UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Implementarea unui algoritm de tip Greedy în Dafny ce rezolvă Problema Bancnotelor

propusă de

Veronica Elisa Chicoș

Sesiunea: iunie, 2022

Coordonator științific

Conf. Dr. Ciobâcă Ștefan

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Implementarea unui algoritm de tip Greedy în Dafny ce rezolvă Problema Bancnotelor

Veronica Elisa Chicoș

Sesiunea: iunie, 2022

Coordonator științific

Conf. Dr. Ciobâcă Ștefan

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Ciobâcă Ștefan.

Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Chicoș Veronica Elisa** domiciliat în **România, jud. Galați, com. Matca, str. 1 decembrie 1918, nr. 7**, născut la data de **09 decembrie 2000**, identificat prin CNP **6001209171714**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2022, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Implementarea unui algoritm de tip Greedy în Dafny ce rezolvă Problema Bancnotelor** elaborată sub îndrumarea domnului **Conf. Dr. Ciobâcă Ștefan**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Implementarea unui algoritm de tip Greedy în Dafny ce rezolvă Problema Bancnotelor**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Veronica Elisa Chicoș**

Data: ..... Semnătura: .....

Cuprins

Motivație

Motivul alegerii acestei teme derivă din dorința de a-mi îmbogăți cunoștințele despre paradigma de programare Greedy.

La prima interacțiune cu acest subiect, mai exact în anul I de facultate la cursul ”Proiectarea algoritmilor”, am considerat că problemele prezentate sunt mai simple comparativ cu problemele pe care trebuia să le rezolv la alte materii. Însă, după ce am învățat mai multe despre algoritmii greedy, am înțeles că partea complicată vine atunci când trebuie să aflăm cea mai bună soluție pentru un anumit input.

Prin alegerea acestei teme, am știut că voi aprofunda și voi înțelege mai bine cum se creează soluțiile optime și totodată voi putea să demonstrez că acea soluție optimă este cu adevărat cea mai bună soluție.

Intenție

În cadrul lucrării voi face o scurtă introducere în paradigma de programare Greedy, urmată de mai multe probleme care se rezolvă cu ajutorul acesteia, printre care și Problema Bancnotelor.

Totodată, voi prezența limbajul Dafny cât și implementarea algoritmului pentru rezolvarea problemei alese de mine.

Introducere

Lucrarea va fi structurată în felul următor:

1. Paradigma Greedy - în acest capitol voi prezenta paradigma Greedy împreună cu câteva probleme și aplicațiile acestora
2. Problema Bancnotelor - în acest capitol voi descrie problema pe care am ales-o pentru licență
3. Limbajul de programare Dafny - în acest capitol voi prezența o introducere în limbajul Dafny
4. Detalii de implementare - în acest capitol voi prezența în detaliu algoritmul implementat

Chapter 1

Paradigma Greedy

Greedy este o strategie de rezolvare a problemelor de optimizare. Metoda presupune luarea unei decizii definitive la fiecare pas în funcție de informațiile cunoscute în prezent fără a ne îngrijora de efectul acesteia în viitor.

1.1 Ce este o problemă de optimizare și cum lucrează metodă Greedy cu aceasta?

O problemă de optimizare este o problema care are un input oarecare dar outputul ei trebuie să fie o valoare maximă sau minimă. În funcție de ce ne cere problema, această poate fi de două feluri:

1. Problema de minimizare $\rightarrow$ rezultatul trebuie să fie de cost minim

Input:
• G = (V,E) - graf hamiltonial care are pe fiecare muchie un cost
Output:
• A $\subseteq \{0, ...m\}$, unde $0 < m <= \|E\|$ iar A este o multime de muchii a caror suma este minima.

În problema de mai sus, avem ca input avem un graf hamiltonian în care fiecare muchie E are un anumit cost. Problema cere să se returneze o mulțime A care conține muchii ale grafului ce formează un ciclu de cost minim.

2. Problema de maximizare → rezultatul trebuie sa fie de câștig maxim

Input: • G = (V,E) - graf neorientat care are pe fiecare muchie un cost • a - nod al grafului G • b - nod al grafului G Output: • A $\subseteq \{0, ...m\}$, unde $0 <= m <= \|E\|$ iar A este o multime de muchii a caror suma este minima.

Problema de mai sus primește ca input un graf în care fiecare muchie E are un câștig și două noduri a și b și se cere să se găsească un drum între cele două noduri care să aibă câștig maxim.

În funcție de decizia făcută de metoda Greedy, vom crea soluții posibile. O soluție posibilă este o submultime care satisface cerințele problemei, în timp ce o soluție optimă respectă cerințele date și are cost minim sau câștig maxim.

Dacă mai multe soluții îndeplinesc criteriile date, atunci acele soluții vor fi considerate ca fiind posibile, în timp ce soluția optimă este cea mai bună dintre toate soluțiile.

Drept urmare, această metodă este folosită pentru a determina soluția oprimă și a rezolva corect problema de optimizare.

1.2 Algoritmi Greedy

1.2.1 Problema Selecției Activităților

Problema activităților este o problemă de maximizare care presupune alegerea cât mai multor activități care pot fi realizate într-un anumit interval de timp fără că acestea să se suprapună.

O posibilă formulare a acestei probleme este:

Input:

- n - numărul de activități,
- aStart[0..n-1] - vector care conține timpul la care încep activitățile,
- aSfarsit[0..n-1] - vector ce conține timpul la care se termină actvitatile astfel încât aStart[i] $<$ aSfarsit[i] pentru orice $0 \leq i \leq$ n-1

Output:

- A $\subseteq$ { 0, ..., n-1 }, unde A este o mulțime de activități care nu se suprapun și este de cardinal maxim;

Spre exmplu putem avea această mulțime de activități reprezentate în tabelul de mai jos:

Nr	Activitate	Început	Final
1	Curs ML	8	10
2	Seminar AI	10	12
3	Curs Pian	11	1
4	Antrenament Înot	12	17
5	Curs CN	16	18
6	Consultații RPA	17	19
7	Laborator CN	18	20

Soluția optimă este $A = \{1, 2, 4, 6\}$

1.2.2 Problema Codurilor Huffman

1.2.3 Problema Bin-packing

1.2.4 Algoritmul Dijkstra

1.2.5 Problema arborelui parțial de cost minim

1.3 Avantaje și dezavantaje ale algoritmilor Greedy

Chapter 2

Problema Bancnotelor

2.1 Ce este Problema Bancnotelor?

Problema bancnotelor este o problemă de minimizare de care ne lovim zilnic. Știind că avem o mulțime de bancnote $B = \{b_1, b_2, b_3\}$ și o suma S de plătit, rezultatul problemei va fi mulțimea de bancnote de cardinal minim pe care o putem folosi pentru a plăti suma S.

