] ++ [6, 1, 8]) ++ [9, 3] = [5, 7] ++ ([6, 1, 8] ++ [9, 3])$$

$$\text{baina } 5:(7:(9:[])) \neq (5:7):(9:[])$$

gainera (5:7):(9:[]) espresioa ez da espresio zuzena.

Osoak diren zenbakiz osatutako zerrenda bat emanda, zerrendako elementu bakoitzari 1 gehituz lortzen den zerrenda itzultzen duen *gehitu* funtzioa kontsideratuko dugu.

Gogoratu: $\text{gehitu}([5, 2]) = [6, 3]$.

Funtzio errekurtsiboa Haskell-ez idatzita:

$\text{gehitu}: ([\text{Int}]) \rightarrow [\text{Int}]$

Hasierako baldintza: {true}

$\text{gehitu}([]) = []$

← kasu simplea

$\text{gehitu}(x:s) = (x + 1) : \text{gehitu}(s)$

← kasu errekurtsiboa

Hasierako baldintza 'true' izateak funtzioari sarrerako datu gisa emango zaion zerrendak inolako baldintzarik ez duela bete behar adierazten du.

Jarraian funtzio errekursiboa ADA* lengoaiaz idatziko da (ADA* lengoiaia ADA eta Haskell-en arteko nahasketa bat da)

Funtzio errekursiboa ADA* lengoaiaz:

function gehitu_re(h: integer) **return** r: [Int] **is**
Hasierako baldintza: {true}

if hutsa_da(h) **then** r := [];
else r := (leh(h) + 1) : gehitu_re(hond(h));
end if;

Bukaerako baldintza: {r = gehitu(h)}

Haskell-en parametroak patroien bidez adierazi daitezke, hau da, zerrenda hutsa daukagunean zer egin behar den adierazteko parametro bezala [] ipini dezakegu eta hutsa ez den zerrenda daukagunean zer egin behar den adierazteko x:s (edo antzeko zerbait) ipini dezakegu. ADA* lengoaiaren aldiz parametroek beti aldagaiak izan behar dute, ezin dira [] eta x:s erako espresioak ipini parametroen lekuan. Horregatik Haskell-etik ADA*-rako itzulpena egin denean [] eta x:s espresioen ordean h aldagaia ipini da. Hor h parametroa zerrenda bat da baina ez dakigu hutsa al den ala ez eta horregatik *gehitu_re* funtzioan *hutsa_da* funtzioa erabili da eta h hutsa ez denean bere elementuak maneiatzeko *leh* eta *hond* funtzioak erabili dira.

Bukaerako baldintzaren bidez *gehitu_re* funtzioak itzuliko duen r emaitza *gehitu* funtzioak h zerrendarentzat itzuliko lukeenaren berdina dela adierazten da. Bukaerako baldintzan hori horrela ipini behar da *gehitu_re* izeneko funtzioa *gehitu* funtzioaren bertsio errekursiboa delako, ez da funtzio bera. Gauza bera egiten dute baina bi funtzio desberdin dira.

Burstall-en metodoko bost urratsak aplikatzerakoan ADA* lengoaiaz idatzitako bertsioa hartuko dugu kontuan, hau da, *gehitu_re* funtzioa:

- **1. urratsa:** Errekurrentzia-erlazioa.

Errekurrentzia-erlazioa kasu errekursiboari dagokion funtzioaren definizioa da.
$$\text{gehitu_re}(h) = (\text{leh}(h) + 1) : \text{gehitu_re}(\text{hond}(h))$$

- **2. urratsa:** Inbariantea kalkulatu.

Inbariantea kalkulatzeko errekurrentzia-erlazioa orokortu behar da. Horretarako onena adibide bat garatzea izaten da hedatze eta tolestatze urratsak emanaz. Tolestatze-uratsen ondoren gelditzen diren espresioek errekurrentzia-erlazioa nola orokortu erakutsiko digute.

| | |
|----------------------------|---|
| gehitu_re([6, -1, 8, 0]) = | |
| hedatu → | = (leh([6, -1, 8, 0]) + 1) : gehitu_re(hond([6, -1, 8, 0])) = |
| tolestatu → | = 7 : gehitu_re([-1, 8, 0]) = |
| hedatu → | = 7 : ((leh([-1, 8, 0]) + 1) : gehitu_re(hond([-1, 8, 0]))) = |
| tolestatu → | = 7 : (0 : gehitu_re([8, 0])) = |
| | $\underbrace{\hspace{10em}}$ |
| | '.' eragilea mantenduz ezin dira elkartu |
| hedatu → | = 7 : (0 : ((leh([8, 0]) + 1) : gehitu_re(hond([8, 0])))) = |
| tolestatu → | = 7 : (0 : (9 : gehitu_re([0]))) = |
| | $\underbrace{\hspace{10em}}$ |
| | '.' eragilea mantenduz ezin dira elkartu |
| hedatu → | = 7 : (0 : (9 : ((leh([0]) + 1) : gehitu_re(hond([0]))))) = |
| tolestatu → | = 7 : (0 : (9 : (1 : gehitu_re([])))) = |
| | $\underbrace{\hspace{10em}}$ |
| | '.' eragilea mantenduz ezin dira elkartu |
| kasu simplea → | = 7 : (0 : (9 : (1 : []))) = |
| | = [7, 0, 9, 1] |

'.' eragilea elkarkorra ez denez, bateratze-urratsetan lortu diren espresioek egitura desberdina dute eta ez dute orokorpenik onartzen, ezin direlako elementuak elkartu. Soluzioa '.' eragilearen ordezkari '+' eragilea erabiltzea da. Beraz, '.' eragilea agertzen denean '+' eragileaz ordezkatzeko dugu tolestatze-urratsa ematean:

| | |
|----------------------------|--|
| gehitu_re([6, -1, 8, 0]) = | |
| hedatu → | = (leh([6, -1, 8, 0]) + 1) : gehitu_re(hond([6, -1, 8, 0])) = |
| tolestatu → | = [7] ++ gehitu_re([-1, 8, 0]) = |
| hedatu → | = [7] ++ ((leh([-1, 8, 0]) + 1) : gehitu_re(hond([-1, 8, 0]))) = |
| tolestatu → | = [7, 0] ++ gehitu_re([8, 0]) = |
| hedatu → | = [7, 0] ++ ((leh([8, 0]) + 1) : gehitu_re(hond([8, 0]))) = |
| tolestatu → | = [7, 0, 9] ++ gehitu_re([0]) = |
| hedatu → | = [7, 0, 9] ++ ((leh([0]) + 1) : gehitu_re(hond([0]))) = |
| tolestatu → | = [7, 0, 9, 1] ++ gehitu_re([]) = |
| kasu simplea → | = [7, 0, 9, 1] ++ [] = |
| | = [7, 0, 9, 1] |

Hedatze-urratsean *gehitu_re* funtzioa bere definizio errekursiboa kontuan hartuz ordeztu behar da.

