Triplete pitagoreice primitive

- Data publicării: 28.02.2022
- Data ultimei modificări: 28.02.2022 (changelog)
- Tema (o arhivă .zip cu fișierul ppt.rkt (plus cryptosystem.rkt în etapa 3) se va încărca pe vmchecker [https://vmchecker.cs.pub.ro/ui/#PP]

Descriere generală și organizare

Tema constă în generarea tripletelor pitagoreice primitive și utilizarea acestora într-un criptosistem simplu.

Tema este împărțită în **4 etape**:

- una pe care o veți rezolva după laboratorul 2 (cu deadline în ziua laboratorului 3, la ora 23:59)
- una pe care o veți rezolva după laboratorul 3 (cu deadline în ziua laboratorului 4, la ora 23:59)
- una pe care o veți rezolva după laboratorul 4 (cu deadline în ziua laboratorului 5, la ora 23:59)
- una pe care o veți rezolva după laboratorul 5 (cu deadline în ziua laboratorului 6, la ora 23:59)

Așa cum se poate observa, **ziua deadline-ului variază în funcție de semigrupa în care sunteți repartizați**. **Restanțierii care refac tema și nu refac laboratorul beneficiază de ultimul deadline** (deci vor avea deadline-uri în zilele de 21.03, 28.03, 04.04, 11.04).

Rezolvările tuturor etapelor pot fi trimise până în ziua laboratorului 6, dar orice exercițiu trimis după deadline se punctează cu **jumătate** din punctaj. Nota finală pe etapă se calculează conform formulei: **n = (n1 + n2) / 2** (n1 = nota obținută înainte de deadline; n2 = nota obținută după deadline). Când toate submisiile sunt înainte de deadline, nota pe ultima submisie este și nota finală (întrucât n1 = n2).

În fiecare etapă, veți folosi ce ați învățat în săptămâna anterioară pentru a dezvolta aplicația.

Pentru fiecare etapă veți primi un **schelet de cod** (dar rezolvarea se bazează în mare măsură pe rezolvările anterioare). Intenția este să puteți rezolva tema utilizând doar indicațiile din schelet (fără a fi necesar să citiți enunțul). Enunțul încearcă să lămurească aspectele care poate nu sunt clare tuturor doar din schelet.

Etapa 1

În această etapă vă veți familiariza cu noțiunea de triplet pitagoreic primitiv (TPP), apoi veți:

- observa trei transformări matriciale suficiente pentru a genera un arbore infinit de TPP
- implementa obținerea celui de-al n-lea TPP din arborele definit anterior

Rezolvarea acestor sarcini presupune utilizarea de:

- liste (tripletele sunt reprezentate ca liste; matricile sunt reprezentate ca liste de liste, fiecare listă interioară fiind un rând în matrice)
- funcții recursive pe stivă, respectiv pe coadă (observați tipul de recursivitate al fiecărei funcții implementate, și atenție la cazurile în care vi se solicită un anumit tip de implementare - chiar dacă obțineți punctaj pe checker, punctajul va fi anulat în cazul în care funcțiile nu sunt implementate conform specificației)
- programare de tip “wishful thinking” - veți implementa funcția get-transformations ca și cum ați avea deja alte funcții care calculează rezultate intermediare utile (de exemplu nivelul din arbore pe care se află un index n), apoi veți avea grijă să implementați toate funcțiile ajutătoare pe care v-ați dorit să le aveți deja

Funcțiile principale pe care va trebui să le implementați sunt:

```
(dot-product X Y)
```

- dot-product calculează produsul scalar a doi vectori X și Y, definit ca suma produselor elementelor care se găsesc pe aceeași poziție în X și Y
- ex: `(dot-product '(1 2 2) '(3 4 5))` va determina calculul `1*3 + 2*4 + 2*5` ⇒ `21`

```
(multiply M V)
```

