UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Implementarea unui algoritm de tip greedy ce rezolva problema bancnotelor in Dafny

propusă de

Veronica Elisa Chicoș

Sesiunea: iunie, 2022

Coordonator științific

Conf. Dr. Ciobaca Stefan

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Implementarea unui algoritm de tip greedy ce rezolva problema bancnotelor in Dafny

Veronica Elisa Chicoș

Sesiunea: iunie, 2022

Coordonator științific

Conf. Dr. Ciobaca Stefan

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Ciobaca Stefan.

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 ce rezolva problema bancnotelor in Dafny** elaborată sub îndrumarea domnului **Conf. Dr. Ciobaca Stefan**, 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 ce rezolva problema bancnotelor in Dafny**, 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 deriva din dorinta de a aprofunda subiectul optimizarii. In timpul facultatii, am invatat ca un cod trebuie sa fie optimizat pentru a creste eficineta acestuia. Drept urmare, consider ca orice actiune, cat de simpla, poate fi optimizata astfel incat sa se realizeze intr-un timp cat mai scurt si cu un numar minim de resurse.

Intentie

In cadrul lucrarii voi face o scurta intorducere in paradigma de programare Greedy, urmata de mai multe probleme care se rezolva cu ajutorul acesteia, printre care si Problema Bancnotelor.

Totodata, voi prezenta limbajul Dafny cat si implementarea algoritmului pentru rezolvarea problemei alese de mine.

Introducere

Lucrarea va fi structurata in felul urmator:

1. Paradigma Greedy - in acest capitol voi prezenta paradigma Greedy impreuna cu cateva probleme si aplicatiile acestora
2. Problema Bancnotelor - in acest capitol voi descrie problema pe care am ales-o pentru licenta
3. Limbajul de programare Dafny - in acest capitol voi prezenta o introducere in limbajul Dafny
4. Detalii de implementare - in acest capitol voi prezenta in 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 in functie de informatiile cunoscute in prezent fara a ne ingrijora de efectul acesteia in viitor.

1.1 Ce este o problema de optimizare si cum lucreaza metoda Greedy cu aceasta?

O problema de optimizare este o problema care are un input oarecare dar outputul ei trebuie sa fie o valoare maxima sau minima. In functie de ce ne cere problema, aceasta poate fi de doua feluri:

1. Problema de minimizare

Please make sure you send in your completed forms by January 1st next year, or the penalty clause 2(a) will apply.

2. Problema de maximizare

Please make sure you send in your completed forms by January 1st next year, or the penalty clause 2(a) will apply.

In functie de decizia facuta de metoda Greedy, vom crea solutii posibile. O solutie posibila este o submultime care satisface cerintele problemei, in timp ce o solutie optima respecta cerintele date si are cost minim sau castig maxim.

Dacă mai multe soluții îndeplinesc criteriile date, atunci acele soluții vor fi considerate ca fiind posibile, in timp ce solutia optima este cea mai buna dintre toate solutiile.

Drept urmare, aceasta metoda este folosita pentru a determina solutia oprima si a rezolva corect problema de optimizare.

1.2 Algoritmi Greedy

1.2.1 Problema selectiei activitatilor

Problema activitatilor este o problema de maximizare care presupune alegerea cat mai multor activitati care pot fi realizate intr-un anumit interval de timp fara ca acestea sa se suprapuna.

O posibila formulare a acestei probleme este:

Input:

- n - numarul de activitati,
- aStart[0..n-1] - vector care contine timpul la care incep activitatile,
- aSfarsit[0..n-1] - vector ce contine timpul la care se termina actvitatile astfel incat aStart[i] $<$ aSfarsit[i] pentru orice 0 $\leq$ i $\leq$ n-1

Output:

- A $\subseteq$ { 0, ..., n-1 }, unde A este o multime de activitati care nu se suprapun si este de cardinal maxim;

Spre exmplu putem avea aceste multimi de activitati reprezentate in tabelul de mai jos:

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

Solutia optima 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 partial de cost minim

1.3 Avantaje si dezavantaje ale algoritmilor Greedy

Chapter 2

Problema Bancnotelor

2.1 Ce este Problema Bancnotelor?

Problema bancnotelor este o problema de minimizare de care ne lovim zilnic. Stiind ca avem o multime de bancnote $B = \{b_1, b_2, b_3\}$ si o suma S de platit, rezultatul problemei va fi multimea de bancnote cu cardinal minim pe care o putem folosi pentru a plati suma S.