Tolestatze-urratsean espresioa sinplifikatu egin behar da ahal diren eragiketarik burutuz.

Adibide honetan tolestatze-urratsetan '.' kendu eta bere ordezkari '+' erabili da.

Garatu dugun adibidean tolestatze-urrats bakoitzaren ondoren zerrenda bat eta *gehitu_re* funtzioari egindako dei bat elkartzen dira '++' eragilearen bidez. Eta gainera funtzioak argumentu bezala beste zerrenda bat du.

Orain errekurrentzia-erlazioa orokortu behar da. Errekurrentzia-erlazioa orokortzeko tolestatze-urratsen ondoren lortutako espresioak hartu behar dira kontuan. Tolestatze-urratsen ondoren lortutako espresio horietan **aldatuz doazen elementuak aldagai berriez ordezkatu** behar dira. Gure adibidean bai kanpoko zerrenda eta bai barrukoa aldatuz doaz.

$$= \underbrace{[7, 0]}_u ++ \underbrace{\text{gehitu_re}([8, 0])}_v =$$

Iterazioaren **inbariantea** honako hau izango da:

$$\text{gehitu_re}(h) = u ++ \text{gehitu_re}(v)$$

Garrantzitsua da inbariantea errekurrentzia-erlazioaren orokorpen bat dela ikustea:

$$\text{gehitu_re}(h) = \underbrace{(\text{leh}(h) + 1)}_u : \underbrace{\text{gehitu_re}(\text{hond}(h))}_v$$

'**(leh(h) + 1) :**' espresioaren ordezkari **'u ++'** ipini da eta **hond(h)** espresioaren ordezkari **v** aldagaia ipini da.

- **3. urratsa:** Aldagai berrien hasieraketa.

Inbariantea lortzerakoan agertu diren aldagai berriak nola hasieratu erabaki behar da urrats honetan.

Hasieran, while-ean lehenengo aldiz sartu aurretik, inbarianteak egiazkoa izan behar du eta hori da aldagaiak nola hasieratu erabakitzeko kontuan hartuko duguna. Hasieratu ondoren u eta v aldagaiek honako berdintza hau bete behar dute:

$$\text{gehitu_re}(h) = u ++ \text{gehitu_re}(v)$$

Berdintza hori bete dadin u eta v aldagaiak berdintzaren ezkerreko aldean beraien posizio bera okupatzen duten balioekin hasieratu beharko dira. Kasu honetan ezkerreko aldean u aldagaiaren leku berean ezer ez dagoenez eta zerrendak elkartzen ari garenez, [] balioa dagoela suposatu behar da:

$$\begin{array}{c} \downarrow \\ [] ++ \text{gehitu_re}(h) = u ++ \text{gehitu_re}(v) \\ \uparrow \end{array}$$

Eta hortik u aldagaia $[]$ balioarekin eta v aldagaia h balioarekin hasieratu beharko ditugula ondoriozta dezakegu:

$$\begin{aligned} u &:= []; \\ v &:= h; \end{aligned}$$

Hasieraketaren ondoren inbariantea bete egiten dela egiazta daiteke:

$$\begin{aligned} \text{gehitu_re}(h) &= u ++ \text{gehitu_re}(v) = \\ &= [] ++ \text{gehitu_re}(h) = \\ &= \text{gehitu_re}(h) \end{aligned}$$

- **4. urratsa:** Bukaera (while-aren baldintza eta bukaerako esleipena).

Urrats honetan while-ean mantentzeko bete behar den B baldintza eta while-a bukatu ondoren burutu beharko den esleipena zein diren erabaki behar da.

- Iterazioan mantentzeko bete beharreko B baldintza kalkulatzeko bide errazena $\neg B$ kalkulatzeko da: $\neg B \equiv \text{"kasu sinplea"}$. Baina kasu sinplea formulatzerakoan aldagai berriak erabili behar dira:

gehitu_re funtzioan kasu sinplea $\text{hutsa_da}(h)$ da eta 3. urratsean ikusi dugu h -ri lotuta doan aldagai berria v dela, beraz $\neg B$ idazterakoan h erabili beharrean v erabili beharko da

$$\begin{aligned} \neg B &\equiv \text{hutsa_da}(v) \text{ eta } B \equiv \neg \text{hutsa_da}(v), \text{ hau da,} \\ B &\equiv \text{not}(\text{hutsa_da}(v)) \end{aligned}$$

- While-a bukatu ondoren zein esleipen burutu behar den erabakitzeko, hau da, funtzioak emaitza bezala zein balio itzuliko duen erabakitzeko, inbariantea hartu eta kasu sinplean gaudela suposatu beharko da ($\text{hutsa_da}(v)$):

$$\begin{aligned} \text{gehitu_re}(h) &= u ++ \text{gehitu_re}(v) = \\ &= u ++ [] \\ &= u \end{aligned}$$

Beraz bukaerako esleipena honako hau izango da:

$$r := u;$$

Gogoratu r aldagaia gehitu_re funtzioak itzultzen duen emaitza dela.

- **5. urratsa:** While-aren barruko aginduak

Urrats honetan aldagai berriak while-aren barruan nola eguneratu behar diren kalkulatu behar da.

While-aren barruko aginduak kalkulatzeko gehitu_re funtzioaren kasu errekurtsiboa hartu behar da kontuan.