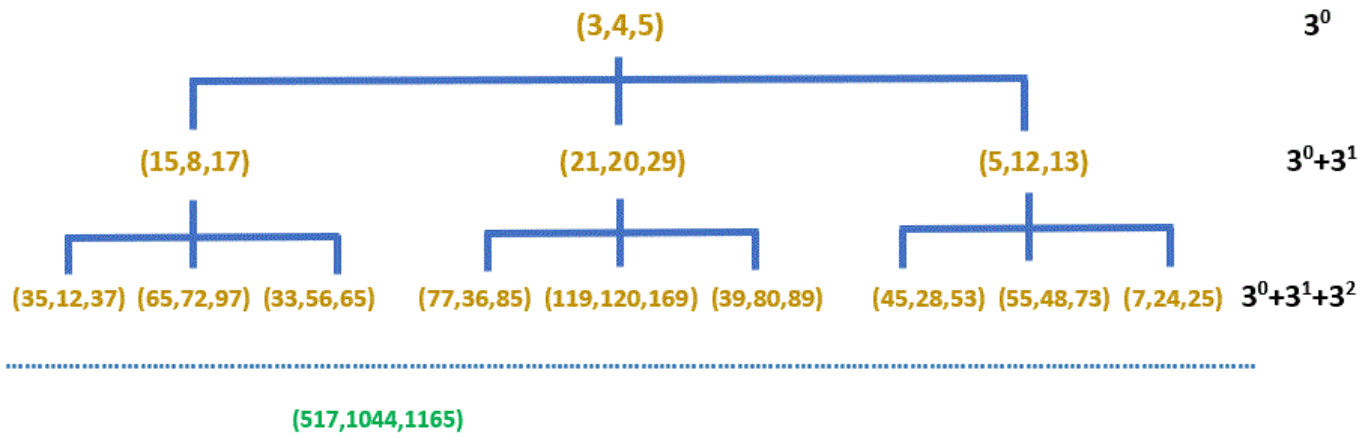
- multiply înmulțește matricea M cu vectorul V (se garantează că numărul de coloane din M coincide cu dimensiunea lui V, nu este necesar să verificați)
- pentru operația de înmulțire, poziționați V "pe verticală" (practic, V este o matrice cu n linii și o singură coloană)
- ex: `(multiply '((1 2 2) (2 1 2) (2 2 3)) '(3 4 5))` ⇒ `'(21 20 29)`

```
(get-transformations n)
```

- înainte de a citi explicațiile de mai jos, citiți neapărat primele 30 de rânduri din scheletul de cod
- get-transformations primește un număr n și determină calea în arborele de TPP de la tripletul (3,4,5) la al n-lea triplet din arbore
- o cale este o succesiune (reprezentată ca listă) de transformări de tip T1, T2 sau T3
- rezultatul întors de get-transformations este o listă numerică (ex: '(2 1 3) înseamnă că trebuie aplicate, în această ordine, T2, T1, T3)

Aceasta este cea mai complexă funcție din etapa 1. Urmează un exemplu de determinare a căii (deși cu siguranță puteți descoperi algoritmul și pe cont propriu).

Să presupunem că ne interesează al 64-lea triplet din arbore.



- pe primul rând este 1 triplet (3^0)
- pe al doilea rând sunt 3 triplete (3^1), ceea ce înseamnă că ultimul triplet de pe acest rând este al patrulea din arbore (3^0+3^1)
- pe al treilea rând sunt 9 triplete (3^2), ceea ce înseamnă că ultimul triplet de pe acest rând este al treisprezecelea din arbore ($3^0+3^1+3^2$)
- al patrulea rând conține 27 de triplete, ducând totalul la 40 (13 + 27)
- al cincilea rând conține 81 de triplete, ducând totalul la 121 – înseamnă că acesta este rândul care conține tripletul căutat, cel cu indexul 64

Pentru a determina calea (succesiunea de transformări) de la rădăcina (3,4,5) la acest triplet, ne uităm la poziția indexului 64 pe rândul său în arbore (adică pe rândul care conține toate tripletele de la indexul 41 la indexul 121). Transformarea T1 (asupra rădăcinii (3,4,5)) conduce către indecșii din prima treime (41 - 67), T2 către indecșii din a doua treime (68 - 94), iar T3 către indecșii din ultima treime (95 - 121).

- rezultat intermediar: 1 (corespunzător transformării T1)

În continuare, ne uităm la poziția lui 64 în intervalul 41 – 67 în care ne-a plasat aplicarea transformării T1. Este în ultima treime (59 - 67), ceea ce înseamnă că am ajuns aici aplicând T3.