2.1.1 Formularea problemei

In limbaj natural, Problema Bancnotelor poate fi formulata astfel:

Se da o multime de bancnote $B = \{b_1, b_2, b_3\}$ si o suma S pe care trebuie sa o platim. Se cere sa se afiseze o multime de bancnote care trebuie sa indeplineasca urmatoarele conditii:

- cardinalul multimii sa fie minim,
- suma elementelor multimii sa fie egata cu S;

Formularea computationala a problemei:

Input:

- un numar natural n - suma care trebuie platita,

Output:

- numerele $n_{500}, n_{200}, n_{100}, n_{50}, n_{20}, n_{10}, n_5, n_1$ (n_i - numarul de bancnote i folosite), astfel incat $\sum_{i \in \{500,200,100,50,20,10,5,1\}} n_i$ sa fie minima si $n = \sum_{i \in \{500,200,100,50,20,10,5,1\}} i \times n_i$.

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

Solutia optima pentru aceasta 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 aleasa

Strategia Greedy consta in alegerea, la fiecare pas, a bancnotei celei mai mari, care este mai mica sau egala cu suma pe care trebuie sa o platim. In functie de alegerea facuta, va trebui sa scadem valoarea bancnotei alese din suma care trebuie platita pentru a ne asigura ca avansam progresiv catre obiectivul final.

Folosind exemplul de mai sus, prima decizie facuta de strategia noastra este alegerea bancnotei de 100 deoarece $100 \leq 157$. Decizia facuta are ca rezultat adaugarea valorii 1 la solutia finala iar suma pe care trebuie sa o platim scade cu 100, astfel incat la pasul urmator va trebui sa o platim 57. Acest algoritm se va repeta pana cand suma va fi 0.

// exemplu desen pasi

2.3 Limitarea Strategiei Greedy

Limitarea strategiei alese este ca exista posibilitatea ca acesta sa nu ofere solutia optima pentru toate datele de intrare.

De exemplu, daca multimea de bancnote este formata din $B = \{1, 6, 9\}$ iar suma care trebuie platita este $s = 12$, solutia optima generata de strategia aleasa va fi $\{3, 0, 1\}$. Cu toate acestea, solutia cu adevarat optima va fi $\{0, 2, 0\}$.
exemplu

Motivul pentru care se produce aceasta "eroare" este ca solutia este construita pas cu pas. La fiecare pas, se alege bancnota care va crea cel mai mic cost pentru solutia curenta chiar daca in viitor ar exista o alta solutie cu cost mai mic.

Chapter 3

Dafny

3.1 Limbajul de programare Dafny

Dafny este un limbaj de programare functional si imperativ care prin intermediul adnotarilor sale permite crearea unui program care sa nu contina erori la runtime si sa faca ceea ce isi doreste programatorul

Un exemplu de adnotare este $requires\ a >= 10$. Astfel, suntem siguri ca variabila a nu va fi mai mica decat 10 scutindu-ne de o structura de decizie pe care ar fi trebui sa o utilizam pentru a verifica acest lucru. Alte erori care pot fi evitate prin folosirea adnitarilor sunt erorile de indexare, impartirea la 0, valori nule etc.

3.2 Avantajele folosirii limbajului Dafny

Preconditiile si postconditiile din acest limbaj de programare stabilesc ce conditii trebuie sa fie adevarate la intrare, respectiv, la iesirea din metoda. Astfel, programele in Dafny sunt verificate pentru corectitudinea globala astfel incat fiecare rulare se va termina si se va ajunge la rezultatul dorit. In plus,adaugarea acestor conditii duc la o intelegere in detaliu a codului.

Chapter 4

Detalii de implementare

In acest capitol voi prezenta detaliile de implementare a algoritmului cat si modul in care am demonstrat corectitudinea codului utilizand Dafny.

