

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**Implementarea și verificarea formală în Dafny a
algoritmului greedy pentru o variantă a
problemei bancnotelor**

propusă de

Alexandra Elena Contur

Sesiunea: februarie, 2023

Coordonator științific

Conf. Dr. Ciobâcă Ștefan

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

**Implementarea și verificarea formală în
Dafny a algoritmului greedy pentru o
variantă a problemei bancnotelor**

Alexandra Elena Contur

Sesiunea: februarie, 2023

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 **Contur Alexandra Elena** domiciliat în **România, jud. Iași, com. Holboca, str. Dascălilor, nr. 10**, născut la data de **04 martie 2001**, identificat prin CNP **6010304226743**, 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 și verificarea formală în Dafny a algoritmului greedy pentru o variantă a problemei 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 și verificarea formală în Dafny a algoritmului greedy pentru o variantă a problemei 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 **Alexandra Elena Contur**

Data:

Semnătura:

Cuprins

1	Context	2
2	Implementarea problemei în Dafny	5
2.1	Reprezentarea datelor de intrare și ieșire	5
2.2	Predicate folosite care garantează optimalitatea soluției	5
3	Verificarea formală a implementării algoritmului	7
3.1	Metoda banknoteMinimum	8
3.2	Funcția addValueToIndex	9
3.3	Metoda maxBanknote	10
3.4	Lema banknoteMaxim	11
3.5	Lema banknoteMaxim32	12
3.6	Lema exchangeArgument	13
3.7	Lema exchangeArgument32	16
	Concluzii	18
	Bibliografie	19

Capitolul 1

Context

În cadrul acestei lucrări am ales să implementez și să verific formal algoritmul greedy pentru **problema bancnotelor**. La fiecare pas se selectează bancnota cea mai mare și se adaugă în soluție. Astfel reprezentăm suma primită printr-un număr minim de bancnote.

Problema bancnotelor la intrare primește un număr natural ce reprezintă o sumă și o rezervă infinită de bancnote de valori nominale predefinite, în funcție de cazul particular ales. Problema determină numărul minim posibil de bancnote în care putem reprezenta restul de valoarea sumei. La ieseire, problema returnează o secvență formată din numerele corespunzătoare aparițiilor fiecărei bancnote.

Reprezentarea generală folosită pentru soluție în problema bancnotelor este: $banknote_1 < banknote_2 < \dots < banknote_n$, unde dacă $banknote_1 = x$, avem x bancnote de valoare 1.

Cazul particular ales pentru a fi implementat în această lucrare este problema bancnotelor în care sistemul de bancnote este format din puteri ale lui doi, până la puterea a 5-a ($2^0, 2^1, 2^2, 2^3, 2^4, 2^5$).

Bancnotele posibile în problema bancnotelor cu sistemul de bancnote menționat sunt: $[1, 2, 4, 8, 16, 32]$.

Un algoritm ce folosește **metoda greedy** face alegerea cea mai bună locală la fiecare pas, construind soluția finală, astfel sperând că aceasta este soluția optimă. Această metoda generează de fiecare dată o soluție mult mai bună decât soluția cea mai costisitoare, dar nu oferă soluția optimă pentru orice problemă.

Exemple de probleme pentru care metoda greedy oferă soluția optimă:

- Problema de selecție a activității
- Problema bancnotelor

•Codificarea Huffman

Verificarea formală în Dafny a algoritmului greedy care rezolvă problema bancnotelor demonstrează faptul că soluția construită este optimă pentru orice sumă dată ca input.

