UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Verificarea unui algoritm pentru varianta discretă a problemei rucsacului în Dafny

propusă de

Alina-Adriana Haidău

Sesiunea: februarie, 2025

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Verificarea unui algoritm pentru varianta discretă a problemei rucsacului în Dafny

Alina-Adriana Haidău

Sesiunea: februarie, 2025

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Ștefan Ciobâcă.

Data: .... Semnătura: ....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Haidău Alina-Adriana** domiciliat în **România, jud. Suceava, com. Bălăceana, nr. 97**, născut la data de **16 iulie 2000**, identificat prin CNP **6000716330223**, 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 **Verificarea unui algoritm pentru varianta discretă a problemei rucsacului în Dafny** elaborată sub îndrumarea domnului **Conf. Dr. Ștefan Ciobâcă**, 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 **Verificarea unui algoritm pentru varianta discretă a problemei rucsacului în 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 **Alina-Adriana Haidău**

Data: .... Semnătura: ....

Cuprins

Motivație

Am ales să implementez și să demonstrez corectitudinea unui algoritm pentru problema rucsacului în Dafny deoarece este o problemă clasică de optimizare în informatică, având aplicabilitate în diverse domenii precum criptografie, economie și logistică.

Pentru a aborda o versiune mai complexă a problemei, am ales varianta discretă a problemei rucsacului. Știam de aceasta din primul an de facultate, unde am învățat în cadrul cursului *Proiectarea algoritmilor* că pentru această variantă a problemei, soluția optimă poate fi obținută folosind paradigma programării dinamice. Astfel, am avut posibilitatea să dobândesc o înțelegere mai profundă a modului în care funcționează aceasta, dar și a felului în care soluțiile optime parțiale sunt construite pas cu pas.

Introducere

În acest capitol introductiv voi face o scurtă prezentare a limbajului Dafny, urmată de câteva puncte esențiale despre paradigma programării dinamice și câteva detalii despre problema rucsacului.

Limbajul de programare Dafny

Dafny este un limbaj de programare imperativ și funcțional, ce oferă suport pentru programarea orientată pe obiecte și ce este destinat verificării formale a corectudinii programelor [1]. A fost creat pentru a ajuta programatorii să scrie cod care este corect din punct de vedere funcțional. Un lucru foarte interesant de știut despre acest limbaj este faptul că a fost conceput astfel încât permite verificarea codului încă din faza de dezvoltare. Datorită faptului că verificatorul Dafny este rulat ca parte a compilatorului, orice eroare matematică (de exemplu, împărțirea la 0) sau logică va fi semnalată către programator, care va trebui să ajute verificatorul prin ajustarea specificațiilor sau a codului. [2]

Programarea dinamică

Programarea dinamică este o metodă de proiectare a unei clase de algoritmi ce rezolvă probleme cu proprietăți similare [3]. Ea se bazează pe conceptul de suprapunere al subproblemelor, însemnând faptul că o problemă poate fi „spartă" în mai multe subprobleme mai mici care sunt mai ușor de rezolvat și care se repetă pe parcursul execuției algoritmului. Soluțiile acestor subprobleme sunt stocate astfel încât să nu fie necesară recalcularea lor, iar rezultatele calculate anterior pot fi reutilizate. [4]

Principalele abordări prin care soluțiile sunt stocate când folosim programarea dinamică sunt:

- **Memoizarea (Top-Down)** este abordarea recursivă, în care rezultatele sunt salvate într-o structură de date de unde vor fi accesate ulterior când este nevoie de rezultatul deja calculat;

- **Tabelizarea (Bottom-Up)** este abordarea în care se pornește de la calcularea celor mai mici subprobleme, construind treptat soluția finală din soluțiile subproblemelor mai mici. [5]

În general, paradigma programării dinamice este folosită pentru a rezolva problemele de optimizare care au ca obiectiv obținerea celei mai bune soluții într-un cadru de optimizare, fie pentru minimizare, fie pentru maximizare [3].

Ce este Problema Rucsacului?

Problema rucsacului este o problemă de optimizare. Presupunând că avem un rucsac care poate susține o anumită greutate maximă numită capacitate, trebuie să alegem un subset de obiecte dintr-o mulțime astfel încât valoarea totală a acestora să fie maximă, iar greutatea totală a obiectelor alese să nu depășească capacitatea rucsacului. Cele mai cunoscute variațiuni ale acestei probleme sunt varianta continuă și varianta discretă [3].

În varianta continuă, obiectele pot fi fracționate în mai multe bucăți, permițând alegerea parțială a unui obiect. Pentru această variantă a problemei, algoritmii greedy sunt mai potriviți deoarece criteriul local raport valoare/greutate conduce la soluția optimă [3].

În varianta discretă fiecare obiect poate fi ales o singură dată sau deloc. Pentru varianta discretă abordările greedy nu pot garanta că la finalul execuției algoritmului soluția obținută este optimă [3]. Acest lucru este datorat naturii acestor algoritmi de a alege varianta cea mai bună în momentul curent și ignoră alte combinații de obiecte care ar putea produce soluții cu un profit mai bun, de aceea acest tip de probleme necesită o abordare mai complexă precum programarea dinamică.

Formularea problemei

Pentru a înțelege mai bine ideea problemei și cum funcționează algoritmul vom considera ca exemplu următoarea instanță a problemei:

Date de intrare:

• $n = 4$, unde n este numărul total de obiecte pe care le avem la dispoziție;

• $c = 8$, unde c reprezintă capacitatea totală a rucsacului;

• *gains* $= [1, 2, 5, 6]$, unde *gains* reprezintă un vector de lungime n cu profitul pe care îl are fiecare obiect;

• *weights* $= [2, 3, 4, 5]$, unde *weights* reprezintă un vector de lungime n cu greutatea fiecărui obiect.

Scopul algoritmului este de a alege obiectele pentru a maximiza profitul obținut, respectând în același timp constrângerea de a nu depăși capacitatea rucsacului. Astfel, la fiecare pas algoritmul va trebui sa aleagă una din cele două opțiuni disponibile: fie obiectul va fi adăugat în rucsac integral, iar greutatea și profitul acestuia vor fi adăugate în soluție, fie obiectul nu va fi inclus în rucsac.

Voi explica următoarele notații care se aplică și în cadrul celorlalte capitole ale lucrării:

$\rightarrow$ i este folosit pentru a memora indexul obiectului curent;

$\rightarrow$ j este folosit pentru a memora valoarea curentă a capacității parțiale a rucsacului;

$\rightarrow$ tuplul (i, j), unde $0 \leq i < n$ și $0 \leq j < c$ reprezintă o subproblemă în care considerăm primele i obiecte și o capacitate parțială j a rucsacului.

În continuare, voi detalia o schiță a algoritmului implementat de către mine care poate fi folosită pentru a rezolva varianta discretă a problemei rucsacului și pentru a obține profitul maxim pentru orice instanță a problemei:

1. **Inițializarea:**

   - Declarăm o matrice *profits* de dimensiune $(n + 1) \times (c + 1)$, unde fiecare celulă *profits*$[i][j]$ reprezintă profitul maxim posibil obținut pentru o subproblemă în care sunt considerate disponibile primele i obiecte, iar rucsacul poate susține o capacitate maximă j;
   - Inițializăm prima linie din matrice cu 0. Acestea reprezintă cazurile în care pentru orice capacitate de la 0 până la j, nu avem obiecte disponibile, deci profitul maxim este 0.

2. **Procesul iterativ:** Pentru fiecare obiect i, unde $1 \leq i < n+1$ și fiecare capacitate j, unde $0 \leq j < c+1$:

(a) Capacitatea rucsacului este 0, deci profitul maxim posibil este de asemenea 0.

(b) Obiectul nu poate fi inclus

• greutatea obiectului curent $\mathit{weights}[i-1]$ depășește capacitatea j, atunci profitul pentru pasul curent rămâne același ca pentru $i-1$ obiecte: $\mathit{profits}[i][j] = \mathit{profits}[i-1][j]$.

(c) Știind că greutatea obiectului curent $\mathit{weights}[i-1]$ nu depășește capacitatea j, obiectul poate fi inclus, dar:

• dacă se obține un profit mai bun decât cel anterior prin includerea obiectului atunci adăugăm câștigul obiectului curent $\mathit{gains}[i-1]$ la valoarea obținută pentru capacitatea rămasă $j - \mathit{weights}[i-1]$: $\mathit{profits}[i][j] = \mathit{profits}[i-1][j = \mathit{weights}[i-1]] + \mathit{gains}[i-1]$.

• dacă se obține un profit cel mult la fel de bun ca profitul anterior obținut pentru aceeași capacitate j prin adăugarea obiectului atunci se va considera profitul maxim obținut fără acest obiect: $\mathit{profits}[i][j] = \mathit{profits}[i-1][j]$.

3. **Rezultatul:** La finalul execuției algoritmului, profitul maxim care se poate obține utilizând toate obiectele pe care le avem la dispoziție pentru un rucsac de capacitate maximă c se află în $\mathit{profits}[n][c]$.

Pentru a stoca soluțiile, algoritmul folosește abordarea Bottom-Up, în care punctul de start îl reprezintă cele mai mici subprobleme reprezentate de cazurile de bază. După cum am văzut în schița algoritmului prezentat anterior, adeseori pe parcusul execuției vom avea nevoie de profitul calculat în pasul anterior obținut pentru $i-1$ obiecte având aceeași capacitate j. Aici intervine proprietatea **problemelor suprapuse** caracteristică programării dinamice, în care avem nevoie de soluția unei subprobleme care a fost deja rezolvată, evitând astfel recalcularea acesteia.

De asemenea, rezolvarea problemei inițiale pentru un rucsac de capacitate c cu n obiecte poate fi construită din rezultatele obținute pentru subproblemele anterioare la care este adăugat profitul obiectului în funcție de capacitatea care permite includerea acestuia. De menționat este că la fiecare pas al procesului iterativ, mereu se va considera cel mai bun rezultat care se poate obține bazat pe două posibile decizii:

• Dacă obiectul curent este inclus în soluție, atunci rezultatul va fi calculat din profitul obiectului, la care se adaugă soluția subproblemei pentru greutatea rămasă $j - \mathit{weights}[i-1]$, soluție despre care știm că este optimă pentru i obiecte si capacitate $j - weights[i-1]$;

• Altfel, dacă obiectul nu este inclus, soluția va rămâne aceeași ca cea pentru $i-1$ obiecte și capacitate j, soluție despre care știm că este optimă pentru $i-1$ obiecte si capacitate j.

Aici intervine cea de-a doua proprietate specifică programării dinamice, numită proprietatea de **substructură optimă**, care constă în faptul că soluția optimă a unei probleme poate fi construită din soluțiile optime ale subproblemelor, subsoluții care sunt optime pentru subproblemele respective.