Iterazioaren barruan joango diren aginduak kalkulatzeko inbariantea hartu, kasu errekursiboan gaudela kontsideratu eta hedatze-urrats bat eta tolestatze-urrats bat eman beharko dira. Tolestatze-urratsaren ondoren lortzen den espresioak u eta v aldagaiak nola eguneratu adieraziko digu:

Kasu errekursiboa: $\neg \text{hutsa_da}(v)$

| | |
|-------------------------|---|
| gehitu_re(h) | = $u ++$ gehitu_re(v) = |
| hedatu \rightarrow | = $u ++ ((\text{leh}(v) + 1) : \text{gehitu_re}(\text{hond}(v)))$ |
| tolestatu \rightarrow | = $(u ++ [(\text{leh}(v) + 1)]) ++ \text{gehitu_re}(\text{hond}(v))$ |

$\underbrace{\hspace{10em}}$
 u

$\underbrace{\hspace{10em}}$
 v

Beraz eguneraketa honako esleipen hauen bidez lortuko da:

$u := u ++ [(\text{leh}(v) + 1)];$
 $v := \text{hond}(v);$

Esleipenen ordena garrantzitsua da hemen. Izan ere u eguneratzerakoan v ere agertzen da eta ondorioz u aldagaia v baino lehenago eguneratu beharko da. Honako eguneraketa hau ez legoke ondo:

~~**$v := \text{hond}(v);$**~~
 ~~**$u := u ++ [(\text{leh}(v) + 1)];$**~~

Eraiki dugun funtzio iteratibo berria honako hau da:

Funtzio iteratiboa ADA* lengoaiatz:

function gehitu_it(h: [Int]) **return** r: [Int] **is**

Hasierako baldintza $\equiv \{\text{true}\}$

$u, v: [\text{Int}];$

$u := [];$

$v := h;$

while {INV} **not** hutsa_da(v) **loop**

$u := u ++ [(\text{leh}(v) + 1)];$

$v := \text{hond}(v);$

end loop;

$r = u;$

Bukaerako baldintza $\equiv \{r = \text{gehitu}(h)\}$

Bukaerako baldintza horren bidez *gehitu_it* funtzio iteratiboak lortzen duen r emaitza Haskell-ez idatzitako *gehitu* funtzioak itzuliko lukeenaren berdina dela adierazten dugu.

6.3.3. KASU SINPLE BAT BAINO GEHIAGO DITUZTEN FUNTZIOAK

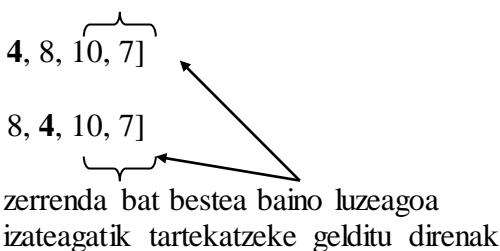
Kasu sinple bat baino gehiago egoteak metodoaren 4. urratsean izango du eragina, bai `while`-aren baldintza kalkulatzekoan eta baita `while`-a amaitu ondoren funtzioaren emaitza zein izango den erabakitzerakoan ere.

Zenbaki osoz osatutako bi zerrenda emanda, zerrenda horietako elementuak tartekatuz osatzen den zerrenda berria itzultzen duen *tartekatu* izeneko funtzioa kontsideratuko dugu. Zerrendetako bat bestea baino luzeagoa denean, tartekatu ezinda gelditzen diren elementuak bukaeran ipiniko dira denak jarraian.

Adibidea:

$$\text{tartekatu}([2, 4], [5, 8, 10, 7]) = [2, 5, 4, 8, 10, 7]$$

$$\text{tartekatu}([5, 8, 10, 7], [2, 4]) = [5, 2, 8, 4, 10, 7]$$



zerrenda bat bestea baino luzeagoa
izateagatik tartekatzeke gelditu direnak

Ariketetako enuntziatuetan funtzio errekursiboa Haskell-ez idatzita etorriko da:

Funtzio errekursiboa Haskell-ez idatzita:

```

tartekatu: ([Int], [Int]) → [Int]
Hasierako baldintza: {true}
tartekatu([], s) = s
tartekatu(x : r, s)
    | hutsa_da(s)           = x : r
    | otherwise             = [x] ++ [leh(s)] ++ tartekatu(r, hond(s))

```

Hasierako baldintza 'true' izateak funtzioari sarrerako datu gisa emango zaizkion zerrendek inolako baldintzarik ez dutela bete behar adierazten du.

Jarraian funtzio errekursiboa ADA* lengoaiaz idatziko da (ADA* lengoia ADA eta Haskell-en arteko nahasketa bat da):

Funtzio errekursiboa ADA*-z idatzita:

```

function tartekatu_re(h, s: [Int]) return q: [Int] is
Hasierako baldintza ≡ {true}
if hutsa_da(h) then q := s;
elseif hutsa_da(s) then q := h;
else q := [leh(h)] ++ [leh(s)] ++ tartekatu_re(hond(h), hond(s));
end if;
Bukaerako baldintza ≡ {q = tartekatu(h, s)}

```

Haskell-en parametroak patroien bidez adierazi daitezke, hau da, zerrenda hutsa daukagunean zer egin behar den adierazteko parametro bezala [] ipini dezakegu eta hutsa ez den zerrenda daukagunean zer egin behar den adierazteko xr (edo antzeko zerbait) ipini dezakegu. ADA* lengoaiaren aldiz parametroek beti aldagaiak izan behar dute, ezin dira [] eta xr erako espresioak ipini parametroen lekuan. Horregatik Haskell-etik ADA*-rako itzulpena egin denean [] eta xr espresioen ordeztu h aldagaia ipini da. Hor h parametroa zerrenda bat da baina ez dakigu hutsa al den ala ez eta horregatik *tartekatu_re* funtzioan *hutsa_da* funtzioa erabili da eta h hutsa ez denean bere elementuak maneiatzeko *leh* eta *hond* funtzioak erabili dira.