Dafny este un limbaj de programare și verificare, care poate verifica corectitudinea funcțională a unui program. Verificarea este posibilă datorită caracteristicilor specifice limbajului precum precondiții, postcondiții, invariante, ș.a.m.d. De asemenea, verificatorul Dafny se asigură că adnotările făcute se îndeplinesc, astfel acesta nu se așteaptă la scrierea unui cod fără erori, ci la scrierea unor adnotări fără erori. [1]

În următoarele rânduri voi explica pe scurt cum funcționează Dafny, folosind un exemplu simplu de cod preluat dintr-un repository public de pe Github, care rezolvă problema Fibonacci. [2].

```
1 function fib (n: nat) : nat
2 {
3   if n == 0 then 0
4   else if n == 1 then 1
5   else fib (n-1) + fib (n-2)
6 }
7
8 method Fib (n: nat) returns (b: nat)
9   ensures b == fib(n);
10 {
11   if (n == 0) { return 0; }
12   var a := 0;
13   b := 1;
14
15   var i := 1;
16   while (i < n)
17     invariant i <= n
18     invariant a == fib(i-1);
19     invariant b == fib(i);
20   {
21     a, b := b, a + b;
22     i := i + 1;
23   }
24 }
```

În exemplul dat putem observa cum metoda Fib are o post condiție *ensures*.

Această postcondiție, dacă se verifică, ne asigură că metoda Fib întradevăr returnează un număr b , care este egal cu rezultatul funcției Fib, care primește ca parametru n .

Dacă am fi folosit un tip de date întreg pentru n , am fi putut avea și o precondiție care să stabilească că n este mai mare decât zero.

Un alt lucru specific limbajului Dafny este prezența celor trei invarianți din bucla while (liniile 17, 18, 19). Acești invarianți au rolul de a se asigura că sunt îndeplinite acele condiții pentru tot parcursul buclei, deoarece Dafny nu poate demonstra de la o iterație la alta ce s-a realizat în buclă.

Capitolul 2

Implementarea problemei în Dafny

2.1 Reprezentarea datelor de intrare și ieșire

În cazul problemei discutate datele de intrare sunt reprezentate de un număr natural **sum** pentru care se verifică dacă se produce soluția finală optimă.

Drept date de ieșire avem soluția de cost minim reprezentată de secvența formată din 6 numere naturale:

- soluție = $b_0, b_1, b_2, b_3, b_4, b_5$, unde $\sum_{k=0}^5 b_k \cdot 2^k = \text{suma}$ [3].

2.2 Predicate folosite care garantează optimalitatea soluției

Soluția este reprezentată ca o secvență de numere naturale, fiind chiar secvența descrisă la secțiunea date de ieșire. Fiecare element din soluție reprezintă numărul de apariții ale bancnotei corespunzătoare în soluție.

Entry point-ul verificării problemei este metoda `banknoteMinimum`, care asigură faptul că soluția returnată este optimă.

- **predicatul `isValidSolution`**: Pentru a fi soluție optimă trebuie ca secvența să fie o soluție validă (cu 6 elemente), care produce suma corectă și care are costul cel mai mic.

```
1 predicate isValidSolution(solution: seq < int > )
2 {
3   hasSixElem(solution) && solution[0] >= 0 && solution[1] >= 0 && solution
4     [2] >= 0 && solution[3] >= 0 && solution[4] >= 0 && solution[5] >= 0
5 }
```

• **predicatul isSolution:** Pentru a fi soluție trebuie ca secvența să fie o soluție validă (cu 6 elemente) și să producă suma corectă.

```
1 predicate isSolution(solution: seq < int > , sum: int)
2     requires isValidSolution(solution)
3 {
4     solutionElementsSum(solution) == sum
5 }
```

• **predicatul isValidSolution:** Pentru a fi soluție validă trebuie ca secvența să aibă 6 elemente (tipuri de bancnote), cu proprietatea că fiecare bancnotă are un număr nul sau pozitiv de apariții în soluție.

```
1 predicate isOptimalSolution(solution: seq < int > , sum: int)
2     requires isValidSolution(solution)
3 {
4     isSolution(solution, sum) &&
5     forall possibleSolution::isValidSolution(possibleSolution) &&
        isSolution(possibleSolution, sum) ==> cost(possibleSolution) >= cost
        (solution)
6 }
```

Capitolul 3

Verificarea formală a implementării algoritmului

În următoarea schemă voi exemplifica modul în care se apelează metodele, funcțiile și lemele pentru a demonstra faptul că implementarea algoritmului greedy rezolva problema bancnotelor și produce soluția optimă.