2.1.1 Formularea problemei

În limbaj natural, Problema Bancnotelor poate fi formulată astfel:

Se dă o mulțime de bancnote $B = \{b_1, b_2, b_3\}$ și o suma S pe care trebuie să o plătim. Se cere să se afișeze o mulțime de bancnote care trebuie să îndeplinească următoarele condiții:

- cardinalul mulțimii să fie minim,
- suma elementelor mulțimii să fie egata cu S;

Formularea computațională a problemei:

Input:

- un număr natural n - suma care trebuie plătită,

Output:

- numerele $n_{500}, n_{200}, n_{100}, n_{50}, n_{20}, n_{10}, n_5, n_1$ (n_i - numărul de bancnote i folosite), astfel încât $\sum_{i \in \{500,200,100,50,20,10,5,1\}} n_i$ să fie minimă și $n = \sum_{i \in \{500,200,100,50,20,10,5,1\}} i \times n_i$.

Spre exemplu, fiind date bancnotele $B = \{100, 50, 10, 5, 1\}$ și $s = 157$ suma care trebuie plătită.

Soluția optimă pentru această formulare a problemei este $\{1, 1, 0, 1, 2\}$ deoarece $s = 1 \cdot 100 + 1 \cdot 50 + 0 \cdot 10, 1 \cdot 5 + 1 \cdot 2$.

2.2 Strategia Greedy aleasă

Strategia Greedy constă în alegerea, la fiecare pas, a bancnotei cele mai mari, care este mai mică sau egală cu suma pe care trebuie să o plătim. În funcție de alegerea făcută, va trebui să scădem valoarea bancnotei alese din suma care trebuie plătită pentru a ne asigura că avansăm progresiv către obiectivul final.

Folosind exemplul de mai sus, prima decizie făcută de strategia noastră este alegerea bancnotei de 100 deoarece $100 \leq 157$. Decizia făcută are ca rezultat adăugarea valorii 1 la soluția finală iar suma pe care trebuie să o plătim scade cu 100, astfel încât la pasul următor va trebui să o plătim 57. Acest algoritm se va repetă până când suma va fi 0.

2.3 Limitarea Strategiei Greedy

Limitarea strategiei alese este că există posibilitatea ca acesta să nu ofere soluția optimă pentru toate datele de intrare.

De exemplu, dacă mulțimea de bancnote este formată din $B = \{1, 6, 9\}$ iar suma care trebuie plătită este $s = 12$, soluția optimă generată de strategia aleasă va fi $\{3, 0, 1\}$. Cu toate acestea, soluția cu adevărat optimă va fi $\{0, 2, 0\}$.

Motivul pentru care se produce această "eroare" este că soluția este construită pas cu pas. La fiecare pas, se alege bancnotă care va crea cel mai mic cost pentru soluția curentă chiar dacă în viitor ar există o altă soluție cu cost mai mic.

Chapter 3

Dafny

3.1 Limbajul de programare Dafny

Dafny este un limbaj de programare funcțional și imperativ care prin intermediul adnotarilor sale permite crearea unui program care să nu conțină erori la rulare și să facă ceea ce își dorește programatorul.

Un exemplu de adnotare este $requires\ a >= 10$. Astfel, suntem siguri că variabila a nu va fi mai mică decât 10 scutindu-ne de o structură de decizie pe care ar fi trebui să o utilizăm pentru a verifica acest lucru. Alte erori care pot fi evitate prin folosirea adnioarilor sunt erorile de indexare, împărțirea la 0, valori nule etc.

3.2 Avantajele folosirii limbajului Dafny

Precondițiile și postcondițiile din acest limbaj de programare stabilesc ce condiții trebuie să fie adevărate la intrare, respectiv, la ieșirea din metodă. Astfel, programele în Dafny sunt verificate pentru corectitudinea globală astfel încât fiecare rulare se va termina și se va ajunge la rezultatul dorit. În plus, adăugarea acestor condiții duc la o înțelegere în detaliu a codului.

Chapter 4

Detalii de implementare

În acest capitol voi prezenta detaliile de implementare a algoritmului cât și modul în care am demonstrat corectitudinea codului utilizând Dafny.

4.1 Reprezentarea datelor de intrare și a celor de ieșire

4.1.1 Varibilele folosite, tipurile și semnificația acestora

- suma : $int \rightarrow$ reprezintă suma pe care trebuie să o plătim
- rest : $int \rightarrow$ reprezintă suma care ne-a rămas de plătit după ce am ales o bancnotă
- solutieFinala : $seq < int > \rightarrow$ reprezintă soluția finală a problemei
- solutieCurenta : $seq < int > \rightarrow$ reprezintă soluția creată până la pasul curent
- solutieOarecare : $seq < int > \rightarrow$ reprezintă o soluție pe care am folosit-o pentru a demonstra corectitudinea programului

4.1.2 Descrierea soluției rezultate

Bancnotele folosite de mine pentru rezolvarea acestei probleme sunt $B = \{1, 5, 10, 20, 50\}$, astfel, soluția problemei va fi o secvență de numere naturale $soluție = \{n_1, n_5, n_{10}, n_{20}, n_{50}\}$ unde n_i va numărul de i bancnote folosite iar

$$n_1 \cdot 1 + n_5 \cdot 5 + n_{10} \cdot 10 + n_{20} \cdot 20 + n_{50} \cdot 50 == suma.$$

4.2 Implementarea algoritmului Greedy

4.2.1 Predicate și funcții

Înainte de a trece la implementarea propriu-zisă a algoritmului, o să încep prin a prezenta predicatele care asigură corectitudinea rezultatului întors de către metoda ce generează soluția optimă pentru suma pe care trebuie să o plătim.

Predicatele sunt metode care returnează o valoare de adevăr și sunt folosite pentru a verifică o anumită proprietate sau mai multe prin intermediul unei singure instrucțiuni.