Bukaerako baldintzaren bidez *tartekatu_re* funtzioak itzuliko duen q emaitza *tartekatu* funtzioak h eta s zerrendentzat itzuliko lukeenaren berdina dela adierazten da. Bukaerako baldintzan hori horrela ipini behar da *tartekatu_re* izeneko funtzioa *tartekatu* funtzioaren bertsio errekurtsiboa delako, ez da funtzio bera. Gauza bera egiten dute baina bi funtzio desberdin dira.

Burstall-en metodoko bost urratsak aplikatzerakoan ADA* lengoaiatzat idatzitako bertsioa hartuko dugu kontuan, hau da, *tartekatu_re* funtzioa:

- **1. urratsa:** Errekurrentzia-erlazioa.

Errekurrentzia-erlazioa kasu errekurtsiboari dagokion funtzioaren definizioa da.

```
tartekatu_re(h, s) = [leh(h)] ++ [leh(s)]
                  ++ tartekatu_re(hond(h), hond(s))
```

- **2. urratsa:** Inbariantea kalkulatu.

Inbariantea kalkulatzeko errekurrentzia-erlazioa orokortu behar da. Horretarako onena adibide bat garatzea izaten da hedatze eta tolestatze urratsak emanaz. Tolestatze-urratsen ondoren gelditzen diren espresioek errekurrentzia-erlazioa nola orokortu erakutsiko digute.

```
tartekatu_re([8, 0], [3, 9, 4, 7]) =
hedatu →      = leh([8, 0]) ++ leh([3, 9, 4, 7]) ++
                tartekatu_re(hond([8, 0]), hond([3, 9, 4, 7])) =
bateratu →    = [8, 3] ++ tartekatu_re([0], [9, 4, 7]) =
hedatu →      = [8, 3] ++ (leh([0]) ++ leh([9, 4, 7]) ++
                tartekatu_re(hond([0]), hond([9, 4, 7]))) =
bateratu →    = [8, 3, 0, 9] ++ tartekatu_re([], [4, 7]) =
kasu simplea → = [8, 3, 0, 9] ++ [4, 7] =
                = [8, 3, 0, 9, 4, 7]
```

Hedatze-urratsean *tartekatu_re* funtzioa bere definizio errekurtsiboa kontuan hartuz ordeztu behar da.

Tolestatze-urratsean espresioa sinplifikatu egin behar da ahal diren eragiketak burutuz.

Garatu dugun adibidean tolestatze-urrats bakoitzaren ondoren zerrenda batez eta argumentu bezala bi zerrenda dituen *tartekatu_re* funtzioaz osatutako espresio bat daukagu.

Tolestatze-urratsen ondoren gelditzen diren espresioetan kanpoko zerrenda eta *tartekatu_re* funtzioaren argumentu diren zerrenda biak aldatuz doazela ikus daiteke.

Orain errekurrentzia-erlazioa orokortu behar da. Errekurrentzia-erlazioa orokortzeko tolestatze-urratsen ondoren lortutako espresioak hartu behar dira kontuan. Tolestatze-urratsen ondoren lortutako espresio horietan **aldatuz doazen elementuak aldagai berriez ordezkatu** behar dira:

$$= \underbrace{[8, 3, 0, 9]}_u ++ \text{tartekatu_re}(\underbrace{[]}_v, \underbrace{[4, 7]}_w) =$$

Iterazioaren **inbariantea** honako hau izango da:

$$\text{tartekatu_re}(h, s) = u ++ \text{tartekatu_re}(v, w)$$

Garrantzitsua da inbariantea errekurrentzia-erlazioaren orokorpen bat dela ikustea:

$$\text{tartekatu_re}(h, s) = \underbrace{[le(h)] ++ [le(s)]}_u ++ \text{tartekatu_re}(\underbrace{\text{hond}(h)}_v, \underbrace{\text{hond}(s)}_w)$$

[le(h)] ++ [le(s)] espresioaren ordezkari **u** aldagaia ipini da, **hond(h)** espresioaren ordezkari **v** aldagaia ipini da eta **hond(s)** espresioaren ordezkari **w** aldagaia ipini da.

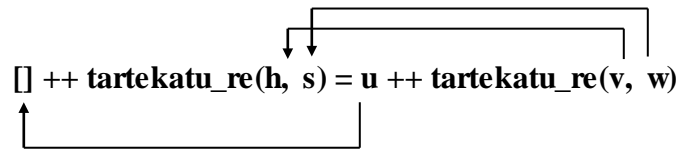
- **3. urratsa:** Aldagai berrien hasieraketa.

Inbariantea lortzerakoan agertu diren aldagai berriak nola hasieratu erabaki behar da urrats honetan.

Hasieran, while-ean lehenengo aldiz sartu aurretik inbarianteak egiazkoa izan behar du eta hori da aldagaiak nola hasieratu erabakitzeko kontuan hartuko duguna. Hasieratu ondoren u, v eta w aldagaiek honako berdintza hau bete behar dute:

$$\text{tartekatu_re}(h, s) = u ++ \text{tartekatu_re}(v, w)$$

Berdintza hori bete dadin u , v eta w aldagaiak berdintzaren ezkerreko aldean beraien posizio bera okupatzen duten balioekin hasieratu beharko dira. Kasu honetan ezkerreko aldean u aldagaiaren leku berean ezer ez dagoenez eta zerrendak elkartzen ari garenez, $[]$ balioa dagoela pentsatu behar da:



Eta hortik u aldagaia $[]$ balioarekin, v aldagaia h balioarekin eta w aldagaia s balioarekin hasieratu beharko ditugula ondoriozta dezakegu:

$u := [];$
 $v := h;$
 $w := s;$

Hasieraketaren ondoren inbariantea bete egiten dela egiazta daiteke:

$\text{tartekatu_re}(h, s) = u ++ \text{tartekatu_re}(v, w) =$
 $= [] ++ \text{tartekatu_re}(h, s) =$
 $= \text{tartekatu_re}(h, s)$

- **4. urratsa:** Bukaera (while-aren baldintza eta bukaerako esleipena).

Urrats honetan while-ean mantentzeko bete behar den B baldintza eta while-a bukatu ondoren burutu beharko den esleipena zein diren erabaki behar da.