Pentru a obține profitul maxim pentru datele de intrare de mai sus putem aplica algoritmul descris și începem prin a stabili cazul de bază: pornim de la $i = 0$ (nu avem niciun obiect la dispoziție) și trecem prin fiecare valoare posibilă a capacității j. Pentru fiecare subproblemă $(0, j)$, unde $0 \leq j < c$, soluția optimă este cea de profit maxim 0:

Matricea *profits* pentru $i = 0$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0

Algoritmul continuă prin parcurgerea fiecărui obiect și fiecare valoare posibilă a greutății rucsacului. Pentru $i = 1$, profitul maxim care se poate obține este 1 deoarece avem un singur obiect la dispoziție indiferent de valoarea capacității rucsacului:

Matricea *profits* pentru $i = 1$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1

Pentru $i = 2$ considerăm doar primele două obiecte și iterăm prin valorile posibile ale capacității:

- Pentru subproblema $(2, 2)$ nu putem adăuga decât primul obiect, iar profitul optim rămâne 1;
- Pentru subproblema $(2, 3)$ putem adăuga al doilea obiect deoarece aduce un profit mai

bun decât cel anterior *profits*$[i-1][j]$ care este 1, iar restricția de a nu depăși capacitatea ruscacului este respectată;

- Pentru subproblema $(2,5)$ putem adăuga ambele obiecte în rucsac, iar profitul este calculat adunând câștigul obiectului cu profitul corespunzător capacității rămase după scăderea greutății obiectului, adică rezultatul subproblemei$(i-1, j-weights[i-1])$ care este 1:

Matricea *profits* pentru $i = 2$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3

Aplicând aceeași logică pentru $i = 3$, vom obține următoarele rezultate:

• Pentru subproblema $(3,4)$ putem adăuga al treilea obiect deoarece aduce un profit mai bun decât rezultatul suproblemei anterioare cu aceeași capacitate $(2,4)$;

• Pentru subproblema $(3,5)$ obținem un profit mai bun decât pentru $(2,5)$;

• Pentru subproblema $(3,6)$ putem adăuga primul și al treilea obiect aducând un profit mai bun decât profitul anterior pentru $(2,6)$;

• Pentru subproblema $(3,7)$ putem adăuga al doilea și al treilea obiect obținând un profit de valoare 7 și greutate $7 \leq j$, mai bun decât profitul anterior pentru $(2,7)$;

• Pentru subproblema $(3,8)$ se obține un profit mai bun decât profitul calculat pentru subproblema anterioară $(2,6)$:

Matricea *profits* pentru $i = 3$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7

Asemănător și pentru $i = 4$, când avem toate obiectele la dispoziție:

• Pentru subproblema $(4, 5)$ profitul calculat este mai bun decât profitul pentru $(3, 5)$;

• Ajungem la $(4, 8)$ care este problema inițială și putem adăuga al doilea și ultimul obiect, iar profitul maxim care se poate obține pornind de la datele de intrare este 8 adunând câștigul obiectului cu rezultatul subproblemei $(3, 3)$:

Matricea *profits* pentru $i = 4$

i\j	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

Capitolul 1

Reprezentarea datelor de intrare și de ieșire

În următoarele capitole voi detalia implementarea algoritmului, modul în care Dafny evaluează corectitudinea, dar și cum am reușit să demonstrez optimalitatea soluției finale.

Pentru a reprezenta problema pentru care algoritmul trebuie să obțină cel mai bun profit pe setul de intrare, am ales să definesc un tip de date `Problem` astfel:

```
datatype Problem = Problem(n: int,
  c: int, gains: seq<int>, weights: seq<int>)
```

unde:

- câmpul `n` reprezintă numărul total de obiecte disponibile;
- câmpul `c` reprezintă capacitatea totală a rucsacului;
- câmpul `gains` este o secvență ce conține profitul fiecărui obiect;
- câmpul `weights` este o secvență ce memorează stochează fiecărui obiect.

Folosind cuvântul cheie `datatype` am definit un nou tip de date numit `Problem` cu un singur constructor având același nume cu tipul, constructor în care câmpurile sunt separate prin virgulă.

Atât profitul, cât și greutatea fiecărui obiect pot fi accesate prin indicele corespunzător poziției din secvență. Spre exemplu, pentru a accesa greutatea celui de-al doilea obiect este folosit operatorul de indexare `weights`[1], deoarece indexarea celor două secvențe începe de la 0. În Dafny, o **secvență** este o colecție de elemente de același tip, iar în cazul acesta, de numere întregi.

Orice instanță a problemei oferită la intrare trebuie să fie validă și să respecte anumite condiții pentru ca algoritmul să poată produce o soluție corectă. Pentru a mă asigura că aceste condiții sunt îndeplinite a fost nevoie să formulez un predicat. **Predicatele** în Dafny sunt funcții ale călor rezultate sunt valori boolene [2]. Acestea sunt folosite pentru a evalua proprietăți care de obicei trebuie să fie adevărate și de aceea ele ajută în procesul de verificare al corectitudinii. Astfel, am definit predicatul `isValidProblem` care verifică dacă avem cel puțin un obiect la dispoziție și o capacitate mai mare decât zero, iar greutatea și câștigul fiecărui obiect au valori strict pozitive:

```
predicate isValidProblem(p: Problem)
{
  |p.gains| == |p.weights| == p.n &&
  p.n > 0 && p.c >= 0 &&
  hasPositiveValues(p.gains) && hasPositiveValues(p.weights)
}
```

Predicatul `hasPositiveValues` este definit astfel:

```
predicate hasPositiveValues(arr: seq<int>)
{
  forall i :: 0 <= i < |arr| ==> arr[i] > 0
}
```

și este folosit, după cum sugerează și numele acestuia, pentru a verifica dacă toate elementele unei secvențe precum `gains` sunt pozitive.

Datele de ieșire sunt reprezentate astfel:

```
(profit: int, solution: Solution)
```

unde:

- `profit` reprezintă profitul maxim care se poate obține pentru o anumită instanță a problemei;
- `solution` reprezintă un sinonim al tipului de date `seq<int>` folosit pentru a îmbunătăți claritatea codului, este declarat anterior astfel:

```
type Solution = seq<int>
```

și este o reprezentare binară a deciziei de includere a obiectului în rucsac. Prin urmare, valoarea 1 reprezintă faptul că obiectul aparține soluției, deci este adăugat în rucsac, iar profitul adus de acesta îmbunătățește câștigul final, iar 0 reprezintă faptul că obiectul nu este adăugat în rucsac, deci profitul obiectului nu este inclus în câștigul maxim obținut. Această secvență reprezintă soluția finală, care trebuie să fie optimă pentru problema completă.

Capitolul 2

Reprezentarea soluțiilor

În cadrul problemei rucsacului putem discuta despre ce reprezintă soluția optimă, cât și soluțiile parțiale optime.

Pentru a stoca rezultatele corespunzătoare câștigurilor subproblemelor am folosit o secvență de secvențe numită `profits`, de tip `seq<seq<int>>`. Deși această variabilă este o secvență de secvențe, ea poate fi privită ca o matrice bidimensională, iar rezultatele pot fi accesate folosind operatorul de indexare similar unui astfel de tablou: `profits`$[i][j]$, unde i și j reprezită mărimea subproblemei pe care încercăm să o rezolvăm.

Pe lângă matricea `profits`, această lucrare se mai bazează și pe o variabilă numită `solutions` de tip `seq<seq<seq<int>>>`, care va stoca fiecare secvență binară corespunzătoare profitului stocat în matricea `profits`. Ea poate fi privită tot ca un tablou multidimensional, iar valorile sunt accesate similar matricei `profits`.

O **soluție optimă** reprezintă selecția de obiecte care produce cel mai bun profit format din câștigul fiecărui obiect care poate fi adăugat în rucsac, dar care în același timp nu depășește capacitatea totală a rucsacului. Această soluție vizează problema completă, deținând decizia de includere a fiecărui obiect, iar profitul calculat trebuie să aibă cea mai mare valoare pe care o putem obține pentru instanța primită. În cazul acestei lucrări, pentru a verifica soluția finală (despre care am discutat în capitolul anterior) am formulat predicatul `isOptimalSolution` astfel:

```
ghost predicate isOptimalSolution(p: Problem,
    solution: Solution)
  requires isValidProblem(p)
  requires isValidPartialSolution(p, solution)
{
```

```
  isOptimalPartialSolution(p, solution, p.n, p.c) &&
  forall s: Solution :: (((isOptimalPartialSolution(
    p, s, p.n, p.c)) ==> gain(p, solution) >= gain(p, s)))
}
```

Cuvântul cheie `ghost` este folosit pentru a marca faptul că acest predicat este folosit doar în partea de verificare și nu este inclus în codul executabil [2].

Condițiile exprimate folosind clauza `requires` sunt numite **precondiții** și reprezintă proprietăți pe care parametrii trebuie să îi respecte înainte de intrarea în corpul unei funcții, a unei metode, leme sau predicat și care vor fi valabile până la ieșirea din acestea [6]. Sunt esențiale pentru Dafny deoarece dacă acestea nu sunt satisfăcute la momentul apelului, atunci nu mai există nicio garanție privind comportamentul corect al programului.

O **soluție parțială** este o secvență binară, de lungime variabilă care nu depășește numărul de obiecte și care respectă constrângerea legată de capacitatea rucsacului. Predicatul formulat pentru a confirma o astfel de soluție este `isPartialSolution`:

```
predicate isPartialSolution(p: Problem, solution: Solution,
        i: int, j: int)
  requires isValidProblem(p)
  requires 0 <= i <= p.n
  requires 0 <= j <= p.c
{
  isValidPartialSolution(p, solution) && |solution| == i &&
  weight(p, solution) <= j
}
```

și depinde de valoarea de adevăr a predicatului `isValidPartialSolution`:

```
predicate isValidPartialSolution(p: Problem, solution: Solution)
  requires isValidProblem(p)
{
  hasAllowedValues(solution) && |solution| <= p.n
}
```

Cu ajutorul celor două predicate am verificat dacă o soluție este parțială și validă: are doar elemente de 0 și 1, lungimea acesteia nu depășește numărul de obiecte și are o greutate cel mult egală cu capacitatea j a rucsacului. Predicatul apelat `hasAllowedValues` este definit astfel:

```
predicate hasAllowedValues(solution: Solution)
{
  forall k :: 0 <= k < |solution| ==> solution[k] == 0
        || solution[k] == 1
}
```

și asigură că toate elementele unei soluții aparțin exclusiv mulțimii $\{0, 1\}$, unde 0 reprezintă faptul că obiectul nu este inclus în rucsac, iar 1 înseamnă că obiectul este inclus în rucsac.

O **soluție parțială optimă** reprezintă o secvență de elemente de 0 și 1, al cărei profit este optim pentru un subset de obiecte sau pentru o capacitate parțială a rucsacului. Luând un exemplu prezentat în introducere, o soluție parțială optimă pentru subproblema $(i = 3, j = 8)$ este $[0, 1, 1]$ deoarece aduce cel mai bun profit pentru un subset ce include doar primele trei obiecte, având la dispoziție un rucsac de capacitate maximă 8. Pentru a verifica dacă avem o astfel de soluție am implementat predicatul `isOptimalPartialSolution`:

```
ghost predicate isOptimalPartialSolution(p: Problem,
    solution: Solution, i: int, j: int)
  requires isValidProblem(p)
  requires 0 <= i <= p.n
  requires 0 <= j <= p.c
{
  isPartialSolution(p, solution, i, j) &&
  forall s: Solution :: (isPartialSolution(p, s, i, j) &&
    |s| == |solution| ==> gain(p, solution) >= gain(p, s))
}
```

Acest predicat verifică dacă orice altă soluție parțială pentru o subproblemă cu i obiecte și capacitate j a rucsacului va avea câștigul mai mic sau cel mult egal cu cel al soluției noastre.

Capitolul 3

Specificații

3.1 Predicate

Alte predicate formulate care au fost necesare în diverse puncte ale demonstrației sunt:

1. Predicatul `isSolution`:

```
predicate isSolution(p: Problem, solution: Solution)
  requires isValidProblem(p)
{
  isValidPartialSolution(p, solution) && |solution| == p.n &&
  weight(p, solution) <= p.c
}
```

   Acesta verifică dacă avem o soluție validă pentru problema completă.

2. Predicatul `isValidSubproblem`:

```
predicate isValidSubproblem(p: Problem, i: int, j: int)
{
  isValidProblem(p) && 1 <= i <= p.n && 1 <= j <= p.c
}
```

   Acesta definește o subproblemă a problemei inițiale, excluzând cazurile de bază.

3. Predicatul `areValidPartialSolutions`:

```
ghost predicate areValidPartialSolutions(p: Problem, profits:
    seq<seq<int>>, solutions: seq<seq<seq<int>>>,
```

```
    partialProfits: seq<int>, partialSolutions: seq<seq<int>>,
    i: int, j: int)
  requires isValidSubproblem(p, i, j)
{
  |partialSolutions| == |partialProfits| == j &&
  (forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)) &&
  (forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k])
}
```

Acesta validează rezultatele obținute doar pentru subproblemele pasului curent, în funcție de modificările cauzate de creșterea numărului de obiecte disponibile.