În Dafny, funcțiile sunt asemănătoare cu funcțiile matematice. Funcțiile conțin o singură expresie care returnează o valoare de tipul declarat in antetul funcției.

1. Predicatul $esteSolutieValida(solutie : seq < int >) \rightarrow$ acest predicat ne asigură că soluția primită ca parametru este de lungime 5 și că fiecare element este mai mare sau egal cu 0. Condiții menționate anterior sunt foarte importante deoarece, având 5 bancnote (1, 5, 10, 20, 50), înseamnă că va trebuie să existe un elemnt al soluției pentru fiecare dintre ele.

```
predicate esteSolutieValida(solutie : seq<int>)
{
  |solutie| == 5 && solutie[0] >= 0 && solutie[1] >= 0 &&
     solutie[2] >= 0 && solutie[3] >=0 && solutie[4] >=0
}
```

2. Predicatul $esteSolutie(solutie : seq < int >, suma : int) \rightarrow$ acest predicat verifică faptul că suma elementelor secvenței $soluție$ înmulțite cu bancnotele folosite în problemă este egală cu $suma$. Dacă cele două sunt egale, atunci secvență $soluție$ devine soluție posibilă pentru $suma$.

```
predicate esteSolutie(solutie : seq<int>, suma : int)
  requires esteSolutieValida(solutie)
{
  solutie[0] * 1 + solutie[1] * 5 + solutie[2] * 10 +
     solutie[3] * 20 + solutie[4] * 50 == suma
}
```

3. Funcția $cost(solutie : seq < int >) : int \rightarrow$ această funcție calculează și returnează câte bancnote au fost folosite pentru a formă soluția posibilă memorată în variabila $solutie$ dată ca parametru.

```
function cost(solutie : seq<int>) : int
  requires esteSolutieValida(solutie)
{
  solutie[0] + solutie[1] + solutie[2] + solutie[3] +
     solutie[4]
}
```

4. Predicatul $esteSolutieOptima(solutie : seq < int >, suma : int) \rightarrow$ acest predicat va returnat valoarea de adevăr dacă soluția primită ca parametru este o soluție posibilă pentru suma pe care trebuie să o plătim și orice altă soluție posibilă are costul mai mare sau egal. Acest lucru se întâmplă deoarece Problema Bancnotelor este o problemă de minimizare iar soluția trebuie să aibe costul minim.

```
predicate esteSolutieOptima(solutie : seq<int>, suma : int)
  requires esteSolutieValida(solutie)
{
  esteSolutie(solutie, suma) &&
  forall solutieOarecare ::
     esteSolutieValida(solutieOarecare) &&
     esteSolutie(solutieOarecare, suma)
  ==> cost(solutieOarecare) >= cost(solutie)
}
```

5. Predicatul $INV(rest : int, suma : int, solutieFinala : seq < int >) \rightarrow$ acest predicat va fi adevărat dacă orice soluție care este validă este și soluție posibilă pentru $rest$ (suma care ne-a rămas de plătit) atunci suma dintre această soluție și cea dată ca parametru trebuie să fie soluție posibilă pentru suma totală care trebuie plătită. Similar se întâmplă și pentru a demosntra că suma celor două soluții este soluție optimă pentru $suma$.

```
predicate INV(rest : int, suma : int, solutieFinala :
   seq<int>)
  requires esteSolutieValida(solutieFinala)
```

```
    {
      forall solutieCurenta :: esteSolutieValida(solutieCurenta)
         ==>
      (esteSolutie(solutieCurenta, rest) ==>
      esteSolutie([solutieFinala[0] + solutieCurenta[0],
         solutieFinala[1] + solutieCurenta[1],
      solutieFinala[2] + solutieCurenta[2], solutieFinala[3] +
         solutieCurenta[3], solutieFinala[4] +
         solutieCurenta[4]], suma)) &&
      (esteSolutieOptima(solutieCurenta, rest) ==>
      esteSolutieOptima([solutieFinala[0] + solutieCurenta[0],
         solutieFinala[1] + solutieCurenta[1],
      solutieFinala[2] + solutieCurenta[2], solutieFinala[3] +
         solutieCurenta[3], solutieFinala[4] +
         solutieCurenta[4]], suma))
    }
```

4.2.2 Algoritmul Greedy

```
method nrMinimBancnote(suma : int) returns (solutie : seq<int>)
  requires suma >= 0
  ensures esteSolutieValida(solutie)
  ensures esteSolutie(solutie, suma)
  ensures esteSolutieOptima(solutie, suma)
{
 var rest := suma;
 var s1 := 0;
 var s5 := 0;
 var s10 := 0;
 var s20 := 0;
 var s50 := 0;
 while(rest > 0)
    decreases rest
    invariant 0 <= rest <= suma
    invariant esteSolutie([s1, s5, s10, s20, s50], suma - rest)
```

```
    invariant INV(rest, suma, [s1, s5, s10, s20, s50])
{
 var i := 0;
 var s := gasireMaxim(rest);
 if( s == 1)
 {
    cazMaxim1(rest, suma, [s1, s5, s10, s20, s50]);
    s1 := s1 + 1;
    assert esteSolutie([s1, s5, s10, s20, s50], suma - (rest - 1));
    assert INV(rest - 1, suma, [s1, s5, s10, s20, s50]);
 }
 else if(s == 5)
 {
    cazMaxim5(rest, suma, [s1, s5, s10, s20, s50]);
    s5 := s5 + 1;
    assert esteSolutie([s1, s5, s10, s20, s50], suma - (rest - 5));
    assert INV(rest - 5, suma, [s1, s5, s10, s20, s50]);

 }
 else if (s == 10)
 {
    cazMaxim10(rest, suma, [s1, s5, s10, s20, s50]);
    s10 := s10 + 1;
    assert esteSolutie([s1, s5, s10, s20, s50],
          suma - (rest - 10));
    assert INV(rest - 10, suma, [s1, s5, s10, s20, s50]);
 }
 else if(s == 20)
 {
    cazMaxim20(rest, suma, [s1, s5, s10, s20, s50]);
    s20 := s20 + 1;
    assert esteSolutie([s1, s5, s10, s20, s50], suma - (rest - 20));
     assert INV(rest - 20, suma, [s1, s5, s10, s20, s50]);
 }
 else
 {
```

```
    cazMaxim50(rest, suma, [s1, s5, s10, s20, s50]);
    s50 := s50 + 1;
    assert esteSolutie([s1, s5, s10, s20, s50], suma - (rest - 50));
    assert INV(rest - 50, suma, [s1, s5, s10, s20, s50]);
  }
   rest := rest - s;
  }
  solutie := [s1, s5, s10, s20, s50];
}
```

Algoritmul propus de mine primește ca parametru suma pe care va trebui să o plătim și returnează o soluție care respectă proprietățile discutate mai sus în secțiunea 4.1.2 *Descrierea soluției rezultate*.