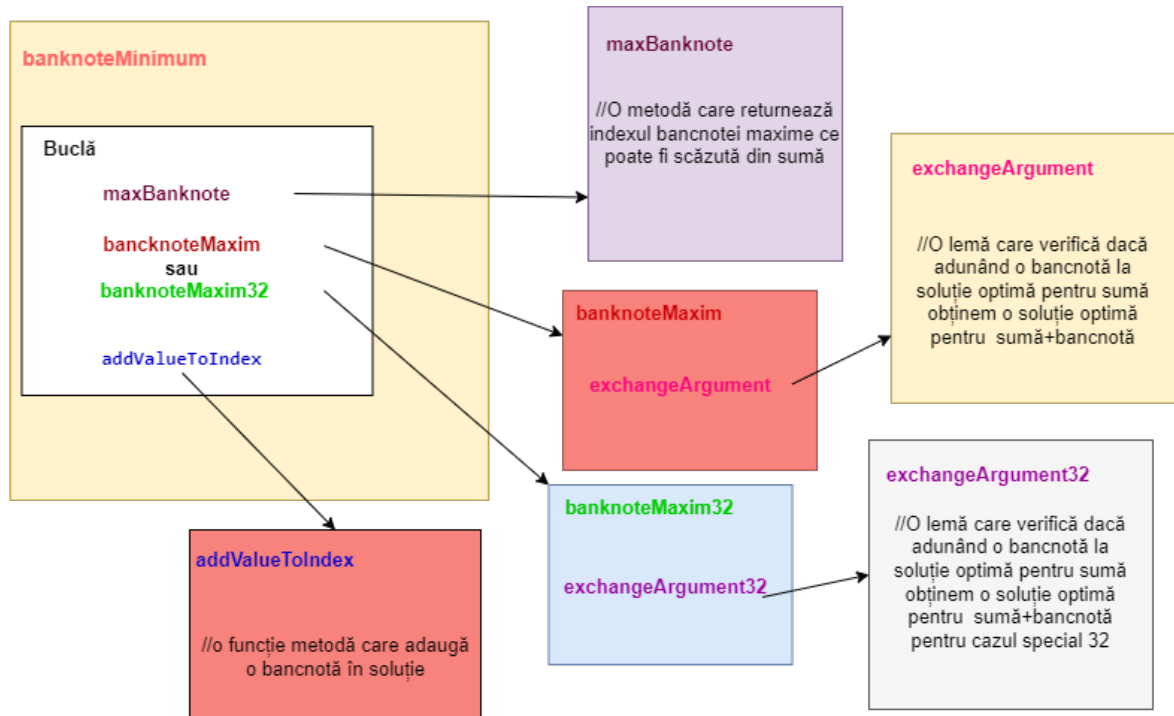


Figura 1: O schemă explicativă a verificării.

3.1 Metoda banknoteMinimum

Metoda banknoteMinimum este metoda principală folosită pentru a returna soluția optimă pentru o sumă dată ca input.

Această metodă are ca precondiții faptul că suma este pozitivă, iar ca postcondiții avem cele 3 predicate menționate anterior care asigură optimalitatea unei soluții.

Metoda folosește o buclă, în care, la fiecare iterație, se adaugă cea mai mare bancnotă mai mică decât sum construind soluția.

În primă fază calculează bancnota cea mai mare ce poate fi dată ca rest, cu ajutorul funcției maxBanknote, detaliată în 3.3 și funcției power, care calculează ridicarea la putere.

Dificultatea principală este de a demonstra că proprietatea de optimalitate se menține pe parcursul buclei.

Invariantii prezenți în while (liniile 12,13,14) au rolul de a menține proprietatea de optimalitate pe parcursul construirii soluției, astfel că la orice pas soluția este optimă pentru $suma - rest$.

Primul invariant are rolul de a se asigura că pe parcursul construirii soluției nu avem de dat rest negativ.

Al doilea invariant, explicat în subcapitolul 2.1, are rolul de a se asigura că soluția construită se menține optimă, aici ne ajuta funcția metodă care garantează că dacă adunăm o anumită bancnotă la soluție, aceasta încă este validă.