4. Predicatul `areValidSolutions`:

```
ghost predicate areValidSolutions(p: Problem, profits:
    seq<seq<int>>, solutions: seq<seq<seq<int>>>, i: int)
  requires isValidSubproblem(p, i, p.c)
{
  i == |profits| == |solutions| && (forall k :: 0 <= k < i
  ==> |profits[k]| == |solutions[k]| == p.c + 1) &&
  (forall k :: 0 <= k < |solutions| ==>
    forall q :: 0 <= q < |solutions[k]| ==>
      isOptimalPartialSolution(p, solutions[k][q], k, q)) &&
  (forall k :: 0 <= k < |solutions| ==>
    forall q :: 0 <= q < |solutions[k]| ==>
      gain(p, solutions[k][q]) == profits[k][q])
}
```

Acesta verifică faptul că soluțiile parțiale ale subproblemelor sunt valide și optime: au lungime corespunzătoare și aduc cel mai bun profit pentru subproblema pe care o calculează. De asemenea, ne asigură că soluțiile obținute la fiecare pas sunt construite pe baza soluțiilor optime ale subproblemelor anterioare.

3.2 Funcții

În Dafny, **funcțiile** nu pot avea efecte secundare și implementează funcții matematice. Corpul funcției reprezintă definiția acesteia, iar de obicei funcțiile nu necesită postcondiții. [2] Acestea returnează un rezultat la ieșire al cărui tip este specificat în semnătura funcției, imediat după lista de parametri. Sunt folosite în special pentru specificații.

Funcțiile importante pe care le-am implementat și care au fost necesare în demonstrație sunt:

1. Funcția `gain`:

```
function gain(p: Problem, solution: Solution): int
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= p.n
{
  if |solution| == 0 then 0
        else computeGain(p, solution, |solution| - 1)
}
```

   Această funcție returnează profitului corespunzător unei soluții valide, rezultat calculat de funcția `computeGain`, sau 0 dacă soluția nu are niciun element.

2. Funcția `computeGain`:

```
function computeGain(p: Problem, solution:
    Solution, i: int) : int
  requires isValidProblem(p)
  requires 0 <= i < |solution|
  requires 0 <= i < |p.gains|
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= |p.gains|
  ensures computeGain(p, solution, i) >= 0
{
  if i == 0 then solution[0] * p.gains[0] else
  solution[i] * p.gains[i] + computeGain(p, solution, i - 1)
}
```

Deoarece funcțiile nu au efecte secundare, parcurgerea soluției se face recursiv, iar finalitatea acesteia este asigurată prin condiția $i == 0$. Această funcție oferă o informație în plus care nu reiese direct din definiția funcției, dar este adevărată, mai exact faptul că la finalul execuției rezultatul obținut va fi mereu pozitiv sau egal cu zero. Acest lucru este specificat folosind clauza `ensures`, iar adnotările de acest gen sunt numite **postcondiții**. Ele sunt expresii logice care trebuie să fie adevărate după executarea logicii unei funcții, metode sau leme [6].

3. Funcția `weight`, asemănătoare din punct de vedere al implementării cu `gain`, returnează greutatea totală a obiectelor corespunzătoare unei soluții valide, sau 0 dacă soluția nu conține niciun obiect.

4. Funcția `computeWeight`, este asemănătoare din punct de vedere al implementării cu `computeGain`, dar aceasta parcurge soluția pentru a calcula greutatea totală a obiectelor corespunzătoare pozițiilor valorilor de 1 din soluție.

5. Funcția `sumAllGains`:

```
function sumAllGains(p: Problem, i: int) : int
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 ensures sumAllGains(p, i) >= 0
{
  if (i == 1) then p.gains[0]
    else p.gains[i - 1] + sumAllGains(p, i -1)
}
```

Rezultatul acestei funcții reprezintă suma tuturor profiturilor obiectelor.

Capitolul 4

Punctul de intrare și arhitectura codului

Execuția algoritmului propus pornește de la o metodă principală numită `solve` ce primește ca parametru o problemă `p: Problem`. O **metodă** în Dafny reprezintă un fragment de cod ce conține operații executabile și ce modifică starea variabilelor definite [6]. Această metodă va trebui să întoarcă două rezultate, un rezultat este valoarea profitului maxim `profit: int` care se poate obține pentru instanța problemei oferită, iar cel de-al doilea rezultat este soluția `solution: Solution` corespunzătoare profitului obținut formată din elemente de 0 și 1:

```
method solve(p: Problem) returns (profit: int, solution: Solution)
  requires isValidProblem(p)
  ensures isSolution(p, solution)
  ensures isOptimalSolution(p, solution)
{
    var profits := [];
    var solutions := [];
    var i := 0;
    var partialProfits, partialSolutions :=
        solves0Objects(p, profits, solutions, i);
    profits := profits + [partialProfits];
    solutions := solutions + [partialSolutions];
    assert forall k :: 0 <= k < |partialSolutions| ==>
        isOptimalPartialSolution(p, partialSolutions[k], i, k);
    assert forall k :: 0 <= k < |solutions| ==>
        forall q :: 0 <= q < |solutions[k]| ==>
```

```
            gain(p, solutions[k][q]) == profits[k][q];
    i := i + 1;
    while i <= p.n
      invariant 0 <= i <= p.n + 1
      invariant |profits| == |solutions| == i
      invariant forall k :: 0 <= k < i ==> |profits[k]| == p.c + 1
      invariant forall k :: 0 <= k < |solutions| ==>
        |solutions[k]| == p.c + 1
      invariant forall k :: 0 <= k < |solutions| ==>
        forall q :: 0 <= q < |solutions[k]| ==>
            isOptimalPartialSolution(p, solutions[k][q], k, q)
      invariant forall k :: 0 <= k < |solutions| ==>
        forall q :: 0 <= q < |solutions[k]| ==>
            gain(p, solutions[k][q]) == profits[k][q]
    {
        partialProfits, partialSolutions :=
            getPartialProfits(p, profits, solutions, i);
        profits := profits + [partialProfits];
        solutions := solutions + [partialSolutions];
        i := i + 1;
    }
    solution := solutions[p.n][p.c];
    assert isOptimalSolution(p, solution);
    profit := profits[p.n][p.c];
}
```

Postcondițiile `isSolution` și `isOptimalSolution` ne asigură că la finalul execuției metodei obținem o soluție validă, doar cu elemente de 0 și 1, de lungime egală cu numărul de obiecte, ce nu depășește capacitatea rucsacului și care are profit maxim.

Începem prin a inițializa matricile `profits` și `solutions` cu secvențe vide, iar numărul de obiecte disponibile este contorizat prin i și este inițializat cu 0. Vom obține rezultatele pentru cazurile de bază, mai exact pentru subproblemele în care $i = 0$ (mulțimea obiectelor disponibile este vidă) și pentru fiecare capacitate parțială a rucsacului invocând o altă metodă, numită `solves0Objects`, implementată astfel:

```
var j := 0;
while j <= p.c
  invariant 0 <= j <= p.c + 1
  invariant |partialProfits| == j
  invariant |partialSolutions| == j
  invariant forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
  invariant forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k]
  {
    partialProfits := partialProfits + [0];
    var currentSolution := [];
    emptySolOptimal(p, currentSolution, i, j);
    assert isOptimalPartialSolution(p, currentSolution, i, j);
    partialSolutions := partialSolutions + [currentSolution];
    j := j + 1;
  }
```

Știm că i este 0 în acest caz, mai exact nu avem niciun obiect la dispoziție, prin urmare cel mai bun profit care se poate obține este 0, iar soluțiile nu trebuie să conțină niciun element, deoarece nu avem ce să adăugăm în rucsac. Am folosit o buclă `while` pentru a considera toate valorile posibile pe care le poate avea capacitatea rucsacului, iar la fiecare iterație soluțiile, respectiv profiturile aferente, sunt memorate în `partialSolutions`, respectiv `partialProfits`.

Structurile repetitive reprezintă, de obicei, o provocare pentru Dafny deoarece nu este mereu clar câte iterații sunt necesare, dar și din cauza faptului că acesta trebuie să verifice anumite proprietăți care nu trebuie să se modifice pe parcursul procesului iterativ, ceea ce complică demonstrarea corectitudinii [6]. Astfel avem nevoie de adnotări pentru a ajuta verificatorul cu proprietățile care rămân adevărate pe parcursul structurii repetitive. Pentru acest lucru este folosită clauza **invariant**. Cu ajutorul acestor adnotări se pot specifica expresii logice care trebuie să fie adevărate pe toată durata structurii, inclusiv înainte de intrarea în buclă [6]. Invariantul

```
invariant forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
```

ne garantează faptul că pe parcursul buclei toate soluțiile calculate își păstrează proprietatea de **soluție parțială optimă**, iar invariantul

```
invariant forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k]
```

are scopul de a ajuta verificatorul să înțeleagă relația dintre cele două secvențe cu rezultate: aplicând funcția `gain` peste o soluție stocată în secvența `partialSolutions`, rezultatul acesta este cel stocat în aceeași poziție în secvența `partialProfits`. Pentru consistență, proprietățile definite de acești invarianți trebuie să fie respectate de toate soluțiile memorate pe parcurs, de aceea se vor repeta și în cadrul invarianților din celelalte metode.

De asemenea, deoarece, din motive de modularitate, proprietățile demonstrate în corpul buclelor sunt vizibile doar în blocul de instrucțiuni al structurii `while` [7], cu ajutorul invarianților acestea pot deveni vizibile și după execuția buclelor, fiind un ajutor important în verificarea postcondițiilor. Această metodă garantează astfel că soluțiile aferente cazurilor de bază sunt optime și respectă limitările impuse față de lungimea rezultatelor stocate.

După ce am obținut soluțiile cazurilor de bază, începem procesul iterativ. Metoda `solve` este cea care ține evidența numărului de obiecte și care salvează în matricile anterior menționate rezultatele concrete, care sunt calculate, verificate și returnate de către metoda `getPartialProfits`, în care fiecare ramură `if` corespunde unei decizii de acceptare sau respingere a obiectului:

```
method getPartialProfits(p: Problem, profits: seq<seq<int>>,
 solutions: seq<seq<seq<int>>>, i: int)
  returns (partialProfits: seq<int>, partialSolutions: seq<seq<int>>)
  requires isValidProblem(p)
  requires 0 < i < p.n + 1
  requires i == |profits| == |solutions|
  requires forall k :: 0 <= k < i ==> |profits[k]| == p.c + 1
  requires forall k :: 0 <= k < i ==> |solutions[k]| == p.c + 1
  requires forall k :: 0 <= k < |solutions| ==>
   forall q :: 0 <= q < |solutions[k]| ==>
    isOptimalPartialSolution(p, solutions[k][q], k, q)
  requires forall k :: 0 <= k < |solutions| ==>
```

```
    forall q :: 0 <= q < |solutions[k]| ==>
      isValidPartialSolution(p, solutions[k][q]) &&
      gain(p, solutions[k][q]) == profits[k][q]
  ensures p.c + 1 == |partialSolutions| == |partialProfits|
  ensures 0 <= |profits| <= p.n + 1
  ensures forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
  ensures forall k :: 0 <= k < |partialSolutions| ==>
    (isValidPartialSolution(p, partialSolutions[k]) &&
    gain(p, partialSolutions[k]) == partialProfits[k])
{
  var j := 0;
  partialProfits := [];
  partialSolutions := [];
  while j <= p.c
   invariant 0 <= j <= p.c + 1
   invariant 0 <= |profits| <= p.n + 1
   invariant j == |partialProfits| == |partialSolutions|
   invariant forall k :: 0 <= k < |partialSolutions| ==>
    isOptimalPartialSolution(p, partialSolutions[k], i, k)
   invariant forall k :: 0 <= k < |partialSolutions| ==>
    gain(p, partialSolutions[k]) == partialProfits[k]
   {
     if j == 0 {
      var currentProfit, currentSolution := solvesCapacity0(p, i, j);
      partialProfits := partialProfits + [currentProfit];
      partialSolutions := partialSolutions + [currentSolution];
      assert isOptimalPartialSolution(p, currentSolution, i, j);
     } else { if p.weights[i - 1] <= j {
       if p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]] >
                       profits[i - 1][j] {
         var currentProfit, currentSolution :=
           solvesAdd1BetterProfit(p, profits, solutions,
           partialProfits, partialSolutions, i, j);
```

```
          partialProfits := partialProfits + [currentProfit];
          partialSolutions := partialSolutions + [currentSolution];
          assert isOptimalPartialSolution(p, currentSolution, i, j);
        } else {
          var currentProfit, currentSolution :=
            solvesAdd0BetterProfit(p, profits, solutions,
            partialProfits, partialSolutions, i, j);
          partialProfits := partialProfits + [currentProfit];
          partialSolutions := partialSolutions + [currentSolution];
          assert isOptimalPartialSolution(p, currentSolution, i, j);
        }
      } else {
          var currentProfit, currentSolution := solvesAdd0TooBig(p,
          profits, solutions, partialProfits, partialSolutions, i, j);
          partialProfits := partialProfits + [currentProfit];
          partialSolutions := partialSolutions + [currentSolution];
          assert isOptimalPartialSolution(p, currentSolution, i, j);
      }
    }
    j := j + 1;
    }
}
```

Metoda primește ca parametri secvențele `profits` și `solutions`, în care sunt salvate rezultatele calculate în iterațiile anterioare pentru a evita recalcularea. Aceasta returnează alte secvențe în care sunt stocate rezultatele iterației curente după ce ne asigurăm că ele sunt corecte și optime pentru subproblemele pe care le rezolvă.