- Iterazioan mantentzeko bete beharreko B baldintza kalkulatzeko bide errazena $\neg B$ kalkulatzeko da: $\neg B \equiv \text{"kasu simplea"}$. Baina orain **bi kasu simple ditugu** eta $\neg B$ kasu sinpleen **disjuntzioa** izango da. Gogoratu gainera kasu sinpleak aldagai berriak erabiliz formulatu behar direla:

$\neg B \equiv \text{hutsa_da}(v) \vee \text{hutsa_da}(w)$
eta ondorioz,

$B \equiv \neg(\text{hutsa_da}(v) \vee \text{hutsa_da}(w))$

eta ADA* programazio lengoia jarraituz hori honela idatziko genuke

$B \equiv \underline{\text{not}} (\text{hutsa_da}(v) \underline{\text{or}} \text{hutsa_da}(w))$, edo nahi izanez gero $\underline{\text{not}} (\text{hutsa_da}(v)) \underline{\text{and}} \underline{\text{not}} (\text{hutsa_da}(w))$ bezala ere idatz daiteke.

- While-a bukatu ondoren zein esleipen burutu behar den erabakitzeke, hau da, funtzioak emaitza bezala zein balio itzuliko duen erabakitzeke, inbariantea hartu eta kasu sinplean gaudela suposatu beharko da. Bi kasu sinple daudenez, kasu bakoitza bere aldetik aztertu beharko da eta kasu bakoitzean emaitza desberdina izango da. Bi kasu sinple daudenez eta while-a kasu horietakoren bat gertatzen denean bukatuko denez, while-a bukatu ondoren **if** bat ipini beharko da kasu sinpleetako bat gertatzeagatik bukatzen denean zer egin egin behar den eta beste kasu sinpleagatik bukatzen denean zer egin behar den zehaztuz:

✓ Lehenengo kasu sinplea: `hutsa_da(v)`

```
tartekatu_re(h, s)    = u ++ tartekatu_re(v, w) =
                      = u ++ w
```

Beraz kasu honetan burutu beharreko esleipena honako hau izango da:

```
q := u ++ w;
```

Gogoratu `q` aldagaia `tartekatu_re` funtzioak itzultzen duen emaitza dela.

✓ Bigarren kasu sinplea: `hutsa_da(w)`

```
tartekatu_re(h, s)    = u ++ tartekatu_re(v, w) =
                      = u ++ v
```

Beraz bigarren kasu honetan burutu beharreko esleipena honako hau izango da:

```
q := u ++ v;
```

Gogoratu `q` aldagaia `tartekatu_re` funtzioak itzultzen duen emaitza dela.

Beraz, while-a bukatu ondoren honako **if** hau ipini beharko da:

```
if hutsa_da(v) then q := u ++ w;
else q := u ++ v;
end if;
```

- **5. urratsa:** While-aren barruko aginduak.

Urrats honetan aldagai berriak while-aren barruan nola eguneratu behar diren kalkulatu behar da.

Bi kasu simple egoteak ez du eraginik urrats honetan, izan ere while-aren barruko aginduak kalkulatzeko *tartekatu_re* funtzioaren kasu errekursiboa hartu behar da kontuan.

Iterazioaren barruan joango diren aginduak kalkulatzeko inbariantea hartu, kasu errekurtsiboan gaudela kontsideratu eta hedatze-urrats bat eta tolestatze-urrats bat eman beharko dira. Tolestatze-urratsaren ondoren lortzen den espresioak u , v eta w aldagaiak nola eguneratu adieraziko digu:

Kasu errekurtsiboa: $\neg(\text{hutsa_da}(v) \vee \text{hutsa_da}(w))$

[illegible]
$$\begin{array}{l} \text{tolestatu} \rightarrow \\ = (\overbrace{\text{u ++ [leh(v)] ++ [leh(w)]}}^{\text{u}}) \\ \quad \underbrace{\text{++ tarteatu_re(hond(v), hond(w))}}_{\substack{\text{v} \qquad \text{w}}} \end{array}$$

Beraz eguneraketa honako esleipen hauen bidez lortuko da:

```

u := u ++ [leh(v)] ++ [leh(w)];
v := hond(v);
w := hond(w);

```

Esleipenen ordena garrantzitsua da hemen. Izan ere, u eguneratzean v eta w ere agertzen dira eta, ondorioz, u aldagaia v eta w baino lehenago eguneratu beharko da. Baina v eta w eguneratzean beraien arteko ordena ez da garrantzitsua. Honako eguneratze hau ondo legoke:

```

u := u ++ [leh(v)] ++ [leh(w)];
w := hond(w);
v := hond(v);

```

Baina beste hau ez dago ondo:

```
v := hond(v);  
u := u ++ [leh(v)] ++ [leh(w)];  
w := hond(w);
```

Eraiki dugun funtzio iteratibo berria honako hau da:

Funtzio iteratiboa ADA* lengoiaiaz:

```
function tartekatu_it(h, s: [Int]) return q: [Int] is
  Hasierako baldintza  $\equiv \{true\}$ 

  u, v, w: [Int];

  u := [];
  v := h;
  w := s;
  while {INV} not (hutsa_da(v) or hutsa_da(w)) loop
    u := u ++ [leh(v)] ++ [leh(w)];
    v := hond(v);
    w := hond(w);
  end loop;
  if hutsa_da(v) then q := u ++ w;
  else q := u ++ v;
  end if;
  Bukaerako baldintza  $\equiv \{q = tartekatu(h, s)\}$ 
```

Bukaerako baldintza horren bidez *tartekatu_it* funtzio iteratiboak lortzen duen q emaitza Haskell-ez idatzitako *tartekatu* funtzioak itzuliko lukeenaren berdina dela adierazten dugu.

6.3.4. KASU ERREKURTSIBO BAT BAINO GEHIAGO DITUZTEN FUNTZIOAK

Kasu errekurtsibo bat baino gehiago egoteak 1, 2 eta 5 urratsetan izango du eragina, hau da, errekkurentzia-erlazioan, inbariantearen kalkuluan eta while-aren barruko aginduen kalkuluan:

- Kasu errekurtsibo bakoitzeko errekkurentzia-erlazio bat egongo da.
- Inbariante bakarra egongo da baina errekkurentzia-erlazio denen orokorpena izango da.
- Inbariantean agertuko diren aldagai berriak while-aren barruan nola eguneratu erabakitzeko, kasu errekurtsibo bakoitza bere aldetik aztertu beharko da. Eta kasu bakoitzean aldagai berriak era desberdinean eguneratu beharko direnez, while-aren barruan if bat ipini beharko dugu.

