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 **4**

2.1 Reprezentarea datelor de intrare 4

2.2 Reprezentarea datelor de ieșire 4

2.3 Structuri de date folosite 4

2.3.1 Condiții ca o soluție finală să fie optimă 4

3 Verificarea formală a problemei **6**

3.1 Schema verificării 6

3.2 Demonstrarea optimalității 6

3.2.1 Metoda banknoteMinimum 7

3.2.2 Metoda maxBanknote 8

3.2.3 Lema banknoteMaxim sau banknoteMaxim32 9

3.2.4 Lema exchangeArgument sau exchangeArgument32 10

3.2.5 Funcția addValueToIndex 12

Concluzii **14**

Bibliografie **15**

Capitolul 1

Context

Dafny este un limbaj de programare și verificare, capabil să verifice corectitudinea funcțională a unui program. Verificarea este posibilă datorită caracteristicilor specifice limbajului precum precondiții, postcondiții, invariante, ș.a.m.d. De asemenea, verificatorul Dafny are grijă ca adnotările făcute să se îndeplinească, astfel acesta ne scapă de povara de a scrie cod fără erori, în schimbul scrierii de adnotări fără erori.[1]

În următoarele rânduri voi arăta un exemplu de cod în Dafny ce rezolvă problema Fibonacci, menționez că următorul cod nu îmi aparține, ci îl voi folosi în scop explicativ. [2]

```
    function fib (n: nat) : nat
{
  if n == 0 then 0
  else if n == 1 then 1
  else fib (n-1) + fib (n-2)
}

method Fib (n: nat) returns (b: nat)
  ensures b == fib(n);
{
  if (n == 0) { return 0; }
  var a := 0;
  b := 1;

  var i := 1;
  while (i < n)
    invariant i <= n
    invariant a == fib(i-1);
```

```
    invariant b == fib(i);
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```

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 intreg 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. Acești invarianți au rolul de a se asigura că sunt îndeplinite acele condiții pentru tot parcursul buclei, deoarece Dafny uită de la un pas la altul ce a realizat în buclă.

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

În cadrul acestei lucrări am ales să implementez **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.

Verificarea formală in Dafny a problemei bancnotelor demonstrează faptul că soluția construită este optimă pentru orice sumă dată ca input.

Reprezentarea generală folosită pentru soluție în problema bancnotelor este: $banknote_1 := 1 < banknote_2 < ... < banknote_n$.

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.

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

Capitolul 2

Implementarea problemei în Dafny

2.1 Reprezentarea datelor de intrare

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

2.2 Reprezentarea datelor de ieșire

Drept date de ieșire avem soluția de cost minim reprezentată de secvența:

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

2.3 Structuri de date folosite

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.

2.3.1 Condiții ca o soluție finală să fie 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.

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

• **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.

• **invariant**:Acest invariant, care apare mai jos, 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
```

• **predicatul addOptimRestEqualsOptimSum**: Acest predicat 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.

Capitolul 3

Verificarea formală a problemei

3.1 Schema verificării

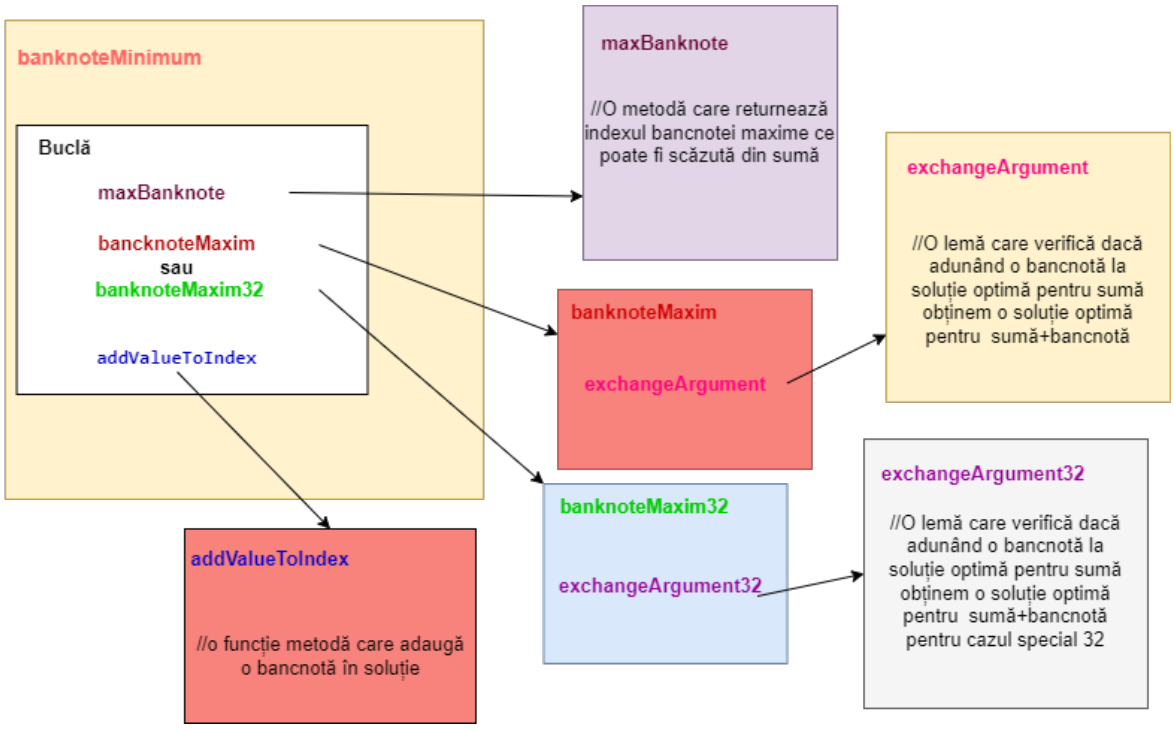

Figura 1: O schemă explicativă a apelurilor de funcții și leme în cadrul verificării.