Al treilea invariant, pentru care voi da exemplu de cod mai jos, nu poate fi verificat de Dafny așa că acest invariant trebuie demonstrat cu ajutorul lemei banknoteMaxim sau banknoteMaxim32 (detaliată în capitolul 3.4, respectiv 3.6), în funcție de bancnota aleasă.

La final bancnota optimă este adăugată în sumă cu ajutorul funcției addValueToIndex, detaliată în subcapitolul 3.2, iar din rest se scade valoarea bancnotei adăugată în sumă.

```
1  method banknoteMinimum(sum: int) returns(solution: seq <int> )
2      requires sum >= 0
3      ensures isValidSolution(solution)
4      ensures isSolution(solution, sum)
5      ensures isOptimalSolution(solution, sum)
6  {
7      var rest := sum;
```

```

8      solution:= [0, 0, 0, 0, 0, 0, 0];
9      var index:= 0;
10     assert isOptimalSolution(solution, sum - rest);
11     while (0 < rest)
12         invariant 0 <= rest <= sum
13         invariant isValidSolution(solution)
14         invariant addOptimRestEqualsOptimSum(rest, sum, solution)
15         decreases rest
16     {
17         index:= maxBanknote(rest);
18         var banknote:= power(2, index);
19         if (index != 5)
20         {
21             banknoteMaxim(rest, sum, solution, index);
22         }
23         else
24         {
25             banknoteMaxim32(rest, sum, solution);
26         }
27         solution:= addValueToIndex(solution, 1, index);
28         rest:= rest - banknote;
29     }
30 }

```

• **predicatul addOptimRestEqualsOptimSum:** Acest predicat (de la linia 14) are rolul de a verifica dacă o soluție pentru suma x adunată cu o soluție optimă pentru suma y are ca rezultat o soluție optimă pentru $x + y$. Acest lucru ne ajută să verificăm optimalitatea unei soluții fără să fie necesară optimalitatea ambelor soluții ce o formează.

Scopul acestor predicate este de a asigura că secvența ce reprezintă datele de ieșire respectă toate condițiile necesare pentru a fi considerată soluție optimă.

3.2 Funcția addValueToIndex

Funcția addValueToIndex este folosită pentru a adăuga unui element din soluție o valoare, folosindu-ne de index-ul acestuia.

Este utilă pentru construirea soluțiilor.

Aceasta are drept precondiții pentru modificarea corectă a sumei faptul că index-

ul trebuie să fie în codomeniu, soluția să fie validă și faptul că valoarea adăugată plus valoarea inițială a elementului din secvență nu reprezintă un număr negativ.

Postcondițiile acestei funcții ne asigură faptul că avem o soluție validă în urma adăugării și faptul că valoarea sumei pe care o reprezintă soluția a crescut cu valoarea la care ne așteptam.

```
1 function method addValueToIndex(solution: seq<int>, value: int, index: int)
    : seq<int>
2     requires 0 <= index <= 5
3     requires isValidSolution(solution)
4     requires solution[index] + value >= 0
5     ensures isValidSolution(solution)
6     ensures solutionElementsSum(solution) + value*power(2, index) ==
        solutionElementsSum(solution[index:=solution[index]+value])
7 {
8     solution[index:= solution[index] + value]
9 }
```

3.3 Metoda maxBanknote

Metoda maxBanknote este folosită pentru a returna un index cu proprietatea:

$$bancnota_{index} \leq rest < bancnota_{index+1}.$$

Avem postcondițiile următoare: indexul să se afle în codomeniu-ul secvenței, bancnota corespunzătoare indexului să fie mai mică decât suma, și un index nu poate fi mai mare decât a fost calculat fără depăși suma. Aceste postcondiții au fost necesare pentru a demonstra faptul că indexul returnat este întradevăr corespunzător celei mai mari bancnote care poate fi dată rest pentru sum.

```
1     method maxBanknote(sum: int) returns(index: int)
2         requires sum > 0
3         ensures 0 <= index <= 5
4         ensures 0 <= power(2, index) <= sum
5         ensures (index != 5 && power(2, index + 1) > sum) || index == 5
6     {
7         index:= 5;
8         if (power(2, index) > sum)
9         {
10             assert power(2, index + 1) > sum;
11             while (power(2, index) > sum && index > 0)
```

```

12         invariant power(2, index + 1) > sum
13     {
14         index := index - 1;
15         assert power(2, index + 1) > sum;
16     }
17 }
18 else
19 {
20     assert index == 5;
21 }
22 }