Preconditiile sunt expresii booleene care trebuie să fie adevărate pentru variabilele date ca parametru, în timp ce postconditiile trebuie să fie adevărate pentru variabilele returnate de metodă. Acestea sunt adăugate la începutul metodei pentru a determina o bună funcționare a programului și totodată corectitudinea acestuia.

Preconditia $requires\ \ suma >= 0$ ne indică faptul că suma trebuie să fie mai mare sau egală cu 0 pentru a se putea realiza metoda.

```
  ensures esteSolutieValida(solutie)
  ensures esteSolutie(solutie, suma)
  ensures esteSolutieOptima(solutie, suma)
```

Aceste trei postconditii se focusează pe soluția returnată de metodă. Soluția trebuie să fie valida, să fie o soluție posibilă pentru suma care trebuie plătită și totodată să fie și soluție oprimă pentru aceasta.

După îndeplinirea cu succes a preconditiilor și a postconditiilor cerute, vom intra în corpul metodei.

La început, vom declara 5 variabile $s_1, s_5, s_{10}, s_{20}, s_{50}$ pe care le inițializăm cu valoarea 0. Aceste variabile vor memora câte bancnote de fiecare tip sunt folosite.

În bucla $while$ se va calcula, la fiecare pas, cu ajutorul uneii metodei $gasireMaxim(rest)$ valoarea bancnotei maxime care este mai mică sau egală cu $rest$. În funcție de valoarea returnată de această metodă, se va incrementa valoarea variabilei corespunzătoare declarată la începutul metodei, va scădea valoarea variabilei $rest$ cu valoarea banc-

notei alese la pasul curent și se va apela o anumită lemma care ajută la demonstrarea corectitudinii programului însă voi reveni ulterior la acest aspect.

Un alt lucru interesant la acest limbaj de programare este prezența invariantilor la începutul buclelor. Invarianții sunt, asemenea postconditiilor și preconditiilor, proprietăți care trebuie respectate pentru a se realiza cu succes bucla while.

```
decreases rest
invariant 0 <= rest <= suma
invariant esteSolutie([s1, s5, s10, s20, s50], suma - rest)
invariant INV(rest, suma, [s1, s5, s10, s20, s50])
```

Acești invarianti ne indică faptul că $rest$ își va schimba valoarea și va descrește dar va fi mereu mai mare decât 0 și mai mică sau egală decât $suma$ și că soluția creată până la pasul curent este o soluție posibilă pentru $suma - rest$. Totodată, vom folosi predicatul INV pentru a verifica fapul că soluția noastră este soluție optimă.

4.2.3 Leme importante în demonstrarea corectitudinii

Lemele sunt metode care sunt folosite pentru a verifica corectitudinea programului. Acestea conțin precondiții iar proprietatea care trebuie demonstrată va fi postconditia lemei.

Uneori, lemele pot fi demonstrate de către Dafny fără a adăuga instrucțiuni, ceea ce înseamnă că un corp fără instrucțiuni servește că argument pentru demonstrație.

În prezentarea agloritmului Greedy creat de mine, în funcție de bancnotă aleasă, va fi apelată o lema care va verifica dacă alegerea făcută va duce la o soluție optimă. Spre exemplu, pentru $s = 1$, unde s este valoarea returnată de metoda $gasireMaxim(rest)$, se va apela lema $cazMaxim1$, pentru $s = 5$ se va apela lema $cazMaxim5$, iar acest lucru se întâmplă pentru fiecare bancnotă existentă în algoritm.

În continuare, o să prezint lemele $cazMaxim1$ și $cazMaxim50$. Primele patru leme, $cazMaxim1$, $cazMaxim5$, $cazMaxim10$, $cazMaxim20$, sunt similare, drep urmare voi prezenta doar lema $cazMaxim1$. Însă, în subsecțiunea următoare, voi reveni asupra lemelor $cazMaxim10$, $cazMaxim20$ deoarece pentru demonstrarea corectitudinii acestor cazuri a fost nevoie de o abordare diferită.

1. Lema cazMaxim1(rest : int, suma : int, solutieFinala : seq<int>)

```
lemma cazMaxim1(rest : int, suma : int, solutieFinala :
   seq<int>)
  requires rest < 5
  requires esteSolutieValida(solutieFinala)
  requires INV(rest, suma, solutieFinala)
  ensures INV(rest-1, suma, [solutieFinala[0] + 1,
     solutieFinala[1], solutieFinala[2], solutieFinala[3],
     solutieFinala[4]])
{

  forall solutieCurenta | esteSolutieValida(solutieCurenta)
     && esteSolutieOptima(solutieCurenta, rest - 1)
  ensures esteSolutieOptima([solutieFinala[0] +
     solutieCurenta[0] + 1, solutieFinala[1] +
     solutieCurenta[1],
  solutieFinala[2] + solutieCurenta[2], solutieFinala[3] +
     solutieCurenta[3], solutieFinala[4] +
     solutieCurenta[4]], suma)
  {
    assert esteSolutie(solutieCurenta, rest - 1);
    assert esteSolutie([solutieCurenta[0] + 1,
       solutieCurenta[1], solutieCurenta[2],
       solutieCurenta[3], solutieCurenta[4]], rest);

    assert forall solutieOarecare ::
       esteSolutieValida(solutieOarecare) &&
       esteSolutie(solutieOarecare, rest - 1)
    ==> cost(solutieOarecare) >= cost(solutieCurenta);

    assert esteSolutie([solutieFinala[0] + solutieCurenta[0]
       + 1, solutieFinala[1] + solutieCurenta[1],
    solutieFinala[2] + solutieCurenta[2], solutieFinala[3] +
       solutieCurenta[3], solutieFinala[4] +
       solutieCurenta[4]], suma);
```

```
        assert forall solutieOarecare ::
            esteSolutieValida(solutieOarecare) &&
            esteSolutie(solutieOarecare, suma)
        ==> cost(solutieOarecare) >= cost([solutieCurenta[0] +
            solutieFinala[0] + 1, solutieCurenta[1] +
            solutieFinala[1],
        solutieCurenta[2] + solutieFinala[2], solutieCurenta[3] +
            solutieFinala[3], solutieCurenta[4] +
            solutieFinala[4]]);
    }

    assert forall solutieCurenta ::
        esteSolutieValida(solutieCurenta)
    && esteSolutieOptima(solutieCurenta, rest - 1) ==>
    esteSolutieOptima([solutieFinala[0] + solutieCurenta[0] +
        1, solutieFinala[1] + solutieCurenta[1],
    solutieFinala[2] + solutieCurenta[2], solutieFinala[3] +
        solutieCurenta[3], solutieFinala[4] +
        solutieCurenta[4]], suma);
}
```