Invarianții folosiți aici sunt necesari pentru verificarea cu succes a metodei și de asemenea corespund cu postcondițiile, deoarece metoda apelantă `solve` trebuie să știe ce fel de rezultate primește înapoi. După cum se poate observa, invarianții sunt similari cu cei descriși pentru metoda `solves0Objects` și au rolul de a ajuta verificatorul să demonstreze corectitudinea postcondițiilor. Fără aceștia, el nu ar știi ce modificări s-ar produce în corpul buclei asupra variabilelor folosite.

Am folosit o buclă `while`, asemănător metodei `solves0Objects`, pentru a trece prin valorile parțiale ale capacității rucsacului, pornind de la 0 și incrementând cu 1 până la capacitatea totală a acestuia. Astfel, având metoda `solve` care iterează prin valorile posibile pentru i (ce reprezintă numărul de obiecte considerate) și metoda `getPartialProfits` care iterează prin valorile posibile pentru j (ce reprezintă capacitatea parțială pe care o poate avea rucsacul), acoperim toate subproblemele de care avem nevoie ca să ajungem la soluția finală.

Numărul de obiecte este fixat ca parametru al metodei `getPartialProfits`, deci trebuie să luăm în considerare doar valorile posibile ale capacității. Astfel, avem patru cazuri pe care trebuie să le rezolvăm:

- cazul în care capacitatea parțială a ruscacului este 0

  ```
  if j == 0
  ```

  este un caz special și poate fi considerat tot un caz de bază. Capacitatea fiind 0, avem un rucsac ce nu poate susține niciun obiect. Acest caz este tratat în metoda `solvesCapacity0`:

  ```
  currentProfit := 0;
  currentSolution := seq(i, y => 0);
  ```

  Astfel, cel mai bun profit care se poate obține este 0, iar soluțiile optime conțin doar elemente de 0 pe fiecare poziție.

- cazul în care greutatea obiectului depășește capacitatea parțială j a rucsacului

  ```
  p.weights[i - 1] > j
  ```

  aferent ultimului `if` din această metodă oferă iarăși o alegere relativ simplă. Obiectul nu poate fi adăugat în rucsac pentru subproblema cu i obiecte și capacitate j, astfel că profitul va rămâne același ca în pasul anterior pentru aceeași capacitate, respectiv `profits`$[i-1][j]$. Soluția pentru subproblema curentă va fi cea de la pasul anterior, respectiv `solutions`$[i-1][j]$, la care se va adăuga un 0 pentru a marca decizia luată în acest pas:

  ```
  currentProfit := profits[i - 1][j];
  currentSolution := solutions[i - 1][j];
  currentSolution := currentSolution + [0];
  ```

Acest caz este tratat în metoda `solvesAdd0TooBig`.

- cazul în care greutatea obiectului curent nu depășeste capacitatea rucsacului și adăugarea acestuia aduce un profit mai bun decât cel anterior pentru aceeași capacitate j:

```
if p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]] >
                     profits[i - 1][j]
```

  este cel care produce elementele de 1 în soluții, element care de această dată este adăugat soluției de pe poziția ce corespunde capacității rămase după ce am inclus greutatea obiectului. Profitul în acest caz este calculat adunând profitul de pe aceeași poziție a secvenței `profits` și câștigul obiectului curent:

```
currentProfit := p.gains[i - 1] +
    profits[i - 1][j - p.weights[i - 1]];
currentSolution := solutions[i - 1][j - p.weights[i - 1]];
currentSolution := currentSolution + [1];
```

  Acest caz este tratat în metoda `solvesAdd1BetterProfit`.

- cazul în care deși greutatea obiectului curent nu depășește capacitatea j, includerea acestuia nu produce un profit mai bun față de excluziunea lui. În acest caz, un 0 este adăugat soluției de la pasul `solutions`$[i-1][j]$, iar profitul rămâne același ca cel de pe poziția corespunzătoare din `profits`. Cazul este tratat în metoda `solvesAdd0BetterProfit` și are implementare similară cu `solvesAdd0TooBig`.

După cum am menționat, fiecare caz este tratat într-o metodă diferită, unde fiecare condiție din `if` din metoda curentă este o precondiție a metodei în care este tratat cazul respectiv. Alte precondiții similare pentru aceste metode vor fi limitările legate de numărul de obiecte, cât și informații despre soluțiile subproblemelor calculate în iterațiile anterioare. Acestea sunt necesare ca specificații, deoarece Danfy nu are acces la codul sursă și la specificațiile metodei apelante, fiecare metodă fiind verificată separat din motive de modularitate.

Capitolul 5

Leme importante

Lemele sunt declarații folosite atunci când unele proprietăți logice nu pot fi demonstrate automat de sistemul Dafny [2]. Acestea reprezintă un instrument de ghidare al verificatorului în demonstrarea corectitudinii programului. Sunt declarate separat și pot avea, asemănător metodelor, precondiții și postcondiții. Astfel, o lemă va fi demonstrată separat luând în calcul toate posibilitățile aplicabile pentru parametri ce îndeplinesc precondițiile, având ca scop verificarea cu succes a postcondiției [2], ce reprezintă proprietatea pe care vrem să o demonstrăm.

În acest capitol voi prezenta cele mai importante leme pe care le-am folosit.

Lema `computeGainAllZeros` este demonstrată prin inducție și este utilă pentru a arăta că dacă avem o soluție ce conține doar elemente de 0, atunci câștigul unei astfel de soluții nu poate fi decât 0:

```
lemma computeGainAllZeros(p: Problem, solution: Solution, i: int)
  requires isValidProblem(p)
  requires 0 <= i < |solution|
  requires 0 <= |solution| <= p.n
  requires forall k :: 0 <= k < |solution| ==> solution[k] == 0
  ensures computeGain(p, solution, i) == 0
{
  if i == 0 {
    assert  computeGain(p, solution, i) == 0;
  } else {
    computeGainAllZeros(p, solution, i - 1);
    assert computeGain(p, solution, i - 1) == 0;
```

```
    assert computeGain(p, solution, i) == 0;
  }
}
```

Am folosit această lemă în cadrul unei alte leme numită `optimalSolCapacity0`, care demonstrează că soluțiile calculate de metoda `solvesCapacity0` pentru subproblemele în care rucsacul nu permite adăugarea niciunui obiect sunt optime. În acest caz, pentru o problemă validă, o soluție parțială care conține doar valori de 0 este cea cu cel mai bun profit, din cauza limitării impuse de către parametrul de capacitate, care este 0:

```
lemma optimalSolCapacity0(p: Problem, solution: Solution, i: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires isPartialSolution(p, solution, i, 0)
 requires forall k :: 0 <= k < |solution| ==> solution[k] == 0
 requires weight(p, solution) == 0
 ensures isOptimalPartialSolution(p, solution, i, 0)
{
  assert isPartialSolution(p, solution, i, 0);
  forall s: Solution |
   isPartialSolution(p, s, i, 0) && |solution| == |s|
  ensures gain(p, solution) >= gain(p, s)
  {
    assert weight(p, solution) == 0;
    assert forall k :: 0 <= k < |solution| ==> solution[k] == 0;
    computeGainAllZeros(p, solution, |solution| - 1);
    gainCapacity0(p, s, i);
    assert gain(p, solution) == 0;
    assert gain(p, s) == 0;
  }
  assert forall s: Solution :: (isPartialSolution(p, s, i, 0) &&
   |s| == |solution| ==> gain(p, solution) >= gain(p, s));
  assert isOptimalPartialSolution(p, solution, i, 0);
}
```

Am folosit un `forall statement` cu ajutorul căruia am demonstrat că pentru toate soluțiile s care îndeplinesc proprietatea de **soluție parțială** de lungime i, câștigul acestora este cel mult egal cu cel al soluției presupusă a fi optimă. Pentru astfel de soluții am demonstrat că ele nu pot fi altele decât cele care conțin tot doar elemente de 0, iar câștigul acestora este de asemenea 0.

O altă lema ce abordează restricții impuse de paramentrul de capacitate este `optimalSolAdd0TooBig`:

```
lemma optimalSolAdd0TooBig(p: Problem, solution: Solution,
    i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 1 <= j <= p.c
 requires isOptimalPartialSolution(p, solution, i - 1, j)
 requires computeWeight(p, solution + [0],
    |solution + [0]| - 1) <= j
 requires p.weights[i - 1] > j
 ensures isOptimalPartialSolution(p, solution + [0], i, j)
{
  var s := solution + [0];
  weightAdd0(p, s);
  if !isOptimalPartialSolution(p, s, i, j) {
    existsOptimalPartialSol(p, i, j);
    var x : Solution :| isOptimalPartialSolution(p, x, i, j);
    gainAddTooBig(p, s, i, j);
    gainAddTooBig(p, x, i, j);
    var x1 := x[..i - 1];
    assert gain(p, x1) == gain(p, x) > gain(p, s);
    assert gain(p, s) == gain(p, solution) < gain(p, x);
    assert gain(p, x1) > gain(p, solution);
    assert isPartialSolution(p, x, i, j);
    assert x[i - 1] == 0;
    computeWeightAdd0(p, x, |x| - 2);
    assert weight(p, x) == weight(p, x1);
    assert isPartialSolution(p, x1, i - 1, j);
```

```
    assert !isOptimalPartialSolution(p, solution, i - 1, j);
    assert false;
  }
}
```

Aceasta este utilă când trebuie să demonstrez că soluția optimă pentru o subproblemă (i, j) nu include obiectul i dacă greutatea acestuia este mai mare decât capacitatea j. Această lemă este demonstrată prin reducere la absurd, iar optimalitatea unei astfel de soluții este arătată prin contrazicerea precondiției ce garantează existența unei soluții optime pentru $(i - 1, j)$.

O lemă foarte importantă în cadrul acestei lucrări este `optimalSolAdd1`, pe care o folosesc pentru a demonstra că adăugarea obiectului i în rucsac produce o soluție optimă pentru o subproblemă (i, j) ce respectă proprietățile specificate prin precondiții:

```
lemma optimalSolAdd1(p: Problem, profit1: int, profit2: int,
    solution1: Solution, solution2: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p,
    solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution1 + [1],
    |solution1 + [1]| - 1) <= j
 requires profit1 == gain(p, solution1)
 requires profit2 == gain(p, solution2)
 requires p.gains[i - 1] + profit1 > profit2
 ensures isOptimalPartialSolution(p, solution1 + [1], i, j)
{
  var s := solution1 + [1];
  if !isOptimalPartialSolution(p, s, i, j){
    existsOptimalPartialSol(p, i, j);
    var x : seq<int> :| isOptimalPartialSolution(p, x, i, j);
    assert gain(p, x) > gain(p, solution1 + [1]);
```

```
    if x[i - 1] == 0 {
      assert gain(p, x) <= profit2 by
      {
        gainAdd0(p, x);
        assert gain(p, x[..i - 1]) == gain(p, x);
        weightAdd0(p, x);
        assert weight(p, x[..i - 1]) <= j;
      }
      assert gain(p, solution1 + [1]) > profit2 by
      {
        gainAdd1(p, solution1 + [1]);
        assert gain(p, solution1 + [1]) ==
            gain(p, solution1) + p.gains[i - 1];
      }
      assert false;
    } else {
      gainAdd1Optimal(p, profit1, profit2,
        solution1, solution2, x, i, j);
      assert gain(p, x) == gain(p, solution1 + [1]);
    }
  }
}
```

Pentru a demonstra această lemă, am folosit metoda reducerii la absurd și am presupus că soluția calculată nu este optimă, prin urmare există o soluție x cu un profit mai bun. Pentru a demonstra că profitul soluției propuse este cel puțin la fel de bun ca profitul soluției x, am arătat că și profiturile subsoluțiilor din care extragem obiectul i sunt egale, iar în acest fel am putut evidenția proprietatea de substructură optimă din cadrul acestei probleme. Astfel am demonstrat că adăugarea obiectului curent aduce un profit la fel de bun ca cel al unei soluții presupuse optime x, deci soluția propusă este și ea optimă pentru subproblema curentă.