```

3.4 Lema banknoteMaxim

Lema banknoteMaxim este tratată diferit de lema banknoteMaxim32 deoarece avem o bancnotă nemărginită superior, bancnota 32, care trebuie tratată separat. Lema banknoteMaxim este folosită pentru a demonstra păstrarea faptului că dacă adăugăm o bancnotă în soluția pe care o construim, în continuare suma acestei soluții cu soluția optimă pentru $rest - bancnota$ va produce soluția optimă pentru suma întreagă.

Pentru a demonstra că o soluție este optimă, încercăm să găsim o soluție cu cost mai mic, dacă nu reușim, atunci știm că soluția găsită este optimă [4]. Avem nevoie să știm că dacă în soluția curentă, optimă pentru $rest - bancnota$ adăugăm bancnota, aceasta devine optimă pentru $rest$. Acest lucru este demonstrat cu ajutorul lemei exchangeArgument. Aceste lemme au nevoie ca indexul să se afle în codomeniu și să fie ales conform metodei maxBanknote, soluția calculată până în prezent să fie validă și să respecte predicatul addOptimRestEqualsOptimSum, așadar acestea sunt precondițiile. Această metodă asigură că predicatul addOptimRestEqualsOptimSum este valid pentru $rest$ dacă scădem bancnota aleasă din rest, de aceea este postcondiție.

```

1 lemma banknoteMaxim(rest: int, sum: int, finalSolution: seq <int> ,
   index: int)
2     requires 0 <= index <= 4
3     requires power(2, index) <= rest < power(2, index + 1)
4     requires isValidSolution(finalSolution)
5     requires addOptimRestEqualsOptimSum(rest, sum, finalSolution)
6     ensures addOptimRestEqualsOptimSum(rest - power(2, index), sum,
   finalSolution[index := finalSolution[index] + 1])

```

```

7      {
8          var banknote:= power(2, index);
9          forall currentSolution | isValidSolution(currentSolution) &&
              isOptimalSolution(currentSolution, rest - banknote)
10             ensures isOptimalSolution(solutionsSum(solutionsSum(
                  currentSolution, finalSolution), [0, 0, 0, 0, 0, 0][index:=
                  1]), sum)
11         {
12             assert isSolution(currentSolution[index:= currentSolution[index
                  ] + 1], rest);
13             exchangeArgument(rest, currentSolution, index);
14         }
15         assert forall currentSolution::isValidSolution(currentSolution) &&
              isOptimalSolution(currentSolution, rest - banknote) ==>
              isOptimalSolution(solutionsSum(solutionsSum(currentSolution,
                  finalSolution), [0, 0, 0, 0, 0, 0][index:= 1]), sum);
16     }

```

3.5 Lema banknoteMaxim32

Lema banknoteMaxim32 este folosită pentru a demonstra invariantul addOptimRestEqualsOptimSum.

Lema este similară cu banknoteMaxim, doar că este specifică bancnotei nemărginite, 32. Avem nevoie să știm că dacă în soluția curentă, optimă pentru $rest - 32$ adaugăm bancnota, aceasta devine optimă pentru $rest$. Acest lucru este demonstrat cu ajutorul lemei exchangeArgument32, detaliat în subcapitolul ???. Aceste lemme au nevoie ca indexul să se afle în codomeniu și să fie ales conform metodei maxBanknote, soluția calculată până în prezent să fie validă și să respecte predicatul addOptimRestEqualsOptimSum, așadar acestea sunt precondițiile. Această metodă se asigură că predicatul addOptimRestEqualsOptimSum este valid pentru $rest$ dacă scădem bancnota 32 din rest, de aceea este postcondiție.

```

1 lemma banknoteMaxim32(rest: int, sum: int, finalSolution: seq < int > )
2     requires rest >= 32
3     requires isValidSolution(finalSolution)
4     requires addOptimRestEqualsOptimSum(rest, sum, finalSolution)
5     ensures addOptimRestEqualsOptimSum(rest - 32, sum, solutionsSum(
        finalSolution, [0, 0, 0, 0, 0, 1]))