4.1 Reprezentarea datelor de intrare si a celor de iesire

4.1.1 Varibilele folosite, tipurile si semnificatia acestora

- suma : $int \rightarrow$ reprezinta suma pe care trebuie sa o platim
- rest : $int \rightarrow$ reprezinta suma care ne-a ramas de platit dupa ce am ales o bancnota
- solutieFinala : $seq < int > \rightarrow$ reprezinta solutia finala a problemei
- solutieCurenta : $seq < int > \rightarrow$ reprezinta solutia creata pana la pasul curent
- solutieOarecare : $seq < int > \rightarrow$ reprezinta o solutie pe care am folosit-o pentru a demonstra corectitudinea programului

4.1.2 Descrierea solutiei rezultate

Bancnotele folosite de mine pentru rezolvarea acestei probleme sunt $B = \{1, 5, 10, 20, 50\}$, astfel, solutia problemei va fi o secventa de numere naturale $solutie = \{n_1, n_5, n_{10}, n_{20}, n_{50}\}$ unde n_i va numarul 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 si functii

Inainte de a trece la implementarea propriu-zisa a algoritmului, o sa incep prin a prezenta predicatele care asigura corectitudinea rezultatului intors de catre metoda ce genereaza solutia optima pentru suma pe care trebuie sa o platim.

Predicatele sunt metode care returneaza o valoare de adevar si sunt folosite pentru a verifica o anumita proprietate sau mai multe prin intermediul unei singure instructiuni.

In Dafny, functiile sunt asemanatoare cu functiile matematice. Functiile contin o singura expresie care returneaza o valoare de tipul declarat in antetul functiei.

1. Predicatul $esteSolutieValida(solutie : seq < int >) \rightarrow$ acest predicat ne asigura ca solutia primita ca parametru este de lungime 5 si ca fiecare element este mai mare sau egal cu 0. Conditii mentionate anterior sunt foarte importante deoarece, avand 5 bancnote (1, 5, 10, 20, 50), inseamna ca va trebuie sa existe un elemnt al solutiei 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 verifica faptul ca suma elementelor secventei $solutie$ inmultime cu bancnotele folosite in problema este egala cu $suma$. Daca cele doua sunt egale, atunci secventa $solutie$ devine solutie posibila 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. Functia $cost(solutie : seq < int >) : int \rightarrow$ aceasta functie calculeaza si returneaza cate bancnote au fost folosite pentru a forma solutia posibila memorate in variabila $solutie$ data 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 adevar daca solutia primita ca parametru este o solutie posibila pentru suma pe care trebuie sa o platim si orice alta solutie posibila are costul mai mare sau egal. Acest lucru se intampla deoarece Problema Bancnotelor este o problema de minimizare iar solutia trebuie sa 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 adevarat daca orice solutie care este valida este si solutie posibila pentru $rest$ (suma care ne-a ramas de platit) atunci suma dintre aceasta solutie si cea data ca parametru trebuie sa fie solutie posibila pentru suma totala care trebuie platita. Similar se intampla si pentru a demosntra ca suma celor doua solutii este solutie optima 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 primeste ca parametru suma pe care va trebui sa o platim si returneaza o solutie care respecta proprietatile discutate mai sus in sectiunea 4.1.2 $Descrierea\ solutiei\ rezultate$.

Preconditiile sunt expresii booleene care trebuie sa fie adevarate pentru variabilele date ca parametru, in timp ce postconditiile trebuie sa fie adevarate pentru variabilele returnate de metoda. Acestea sunt adaugate la inceputul metodei pentru a determina o buna functionare a programului si totodata corectitudinea acestuia.

Preconditia $requires\ \ suma >= 0$ ne indica faptul ca suma trebuie sa fie mai mare sau egala cu 0 pentru a se putea parcurge metoda.

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

Aceste trei postconditii se focuseaza pe solutia returnata de metoda. Solutia trebuie sa fie valida, sa fie o solutie posibila pentru suma care trebuie platita si totodata sa fie si solutie oprima pentru aceasta.