O altă lemă similară din punct de vedere al demonstrației, dar care tratează optimalitatea soluției pentru care adăugarea obiectului nu produce un profit mai bun, este `optimalSolAdd0`. Această lemă este demonstrată, de asemenea, folosind proprietatea de substructură optimă și analizând modificările aduse asupra profitului și greutății soluțiilor prin adăugarea sau eliminarea unui element de 0 sau 1:

```
lemma optimalSolAdd0(p: Problem, profit1: int, profit2: int,
    solution1: Solution, solution2: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p,
    solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution2 + [0],
    |solution2 + [0]| - 1) <= j
 requires profit1 == gain(p, solution1)
 requires profit2 == gain(p, solution2)
 requires p.gains[i - 1] + profit1 <= profit2
 ensures isOptimalPartialSolution(p, solution2 + [0], i, j)
{
  if !isOptimalPartialSolution(p, solution2 + [0], i, j) {
    existsOptimalPartialSol(p, i, j);
    var x : Solution :| isOptimalPartialSolution(p, x, i, j);
    if x[i - 1] == 1 {
      var x1 := x[..i - 1];
      assert gain(p, x1) == profit1 by {
        optimalSolRemove1(p, x, i, j);
        assert x1 == x[..|x| - 1];
        assert isOptimalPartialSolution(p, x1, i - 1,
            j - p.weights[i - 1]);
      }
      gainAdd1(p, x);
      gainAdd0(p, solution2 + [0]);
      assert gain(p, x) == gain(p, x1) +
        p.gains[i - 1] <= gain(p, solution2 + [0]);
      assert false;
```

```
    }
    assert x[i - 1] == 0;
    gainAdd0Optimal(p, profit1, profit2, solution1,
        solution2, x, i, j);
    assert gain(p, x) == gain(p, solution2 + [0]);
  }
}
```

Lemele `optimalSolRemove1` și `optimalSolRemove0` sunt similare în demonstrație și enunță proprietatea de substructură optimă necesară pentru verificarea lemelor prezentate anterior. Lema `optimalSolRemove1` demonstrează că având o soluție optimă pentru o subproblemă (i, j) ce include obiectul i, atunci prin extragerea acestuia obținem o soluție optimă pentru subproblema $(i-1, j-\texttt{p.weights}[i-1])$, ce consideră primele $i-1$ obiecte și o capacitate ce rămâne după eliminarea greutății obiectului i:

```
lemma optimalSolRemove1(p: Problem, solution: Solution,
    i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires isOptimalPartialSolution(p, solution, i, j)
 requires solution[i - 1] == 1
 ensures isOptimalPartialSolution(p, solution[..i - 1],
    i - 1, j - p.weights[i - 1])
{
  var s := solution[..i - 1];
  weightAdd1(p, solution);
  assert isPartialSolution(p, solution[..i - 1],
    i - 1, j - p.weights[i - 1]);
  if !isOptimalPartialSolution(p, solution[..i - 1],
    i - 1, j - p.weights[i - 1]) {
    gainAdd1(p, solution);
    existsOptimalPartialSol(p, i - 1, j - p.weights[i - 1]);
    var x : Solution :| isOptimalPartialSolution(p,
        x, i - 1, j - p.weights[i - 1]);
```

```
    assert |x| == |solution[..i - 1]|;
    assert gain(p, x) > gain(p, solution[..i - 1]);
    var x1 := x + [1];
    gainAdd1(p, x1);
    weightAdd1(p, x1);
    assert isOptimalPartialSolution(p, x1, i, j);
    assert s == solution[..|solution| - 1];
    assert x == x1[..|x1| - 1];
    assert gain(p, x1) == gain(p, x) + p.gains[i - 1] >
        gain(p, s) + p.gains[i- 1] == gain(p, solution);
    assert gain(p, x1) > gain(p, solution);
    assert false;
  }
}
```

În lemele prezentate anterior, folosite pentru a demonstra optimalitatea soluțiilor, a fost necesară presupunerea că avem la îndemână o altă soluție optimă pentru subproblema (i, j) pe care încercăm să o rezolvăm. Pentru a exprima faptul că există o astfel de soluție, am folosit cuantificatorul existențial `exists` în cadrul unei instrucțiuni `assert`, însă sistemul Dafny nu poate demonstra automat această proprietate [8]. Astfel, a fost nevoie să formulez o lemă numită `existsOptimalPartialSol`, ce folosește acest cuantificator și să ghidez verificatorul să demonstreze existența unei soluții optime pentru o subproblemă cu i obiecte și un rucsac de capacitate j:

```
lemma existsOptimalPartialSol(p: Problem, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 ensures exists s :: isOptimalPartialSolution(p, s, i, j)
{
  var k : int := 0;
  var completeSol := seq(i, y => 1);
  assert forall q :: 0 <= q < i ==> completeSol[q] == 1;
  var sum := sumAllGains(p, i);
  assert forall k :: 0 <= k < i ==> p.gains[k] > 0;
```

```
  if !exists s :: isOptimalPartialSolution(p, s, i, j) {
    var q := 0;
    var currentSol := seq(i, y => 0);
    computeWeightAllZeros(p, currentSol, |currentSol| - 1);
    computeGainAllZeros(p, currentSol, |currentSol| - 1);
    assert computeGain(p, currentSol, |currentSol|-1) == 0 >= q;
    assert sum == sumAllGains(p, i);
    while q < sum + 1
      invariant 0 <= q <= sum + 1
      invariant !exists s :: isOptimalPartialSolution(p, s, i, j)
      invariant !isOptimalPartialSolution(p, currentSol, i, j)
      invariant isPartialSolution(p, currentSol, i, j)
      invariant computeGain(p, currentSol, |currentSol|-1) >= q
    {
      assert exists s_i :: isPartialSolution(p, s_i, i, j) &&
        gain(p, s_i) > gain(p, currentSol);
      var s_i :| isPartialSolution(p, s_i, i, j) &&
        gain(p, s_i) > gain(p, currentSol);
      currentSol := s_i;
      q := computeGain(p, s_i, |s_i| - 1);
      gainUpperBound(p, s_i, i);
    }
    assert computeGain(p, currentSol, |currentSol|-1) >= sum + 1;
    gainUpperBound(p, currentSol, i);
    assert false;
  }
}
```

Pentru a demonstra acest lucru am presupus prin reducere la absurd că nu există o soluție optimă pentru subproblema (i, j), ceea ce înseamnă că pentru orice soluție există alta cu un profit mai mare. Pentru a itera prin soluțiile posibile am folosit o buclă `while`, iar invarianții sunt cei care asigură verificatorul că la fiecare iterație va exista o soluție cu un profit mai bun decât cea curentă. După un număr finit de pași, vom ajunge la o soluție ce are un profit mai mare decât valoarea tuturor obiectelor, lucru care nu este posibil, deci presupunerea

```
!exists s :: isOptimalPartialSolution(p, s, i, j)
```

pe care am făcut-o este falsă, iar postcondiția este verificată cu succes.

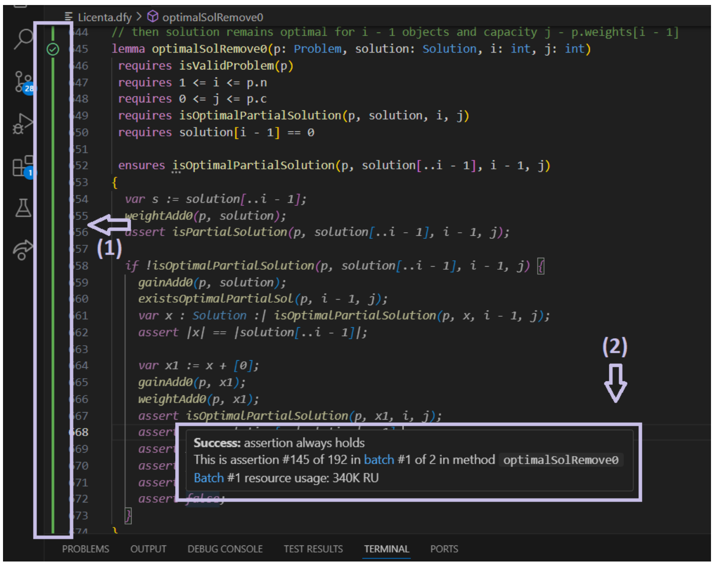
Capitolul 6

Provocări și aspecte practice ale procesului de lucru

În acest capitol aș dori să detaliez unele aspecte care consider că sunt destul de importante de menționat și unele provocări întâlnite pe parcursul procesului de dezvoltare.

Pentru realizarea proiectului am folosit Visual Studio Code ca mediu de dezolvare, acesta oferind o interfață intuitivă, dar și multe alte funcționalități care au ușurat procesul de dezvoltare și care sunt disponibile prin intermediul serverului de limbaj Dafny. Acesta furnizează instrumente pentru evidențierea sintaxei, verificarea codului pe măsură ce instrucțiunile sunt tastate și feedback vizual pentru starea curentă a verificării formale a programului [2].

Aceasta din urmă a fost, in mod special, foarte folositor pentru a mă asigura că verificarea formală a metodelor și a lemelor a trecut cu succes. Evidențierea verificării poate fi observată în partea din stânga a editorului (săgeata 1):



De exemplu, pentru lema `optimalSolRemove0`, Dafny a reușit să verifice cu succes corectitudinea acesteia, iar acest lucru este evidențiat printr-o bară verticală de culoare verde pe toată lungimea declarației acesteia. În funcție de starea procesului de verificare, această bară își va schimba stilul în mod dinamic pentru a ajuta programatorul în procesul de dezvoltare al codului.

De asemenea, dacă plasăm cursorul peste o instrucțiune, o fereastră pop-up apare în care putem afla informații despre stadiul verificării acesteia (săgeata 2). Acest lucru este de ajutor mai mult în cazul invarianților, pentru a afla dacă verificatorul nu poate demonstra validitatea acestora înainte de execuția buclei, sau în timpul acesteia.

Una dintre cele mai comune provocări pe care le-am întâmpinat pe parcursul procesului de implementare a fost depășirea timpului limită alocat pentru verificarea unei leme sau a unei metode, care poate apărea atunci când unele specificații nu sunt complete sau când demonstrația este prea complexă pentru verificator, dar există și cazuri în care acesta se pierde făcând lanțuri inutile de raționament [7]. În astfel de cazuri am avut la îndemână următoarele posibilități:

- Utilizarea instrucțiunilor `assert`: acestea sunt folosite pentru a verifica valoarea de adevăr a unei expresii logice necesare în demonstrație [2]. Cu ajutorul acestora am verificat dacă Dafny poate aproba raționamentul pe care l-am aplicat în demonstrarea postcondițiilor deoarece de foarte multe ori a fost nevoie să ghidez verificatorul spre o anumită direcție logică, dar și unele proprietăți care trebuiau să fie adevărate după apelarea unor leme pentru a continua verificarea. Un exemplu foarte bun pentru ambele cazuri este lema `gainAddTooBig`:

```
lemma gainAddTooBig(p: Problem, solution: Solution,
    i: int, j: int)
 ...
 requires isPartialSolution(p, solution, i, j)
 requires p.weights[i - 1] > j
 ensures solution[i - 1] == 0
 ensures gain(p, solution[..i - 1]) == gain(p, solution)
 {
    if solution[i - 1] == 1 {
      assert computeWeight(p, solution, |solution| - 1) ==
```

```
          computeWeight(p, solution, |solution[..i]| - 1) +
          p.weights[i - 1];
        assert weight(p, solution) >= p.weights[i - 1] > j;
        assert !isPartialSolution(p, solution, i, j);
        assert false;
      }
      assert solution[i - 1] == 0;
      computeGainAdd0(p, solution, |solution| - 2);
      assert gain(p, solution[..i - 1]) == gain(p, solution);
}
```

unde `assert-urile` din instrucțiunea `if` urmăresc să ajungem la o contradicție prin faptul că soluția nu are cum să fie parțială dacă greutatea obiectului depășește capacitatea j, propoziție care este falsă deoarece avem o precondiție care asigură exact acest lucru, iar instrucțiunea

```
assert gain(p, solution[..i - 1]) == gain(p, solution);
```

se asigură că după apelarea lemei `computeGainAdd0`, verificatorul știe că profitul unei soluții rămâne neschimbat dacă eliminăm ultima valoare de 0 din soluție.