```



```

6 {
7   forall currentSolution | isValidSolution(currentSolution) && isSolution(
      currentSolution, rest - 32)
8     ensures isSolution(solutionsSum(solutionsSum(finalSolution,
      currentSolution), [0, 0, 0, 0, 0, 1]), sum)
9   {
10    assert isSolution(solutionsSum(currentSolution, [0, 0, 0, 0, 0, 1]),
      rest);
11  }
12
13  forall currentSolution | isValidSolution(currentSolution) &&
      isOptimalSolution(currentSolution, rest - 32)
14    ensures isOptimalSolution(solutionsSum(solutionsSum(finalSolution,
      currentSolution), [0, 0, 0, 0, 0, 1]), sum)
15  {
16    forall someSolution | isValidSolution(someSolution) && isSolution(
      someSolution, sum)
17      ensures cost(someSolution) >= cost(solutionsSum(solutionsSum(
      finalSolution, currentSolution), [0, 0, 0, 0, 0, 1]))
18    {
19      exchangeArgument32(rest, sum, currentSolution);
20    }
21  }
22 }

```

3.6 Lema exchangeArgument

Lema exchangeArgument este tratată diferit de lema exchangeArgument32 deoarece avem o bancnotă nemărginită superior, bancnota 32, care trebuie tratată separat.

Lema exchangeArgument este o leamnă tipică algoritmilor greedy care se bazează pe ideea că modificând progresiv dintr-o soluție produsă de orice alt algoritm într-o soluție produsă de algoritmul greedy, dacă folosim un mod care să nu înrăutățească calitatea soluției, atunci costul soluției produse este cel puțin la fel de mic ca cea a oricărei alte soluții, acesta fiind argumentul de schimb.

Presupunem că avem o soluție optimă oarecare și arătăm că poate fi transformată într-o soluție optimă care corespunde celei construite cu metoda greedy.

În lema exchangeArgument presupunem că soluția nu este optimă dacă adăugăm

bancnota aleasă și ajungem la o contradicție. Știm că soluția e optimă pentru $rest - bancnota$, dar nu pentru $rest$, dacă adăugăm bancnota (linia 13).

Considerăm că există o altă soluție optimă pentru $rest$, numită `optimalSolution` (linia 15).

Următoarea verificare este specifică cazurilor 1, 2, 4, 8, 16.

Verificăm în presupusa soluție pentru $rest$ că nu avem bancnota în soluție deja (linia 17), altfel dacă din acea presupusă soluție am scădea bancnota, atunci am avea un cost mai mic decât costul soluției noastre care e optimă pentru $rest - bancnota$ din precondiție, deci avem o contradicție.

Ulterior, demonstrăm că invariantul, explicat mai jos, de la linia 19, se menține pe parcursul unei bucle.

Nu există două bancnote de aceeași valoare în soluție, mai mici de 32.

În acest mod știm că soluția presupusă nu poate fi optimă, deoarece, dacă respectă proprietatea suma produsă de aceasta nu poate fi egală cu $rest$.

Deoarece :

- $\sum_{k=0}^{index-1} 2^k = 2^{index} - 1$
- $rest \geq 2^{index}$

Deci presupusa soluție pentru $rest$ nu poate fi egală decât cu cel mult $rest - 1$ (linia 45), astfel nu poate fi soluție optimă și se ajunge la o contradicție. Precondițiile și postcondițiile necesare pentru acestor lemme sunt aceleași ca în lemma `banknoteMaxim`.

```

1 lemma exchangeArgument (rest: int, currentSolution: seq <int> , index: int)
2   requires 0 <= index <= 4
3   requires power(2, index) <= rest < power(2, index + 1)
4   requires isValidSolution(currentSolution)
5   requires isOptimalSolution(currentSolution, rest - power(2, index))
6   ensures isOptimalSolution(currentSolution[index:= currentSolution[index
7     ] + 1], rest)
7 {
8   var banknote:= power(2, index);
9   var solution:= currentSolution[index:= currentSolution[index] + 1];
10  assert isValidSolution(solution);
11  assert isSolution(solution, rest);
12  var i:= index;
13  if (!isOptimalSolution(solution, rest))
14  {