Dupa indeplinirea cu succes a preconditiilor si a postconditiilor cerute, vom intra in corpul metodei.

La inceput, vom declara 5 variabile $s_1, s_5, s_{10}, s_{20}, s_{50}$ pe care le initializam cu valoarea 0. Aceste variabile vor memora cate bancnote din fiecare tip sunt folosite.

In bucla $while$ se va calcula, la fiecare pas, cu ajutorul uneii metode $gasireMaxim(rest)$ valoarea bancnotei maxime care este mai mica sau egala cu $rest$. In functie de valoarea returnata de aceasta metoda, se va incrementa valoarea unei variabilele declarate la inceputul metodei, va scadea valoarea variabilei $rest$ cu valoarea bancnotei alese la pasul curent si se va apela o anumita lemma care ajuta la demonstrarea corectitudinii

programului insa voi reveni ulterior la acest aspect.

Un alt lucru interesant la acest limbaj de programare este prezenta invariantilor la inceputul buclelor. Invariantii sunt, asemenea postconditiilor si preconditiilor, proprietati 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])
```

Acesti invarianti ne indica faptul ca $rest$ isi va schimba valoarea si va descreste dar va fi mereu mai mare decar 0 si mai mica sau egala decat $suma$ si ca solutia creata pana la pasul curent este o solutie posibila pentru $suma - rest$. Totodata, vom folosi predicatul INV pentru a verifica fapul ca solutia noastra este solutie optima.

4.2.3 Leme importante in demonstrarea corectitudinii

Lemele sunt metode care sunt folosite pentru a verifica corectitudinea programului. Acestea contin preconditii iar proprietatea care trebuie demonstrata va fi postconditia lemei.

Uneori, lemele pot fi demonstrate de catre Dafny fara a adauga intructiuni, ceea ce inseamna ca un corp fara instructiuni serveste ca argument pentru demonstratie.

In prezentarea agloritmului Greedy creat de mine, in functie de bancnota aleasa, va fi apelata o lema care va verifica daca alegerea facuta va duce la o solutie optima. Spre exemplu, pentru $s = 1$, unde s este valoarea returnata de metoda $gasireMaxim(rest)$, se va apela lema $cazMaxim1$, pentru $s = 5$ se va apela lema $cazMaxim5$, iar acest lucru se intampla pentru fiecare bancnota care este posibil sa fie aleasa.

In continuare, o sa prezint lemele $cazMaxim1$ si $cazMaxim50$. Primele patru leme, $cazMaxim1$, $cazMaxim5$, $cazMaxim10$, $cazMaxim20$, sunt similare, drep urmare voi prezenta doar lema $cazMaxim1$. Insa, in subsectiunea urmatoare, voi reveni asupra lemelor $cazMaxim10$, $cazMaxim20$ deoarece contin o demonstratie interesanta a unei proprietati.

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 caruia vom demonstra ca daca vom alege bancnota 1, vom crea o solutie oprima pentru variabila $suma$.

Acest lucru il vom demonstra cu ajutorul assert-urilor. Cu ajutorul acestora vom verifica ce ”stie” verificatorul Dafny si ce va trebui demonstrat separat cu ajutorul lemelor.

Forall statement-ul despre care am amintit mai sus va crea o solutie numita $solutieCurenta$ despre care stim ca este o solutie valida si ca este o solutie oprima pentru $rest - 1$. Spre exemplu, daca $rest = 4$, $suma = 54$ si $solutieFinala = [0, 0, 0, 0, 1]$ solutia aleasa de forall statement va fi $[3, 0, 0, 0, 0]$ care este solutie optima pentru 3.

Atfel, stiind ca $solutieCurenta$ este solutie optima pentru $rest - 1$, este evidet ca aceasta este o solutie posibila pentru $rest - 1$. Vom incerca sa modificam

$solutieCurenta[0] + 1$, pentru a crea o solutie posibila pentru $rest$.

Urmatorul pas este sa formam o noua solutie prin adunarea celor doua solutii cunoscute pentru a crea o solutie oprima 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 lema are rolul de a demonstra ca prin alegerea bancnotei de 50 se va crea o solutie optima pentru $suma$.