De asemenea, folosind structura `assert ... by` putem reduce sarcina verificatorului deoarece pașii din acest bloc de instrucțiuni sunt utilizați doar pentru verificarea condiției din structură [7]:

```
assert gain(p, x[..i - 1]) == gain(p, solution1) by
{
  optimalSolRemove1(p, x, i, j);
  assert isOptimalPartialSolution(p, x[..i - 1],
    i - 1, j - p.weights[i - 1]);
}
```

- Utilizarea instrucțiunii `assume`: această instrucțiune este foarte utilă pentru a determina la ce linie întâlnește verificatorul probleme. O instrucțiune `assume` instruiește sistemul să trateze condiția ca fiind adevarată, indiferent dacă aceasta este adevărata sau nu [2]. De aceea, de multe ori am folosit-o pentru a afla de ce informații în plus are nevoie verificatorul sau pentru a putea continua următorii pași ai demonstrației.

De exemplu, mi-a fost de ajutor în cazul lemelor în care presupun că există o soluție mai bună decât cea pentru care vreau să demonstrez că este optimă, întrucât am putut continua implementarea acestora, după care am revenit să formulez o lemă specială pentru a demonstra existența unor astfel de soluții.

Utilizarea instrucțiunii `assume false`, care îi indică verificatorului să nu mai încerce demonstrarea condițiilor care urmează după aceasta și să accepte orice afirmație ulterioară ca fiind adevărată [2]. Această instrucțiune a fost foarte folositoare pentru a identifica ce `assert`-uri nu reușea Dafny să demonstreze pentru lemele mai complexe. Prin mutarea acestei instrucțiuni linie cu linie, am reușit să determin astfel instrucțiunile problematice.

- **Modularizare**: în leme precum `optimalSolAdd1` și `optimalSolAdd0`, unde deși demonstrația era corectă și fiecare caz (când ultimul element este fie 0, fie 1) se verifica cu succes separat, dacă ambele erau tratate în cadrul aceleași leme obțineam acest *timeout*, deoarece demonstrația necesita mai multe resurse decât erau alocate. Soluția a fost astfel să tratez unul dintre cele două cazuri posibile într-o lemă separată.

Concluzii

În cadrul acestei lucrări am implementat și demonstrat corectitudinea unui algoritm bazat pe tehnica programării dinamice ce rezolvă varianta discretă a problemei rucsacului folosind Dafny, un limbaj de programare dezvoltat pentru verificarea formală a programelor. De asemenea, cu ajutorul sistemului Dafny, am reușit nu doar să demonstrez corectitudinea algoritmului, dar și faptul că acesta construiește la fiecare pas cea mai bună soluție pentru dimensiunea curentă a problemei și resursele disponibile, garantând că soluția finală este cea optimă global.

Sinteza elementelor din cadrul acestei implementări este următoarea:

Elemente	**Număr**
Linii de cod	1013
Comentarii	24
Tipuri de date declarate	1
Sinonime ale tipurilor de date	1
Funcții	5
Predicate	11
Leme	26
Metode	8
Assert-uri	115
Precondiții	201
Postcondiții	50
Invarianți	26

Pe parcursul procesului de dezvoltare am aprofundat modul în care lucrează algoritmul și am înțeles mai bine cum sunt construite soluțiile parțiale. De asemenea, am dobândit o înțelegere mai detaliată asupra limbajului Dafny, în special a modului în care trebuie să definesc specificațiile și constrângerile unei probleme. O dificultate semnificativă pe care am întâmpinat-o a fost formularea și verificarea lemelor pentru care de foarte multe ori am obținut *timeout-uri* datorită complexității acestora sau a anumitor precondiții lipsă.

Pe baza acestei lucrări consider că sunt deschise oportunități spre îmbunătațire, de exemplu, prin optimizarea modului în care sunt memorate soluțiile parțiale. În implementarea curentă este utilizată o întreagă matrice pentru stocarea soluțiilor, însă acest lucru poate fi îmbunătățit prin utilizarea unui vector ce înlocuiește această matrice, deoarece pentru a rezolva fiecare subproblemă în care avem disponibile primele i obiecte avem nevoie doar de soluțiile subproblemelor ce consideră primele $i - 1$ obiecte.

De asemenea, ar fi foarte interesant să realizez o comparație între implementarea algoritmului în Dafny și alte implementări în diferite limbaje de verificare formală.

Această implementare poate fi utilă pentru cei care au nevoie să verifice automat instanțe specifice ale problemei rucsacului.

Codul sursă poate fi găsit la adresa `https://github.com/Haidau-Alina-Adriana/Licenta`. Pentru verificarea formală a acestei implementări am folosit versiunea 4.10.0 a serverului de limbaj Dafny.

Bibliografie

[1] "Dafny — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Dafny, 2024. Accessed: January 2025.

[2] K. R. M. Leino, R. L. Ford, and D. R. Cok, "Dafny reference manual." https://dafny.org/dafny/DafnyRef/DafnyRef, 2021. Accessed: January 2025.

[3] D. Lucanu and Ş. Ciobâcă, "Algorithm design - Lectures." https://sites.google.com/view/fii-pa/2024/lectures. Facultatea de informatică, Universitatea "Alexandru-Ioan Cuza" din Iași, Accessed: January 2025.

[4] Javatpoint, "Dynamic programming." https://www.javatpoint.com/dynamic-programming. Accessed: January 2025.

[5] GeeksforGeeks, "Dynamic programming (DP) introduction." https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials. Accessed: January 2025.

[6] J. Koenig and K. R. M. Leino, "Getting started with Dafny: A guide," in *Software Safety and Security - Tools for Analysis and Verification* (T. Nipkow, O. Grumberg, and B. Hauptmann, eds.), vol. 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pp. 152–181, IOS Press, 2012.

[7] "Verification optimization in Dafny." https://dafny.org/latest/VerificationOptimization/VerificationOptimization. Accessed: January 2025.

[8] J. Wilcox, "Frequently asked qustions." https://github.com/dafny-lang/dafny/wiki/FAQ/how-does-dafny-handle-quantifiers-ive-heard-about-triggers-what-are-those, 2022.