- rezultat intermediar: 1, 3

Constatăm apoi că ne aflăm în a doua treime a intervalului 59 – 67 (via T2), în ultima treime a intervalului 62 – 64 (via T3), și, în final, că intervalul s-a redus la numărul 64 deci calea este completă.

- rezultat final: 1, 3, 2, 3 (`(get-transformations 64` ⇒ `'(1 3 2 3)`)

```
(apply-matrix-transformations Ts ppt)
```

- Ts este o listă numerică (ex: `'(1 3 2 3)` care codifică succesiunea de transformări T1, T3, T2, T3)
- ppt este un triplet inițial (este traducerea TPP în engleză: primitive pythagorean triple)
- apply-matrix-transformations pornește de la ppt și aplică, pe rând, transformările corespunzătoare numerelor din Ts (ex: pentru `Ts = '(1 3 2 3)`, calculează `T3·T2·T3·T1·ppt`)
- ex: `(apply-matrix-transformations '(1 3 2 3) '(3 4 5))` ⇒ `'(517 1044 1165)`

```
(get-nth-ppt-from-matrix-transformations n)
```

- get-nth-ppt-from-matrix-transformations calculează al n-lea TPP din arbore
- ex: `(get-nth-ppt-from-matrix-transformations 64)` ⇒ `'(517 1044 1165)`

Etapa 2

În această etapă veți implementa o metodă alternativă de generare a aceluiași arbore infinit de TPP, bazată pe cvartete în loc de triplete. Metoda este descrisă în amănunt în scheletul de cod.

Pe lângă implementarea noii metode, veți modifica și implementarea metodei anterioare, astfel încât să exploatați faptul că funcțiile sunt valori de ordinul întâi. Scopul este consolidarea cunoștințelor legate de:

- funcționale
- funcții curry și uncurry
- abstractizarea funcțiilor cu implementări similare

De asemenea, vă încurajăm să valorificați oportunitățile de utilizare a funcțiilor anonime și a funcțiilor de bibliotecă `curry` și `uncurry`, deși enunțul nu vă cere în mod explicit.

Principalele funcții noi pe care va trebui să le implementați sunt:

```
(apply-functional-transformations Fs tuple)
```

- Fs este o listă de funcții unare pe tupluri (ex: `(list reverse cdr)`)
- tuple este un tuplu inițial
- apply-functional-transformations pornește de la tuple și aplică, pe rând, funcțiile din Fs
- ex: `(apply-functional-transformations (list reverse ((curry map) add1) cdr) '(3 4 5))` ⇒ `'(5 4)`
  - `(reverse '(3 4 5))` ⇒ `'(5 4 3)`
  - `(((curry map) add1) '(5 4 3))` ⇒ `'(6 5 4)`
  - `(cdr '(6 5 4))` ⇒ `'(5 4)`

```
get-nth-tuple
```

- get-nth-tuple determină al n-lea tuplu dintr-un arbore care pleacă:
  - de la un tuplu inițial dat
  - cu trei transformări date, pe baza cărora se calculează generația următoare
- signatura nu este precizată întrucât primul pas este să determinați voi înșivă modul în care ar fi cel mai bine ca get-nth-tuple să își primească parametrii
- ex: pentru n = 5, tuplul de start (1,1,2,3) și funcțiile corespunzătoare transformărilor Q1, Q2, Q3 ⇒ `'(5 1 6 7)`

Funcția get-nth-tuple este un exemplu de abstractizare. Abstractizare înseamnă că, dacă ar exista o a treia metodă de generare a arborelui, bazată pe, de exemplu, cvintete și niște funcții capabile să calculeze dintr-un cvintet trei cvintete următoare, atunci get-nth-tuple ar funcționa fără modificări și pentru această metodă (apelat pe argumente adecvate). Cu alte cuvinte, în funcție nu trebuie să verificăm cu ce fel de tupluri lucrăm, ci funcția trebuie să aplice “orbește” funcțiile primite ca parametru pe tuplul primit ca parametru, știind că acestea vor produce rezultatul așteptat (triplet, cvartet, cvintet sau orice altceva).

Pe baza lui get-nth-tuple veți implementa ultimele funcții din temă - care ne permit obținerea celui de-al n-lea TPP din arbore prin cele două metode distincte, fără a scrie două bucăți foarte similare de cod.