Ceea ce stim este ca pentru orice solutie valida si posibila pentru $rest$, suma acesteia cu $solutieFinala$ va fi solutie pentru $suma$. Astfel,folosind un forall statement vom demonstra ca pentru orice $solutieCurenta$ care este solutie posibila pentru $rest - 50$, suma acesteia cu $solutieFinala$ va crea o solutie pentru $suma$.

// de reformulat

Urmatorul lucru care trebuie sa il demonstram este ca suma elementelor celor doua solutii este solutie optima pentru $suma$. Pentru aceasta demonstratie, am folosit un exchange argument pe care il voi prezenta in subsectiunea urmatoare.

4.2.4 Exchange Arguments

Exchange arguments este o tehnica folosita pentru a demonstra optimitatea unei solutii. Tehnica presupune modificarea unei solutii oarecare pentru a obtine o solutie optima pentru agloritmul greedy fara a-i modifica costul.

Dupa cum am mentionat anterior, pentru lemele $cazMaxim10$,$cazMaxim20$ si $cazMaxim50$ demonstrarea faptului ca suma dintre $solutieCurenta$ si $solutieFinala$ este solutie optima pentru $suma$ nu a mers inductiv ca in celalte leme, asa ca va trebui "sa ajutam" verificatorul Dafny.

In programul scris de mine, exista doua situatii in care intervine folosirea acestei tehnici:

- variabila $rest$ apartine unui interval, cum ar fi $10 <= rest < 20$ si $20 <= rest < 50$
- variabila $rest$ are doar limita inferioara, cum ar fi $rest >= 50$.

In ceea ce urmeaza o sa prezint cate un caz din fiecare situatie.

1. Situatia in care variabila $10 <= rest < 20$ este reprezentata 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 ca solutia curenta construita astfel $[solutieCurenta[0], solutieCurenta[1], solutieCurenta[2]+1, solutieCurenta[3], solutieCurenta[4]]$ este solutie optima pentru variabila $rest$.

Pentru a demonstra acest lucru, vom crea o noua solutie numita $solutieOptima$ care are cost mai mic decat solutia curenta. Stim deja ca elementele $solutieOptima[3] == 0$ si $solutieOptima[4] == 0$ asa ca ne putem concentra atentia pe primele 3 elemente ale solutiei.

In cele ce urmeaza, vom cauta toate combinatiile din numere 1, 5 si 10 a caror suma este 10. Pentru fiecare combinatie se va crea o noua solutie $solutieOprima'$ astfel: se va scadea cate o bancnota cu valoarea respectiva din $solutieOptima$ pana cand suma acestora va fi 10.

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

Stiind ca am scazut n bancnote din $solutieOptima$ pentru a crea $solutieOptima'$ costul acesteia va fi $cost(solutieOptima) - n$, iar incercand sa comparam acest cost cu costul solutiei curente vom ajunge la o contradictie.

2. Situatia in care variabila $rest <= 50$ este reprezentata de lema exchangeArgumentCaz50(rest : int, suma : int, solutieOarecare : seq<int>, solutieCurenta : seq<int>)

   Codul pentru aceasta lema este foarte lung si nu il voi insera pe tot in acest document. Voi prezenta anumite instructiuni care au o importanta deosebita, iar secventele repetitive le voi inlocui cu un comentariu sugestiv.

   Acesta lema are drept scop demonstrarea faptului ca variabila $solutieCurenta$ are un cost mai mic decat 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);
```

Incepem demonstratia prin a verifica daca $solutieOarecare$ are costul mai mic decat $cost(solutieCurenta)$. In caz afirmativ, vom incepe sa comparam elementele din cele doua solutii: $solutieOarecare[4] > solutieCurenta[4] + 1$, $solutieOarecare[3] > solutieCurenta[3]$, $solutieOarecare[2] > solutieCurenta[2]$, $solutieOarecare[1] > solutieCurenta[1]$ si $solutieOarecare[0] > solutieCurenta[0]$.

Secventa de cod adaugata mai sus contine doar compararea elementelor de pe pozitia 4 deoarece aceasta este cea mai complexa parte.