Osoa den x zenbaki bat eta Orik ez duen zenbaki osozko s zerrenda bat emanda, x -ren zatitzaileak diren s -ko elementu denak hasieran eta x -ren zatitzaileak ez diren s -ko elementuak bukaeran dituen zerrenda berria itzultzen duen *banandu* izeneko funtzioa kontsideratuko dugu.

Adibidea:

`banandu(20, [3, 8, 10, 9, 4, 7]) = [10, 4, 7, 9, 8]`

20ren zatitzaile denak hasieran eta ordena mantenduz ageri dira eta 20ren zatitzaileak ez direnak bukaeran eta alderantzizko ordenan ageri dira. Funtzio honek horrela kalkulatu du emaitza beti.

Funtzio errekurtsiboa Haskell-ez idatzita:

`banandu: (Int, [Int]) → [Int]`

Hasierako baldintza: `{¬badago(0, s)}`

`banandu(x, s)`

| `hutsa_da(s)` = `[]`

| `x mod leh(s) == 0` = `[leh(s)] ++ banandu(x, hond(s))`

| `x mod leh(s) /= 0` = `banandu(x, hond(s)) ++ [leh(s)]`

Kasu honetan Haskell-ez idatzitako funtzioan zerrenda hutsa al den ala ez erabakitzeko `[]` eta `z:r` kasuak era esplizitoan bereizi beharrean zerrendari `s` deitu zaio eta gero *hutsa_da*, *leh* eta *hond* funtzioak erabili dira kasuak bereizteko eta elementuak eskuratzeko eta maneiatzeko. Ariketetan ere batzuetan patroiak (`[]` eta `z:r`) erabiliko dira eta beste batzuetan aldiz `[]` eta `z:r` kasuak zuzenean bereizi beharrean aldagai bakar bat erabiliko da.

Hasierako baldintza '`¬badago(0, s)`' izateak funtzioari sarrerako datu bezala emandako zerrendan 0 balioa ez dela agertuko ziurtatzen du.

Jarraian funtzio errekursiboa ADA* lengoiaz idatziko da (ADA* lengoia ADA eta Haskell-en arteko nahasketa bat da):

Funtzio errekursiboa ADA* lengoiaz:

```
function banandu_re(x: Int, s: [Int]) return h: [Int] is
Hasierako baldintza  $\equiv \{\neg \text{badago}(0, s)\}$ 
if hutsa_da(s) then h := [];
elseif x mod leh(s) = 0 then h := [leh(s)] ++ banandu_re(x, hond(s));
else h := banandu_re(x, hond(s)) ++ [leh(s)];
end if;
{h = banandu(x, s)}
```

Kasu honetan Haskell-ez idatzitako funtziotik ADA* lengoiaz idatzitako funtzioa lortzeko itzulpena nahiko zuzena izan da Haskell-en ere zerrenda s gisa agertzen zelako eta ez [] eta zr gisa.

Burstall-en metodoko bost urratsak aplikatzerakoan ADA* lengoiaz idatzitako bertsioa hartuko dugu kontuan, hau da, *banandu_re* funtzioa:

- **1. urratsa:** Errekurrentzia-erlazioa.
Errekurrentzia-erlazioa kasu errekursiboari dagokion funtzioaren definizioa da. Funtzio honetan bi kasu errekursibo daudenez, bi errekurrentzia-erlazio egongo dira.

x mod leh(s) = 0 bada:

$$\text{banandu_re}(x, s) = [\text{leh}(s)] ++ \text{banandu_re}(x, \text{hond}(s))$$

x mod leh(s) \neq 0 bada:

$$\text{banandu_re}(x, s) = \text{banandu_re}(x, \text{hond}(s)) ++ [\text{leh}(s)]$$

- **2. urratsa:** Inbariantea kalkulatu.

Inbariantea kalkulatzeko errekurrentzia-erlazio biak orokortu behar dira. Horretarako onena adibide bat garatzea izaten da hedatze eta tolestatze urratsak emanez. Tolestatze-urratsen ondoren gelditzen diren espresioek errekurrentzia-erlazioak nola orokortu erakutsiko digute.

```

banandu_re(20, [8, 10, 9, 4, 7]) =
hedatu →      = banandu_re(20, hond([8, 10, 9, 4, 7]) ++ [leh([8, 10, 9, 4, 7])]) =
tolestatu →    = banandu_re(20, [10, 9, 4, 7]) ++ [8] =
hedatu →      = ([leh([10, 9, 4, 7])]) ++ banandu_re(20, hond([10, 9, 4, 7])) ++ [8] =
tolestatu →    = [10] ++ banandu_re(20, [9, 4, 7]) ++ [8] =
hedatu →      = [10] ++ (banandu_re(20, hond([9, 4, 7])) ++ [leh([9, 4, 7])]) ++ [8] =
tolestatu →    = [10] ++ banandu_re(20, [4, 7]) ++ [9, 8] =
hedatu →      = [10] ++ ([leh([4, 7])]) ++ banandu_re(20, hond([4, 7])) ++ [9, 8] =
tolestatu →    = [10, 4] ++ banandu_re(20, [7]) ++ [9, 8] =
hedatu →      = [10, 4] ++ (banandu_re(20, hond([7])) ++ [leh([7])]) ++ [9, 8] =
tolestatu →    = [10, 4] ++ banandu_re(20, []) ++ [7, 9, 8] =
kasu sinplea → = [10, 4] ++ [] ++ [7, 9, 8] =
               = [10, 4, 7, 9, 8]

```

Hedatze-urratsean *banandu_re* funtzioa bere definizio errekursiboa kontuan hartuz ordezkatu behar da.

Tolestatze-urratsean espresioa sinplifikatu egin behar da ahal diren eragiketak burutuz.