Etapa 3

În această etapă veți implementa un criptosistem cu chei simetrice, în care generarea cheii este bazată pe teoria tripletelor pitagoreice. Vă veți exersa cunoștințele legate de:

- funcționale, funcții curry și uncurry, abstractizarea proceselor similare (la fel ca și săptămâna trecută - acestea sunt concepte centrale programării funcționale și, odată studiate, trebuie să devină un stil de a programa)
- expresii de legare statică a variabilelor (let-uri)

Pe scurt, criptosistemul funcționează astfel:

- cheia se generează pe baza unui număr n
  - se determină al n-lea cvartet din arborele TPP de cvartete generat data trecută
  - pe baza valorilor e și f din cvartet se calculează un tuplu de 9 elemente, conform formulelor descrise în scheletul de cod
- mesajele (reprezentate ca șiruri de caractere) sunt convertite în liste de coduri
  - din start, ne limităm la mesaje care conțin doar litere mici și spații
  - spațiului îi asociem codul 0, literelor de la 'a' la 'z' le asociem coduri de la 1 la 26
- algoritmul de criptare este:
  - extindem/trunchiem cheia la dimensiunea mesajului
  - pentru fiecare index din cele 2 liste de coduri (mesajul și cheia), aplicăm formula (m + k) mod 27
    - m reprezintă un cod din mesaj, k reprezintă codul de pe aceeași poziție în cheie
- algoritmul de decriptare este similar, formula fiind (c - k) mod 27 (c reprezintă un cod din mesajul criptat)

Veți lucra doar în fișierul cryptosystem.rkt, însă este necesar să aveți în același folder și fișierul ppt.rkt din etapa 2 (iar arhiva încărcată pe vmchecker va conține ambele fișiere).

Funcțiile pe care va trebui să le implementați sunt:

```
(key n)
```

- key calculează cheia de criptare/decriptare pe baza numărului n
- ex: `(key 3)` ⇒ `'(24 16 20 11 3 7 21 20 2)`
  - al treilea cvartet din arborele TPP este (3,2,5,7) ⇒ e = 2, f = 5
  - se aplică formulele pentru a1, b1, … b3, c3 ⇒ `'(24 70 74 -16 30 34 21 20 29)`
  - se aduce fiecare valoare în intervalul 0 - 26 ⇒ `'(24 16 20 11 3 7 21 20 2)`

```
(message->codes message)
(codes->message codes)
```

- message→codes primește un string message și întoarce o listă de numere în intervalul 0 - 26
  - 0 pentru fiecare spațiu din string
  - un număr între 1 și 26 pentru fiecare literă mică ('a' devine 1 … 'z' devine 26)
- codes→message efectuează procesul invers
- ex: `(message->codes "do or do not")` ⇒ `'(4 15 0 15 18 0 4 15 0 14 15 20)`
- ex: `(codes->message '(4 15 0 15 18 0 4 15 0 14 15 20))` ⇒ `"do or do not"`

```
(extend-key key size)
```

- extend-key primește o listă numerică reprezentând cheia și aduce această listă la dimensiunea size prin concatenări succesive și/sau trunchiere atunci când este cazul
- ex: `(extend-key '(24 16 20 11 3 7 21 20 2) 21)` ⇒ `'(24 16 20 11 3 7 21 20 2 24 16 20 11 3 7 21 20 2 24 16 20)`
- ex: `(extend-key '(24 16 20 11 3 7 21 20 2) 7)` ⇒ `'(24 16 20 11 3 7 21)`

```
(encrypt-codes message key)
(decrypt-codes message key)
(encrypt-message message key)
(decrypt-message message key)
```

- primele două funcții realizează criptare/decriptare pe liste de coduri (argumentul message este o listă de coduri, valoarea întoarsă este o listă de coduri)
- ultimele două funcții realizează criptare/decriptare pe stringuri (argumentul message este un string, valoarea întoarsă este un string)
- ex: `(encrypt-message "there is no try" (key 45))` ⇒ `"vzccfh nepfyesf"`
  - "there is no try" are asociate codurile `'(20 8 5 18 5 0 9 19 0 14 15 0 20 18 25)`
  - `(key 45)` ⇒ `'(2 18 25 12 1 8 18 22 5)`
  - mesajul având dimensiune 15, cheia extinsă este `'(2 18 25 12 1 8 18 22 5 2 18 25 12 1 8)`
  - (20 + 2) mod 27 = 22 (codul asociat caracterului 'v'), (8 + 18) mod 27 = 26 (codul asociat lui 'z'), etc.