Am creat un forall statement cu ajutorul căruia vom demonstra că dacă vom alege bancnota de valoare 1, vom crea o soluție oprimă pentru variabila $suma$.

Acest lucru îl vom demonstra cu ajutorul assert-urilor. Cu ajutorul acestora vom verifică ce ”știe” verificatorul Dafny și ce va trebui demonstrat separat cu ajutorul lemelor.

Forall statement-ul despre care am amintit mai sus va crea o soluție numită $solutieCurenta$ despre care știm că este o soluție validă și că este o soluție oprimă pentru $rest - 1$. Spre exemplu, dacă $rest = 4$, $suma = 54$ și $solutieFinala = [0, 0, 0, 0, 1]$ soluția aleasă de forall statement va fi $[3, 0, 0, 0, 0]$ care este soluție optimă pentru 3.

Atfel, știind că $solutieCurenta$ este soluție optimă pentru $rest - 1$, este evidet că aceasta este o soluție posibilă pentru $rest - 1$. Vom încerca să modificăm

$solutieCurenta[0] + 1$, pentru a crea o soluție posibilă pentru $rest$.

Următorul pas este să formăm o nouă soluție prin adunarea celor două soluții cunoscute pentru a crea o soluție oprimă pentru variabila $suma$: $[solutieFinala[0]+ solutieCurenta[0] + 1, solutieFinala[1] + solutieCurenta[1], solutieFinala[2] + solutieCurenta[2], solutieFinala[3] + solutieCurenta[3], solutieFinala[4] + solutieCurenta[4]]$.

2. Lema cazMaxim50(rest : int, suma : int, solutieFinala : seq<int>)

```
lemma cazMaxim50(rest : int, suma : int, solutieFinala :
   seq<int>)
  requires rest >= 50
  requires esteSolutieValida(solutieFinala)
  requires INV(rest, suma, solutieFinala)
  ensures INV(rest - 50, suma, [solutieFinala[0],
     solutieFinala[1], solutieFinala[2], solutieFinala[3], 1 +
     solutieFinala[4]])
{
  assert forall solutieCurenta ::
     esteSolutieValida(solutieCurenta) ==>
  (esteSolutie(solutieCurenta, rest) ==>
  esteSolutie([solutieFinala[0] + solutieCurenta[0],
     solutieFinala[1] + solutieCurenta[1], solutieFinala[2] +
     solutieCurenta[2], solutieFinala[3] +
     solutieCurenta[3],solutieFinala[4] + solutieCurenta[4]],
     suma));

  forall solutieCurenta | esteSolutieValida(solutieCurenta)
  && esteSolutie(solutieCurenta, rest - 50)
    ensures esteSolutie([solutieFinala[0] + solutieCurenta[0],
       solutieFinala[1] + solutieCurenta[1], solutieFinala[2] +
       solutieCurenta[2], solutieFinala[3] + solutieCurenta[3],
       1 + solutieFinala[4] + solutieCurenta[4]], suma)
  {
    assert esteSolutie([solutieCurenta[0], solutieCurenta[1],
       solutieCurenta[2], solutieCurenta[3], 1 +
```

```
      solutieCurenta[4]], rest);
}
forall solutieCurenta | esteSolutieValida(solutieCurenta)
&& esteSolutieOptima(solutieCurenta, rest - 50)
  ensures esteSolutieOptima([solutieFinala[0] +
      solutieCurenta[0], solutieFinala[1] + solutieCurenta[1],
      solutieFinala[2] + solutieCurenta[2], solutieFinala[3] +
      solutieCurenta[3], 1 + solutieFinala[4] +
      solutieCurenta[4]], suma)
{
  assert esteSolutie(solutieCurenta, rest - 50);
  assert esteSolutie([solutieCurenta[0], solutieCurenta[1],
      solutieCurenta[2], solutieCurenta[3], 1 +
      solutieCurenta[4]], rest);

  assert forall solutieOarecare ::
      esteSolutieValida(solutieOarecare)
  && esteSolutie(solutieOarecare, rest - 50)
  ==> cost(solutieOarecare) >= cost(solutieCurenta);

  assert esteSolutie([solutieFinala[0] + solutieCurenta[0],
      solutieFinala[1] + solutieCurenta[1], solutieFinala[2] +
      solutieCurenta[2], solutieFinala[3] + solutieCurenta[3],
      1 + solutieFinala[4] + solutieCurenta[4]], suma);

  forall solutieOarecare | esteSolutieValida(solutieOarecare)
  && esteSolutie(solutieOarecare, suma)
    ensures cost(solutieOarecare) >= cost([solutieFinala[0] +
        solutieCurenta[0], solutieFinala[1] +
        solutieCurenta[1], solutieFinala[2] +
        solutieCurenta[2], solutieFinala[3] +
        solutieCurenta[3], 1 + solutieFinala[4] +
        solutieCurenta[4]])
  {
    solutieFianalaAreCostMinim(rest, suma, solutieOarecare,
        solutieFinala, solutieCurenta);
```

```
  }
  assert forall solutieOarecare ::
     esteSolutieValida(solutieOarecare)
  && esteSolutie(solutieOarecare, suma)
  ==> cost(solutieOarecare) >= cost([solutieFinala[0] +
     solutieCurenta[0], solutieFinala[1] + solutieCurenta[1],
     solutieFinala[2] + solutieCurenta[2], solutieFinala[3] +
     solutieCurenta[3], 1 + solutieFinala[4] +
     solutieCurenta[4]]);
}
assert forall solutieCurenta ::
   esteSolutieValida(solutieCurenta)
&& esteSolutieOptima(solutieCurenta, rest - 50) ==>
esteSolutieOptima([solutieFinala[0] + solutieCurenta[0],
   solutieFinala[1] + solutieCurenta[1], solutieFinala[2] +
   solutieCurenta[2], solutieFinala[3] + solutieCurenta[3], 1
   + solutieFinala[4] + solutieCurenta[4]], suma);

assert forall solutieCurenta ::
   esteSolutieValida(solutieCurenta) ==>
(esteSolutie(solutieCurenta, rest - 50) ==>
esteSolutie([solutieFinala[0] + solutieCurenta[0],
   solutieFinala[1] + solutieCurenta[1], solutieFinala[2] +
   solutieCurenta[2], solutieFinala[3] + solutieCurenta[3], 1
   + solutieFinala[4] + solutieCurenta[4]], suma));

assert forall solutieCurenta ::
   esteSolutieValida(solutieCurenta) ==>
(esteSolutieOptima(solutieCurenta, rest - 50) ==>
esteSolutieOptima([solutieFinala[0] + solutieCurenta[0],
   solutieFinala[1] + solutieCurenta[1], solutieFinala[2] +
   solutieCurenta[2], solutieFinala[3] + solutieCurenta[3], 1
   + solutieFinala[4] + solutieCurenta[4]], suma));

assert INV(rest - 50, suma, [solutieFinala[0],
   solutieFinala[1], solutieFinala[2], solutieFinala[3], 1 +
```

```
        solutieFinala[4]]);
}
```