Codul sursă

```
datatype Problem = Problem(n: int, c: int, gains: seq<int>, weights: seq<int>)

type Solution = seq<int>

function gain(p: Problem, solution: Solution): int
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= p.n
{
  if |solution| == 0 then 0 else computeGain(p, solution, |solution| - 1)
}

function computeGain(p: Problem, solution: Solution, i: int) : int
  requires isValidProblem(p)
  requires 0 <= i < |solution|
  requires 0 <= i < |p.gains|
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= |p.gains|
  ensures computeGain(p, solution, i) >= 0
{
  if i == 0 then solution[0] * p.gains[0] else
        solution[i] * p.gains[i] + computeGain(p, solution, i - 1)
}

function weight(p: Problem, solution: Solution): int
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= p.n
{
  if |solution| == 0 then 0 else computeWeight(p, solution, |solution| - 1)
}

function computeWeight(p: Problem, solution: Solution, i: int) : int
  requires forall i :: 0 <= i < |p.weights| ==> p.weights[i] > 0
  requires 0 <= i < |solution|
  requires 0 <= i < |p.weights|
  requires hasAllowedValues(solution)
  requires 0 <= |solution| <= |p.weights|
  ensures computeWeight(p, solution, i) >= 0
{
  if i == 0 then solution[0] * p.weights[0] else
        solution[i] * p.weights[i] + computeWeight(p, solution, i - 1)
}

function sumAllGains(p: Problem, i: int) : int
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 ensures sumAllGains(p, i) >= 0
{
```

```
  if (i == 1) then p.gains[0] else p.gains[i - 1] + sumAllGains(p, i -1)
}

predicate hasPositiveValues(arr: seq<int>)
{
  forall i :: 0 <= i < |arr| ==> arr[i] > 0
}

predicate hasAllowedValues(solution: Solution)
{
  forall k :: 0 <= k < |solution| ==> solution[k] == 0 || solution[k] == 1
}

predicate isValidProblem(p: Problem)
{
  |p.gains| == |p.weights| == p.n &&
  p.n > 0 && p.c >= 0 &&
  hasPositiveValues(p.gains) && hasPositiveValues(p.weights)
}

predicate isValidPartialSolution(p: Problem, solution: Solution)
  requires isValidProblem(p)
{
  hasAllowedValues(solution) && |solution| <= p.n
}

predicate isPartialSolution(p: Problem, solution: Solution, i: int, j: int)
  requires isValidProblem(p)
  requires 0 <= i <= p.n
  requires 0 <= j <= p.c
{
  isValidPartialSolution(p, solution) && |solution| == i &&
  weight(p, solution) <= j
}

predicate isSolution(p: Problem, solution: Solution)
  requires isValidProblem(p)
{
  isValidPartialSolution(p, solution) && |solution| == p.n &&
  weight(p, solution) <= p.c
}

ghost predicate isOptimalPartialSolution(p: Problem, solution: Solution, i: int, j: int)
  requires isValidProblem(p)
  requires 0 <= i <= p.n
  requires 0 <= j <= p.c
{
  isPartialSolution(p, solution, i, j) &&
  forall s: Solution :: (isPartialSolution(p, s, i, j) && |s| == |solution|
      ==> gain(p, solution) >= gain(p, s))
}
```

```
ghost predicate isOptimalSolution(p: Problem, solution: Solution)
  requires isValidProblem(p)
  requires isValidPartialSolution(p, solution)
{
  isOptimalPartialSolution(p, solution, p.n, p.c) &&
  forall s: Solution :: (((isOptimalPartialSolution(p, s, p.n, p.c)) ==>
  gain(p, solution) >= gain(p, s)))
}

predicate isValidSubproblem(p: Problem, i: int, j: int)
{
  isValidProblem(p) &&
  1 <= i <= p.n &&
  1 <= j <= p.c
}

ghost predicate areValidSolutions(p: Problem, profits: seq<seq<int>>, solutions: seq<seq<seq<int>>>,
  i: int)
  requires isValidSubproblem(p, i, p.c)
{
  i == |profits| == |solutions| &&
  (forall k :: 0 <= k < i ==> |profits[k]| == |solutions[k]| == p.c + 1) &&
  (forall k :: 0 <= k < |solutions| ==> forall q :: 0 <= q < |solutions[k]| ==>
                  isOptimalPartialSolution(p, solutions[k][q], k, q)) &&
  (forall k :: 0 <= k < |solutions| ==> forall q :: 0 <= q < |solutions[k]| ==>
              gain(p, solutions[k][q]) == profits[k][q])
}

ghost predicate areValidPartialSolutions(p: Problem, profits: seq<seq<int>>, solutions:
        seq<seq<seq<int>>>, partialProfits: seq<int>, partialSolutions: seq<seq<int>>, i: int, j: int)
  requires isValidSubproblem(p, i, j)
{
  |partialSolutions| == |partialProfits| == j &&
  (forall k :: 0 <= k < |partialSolutions| ==> isOptimalPartialSolution(p, partialSolutions[k], i, k)) &&
  (forall k :: 0 <= k < |partialSolutions| ==> gain(p, partialSolutions[k]) == partialProfits[k])
}

// lemma which proves that adding 1 to a solution it will not exceed capacity j
lemma computeWeightFits1(p: Problem, solution: Solution, i: int, j: int)
  requires isValidProblem(p)
  requires 0 <= |solution| < p.n
  requires hasAllowedValues(solution)
  requires 0 <= j <= p.c + 1
  requires i == |solution|
  requires weight(p, solution) <= j - p.weights[i]
  ensures computeWeight(p, solution + [1], |solution + [1]| - 1) <= j
{
  var s := solution + [1];
  assert solution == s[..|s| - 1];
  for a := 0 to |s[..|s| - 1]|
```

```
    invariant 0 <= a <= |s[..|s| - 1]| + 1
    invariant forall k :: 0 <= k < a ==> computeWeight(p, solution, k) == computeWeight(p, s, k)
  { }
}

// lemma which proves that adding 0 to a solution will not exceed capacity j
lemma computeWeightFits0(p: Problem, solution: Solution, j: int)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= |solution| < p.n
  requires weight(p, solution) <= j
  ensures computeWeight(p, solution + [0], |solution + [0]| - 1) <= j
{
  if |solution| == 0 {
    assert weight(p, solution) <= j;
  } else {
    var s := solution + [0];
    assert s[..|s| - 1] == solution;
    computeWeightAdd0(p, s, |s| - 2);
    assert computeWeight(p, solution + [0], |solution + [0]| - 1) <= j;
  }
}

lemma computeWeightAdd0(p: Problem, solution: Solution, i: int)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= i < |solution| - 1
  requires 0 <= |solution| <= |p.weights|
  requires solution[|solution| - 1] == 0
  ensures computeWeight(p, solution, i) == computeWeight(p, solution[..|solution| - 1], i)
{ }

lemma weightAdd0(p: Problem, solution: Solution)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 < |solution| <= p.n
  requires solution[|solution| - 1] == 0
  ensures weight(p, solution) == weight(p, solution[..|solution| - 1])
{
  if |solution| == 1 {

  } else {
    assert computeWeight(p, solution, |solution| - 1) == computeWeight(p, solution, |solution| - 2);
    computeWeightAdd0(p, solution, |solution| - 2);
  }
}

lemma computeGainAdd0(p: Problem, solution: Solution, i: int)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= i < |solution| - 1
```

```
  requires 0 <= |solution| <= |p.gains|
  requires solution[|solution| - 1] == 0
  ensures computeGain(p, solution, i) == computeGain(p, solution[..|solution| - 1], i)
{ }

lemma gainAdd0(p: Problem, solution: Solution)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 < |solution| <= p.n
  requires solution[|solution| - 1] == 0
  ensures gain(p, solution) == gain(p, solution[..|solution| - 1])
{
  if |solution| == 1 {

  } else {
    assert computeGain(p, solution, |solution| - 1) == computeGain(p, solution, |solution| - 2);
    computeGainAdd0(p, solution, |solution| - 2);
  }
}

lemma computeGainRemoveLast(p: Problem, solution: Solution, i: int)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= i < |solution| <= |p.gains|
  requires i <= |solution[..|solution| - 1]| - 1
  ensures computeGain(p, solution, i) == computeGain(p, solution[..|solution| - 1], i)
{
  assert solution == solution[..|solution| - 1] + [solution[|solution| - 1]];
}

lemma gainAdd1(p: Problem, solution: Solution)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 < |solution| <= p.n
  requires solution[|solution| - 1] == 1
  ensures gain(p, solution) == gain(p, solution[..|solution| - 1]) + p.gains[|solution| - 1]
{
   if |solution| == 1 {

  } else {
    computeGainRemoveLast(p, solution, |solution[..|solution| - 1]| - 1);
    assert computeGain(p, solution, |solution[..|solution| - 1]| - 1) ==
           computeGain(p, solution[..|solution| - 1], |solution[..|solution| - 1]| - 1);
  }
}

lemma computeWeightRemoveLast(p: Problem, solution: Solution, i: int)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 <= i < |solution| <= |p.gains|
  requires solution == solution[..|solution| - 1] + [solution[|solution| - 1]]
```

```
  requires i <= |solution[..|solution| - 1]| - 1
  ensures computeWeight(p, solution, i) == computeWeight(p, solution[..|solution| - 1], i)
{ }

lemma weightAdd1(p: Problem, solution: Solution)
  requires isValidProblem(p)
  requires hasAllowedValues(solution)
  requires 0 < |solution| <= p.n
  requires solution[|solution| - 1] == 1
  ensures weight(p, solution) == weight(p, solution[..|solution| - 1]) + p.weights[|solution| - 1]
{
  if |solution| == 1 {

  } else {
    computeWeightRemoveLast(p, solution, |solution[..|solution| - 1]| - 1);
    assert computeWeight(p, solution, |solution[..|solution| - 1]| - 1) ==
           computeWeight(p, solution[..|solution| - 1], |solution[..|solution| - 1]| - 1);
  }
}

lemma emptySolOptimal(p: Problem, solution: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 0 <= j <= p.c
 requires |solution| == i == 0
 requires isPartialSolution(p, solution, i, j)
 ensures isOptimalPartialSolution(p, solution, i, j)
{
  forall s: Solution  | isPartialSolution(p, s, i, j) && |solution| == |s|
  ensures gain(p, solution) >= gain(p, s)
  { }
  assert forall s: Solution :: ((isPartialSolution(p, s, i, j) && |s| == |solution|
     ==> gain(p, solution) >= gain(p, s)));
}

lemma computeWeightAllZeros(p: Problem, solution: Solution, i: int)
  requires isValidProblem(p)
  requires 0 <= i < |solution|
  requires 0 <= |solution| <= p.n
  requires forall k :: 0 <= k < |solution| ==> solution[k] == 0
  ensures computeWeight(p, solution, i) == 0
{
  if i == 0 {
    assert  computeWeight(p, solution, i) == 0;
  } else {
    computeWeightAllZeros(p, solution, i - 1);
    assert computeWeight(p, solution, i - 1) == 0;
    assert computeWeight(p, solution, i) == 0;
  }
}

lemma computeGainAllZeros(p: Problem, solution: Solution, i: int)
```

```
  requires isValidProblem(p)
  requires 0 <= i < |solution|
  requires 0 <= |solution| <= p.n
  requires forall k :: 0 <= k < |solution| ==> solution[k] == 0
  ensures computeGain(p, solution, i) == 0
{
  if i == 0 {
    assert  computeGain(p, solution, i) == 0;
  } else {
    computeGainAllZeros(p, solution, i - 1);
    assert computeGain(p, solution, i - 1) == 0;
    assert computeGain(p, solution, i) == 0;
  }
}

//lemma which proves that if a partial solution has weight 0,
//then any subsolution will have weight 0 as well
lemma {:induction false} computeWeightCapacity0(p: Problem, solution: Solution, i: int, idx: int, x: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= |solution| <= p.n
 requires isPartialSolution(p, solution, i, 0)
 requires 0 <= x <= |solution| - 1
 requires computeWeight(p, solution, x) == 0
 requires 0 <= idx <= x
 ensures computeWeight(p, solution, idx) == 0
{
  if idx == x {
    assert computeWeight(p, solution, idx) == 0;
  } else {
    assert computeWeight(p, solution, x - 1) == 0 by
    {
      assert 0 <= idx < x;
      assert computeWeight(p, solution, x) == 0;
      assert p.weights[x] > 0;
      assert 0 <= solution[x] <= 1;
      assert solution[x] * p.weights[x] >= 0;
      assert computeWeight(p, solution, x) == solution[x] * p.weights[x] +
            computeWeight(p, solution, x - 1);
    }
    computeWeightCapacity0(p, solution, i, idx, x - 1);
    assert computeWeight(p, solution, idx) == 0;
  }
}

// lemma which proves that if a partial solution for capacity 0 with only zeros will have a gain of 0
lemma gainCapacity0(p: Problem, solution: Solution, i: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires |solution| == i
 requires isPartialSolution(p, solution, i, 0)
```

```
 ensures gain(p, solution) == 0
{
  var idx: int := 0;
  while idx < |solution|
    invariant 0 <= idx <= |solution|
    invariant forall k :: 0 <= k < idx ==> solution[k] == 0
    invariant idx > 0 ==> computeWeight(p, solution, idx - 1) == 0
  {
    assert p.weights[idx] > 0;
    computeWeightCapacity0(p, solution, i, idx, |solution| - 1);
    assert p.weights[idx] > 0 && computeWeight(p, solution, idx) == 0 ==> solution[idx] == 0;
    idx := idx + 1;
  }
  assert forall k :: 0 <= k < |solution| ==> solution[k] == 0;
  computeGainAllZeros(p, solution, |solution| - 1);
  assert gain(p, solution) == 0;
}

// lemma which proves that for allowed capacity j = 0, a solution with gain 0 is optimal
lemma optimalSolCapacity0(p: Problem, solution: Solution, i: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires isPartialSolution(p, solution, i, 0)
 requires forall k :: 0 <= k < |solution| ==> solution[k] == 0
 requires weight(p, solution) == 0
 ensures isOptimalPartialSolution(p, solution, i, 0)
{
  assert isPartialSolution(p, solution, i, 0);
  forall s: Solution | isPartialSolution(p, s, i, 0) && |solution| == |s|
  ensures gain(p, solution) >= gain(p, s)
  {
    assert weight(p, solution) == 0;
    assert forall k :: 0 <= k < |solution| ==> solution[k] == 0;
    computeGainAllZeros(p, solution, |solution| - 1);
    gainCapacity0(p, s, i);
    assert gain(p, solution) == 0;
    assert gain(p, s) == 0;
  }
  assert forall s: Solution :: (isPartialSolution(p, s, i, 0) && |s| == |solution| ==>
        gain(p, solution) >= gain(p, s));