Daca $solutieOarecare[4] > solutieCurenta[4] + 1$ atunci solutia $[solutieOarecare[0], solutieOarecare[1], solutieOarecare[2], solutieOarecare[3], solutieOarecare[4] - 1]$ are costul mai mic decat costul solutiei curente si este solutie optima pentru $rest - 50$, lucru care este fals.

In schimb, daca $solutieOarecare[4] < solutieCurenta[4] + 1$ vom crea o noua solutie numita $nouaSolutieOarecare$ formata din elementele secventei $solutieOarecare$ din care am scazut bancnotele a caror suma este egala cu 50 si am incrementat valoarea elementului de pe pozitia 4, apoi, cu ajutorul unui apel recursiv, vom compara $nouaSolutieOarecare$ cu $solutieCurenta$ pana cand acestea vor fi egale.

Similar se intampla si cu celelalte trei comparatii pe care le-am enumerat mai sus. Se vor crea noi solutii pana cand elementele din $solutieOarecare$ si $solutieCurenta$ vor fi egale.

4.3 Probleme ”interesante” intalnite

Pe parcursul realizarii acestui algoritm si a demonstrarii corectitudinii sale am intampinat mai multe probleme care m-au pus in dificultate.

Problema care m-a ”incurcat” am intampinat-o la lema exchangeArgumentCaz50(rest: int, suma : int, solutieOarecare : seq<int>, solutieCurenta : seq<int>). Dupa cum am prezentat anterior, pentru a demonstra ca $solutieOarecare[4] == solutieCurenta[4]+1$, cu alte cuvinte, daca ambele solutii contineau acelasi numar de bancnote de valoare 50, a trebuit sa caut toate combinatiile de valori de 1,5,10 si 20 a caror suma este 50 si sa creez o noua solutie.

- Problema 1: Dupa ce am adaugat in lema toate cele 53(!! de verificat) de cazuri posibile, lema nu putea sa demonstreze ca $solutieOarecare[4] == solutieCurenta[4] + 1$, ceea ce insemna ca imi lipseste un caz.

  Rezulvarea problemei 1: In ultima instructiune $else$ am adaugat cateva 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 insemna ca aveam 60 de lei formati din 3 bancnote de 20 de lei.

- Problema 2: Cum as putea sa tratez acest caz?

  Incercarea 1: Am incercat sa verific daca $solutieOarecare[3] >= 5$ astfel incat sa pot adauga 2 bancnote de 50 de lei $solutieOarecare[4]+2 \rightarrow$ eroare: $solutieOarecare[3]$ nu este mai mare sau egal cu 5.

  Incercarea 2: Am incercat sa verific daca $solutieOarecare[0]+5 \cdot solutieOarecare[1]+10 \cdot solutieOarecare[2] >= 100$ pentru a putea adauga valoarea 5 la $solutieOarecare[3]$ si sa pot reveni la ideea de la ”Incercarea 1” $\rightarrow$ eroare: suma elementelor nu este mai mare sau egala cu 100.

  Incercare 3: Am incercat sa tratez cazul astfel: cele 3 bancnote de 20 le voi transforma intr-o bancnota de 50 iar bancnota de 10 ramasa o voi aduna la celalate pe care le am in solutie, astfel, creand o noua solutie 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 "decrease clase might not decrease" aparuse deoarece in adnotatiile lemei initial exista $decreases\ solutieOarecare[0], solutieOarecare[1], solutieOarecare[2], solutieOarecare[3]$ ceea ce ne asigura ca elementele secventei $solutieOarecare$ vor descreste. In incercarea mea de a trata cazul ramas, incercam sa adaug "restul" in $solutieOarecare[2]$ si incalcam adnotarea specificata mai sus.

  Rezolvarea problemei 2: Cum scopul pricipal al lemei era de a demonstra ca $solutieCurenta$ are cost minim am modificat adnotarea astfel $decreases\ solutieOarecare[0] + solutieOarecare[1] + solutieOarecare[2] + solutieOarecare[3]$ deoarece este suficient sa descreasca suma acestora pentru a ajunge la o solutie 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