Acesta lemă are rolul de a demonstra că prin alegerea bancnotei de 50 se va crea o soluție optimă pentru $suma$.

Știm că pentru orice soluție validă și posibilă pentru $rest$, suma dintre această soluție și $solutieFinala$ va fi o soluție posibilă pentru $suma$. Astfel, folosind un forall statement vom demonstra, similiar ca în cazul precedent, că prin însumarea secevetei $solutieFinala$ cu orice altă secvență $solutieCurenta$, care este soluție posibilă pentru $rest - 50$, vom crea o nouă soluție posibilă pentru $suma$.

Următorul lucru care trebuie să îl demonstrăm este că suma elementelor celor două soluții este soluție optimă pentru $suma$. Pentru această demonstrație, am folosit un exchange argument pe care îl voi prezenta în subsecțiunea următoare.

4.2.4 Exchange Arguments

Exchange arguments este o tehnică folosită pentru a demonstra optimitatea unei soluții. Tehnica presupune modificarea unei soluții oarecare cu scopul de a obține o soluție optimă pentru agloritmul greedy fără a-i modifica costul.

După cum am menționat anterior, pentru lemele $cazMaxim10$,$cazMaxim20$ și $cazMaxim50$ demonstrarea faptului că suma dintre $solutieCurenta$ și $solutieFinala$ este soluție optimă pentru $suma$ nu a mers "de la sine"ca în celalte leme, așa că va trebui "să ajutăm" verificatorul Dafny.

În programul scris de mine, există două situații în care intervine folosirea acestei tehnici:

- variabila $rest$ aparține unui interval, cum ar fi $10 <= rest < 20$ si $20 <= rest < 50$
- variabila $rest$ are doar limită inferioară, cum ar fi $rest >= 50$.

  În ceea ce urmează o să prezint câte un caz din fiecare situație.

1. Situația în care variabila $10 <= rest < 20$ este reprezentată de lema exchangeArgumentCaz10(rest : int, solutieCurenta : seq<int>)

```
lemma exchangeArgumentCaz10(rest : int, solutieCurenta :
   seq<int>)
  requires 10 <= rest < 20
  requires esteSolutieValida(solutieCurenta)
  requires esteSolutieOptima(solutieCurenta, rest - 10)
  ensures esteSolutieOptima([solutieCurenta[0],
     solutieCurenta[1], solutieCurenta[2] + 1,
     solutieCurenta[3], solutieCurenta[4]], rest)
{
  assert esteSolutie([solutieCurenta[0], solutieCurenta[1],
     solutieCurenta[2] + 1, solutieCurenta[3],
     solutieCurenta[4]], rest);
  if(!esteSolutieOptima([solutieCurenta[0], solutieCurenta[1],
     solutieCurenta[2] + 1, solutieCurenta[3],
     solutieCurenta[4]], rest))
  {
    var solutieOptima :|esteSolutieValida(solutieOptima) &&
       esteSolutie(solutieOptima, rest) && cost(solutieOptima)
       < cost([solutieCurenta[0], solutieCurenta[1],
       solutieCurenta[2] + 1, solutieCurenta[3],
       solutieCurenta[4]]);
    assert cost([solutieCurenta[0], solutieCurenta[1],
       solutieCurenta[2] + 1, solutieCurenta[3],
       solutieCurenta[4]]) == cost(solutieCurenta) + 1;
    assert solutieOptima[3] == 0;
    assert solutieOptima[4] == 0;
    if(solutieOptima[2] >= 1)
    {
      var solutieOptima' := [solutieOptima[0],
         solutieOptima[1], solutieOptima[2] - 1,
         solutieOptima[3], solutieOptima[4]];
      assert esteSolutie(solutieOptima', rest - 10);
      assert cost(solutieOptima') == cost(solutieOptima) - 1;
      assert cost(solutieOptima) - 1 < cost(solutieCurenta);
      assert false;
```

```
}
else if(solutieOptima[1] >= 2)
{
  var solutieOptima' := [solutieOptima[0], solutieOptima[1]
     - 2, solutieOptima[2], solutieOptima[3],
     solutieOptima[4]];
  assert esteSolutie(solutieOptima', rest - 10);
  assert cost(solutieOptima') == cost(solutieOptima) - 2;
  assert cost(solutieOptima) - 2 < cost(solutieCurenta);
  assert false;
}else if(solutieOptima[1] >= 1 && solutieOptima[0] >= 5)
{
  var solutieOptima' := [solutieOptima[0] - 5,
     solutieOptima[1] - 1, solutieOptima[2],
     solutieOptima[3], solutieOptima[4]];
  assert esteSolutie(solutieOptima', rest - 10);
  assert cost(solutieOptima') == cost(solutieOptima) - 6;
  assert cost(solutieOptima) - 6 < cost(solutieCurenta);
  assert false;
}
else if(solutieOptima[0] >= 10)
{
  var solutieOptima' := [solutieOptima[0] - 10,
     solutieOptima[1], solutieOptima[2], solutieOptima[3],
     solutieOptima[4]];
  assert esteSolutie(solutieOptima', rest - 10);
  assert cost(solutieOptima') == cost(solutieOptima) - 10;
  assert cost(solutieOptima) - 10 < cost(solutieCurenta);
  assert false;
}
else{
  assert false;
}}}
```