3.2 Demonstrarea optimalității

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]

3.2.1 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 o bancnotă în soluție.

Invarianții prezenți în while au rolul de a menține proprietatea de optimalitate pe parcursul construirii soluției, astfel că la orice pas soluția este optimă pentru $suma - rest$ de la momentul respectiv.

În primă fază calculează bancnota cea mai mare ce poate fi dată ca rest, cu ajutorul funcției maxBanknote și funcției power.

Ulterior, folosește lema banknoteMaxim sau banknoteMaxim32, în funcție de bancnota aleasă, pentru a demonstra că dacă adaugăm bancnota aleasă la soluția calculată până în prezent, noua soluție pentru $sum - rest$ adunată cu o soluție optimă pentru $rest - bancnota$ produce o soluție optimă pentru 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];
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);
```

```
            }
            else
            {
                banknoteMaxim32(rest, sum, solution);
            }
            solution:= addValueToIndex(solution,1,index);
            rest:= rest - banknote;
        }
    }
```

3.2.2 Metoda maxBanknote

Metoda maxBanknote este folosită pentru a returna un index cu proprietatea: $bancnota_{index} <= rest < bancnota_{index+1}$.

Avem postcondițiile următoare: indexul să se afle în range-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.

```
    method maxBanknote(sum: int) returns(index: int)
      requires sum > 0
      ensures 0 <= index <= 5
      ensures 0 <= power(2, index) <= sum
      ensures(index != 5 && power(2, index + 1) > sum) || index == 5
    {
      index:= 5;
      if (power(2, index) > sum)
      {
        assert power(2, index + 1) > sum;
        while (power(2, index) > sum && index > 0)
          invariant power(2, index + 1) > sum
          {
            index:= index - 1;
            assert power(2, index + 1) > sum;
          }
      }
        else
      {
```

```
20          assert index == 5;
21        }
22      }
```

3.2.3 Lema banknoteMaxim sau banknoteMaxim32

Lema banknoteMaxim, respectiv banknoteMaxim32, este folosită pentru a demonstra faptul că dacă adăugam 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 acest lucru avem nevoie să știm că dacă în soluția curentă, optimă pentru $rest - bancnota$ adaugăm bancnota aceasta devine optimă pentru $rest$. Acest lucru este demonstrat cu ajutorul lemei exchangeArgument, respectiv exchangeArgument32. Cele doua leme sunt similare așa că voi da exemplu de cod doar pentru una dintre ele.

Aceste lemme au nevoie ca indexul să se afle în range 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 și 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      }
```

3.2.4 Lema exchangeArgument sau exchangeArgument32

Lema exchangeArgument, respectiv exchangeArgument32, presupune că nu este optimă soluția dacă adaugăm bancnota aleasă și ajunge la o contradicție. Știm că soluția e optimă pentru $rest - bancnota$, dar nu pentru rest, dacă adaugăm bancnota.

Consideră că există o altă soluție optimă pentru rest.

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

• Verificăm în presupusa soluție pentru rest că nu avem bancnota în soluție deja, altfel dacă din acea presupusă solutie 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, verificăm demonstrăm că invariantul explicat în subsecțiunea 2.3.1 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 :

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

•$rest >= 2^{index}$

Deci presupusa soluție pentru rest nu poate fi egală decât cu cel mult $rest - 1$, 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, respectiv banknoteMaxim32.

Lemma exchangeArgument este o lemă 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.

```
lemma exchangeArgument(rest: int, currentSolution: seq <int> , index: int)
    requires 0 <= index <= 4
    requires power(2, index) <= rest < power(2, index + 1)
    requires isValidSolution(currentSolution)
    requires isOptimalSolution(currentSolution, rest - power(2, index))
    ensures isOptimalSolution(currentSolution[index:= currentSolution[index
        ] + 1], rest)
{
    var banknote:= power(2, index);
    var solution:= currentSolution[index:= currentSolution[index] + 1];
    assert isValidSolution(solution);
    assert isSolution(solution, rest);
    var i:= index;
    if (!isOptimalSolution(solution, rest))
    {
    var optimalSolution:| isValidSolution(optimalSolution) && isSolution(
        optimalSolution, rest) &&
        isOptimalSolution(optimalSolution, rest) && cost(optimalSolution) <
            cost(solution);
    if (optimalSolution[index] -1 >= 0)
    {
        var betterSolution:= addValueToIndex(optimalSolution,-1,index);
        assert isSolution(betterSolution, rest - banknote);
        assert cost(betterSolution) == cost(optimalSolution) - 1;
        assert cost(optimalSolution) - 1 < cost(currentSolution);
        assert false;
    }
    else
    {
        while (0 < i)
        invariant 0 <= i <= index
        invariant forall x::index >= x >= i ==> optimalSolution[x] <= 1
        {
        i:= i - 1;
        assert isOptimalSolution(optimalSolution, rest);
        if (optimalSolution[i] > 1)
        {
            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  }
```

3.2.5 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 range, soluția să fie validă și faptul că valoarea adaugată 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 adaugării 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 }
```

Concluzie

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

Implementarea unei metode greedy ce rezolvă problema 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.

Parcurgerea 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.

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.