Etapa 4

În etapele 1 și 2 ați studiat două metode de generare a unui arbore infinit de TPP-uri, punând accent pe algoritmul de a determina al n-lea TPP din arbore. În etapa finală vă veți concentra pe modul în care obținem o întreagă secvență ordonată de TPP, unde ordinea depinde de metoda de generare folosită. Mai întâi veți implementa fluxul de triplete corespunzător indexării cu care am lucrat în etapele anterioare, apoi un alt flux care conține aceleași triplete într-o altă ordine, corespunzătoare unei noi metode de generare.

Pentru a completa cu succes etapa, veți lucra cu:

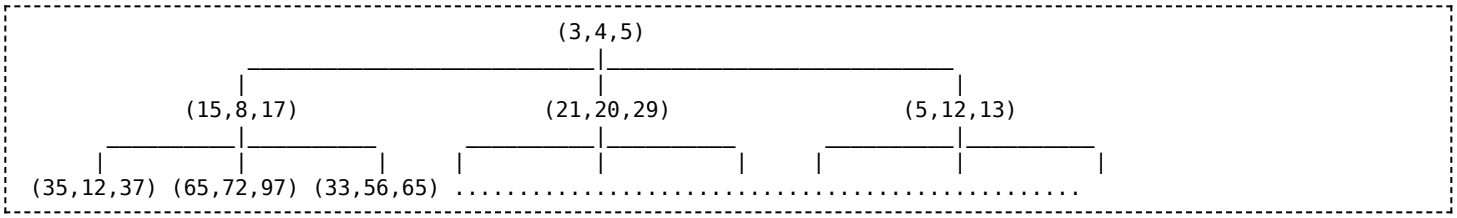
- recursivitate, funcționale, expresii de legare statică - acestea sunt facilități ale stilului funcțional de a programa pe care le folosim cu fiecare ocazie pentru a avea un cod mai eficient, mai compact și mai lizibil
- fluxuri - conceptul central acestei etape

Funcțiile (sau fluxurile) principale pe care trebuie să le implementați sunt:

```
ppt-stream-in-tree-order
```

- ppt-stream-in-tree-order este un flux care corespunde parcurgerii în lățime (BFS) a arborelui TPP pe care îl cunoaștem

Exemplu:



- la începutul BFS, coada Q de noduri care trebuie vizitate e inițializată la {(3,4,5)}, iar fluxul F pe care îl construim încă nu conține niciun element
- în prima iterație:
  - F = {(3,4,5)} - primul element din Q este trecut în rezultat
  - Q = {(15,8,17), (21,20,29), (5,12,13)} - succesorii nodului vizitat sunt adăugați la sfârșitul cozii Q
- în a doua iterație:
  - F = {(3,4,5), (15,8,17)} - primul element din Q este trecut în rezultat
  - Q = {(21,20,29), (5,12,13), (35,12,37), (65,72,97), (33,56,65)} - succesorii nodului vizitat sunt adăugați la sfârșitul cozii Q
- construim un flux infinit, așadar iterațiile continuă la infinit

```
(pairs G H)
```

- pairs primește 2 fluxuri ordonate G și H
- notăm cu g un element din G și cu h un element din H (pentru un același index considerăm, fără să verificăm, că va fi întotdeauna adevărat că g < h, în sensul că în final pairs este menit să fie apelat doar pe fluxurile G = {1,3,5,7 …} și H = {3,5,7,9 …} care respectă această condiție)
- pairs întoarce un flux infinit de perechi de forma (g, h) conform algoritmului descris în schelet, în ordinea crescătoare a valorii lui h (și în ordinea crescătoare a valorii lui g, dacă perechile au același h)
- Atenție: pairs nu face nicio verificare (nu verifică dacă g și h sunt impare, nu verifică dacă g<h, nu verifică dacă cele două numere sunt prime între ele)! Aceste condiții vor fi asigurate de funcțiile următoare, pairs doar implementează metoda de construcție descrisă în schelet. În checker, pairs este apelat inclusiv pe fluxuri pentru care niciuna din condițiile anterioare nu se respectă, asigurând astfel că nu faceți verificările în altă parte decât se precizează în cerință.