Cu ajutorul acestei leme vom demonstra faptul că soluția curentă construită astfel $[solutieCurenta[0], solutieCurenta[1], solutieCurenta[2]+1, solutieCurenta[3],$

$solutieCurenta[4]]$ este soluție optimă pentru variabila $rest$.

Pentru a demonstra acest lucru, vom crea o nouă soluție numită $solutieOptima$ care are cost mai mic decât soluția curentă. Știm deja că elementele $solutieOptima[3] == 0$ și $solutieOptima[4] == 0$ așa că ne putem concentra atenția pe primele 3 elemente ale soluției.

În cele ce urmează, vom căuta toate combinațiile formate din numerele 1, 5 și 10 a căror sumă este 10. Pentru fiecare combinație se va crea o nouă soluție $solutieOprima'$ astfel: se vor scădea bancnotele cu valoarea respectivă din $solutieOptima$.

De exemplu, pentru cazul $1 + 1 + 1 + 1 + 1 + 5 = 10$ noua soluție va fi $varsolutieOptima' := [solutieOptima[0]-5, solutieOptima[1]-1, solutieOptima[2], solutieOptima[3], solutieOptima[4]]$, cu alte cuvinte am scăzut 5 bancnote de valoare 1 și o bancnotă de valoare 5.

Știind că am scăzut n bancnote din $solutieOptima$ pentru a crea $solutieOptima'$ costul acesteia va fi $cost(solutieOptima) - n$, iar încercând să comparăm acest cost cu costul soluției curente vom ajunge la o contradicție.

2. Situația în care variabila $rest <= 50$ este reprezentată de lema exchangeArgumentCaz50(rest : int, suma : int, solutieOarecare : seq<int>, solutieCurenta : seq<int>)

   Codul pentru această lema este foarte lung și nu îl voi insera pe tot în acest document. Voi prezența anumite instrucțiuni care au o importantă deosebită, iar secvențele repetitive le voi înlocui cu un comentariu sugestiv.

   Acesta lemă are drept scop demonstrarea faptului că variabila $solutieCurenta$ are un cost mai mic decât costul variabilei $solutieOarecare$.

```
assert esteSolutie(solutieOarecare, rest);
assert esteSolutie([solutieCurenta[0], solutieCurenta[1],
   solutieCurenta[2], solutieCurenta[3], 1 +
   solutieCurenta[4]], rest);
if(cost(solutieOarecare) < cost([solutieCurenta[0],
   solutieCurenta[1], solutieCurenta[2], solutieCurenta[3], 1 +
   solutieCurenta[4]]))
```

```
{
  if(solutieOarecare[4] > solutieCurenta[4] + 1)
  {
    assert cost([solutieOarecare[0], solutieOarecare[1],
       solutieOarecare[2], solutieOarecare[3],
       solutieOarecare[4] - 1]) < cost(solutieCurenta);
    assert esteSolutieOptima([solutieOarecare[0],
       solutieOarecare[1], solutieOarecare[2],
       solutieOarecare[3], solutieOarecare[4] - 1], rest - 50);
    assert false;
  }
  else if(solutieOarecare[4] < solutieCurenta[4] + 1)
  {
    assert (solutieOarecare[0] + (5 * solutieOarecare[1])+(10 *
       solutieOarecare[2]) + (20 * solutieOarecare[3])) >= 50;

    if(solutieOarecare[2] >= 1 && solutieOarecare[3] >= 2)
    {
      var nouaSolutieOarecare := [solutieOarecare[0],
         solutieOarecare[1], solutieOarecare[2] - 1,
         solutieOarecare[3] - 2, solutieOarecare[4] + 1];
      exchangeArgumentCaz50(rest, suma, nouaSolutieOarecare,
         solutieCurenta);
    }
    //combinatiile de 1,5,10 si 20 a caror suma este 50
    else{
      assert solutieOarecare[0] >= 0;
      assert solutieOarecare[1] >= 0;
      assert solutieOarecare[2] >= 0;
      assert solutieOarecare[3] >= 3;
      if(solutieOarecare[3] >= 3)
      {
        var nouaSolutieOarecare := [solutieOarecare[0],
           solutieOarecare[1], solutieOarecare[2] + 1,
           solutieOarecare[3] - 3, solutieOarecare[4] + 1];
        assert cost(nouaSolutieOarecare) <
```

```
            cost(solutieOarecare);
          exchangeArgumentCaz50(rest, suma, nouaSolutieOarecare,
            solutieCurenta);
        }
      }
    }
    assert solutieOarecare[4] == (solutieCurenta[4] + 1);
```

Începem demonstrația prin a verifica dacă $solutieOarecare$ are costul mai mic decât $cost(solutieCurenta)$. În caz afirmativ, vom începe să comparăm elementele din cele două soluții: $solutieOarecare[4] > solutieCurenta[4] + 1$, $solutieOarecare[3] > solutieCurenta[3]$, $solutieOarecare[2] > solutieCurenta[2]$, $solutieOarecare[1] > solutieCurenta[1]$ și $solutieOarecare[0] > solutieCurenta[0]$.

Secventa de cod adăugată mai sus conține doar compararea elementelor de pe poziția 4 deoarece această este cea mai complexă parte.

Dacă $solutieOarecare[4] > solutieCurenta[4] + 1$ atunci soluția $[solutieOarecare[0], solutieOarecare[1], solutieOarecare[2], solutieOarecare[3], solutieOarecare[4] - 1]$ are costul mai mic decât costul soluției curente și este soluție optimă pentru $rest - 50$, lucru care este fals.