```

```

15     var optimalSolution:| isValidSolution(optimalSolution) &&
        isSolution(optimalSolution, rest) &&
16         isOptimalSolution(optimalSolution, rest) && cost(
            optimalSolution) < cost(solution);
17     if (optimalSolution[index] - 1 >= 0)
18     {
19         var betterSolution:= addValueToIndex(optimalSolution, -1, index);
20         assert isSolution(betterSolution, rest - banknote);
21         assert cost(betterSolution) == cost(optimalSolution) - 1;
22         assert cost(optimalSolution) - 1 < cost(currentSolution);
23         assert false;
24     }
25     else
26     {
27         while (0 < i)
28             invariant 0 <= i <= index
29             invariant forall x::index >= x >= i ==> optimalSolution[x] <= 1
30             {
31                 i:= i - 1;
32                 assert isOptimalSolution(optimalSolution, rest);
33                 if (optimalSolution[i] > 1)
34                 {
35                     var optimalSolution' := optimalSolution[i:=optimalSolution[
                        i]-2];
36                     optimalSolution' := optimalSolution' [i + 1:=
                        optimalSolution'[i+1]+1];
37                     assert isSolution(optimalSolution', rest);
38                     assert cost(optimalSolution') == cost(optimalSolution) - 1;
39                     assert cost(optimalSolution') < cost(optimalSolution);
40                     assert false;
41                 }
42             }
43         assert solutionElementsSum(optimalSolution) <= banknote - 1;
44         assert rest >= banknote;
45         assert solutionElementsSum(optimalSolution) <= rest - 1;
46         assert isOptimalSolution(optimalSolution, rest);
47         assert false;
48     }
49 }
50 }

```

- **invariant:** Acest invariant (de la linia 29) se asigură că nu avem în soluția optimă mai mult de o bancnotă de valoarea 1, 2, 4, 8 sau 16, deoarece aparițiile multiple ale acestora pot fi înlocuite cu o bancnotă superioară pentru optimalitate (2 bancnote de valoare 1 au costul mai mare decât o bancnotă de valoare 2).

```
1 invariant forall index :: 4 >= index >= i ==> optimalSolution[index] <=
    1
```

3.7 Lema exchangeArgument32

Presupunem că avem o soluție optimă oarecare și arătăm că poate fi transformată într-o soluție optimă care corespunde celei construite cu metoda greedy.

În lema exchangeArgument32 presupunem că soluția nu este optimă dacă adăugăm bancnota 32 și ajungem la o contradicție. Știm că soluția e optimă pentru $rest - 32$, dar nu pentru $rest$, dacă adăugăm bancnota (linia 10).

Considerăm că există o altă soluție optimă pentru $rest$, numită `optimalSolution` (linia 15).

Demonstrăm că invariantul, explicat anterior, de la linia 25, se menține pe parcursul buclei.

Nu există două bancnote de aceeași valoare în soluție, mai mici de 32.

În acest mod știm că soluția presupusă nu poate fi optimă, deoarece, dacă respectă proprietatea suma produsă de aceasta nu poate fi egală cu $rest$.