Exemplu:

- la început, G = {1,3,5,7 …} și H = {3,5,7,9 …}
- fluxul rezultat va fi {(1,3), (1,5), (3,5), (1,7), (3,7), (5,7), (1,9), (3,9), (5,9) …} (metoda de construcție e detaliată în schelet)
- observați că în rezultatul funcției pairs apar și perechi de numere neprime între ele (de exemplu, perechea (3,9))

```
gh-pairs-stream
```

- gh-pairs-stream reprezintă fluxul de perechi (g,h) care respectă condițiile
  - g, h impare

    - g < h
    - g, h prime între ele
- fluxul rezultat va fi {(1,3), (1,5), (3,5), (1,7), (3,7), (5,7), (1,9), ~~(3,9),~~ (5,9) …}

```
ppt-stream-in-pair-order
```

- ppt-stream-in-pair-order reprezintă fluxul de TPP corespunzător fluxului anterior de perechi
- formulele de obținere a unui triplet (a,b,c) dintr-o pereche (g,h) sunt date în schelet
- fluxul rezultat va fi {(3,4,5), (5,12,13), (15,8,17), (7,24,25), (21,20,29) …}

Acesta este punctul terminus al temei 1. În fiecare etapă am încercat să valorificăm atât conceptele studiate în etapele anterioare, cât și concepte nou descoperite. Deși nu există o etapă 5, există o temă 2, în Haskell, către care fluxurile sunt o punte foarte potrivită. Haskell va avea și recursivitate, și funcționale, și legări locale, și liste infinite, și va aduce în plus alte facilități care sperăm să vă placă și să vă deschidă apetitul pentru arta de a programa. Felicitări pentru că ați ajuns în acest punct al temei 1 și mult succes în continuare!

Precizări

- În etapele 1, 2 și 4, veți implementa funcțiile din fișierul **ppt.rkt**. În etapa 3, veți lucra în fișierul **cryptosystem.rkt**. Pentru testare, veți rula codul din fișierul **checker.rkt**.
- Tema se va încărca pe vmchecker. Testele de vmchecker sunt aceleași cu cele din checker.rkt.
- Tema este în primul rând o temă de programare funcțională - pentru care folosim Racket. Racket este un limbaj multiparadigmă, care conține și elemente “ne-funcționale” (de exemplu proceduri cu efecte laterale), pe care **nu** este permis să le folosiți în rezolvare.
- Pentru fiecare etapă, checker-ul vă oferă un punctaj între 0 și 120 de puncte. Pentru a obține cele 1.33p din nota finală cu care este creditată tema de Racket, este suficient să acumulați 400 de puncte de-a lungul celor 4 etape. Un punctaj între 400 și 480 se transformă într-un bonus proporțional.
- Veți prezenta tema asistentului, care poate modifica punctajul dat de checker dacă observă nereguli precum răspunsuri hardcodate, proceduri cu efecte laterale, implementări neconforme cu restricțiile din enunț.

Resurse

- etapa 1
- etapa 2
- etapa 3
- etapa 4

Changelog

- 27.03 (ora 15:55) - Etapa 4 - Am adăugat (în schelet) precizarea că puteți folosi recursivitate explicită pentru a genera fluxurile G și H.
- 27.03 (ora 15:55) - Etapa 4 - Am adăugat (în această pagină) explicații despre comportamentul funcției pairs.
- 21.03 (ora 23:21) - Am publicat etapa 4.
- 14.03 (ora 23:14) - Am publicat etapa 3.
- 07.03 (ora 23:14) - Am publicat etapa 2.
- 28.02 (ora 23:25) - Am publicat etapa 1 (enunț + schelet), și etapele 2-4 (momentan doar enunț).

Referinţe

- Pythagorean Triples and Cryptographic Coding [https://arxiv.org/ftp/arxiv/papers/1004/1004.3770.pdf]
- Data Encryption and Decryption Using New Pythagorean Triple Algorithm [http://www.iaeng.org/publication/WCE2014/WCE2014_pp516-519.pdf]

pp/22/teme/racket-pitagora.txt · Last modified: 2022/03/27 17:00 by mihaela.balint