În schimb, dacă $solutieOarecare[4] < solutieCurenta[4] + 1$ vom crea o nouă soluție numită $nouaSolutieOarecare$ formată din elementele secvenței $solutieOarecare$ din care am scăzut bancnotele a căror sumă este egală cu 50 și am incrementat valoarea elementului de pe poziția 4, apoi, cu ajutorul unui apel recursiv, vom compara $nouaSolutieOarecare$ cu $solutieCurenta$ până când acestea vor fi egale.

Similar se întâmplă și cu celelalte trei comparații pe care le-am enumerat mai sus. Se vor crea noi soluții până când elementele din $solutieOarecare$ și $solutieCurenta$ vor fi egale.

4.3 Probleme ”interesante” intalnite

Pe parcursul realizării acestui algoritm și a demonstrării corectitudinii sale am întâmpinat mai multe probleme care m-au pus în dificultate.

Problema care m-a "încurcat" cel mai mult am întâmpinat-o la lema exchangeArgumentCaz50(rest: int, suma : int, solutieOarecare : seq<int>, solutieCurenta : seq<int>). După cum am prezentat anterior, pentru a demonstra că $solutieOarecare[4] == solutieCurenta[4]-1$, cu alte cuvinte, dacă ambele soluții conțineau același număr de bancnote de valoare 50, a trebuit să caut toate combinațiile de valori de 1,5,10 și 20 a căror suma este 50 și să creez o nouă soluție.

- Problema 1: După ce am adăugat în lemă toate cele 56 de cazuri posibile, lema nu putea să demonstreze că $solutieOarecare[4] == solutieCurenta[4] + 1$, ceea ce însemna că îmi lipsește un caz.

  Rezulvarea problemei 1: În ultima instrucțiune $else$ am adăugat câteva assert-uri pentru a afla ce caz sau cazuri am ratat. Spre surprinderea mea, $solutieOarecare[0] == solutieOarecare[1] == solutieOarecare[2] == 0$ iar $solutieOarecare[3] == 3$ asta însemna că aveam 60 de lei formați din 3 bancnote de 20 de lei.

- Problema 2: Cum aș putea să tratez acest caz?

  Încercarea 1: Am încercat să verific dacă $solutieOarecare[3] >= 5$ astfel încât să pot adaugă 2 bancnote de 50 de lei $solutieOarecare[4]+2 \rightarrow$ eroare: $solutieOarecare[3]$ nu este mai mare sau egal cu 5.

  Încercarea 2: Am încercat să verific dacă $solutieOarecare[0]+5 \cdot solutieOarecare[1]+10 \cdot solutieOarecare[2] >= 100$ pentru a putea adaugă valoarea 5 la $solutieOarecare[3]$ și să pot reveni la ideea de la "Încercarea 1" $\rightarrow$ eroare: suma elementelor nu este mai mare sau egală cu 100.

  Încercare 3: Am încercat să tratez cazul astfel: cele 3 bancnote de 20 le voi transforma într-o bancnotă de 50 iar bancnota de 10 lei rămasă o voi aduna la celalate pe care le am în soluție, astfel, creând o nouă soluție cu un cost mai mic ($nouaSolutieOarecare := [solutieOarecare[0], solutieOarecare[1], solutieOarecare[2] + 1, solutieOarecare[3] - 3, solutieOarecare[4] + 1]$) $\rightarrow$ eroare: "decreases clase might not decrease"

- Problema 3: Eroarea "decreases clase might not decrease" apăruse deoarece în adnotările lemei inițial exista $decreases\ solutieOarecare[0], solutieOarecare[1], solutieOarecare[2], solutieOarecare[3]$ ceea ce ne asigura că elementele secvenței

$solutieOarecare$ vor descrește. În încercarea mea de a trata cazul rămas, voiam să adaug "restul" în $solutieOarecare[2]$ și încălcam adnotarea specificată mai sus.

Rezolvarea problemei 2: Cum scopul pricipal al lemei era de a demonstra că $solutieCurenta$ are cost minim am modificat adnotarea astfel $decreases$ $solutieOarecare[0] + solutieOarecare[1] + solutieOarecare[2] + solutieOarecare[3]$ deoarece este suficient să descrească suma acestora pentru a ajunge la o soluție cu cost mai mic.

Concluzii

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Nunc mattis enim ut tellus elementum sagittis vitae et. Placerat in egestas erat imperdiet sed euismod. Urna id volutpat lacus laoreet non curabitur gravida. Blandit turpis cursus in hac habitasse platea. Eget nunc lobortis mattis aliquam faucibus. Est pellentesque elit ullamcorper dignissim cras tincidunt lobortis feugiat. Viverra maecenas accumsan lacus vel facilisis volutpat est. Non odio euismod lacinia at quis risus sed vulputate odio. Consequat ac felis donec et odio pellentesque diam volutpat commodo. Etiam sit amet nisl purus in. Tortor condimentum lacinia quis vel eros donec. Phasellus egestas tellus rutrum tellus pellentesque eu tincidunt. Aliquam id diam maecenas ultricies mi eget mauris pharetra. Enim eu turpis egestas pretium.

Bibliografie

1. https://github.com/ElisaChicos/Licenta
2. https://www.jeffyang.io/blog/min_number_of_coins_for_exchange/
3. https://www.baeldung.com/cs/min-number-of-coins-algorithm
4. https://jeffe.cs.illinois.edu/teaching/algorithms/book/04-greedy.pdf
5. https://www.guru99.com/greedy-algorithm.html
6. https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYX Vsd GRvbWFpbnxmaWljb3Vyc2VwYXxneDo2ZTIzNDNhOTZmZmFmZjdk
7. https://brilliant.org/wiki/greedy-algorithm/
8. https://techvidvan.com/tutorials/greedy-algorithm/
9. https://www.hackerearth.com/practice/algorithms/greedy/basics-of -greedy-algorithms/tutorial/
10. https://progressivecoder.com/coin-change-problem-using-greedy-algorithm/
11. https://dafny.org/dafny/OnlineTutorial/guide
12. https://dafny.org/dafny/QuickReference
13. http://www.doc.ic.ac.uk/ scd/Dafny_Material/Lectures.pdf
14. https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/
15. https://arxiv.org/pdf/1701.04481.pdf
16. https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/handouts/120%20Guid%20to%20Greedy%20Algorithms.pdf