Garatu dugun adibidean tolestatze-urrats bakoitzean zerrenda bat, dei errekursibo bat eta beste zerrenda bat elkartuz osatutako espresio bat daukagu. Dei errekursiboan argumentu bezala zenbaki oso bat eta hirugarren zerrenda bat agertzen dira.

Orain errekurrentzia-erlazioa orokortu behar da. Errekurrentzia-erlazioa orokortzeko tolestatze-urratsen ondoren lortutako espresioak hartu behar dira kontuan. Tolestatze-urratsen ondoren lortutako espresio horietan **aldatuz doazen elementuak aldagai berriez ordezkatu** behar dira. Gure adibidean bai kanpoko zerrenda biak eta bai dei errekursiboko zerrenda aldatuz doaz. Dei errekursiboko zenbakia beti berdin mantentzen da eta ondorioz hori ez da ordezkatu behar:

$$\begin{array}{c}
 = [10, 4] ++ banandu_re(20, []) ++ [7, 9, 8] = \\
 \underbrace{\hspace{1.5cm}}_u \qquad \underbrace{\hspace{1.5cm}}_v \quad \underbrace{\hspace{1.5cm}}_w
 \end{array}$$

Iterazioaren **inbariantea** honako hau izango da:

$$\mathbf{banandu_re(x, s) = u ++ banandu_re(x, v) ++ w}$$

Garrantzitsua da inbariantea errekurrentzia-erlazio bien orokorpen bat dela ikustea:

$x \bmod \text{leh}(s) = 0$:

$$\text{banandu_re}(x, s) = \underbrace{[\text{leh}(s)]}_{u} ++ \text{banandu_re}(x, \underbrace{\text{hond}(s)}_v) ++ \underbrace{[]}_w$$

$x \bmod \text{leh}(s) \neq 0$:

$$\text{banandu_re}(x, s) = \underbrace{[]}_u ++ \text{banandu_re}(x, \underbrace{\text{hond}(s)}_v) ++ \underbrace{[\text{leh}(s)]}_w$$

$[\text{leh}(s)]$ eta $[]$ espresioen ordeztu u ipini da, $\text{hond}(s)$ espresioaren ordeztu v aldagaia ipini da eta $[]$ eta $[\text{leh}(s)]$ espresioen ordeztu w ipini da. Errekurrentzia-erlazioetan ipini diren zerrenda hutsak errekurrentzia-erlazio biek formato bera izan dezaten ipini dira: 'zerrenda' ++ 'dei errekurtsiboa' ++ 'zerrenda'.

- **3. urratsa:** Aldagai berrien hasieraketa.

Inbariantea lortzerakoan agertu diren aldagai berriak nola hasieratu erabaki behar da urrats honetan.

Hasieran, while-ean lehenengo aldiz sartu aurretik inbarianteak egiazkoa izan behar du eta hori da aldagaiak nola hasieratu erabakitzeko kontuan hartuko duguna. Hasieratu ondoren u , v eta w aldagaiek honako berdintza hau bete behar dute:

$$\text{banandu_re}(x, s) = u ++ \text{banandu_re}(x, v) ++ w$$

Berdintza hori bete dadin u , v eta w aldagaiak berdintzaren ezkerreko aldean beraien posizio bera okupatzen duten balioekin hasieratu beharko dira. Kasu honetan ezkerreko aldean u eta w aldagaien leku berean ezer ez dagoenez eta zerrendak elkartzen ari garenez, leku bietan $[]$ balioa dagoela suposatu behar da:

$$\begin{array}{c} \uparrow \qquad \qquad \qquad \downarrow \qquad \qquad \qquad \downarrow \\ [] ++ \text{banandu_re}(x, s) ++ [] = u ++ \text{banandu_re}(x, v) ++ w \end{array}$$

Eta hortik u eta w aldagaiak $[]$ balioarekin eta v aldagaia s balioarekin hasieratu beharko ditugula ondoriozta dezakegu:

```
u := [];
v := s;
w := [];
```


Hasieraketaren ondoren inbariantea bete egiten dela egiazta daiteke:

$$\begin{aligned} \text{banandu_re}(x, s) &= u ++ \text{banandu_re}(x, v) ++ w = \\ &= [] ++ \text{banandu_re}(x, s) ++ [] = \\ &= \text{banandu_re}(x, s) \end{aligned}$$

- **4. urratsa:** Bukaera (while-aren baldintza eta bukaerako esleipena).

Urrats honetan while-ean mantentzeko bete behar den B baldintza eta while-a bukatu ondoren burutu beharko den esleipena zein diren erabaki behar da.

- Iterazioan mantentzeko bete beharreko B baldintza kalkulatzekoan bide errazena $\neg B$ kalkulatzeko da: $\neg B \equiv \text{"kasu simplea"}$. Baina kasu sinplea formulatzekoan aldagai berriak erabili behar dira:

$$\begin{aligned} \neg B &\equiv \text{hutsa_da}(v) \\ &\text{eta,} \\ B &\equiv \neg \text{hutsa_da}(v), \end{aligned}$$

- While-a bukatu ondoren zein esleipen burutu behar den erabakitzeko, hau da, funtzioak emaitza bezala zein balio itzuliko duen erabakitzeko, inbariantea hartu eta kasu sinplean gaudela suposatu beharko da ($\text{hutsa_da}(v)$):

$$\begin{aligned} \text{banandu_re}(x, s) &= u ++ \text{banandu_re}(x, v) ++ w = \\ &= u ++ \text{banandu_re}(x, []) ++ w = \\ &= u ++ w \end{aligned}$$

Beraz bukaerako esleipena honako hau izango da:

$$h := u ++ w;$$

Gogoratu h aldagaia *banandu_re* funtzioak itzultzen duen emaitza dela.

- **5. urratsa:** While-aren barruko aginduak

Urrats honetan aldagai berriak while-aren barruan nola eguneratu behar diren kalkulatu behar da.

While-aren barruko aginduak kalkulatzeko *banandu_re* funtzioaren kasu errekurtsiboak hartu behar dira kontuan.