- $\sum_{k=0}^{index-1} 2^k = 2^{index} - 1$
- $rest \geq 2^{index}$

Deci presupusa soluție pentru $rest$ nu poate fi egală decât cu cel mult $rest - 1$ (linia 38), astfel nu poate fi soluție optimă și se ajunge la o contradicție. Precondițiile și postcondițiile necesare pentru acestor lemme sunt aceleași ca în lema banknoteMaxim32.

```
1
2 lemma exchangeArgument32 (rest: int, sum: int, optimalSolution: seq < int >
    )
3   requires 32 <= rest
4   requires isValidSolution (optimalSolution)
5   requires isOptimalSolution (optimalSolution, rest - 32)
6   ensures isOptimalSolution (optimalSolution[5:= optimalSolution[5] + 1],
    rest)
```

```

7 {
8   var solution:= optimalSolution[5:= optimalSolution[5] + 1];
9   var i:= 4;
10  if (!isOptimalSolution(solution, rest))
11  {
12    if (optimalSolution[i] > 1)
13    {
14      var solution:= optimalSolution[i:= optimalSolution[i] - 2];
15      solution:= solution[i + 1:= solution[i + 1] + 1];
16      assert isSolution(solution, rest - 32);
17      assert cost(solution) == cost(optimalSolution) - 1;
18      assert cost(optimalSolution) - 1 < cost(solution);
19      assert false;
20    }
21    else
22    {
23      while (0 < i)
24        invariant 0 <= i <= 4
25        invariant forall index::4 >= index >= i ==> optimalSolution
26          [index] <= 1
27      {
28        i:= i - 1;
29        if (optimalSolution[i] > 1)
30        {
31          var solution:= optimalSolution[i:= optimalSolution[i] -
32            2];
33          solution:= solution[i + 1:= solution[i + 1] + 1];
34          assert isSolution(solution, rest - 32);
35          assert cost(solution) == cost(optimalSolution) - 1;
36          assert cost(optimalSolution) - 1 < cost(solution);
37          assert false;
38        }
39      }
40      assert solutionElementsSum(optimalSolution) <= rest - 1;
41      assert isOptimalSolution(solution, rest);
42      assert false;
43    }
44  }
45 }

```

Concluzie

În cadrul acestei lucrări am realizat verificarea formală a unei implementări a metodei greedy ce rezolvă o variantă a problemei bancnotelor cu bancnote puteri ale lui 2.

Implementarea unei metode greedy ce rezolvă varianta clasică a problemei bancnotelor este deja existentă, în cadrul unei lucrări de licență prezentată în 2022 la Facultatea de Informatică Iași [5].

Una dintre dificultățile întâlnite în acest proces a fost crearea invariantului care menține proprietatea de soluție optimă, așa a apărut predicatul `addOptimRestEqualsOptimSum`.

Acest predicat afirmă că o soluție optimă pentru o sumă x adunată cu o soluție pentru suma y are ca rezultat o soluție optimă pentru suma $x + y$. Astfel, secvența ce reprezintă soluția care se construiește pe parcursul buclei are garanția că atât timp cât se adună cu soluții locale optime, în final, vom avea o soluție finală optimă.

Parcursul acestor etape pentru verificare m-a făcut să aprofundez noțiunea de soluție și să înțeleg în amănunt formarea acesteia prin metoda greedy.

Pe viitor îmi propun să găsesc o modalitate de a unifica metodele `banknoteMaxim` pentru a funcționa pentru orice caz, astfel optimizând viteza verificării.

Un alt lucru interesant de făcut în viitor, ar fi găsirea unei proprietăți care să se aplice pentru orice sistem de bancnote, pentru a verifica faptul că produce o soluție optimă.

Bibliografie

- [1] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*, pp. 348–370, Springer, 2010.
- [2] S. T. Q. Assurance and Maintenance, "dafny." Github, 2018. <https://github.com/stqam/dafny>, data accesării: 02-01-2023.
- [3] J. Nielsen, "How to sum consecutive powers of 2," 2020. <https://jarednielsen.com/sum-consecutive-powers-2/>, data accesării: 23-12-2022.
- [4] D. Kozen and S. Zaks, "Optimal bounds for the change-making problem," in *Automata, Languages and Programming: 20th International Colloquium, ICALP 93 Lund, Sweden, July 5–9, 1993 Proceedings 20*, pp. 150–161, Springer, 1993.
- [5] C. Elisa, "Implementarea unui algoritm de tip greedy în dafny ce rezolvă problema bancnotelor." Github, 2022. <https://github.com/ElisaChicos/Licenta>, data accesării: 05-12-2022.