Bi kasu errekurtsibo daudenez, bakoitza bere aldetik aztertu beharko da. Hasteko, inbariantea hartu eta lehenengo kasu errekurtsiboan gaudela kontsideratu beharko da eta hedatze-urrats bat eta tolestatze-urrats bat emango dira. Horrela lehenengo kasu errekurtsiboan gaudenean aldagaiak nola eguneratu behar ditugun jakingo dugu. Gero bigarren kasu errekurtsiboan gaudela suposatuz berdina egin beharko da. Bi kasu daudenez eta kasu bakoitzean aldagaiak eguneratzeko era desberdina izangoenez while-aren barruan **if** bat ipini beharko da.

- Lehenengo kasu errekursiboa: $x \bmod \text{leh}(v) = 0$

$\text{banandu_re}(x, s) = u ++ \text{banandu_re}(x, v) ++ w =$
 hedatu $\rightarrow = u ++ ([\text{leh}(v)] ++$
 $++ \text{banandu_re}(x, \text{hond}(v))) ++ w =$
 tolestatu $\rightarrow = \underbrace{(u ++ [\text{leh}(v)])}_{u} ++ \underbrace{\text{banandu_re}(x, \text{hond}(v))}_{v} ++ \underbrace{w}_{w}$

Beraz eguneraketa honako esleipen hauen bidez lortuko da:

$u := u ++ [\text{leh}(v)];$
 $v := \text{hond}(v);$
 $w := w;$

Esleipenen ordena garrantzitsua da hemen. Izan ere u eguneratzerakoan v ere agertzen da eta ondorioz u aldagaia v baino lehenago eguneratu beharko da. Beste aldetik, w aldagaiaren eguneraketak u eta v -rengan eraginik ez duenez, w edozein unetan egunera daiteke. Honako eguneraketa hau ondo legoke.

:

$w := w;$
 $u := u ++ [\text{leh}(v)];$
 $v := \text{hond}(v);$

Baina beste eguneraketa hau ez legoke ondo:

~~$v := \text{hond}(v);$
 $u := u ++ [\text{leh}(v)];$
 $w := w;$~~

- Bigarren kasu errekursiboa: $x \bmod \text{leh}(v) \neq 0$

$\text{banandu_re}(x, s) = u ++ \text{banandu_re}(x, v) ++ w =$
 hedatu $\rightarrow = u ++ (\text{banandu_re}(x, \text{hond}(v)) ++ [\text{leh}(v)]) ++ w =$
 tolestatu $\rightarrow = \underbrace{u}_{u} ++ \underbrace{\text{banandu_re}(x, \text{hond}(v))}_{v} ++ \underbrace{([\text{leh}(v)] ++ w)}_{w}$

Beraz bigarren kasu honetan eguneraketa honako esleipen hauen bidez lortuko da:

$u := u;$
 $w := [\text{leh}(v)] ++ w;$
 $v := \text{hond}(v);$

Esleipenen ordena garrantzitsua da hemen ere, izan ere w aldagaiaren eguneraketan v agertzen denez, w aldagaia v baino lehenago eguneratu behar da. Baina u aldagaiak beste aldagaietan ez duenez eraginik, edozein unetan egunera daiteke.

Bukatzeko, kasu errekursibo bietako eguneraketak batera hartuz, honako if agindu hau planteatu behar da:

```
if  $x \bmod \text{leh}(s) = 0$  then  $u := u ++ [\text{leh}(v)]$ ;  
                                 $v := \text{hond}(v)$ ;  
                                 $w := w$ ;  
else  $u := u$ ;  
       $w := [\text{leh}(v)] ++ w$ ;  
       $v := \text{hond}(v)$ ;  
end if;
```

If agindu hori $w := w$ eta $u := u$ esleipenak kenduz eta $v := \text{hond}(v)$ esleipena if-etik kanpora ateraz sinplifika daiteke, baina guk horrela lagako dugu metodoa jarraituz kasu bakoitzean zer atera den hobeto ikusteko.

Eraiki dugun funtzio iteratibo berria honako hau da:

Programa iteratiboa ADA* lengoaiatz:

function banandu_it(x : Int, s : [Int]) **return** h : [Int] **is**
Hasierako baldintza $\equiv \{\neg \text{badago}(0, s)\}$

u, v, w : [Int];

$u := []$;

$v := s$;

$u := []$;

while {INB} **not** hutsa_da(v) **loop**

```
  if  $x \bmod \text{leh}(s) = 0$  then  
     $u := u ++ [\text{leh}(v)]$ ;  
     $v := \text{hond}(v)$ ;  
     $w := w$ ;
```

else

$u := u$;

$w := [\text{leh}(v)] ++ w$;

$v := \text{hond}(v)$;

end if;

end loop;

$h := u ++ w$;

Bukaerako baldintza $\equiv \{h = \text{banandu}(x, s)\}$

Hasierako eta bukerako baldintzen bidez, *banandu_it* funtzioak datu bezala zerorik ez duen s zerrenda eta x elementua hartu eta emaitza moduan Haskell-ez idatzitako *banandu* funtzioak itzuliko lukeenaren berdina itzuliko duela adierazten da.

6.4. LABURPENA ETA ONDORIOAK

Gai honetan programa errekurtsiboak iteratibo bihurtzeko balio duen Burstall-en metodoa aurkeztu da eta adibideak garatuz metodo hori lau eratako funtzioetan nola aplikatzen den azaldu da:

- A) Oinarrizko funtzio errekurtsiboak (kasu simple bat eta kasu errekurtsibo bat).
- B) Elkarkorra ez den eragiketa duten funtzio errekurtsiboak.
- C) Kasu simple bat baino gehiago dituzten funtzioak.
- D) Kasu errekurtsibo bat baino gehiago dituzten funtzioak.

Gai honetako helburuetako bat errekurtsibitatearen eta iterazioaren arteko erlazioa zein den ulertzea da. Garatu diren lau adibideak begiratzuz gero honako ondorio hau atera dezakegu:

Dei errekurtsibo bakoitza while-eko buelta bat da.

Ondorio honek errekurtsioaren eta iterazioaren arteko erlazioa ulertzen hasteko balio digu.