  assert isOptimalPartialSolution(p, solution, i, 0);
}

// this lemma is a helper for optimalSolAdd0, x is an assumed better
// solution than solution2 + [0], solution2 is the optimal solution when
// considering the first i - 1 objects; adding a zero to solution2 will obtain a profit as good as x
lemma gainAdd0Optimal(p: Problem, profit1: int, profit2: int, solution1: Solution,
    solution2: Solution, x: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
```

```
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p, solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution2 + [0], |solution2 + [0]| - 1) <= j
 requires profit1 == gain(p, solution1)
 requires profit2 == gain(p, solution2)
 requires p.gains[i - 1] + profit1 <= profit2
 requires isOptimalPartialSolution(p, x, i, j)
 requires x[i - 1] == 0
 ensures gain(p, x) == gain(p, solution2 + [0])
{
  var x1 := x[..i - 1];
  assert x1 == x[..|x| - 1];
  gainAdd0(p, x);
  assert gain(p, x) == gain(p, x[..i - 1]) == gain(p, x1);
  optimalSolRemove0(p, x, i, j);
  assert isOptimalPartialSolution(p, x1, i - 1, j);
  assert gain(p, x1) == gain(p, solution2);
  gainAdd0(p, solution2 + [0]);
  assert gain(p, solution2) == gain(p, solution2 + [0]);
  gainAdd0(p, x);
  assert x == x1 + [0];
  assert gain(p, x1 + [0]) == gain(p, x) == gain(p, solution2 + [0]);
}

lemma optimalSolAdd0(p: Problem, profit1: int, profit2: int, solution1: Solution,
    solution2: Solution, i: int, j: int)
                                    // profit1 = profits[i - 1][j - p.weights[i - 1]]
                                    // profit2 = profits[i - 1][j]
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p, solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution2 + [0], |solution2 + [0]| - 1) <= j
 requires profit1 == gain(p, solution1)
 requires profit2 == gain(p, solution2)
 requires p.gains[i - 1] + profit1 <= profit2
 ensures isOptimalPartialSolution(p, solution2 + [0], i, j)
{
  if !isOptimalPartialSolution(p, solution2 + [0], i, j) {
    existsOptimalPartialSol(p, i, j);
    var x : Solution :| isOptimalPartialSolution(p, x, i, j);
    if x[i - 1] == 1 {
      var x1 := x[..i - 1];
      assert gain(p, x1) == profit1 by {
        optimalSolRemove1(p, x, i, j);
        assert x1 == x[..|x| - 1];
        assert isOptimalPartialSolution(p, x1, i - 1, j - p.weights[i - 1]);
      }
```

```
      gainAdd1(p, x);
      gainAdd0(p, solution2 + [0]);
      assert gain(p, x) == gain(p, x1) + p.gains[i - 1] <= gain(p, solution2 + [0]);
      assert false;
    }
    assert x[i - 1] == 0;
    gainAdd0Optimal(p, profit1, profit2, solution1, solution2, x, i, j);
    assert gain(p, x) == gain(p, solution2 + [0]);
  }
}

// if the last element's weight is bigger than allowed capacity j
// then it means the last element must be 0
lemma gainAddTooBig(p: Problem, solution: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 1 <= j <= p.c
 requires isPartialSolution(p, solution, i, j)
 requires |solution| >= 2
 requires p.weights[i - 1] > j
 ensures solution[i - 1] == 0
 ensures gain(p, solution[..i - 1]) == gain(p, solution)
 {
    if solution[i - 1] == 1 {
      assert computeWeight(p, solution, |solution| - 1) ==
        computeWeight(p, solution, |solution[..i]| - 1) + p.weights[i - 1];
      assert weight(p, solution) >= p.weights[i - 1] > j;
      assert !isPartialSolution(p, solution, i, j);
      assert false;
    }
    assert solution[i - 1] == 0;
    computeGainAdd0(p, solution, |solution| - 2);
    assert gain(p, solution[..i - 1]) == gain(p, solution);
}

lemma gainUpperBound(p: Problem, solution: Solution, i: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= |solution| <= |p.gains|
 requires isValidPartialSolution(p, solution) && |solution| >= i
 ensures computeGain(p, solution, i - 1) <= sumAllGains(p, i)
{
  var completeSol := seq(i, y => 1);
  assert forall q :: 0 <= q < i ==> completeSol[q] == 1;
  if i > 1 {
    gainUpperBound(p, solution, i - 1);
    assert computeGain(p, solution, i - 2) <= sumAllGains(p, i - 1);
  } else {
  }
}
```

```
lemma existsOptimalPartialSol(p: Problem, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 ensures exists s :: isOptimalPartialSolution(p, s, i, j)
{
  var k : int := 0;
  var completeSol := seq(i, y => 1);
  assert forall q :: 0 <= q < i ==> completeSol[q] == 1;
  var sum := sumAllGains(p, i);
  assert forall k :: 0 <= k < i ==> p.gains[k] > 0;
  if !exists s :: isOptimalPartialSolution(p, s, i, j) {
    var q := 0;
    var currentSol := seq(i, y => 0);
    computeWeightAllZeros(p, currentSol, |currentSol| - 1);
    computeGainAllZeros(p, currentSol, |currentSol| - 1);
    assert computeGain(p, currentSol, |currentSol| - 1) == 0 >= q;
    assert sum == sumAllGains(p, i);
    while q < sum + 1
      invariant 0 <= q <= sum + 1
      invariant !exists s :: isOptimalPartialSolution(p, s, i, j)
      invariant !isOptimalPartialSolution(p, currentSol, i, j)
      invariant isPartialSolution(p, currentSol, i, j)
      invariant computeGain(p, currentSol, |currentSol| - 1) >= q
    {
      assert exists s_i :: isPartialSolution(p, s_i, i, j) && gain(p, s_i) > gain(p, currentSol);
      var s_i :| isPartialSolution(p, s_i, i, j) && gain(p, s_i) > gain(p, currentSol);
      currentSol := s_i;
      q := computeGain(p, s_i, |s_i| - 1);
      gainUpperBound(p, s_i, i);

    }
    assert computeGain(p, currentSol, |currentSol| - 1) >= sum + 1;
    gainUpperBound(p, currentSol, i);
    assert false;
  }
}

lemma optimalSolAdd0TooBig(p: Problem, solution: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 1 <= j <= p.c
 requires isOptimalPartialSolution(p, solution, i - 1, j)
 requires computeWeight(p, solution + [0], |solution + [0]| - 1) <= j
 requires p.weights[i - 1] > j
 ensures isOptimalPartialSolution(p, solution + [0], i, j)
{
  var s := solution + [0];
  weightAdd0(p, s);
  if !isOptimalPartialSolution(p, s, i, j) {
    existsOptimalPartialSol(p, i, j);
```

```
    var x : Solution :| isOptimalPartialSolution(p, x, i, j);
    gainAddTooBig(p, s, i, j);
    gainAddTooBig(p, x, i, j);
    var x1 := x[..i - 1];
    assert gain(p, x1) == gain(p, x) > gain(p, s);
    assert gain(p, s) == gain(p, solution) < gain(p, x);
    assert gain(p, x1) > gain(p, solution);
    assert isPartialSolution(p, x, i, j);
    assert x[i - 1] == 0;
    computeWeightAdd0(p, x, |x| - 2);
    assert weight(p, x) == weight(p, x1);
    assert isPartialSolution(p, x1, i - 1, j);
    assert !isOptimalPartialSolution(p, solution, i - 1, j);
    assert false;
  }
}

// if solution is optimal for i objects and capacity j, if the last element which is 1 will be
// removed, then solution remains optimal for i - 1 objects and capacity j - p.weights[i - 1]
lemma optimalSolRemove1(p: Problem, solution: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires isOptimalPartialSolution(p, solution, i, j)
 requires solution[i - 1] == 1
 ensures isOptimalPartialSolution(p, solution[..i - 1], i - 1, j - p.weights[i - 1])
{
  var s := solution[..i - 1];
  weightAdd1(p, solution);
  assert isPartialSolution(p, solution[..i - 1], i - 1, j - p.weights[i - 1]);
  if !isOptimalPartialSolution(p, solution[..i - 1], i - 1, j - p.weights[i - 1]) {
    gainAdd1(p, solution);
    existsOptimalPartialSol(p, i - 1, j - p.weights[i - 1]);
    var x : Solution :| isOptimalPartialSolution(p, x, i - 1, j - p.weights[i - 1]);
    assert |x| == |solution[..i - 1]|;
    assert gain(p, x) > gain(p, solution[..i - 1]);
    var x1 := x + [1];
    gainAdd1(p, x1);
    weightAdd1(p, x1);
    assert isOptimalPartialSolution(p, x1, i, j);
    assert s == solution[..|solution| - 1];
    assert x == x1[..|x1| - 1];
    assert gain(p, x1) == gain(p, x) + p.gains[i - 1] > gain(p, s) + p.gains[i- 1] == gain(p, solution);
    assert gain(p, x1) > gain(p, solution);
    assert false;
  }
}

// if solution is optimal for i objects and capacity j, if the last element which is 0 will be
// removed, then solution remains optimal for i - 1 objects and capacity j - p.weights[i - 1]
lemma optimalSolRemove0(p: Problem, solution: Solution, i: int, j: int)
```

```
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires isOptimalPartialSolution(p, solution, i, j)
 requires solution[i - 1] == 0
 ensures isOptimalPartialSolution(p, solution[..i - 1], i - 1, j)
{
  var s := solution[..i - 1];
  weightAdd0(p, solution);
  assert isPartialSolution(p, solution[..i - 1], i - 1, j);
  if !isOptimalPartialSolution(p, solution[..i - 1], i - 1, j) {
    gainAdd0(p, solution);
    existsOptimalPartialSol(p, i - 1, j);
    var x : Solution :| isOptimalPartialSolution(p, x, i - 1, j);
    assert |x| == |solution[..i - 1]|;
    var x1 := x + [0];
    gainAdd0(p, x1);
    weightAdd0(p, x1);
    assert isOptimalPartialSolution(p, x1, i, j);
    assert s == solution[..|solution| - 1];
    assert x == x1[..|x1| - 1];
    assert gain(p, x1) == gain(p, x) >= gain(p, s) == gain(p, solution);
    assert gain(p, x1) == gain(p, solution);
    assert false;
  }
}

// this lemma is a helper for optimalSolAdd1, x is an assumed better
// solution than solution1 + [1], solution1 is the optimal solution if
// the first i - 1 objects are taken; adding a one to solution1 will obtain a profit as good as x
lemma gainAdd1Optimal(p: Problem, profit1: int, profit2: int, solution1: Solution,
    solution2: Solution, x: Solution, i: int, j: int)
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p, solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution1 + [1], |solution1 + [1]| - 1) <= j
 requires p.gains[i - 1] + profit1 > profit2
 requires isOptimalPartialSolution(p, x, i, j)
 requires x[i - 1] == 1
 ensures gain(p, x) == gain(p, solution1 + [1])
{
   assert gain(p, x) == gain(p, solution1 + [1]) by {
     gainAdd1(p, solution1 + [1]);
     gainAdd1(p, x);
     assert x == x[..i - 1] + [1];
     assert gain(p, x[..i - 1]) == gain(p, solution1) by
     {
       optimalSolRemove1(p, x, i, j);
```

```
        assert isOptimalPartialSolution(p, x[..i - 1], i - 1, j - p.weights[i - 1]);
      }
  }
}

lemma optimalSolAdd1(p: Problem, profit1: int, profit2: int, solution1: Solution,
    solution2: Solution, i: int, j: int)
                                    // profit1 = profits[i - 1][j - p.weights[i - 1]]
                                    // profit2 = profits[i - 1][j]
 requires isValidProblem(p)
 requires 1 <= i <= p.n
 requires 0 <= j <= p.c
 requires p.weights[i - 1] <= j
 requires isOptimalPartialSolution(p, solution1, i - 1, j - p.weights[i - 1])
 requires isOptimalPartialSolution(p, solution2, i - 1, j)
 requires computeWeight(p, solution1 + [1], |solution1 + [1]| - 1) <= j
 requires profit1 == gain(p, solution1)
 requires profit2 == gain(p, solution2)
 requires p.gains[i - 1] + profit1 > profit2
 ensures isOptimalPartialSolution(p, solution1 + [1], i, j)
{
  var s := solution1 + [1];
  if !isOptimalPartialSolution(p, s, i, j){
    existsOptimalPartialSol(p, i, j);
    var x : seq<int> :| isOptimalPartialSolution(p, x, i, j);
    assert gain(p, x) > gain(p, solution1 + [1]);
    if x[i - 1] == 0 {
      assert gain(p, x) <= profit2 by
      {
        gainAdd0(p, x);
        assert gain(p, x[..i - 1]) == gain(p, x);
        weightAdd0(p, x);
        assert weight(p, x[..i - 1]) <= j;
      }
      assert gain(p, solution1 + [1]) > profit2 by
      {
        gainAdd1(p, solution1 + [1]);
        assert gain(p, solution1 + [1]) == gain(p, solution1) + p.gains[i - 1];
      }
      assert false;
    } else {
      gainAdd1Optimal(p, profit1, profit2, solution1, solution2, x, i, j);
      assert gain(p, x) == gain(p, solution1 + [1]);
    }
  }
}

// the optimal solution/obtained profit for the n objects and
// capacity c will be the last element from solutions/profits
method solve(p: Problem) returns (profit: int, solution: Solution)
  requires isValidProblem(p)
```

```
  ensures isSolution(p, solution)
  ensures isOptimalSolution(p, solution)
{
    var profits := [];
    var solutions := [];
    var i := 0;
    var partialProfits, partialSolutions := solves0Objects(p, profits, solutions, i);
    profits := profits + [partialProfits];
    solutions := solutions + [partialSolutions];
    assert forall k :: 0 <= k < |partialSolutions| ==>
        isOptimalPartialSolution(p, partialSolutions[k], i, k);
    assert forall k :: 0 <= k < |solutions| ==> forall q :: 0 <= q < |solutions[k]| ==>
        gain(p, solutions[k][q]) == profits[k][q];

    i := i + 1;
    while i <= p.n
      invariant 0 <= i <= p.n + 1
      invariant |profits| == |solutions| == i
      invariant forall k :: 0 <= k < i ==> |profits[k]| == p.c + 1
      invariant forall k :: 0 <= k < |solutions| ==> |solutions[k]| == p.c + 1
      invariant forall k :: 0 <= k < |solutions| ==> forall q :: 0 <= q < |solutions[k]| ==>
        isOptimalPartialSolution(p, solutions[k][q], k, q)
      invariant forall k :: 0 <= k < |solutions| ==> forall q :: 0 <= q < |solutions[k]| ==>
        gain(p, solutions[k][q]) == profits[k][q]
    {
        partialProfits, partialSolutions := getPartialProfits(p, profits, solutions, i);
        profits := profits + [partialProfits];
        solutions := solutions + [partialSolutions];
        i := i + 1;
    }
    solution := solutions[p.n][p.c];
    assert isOptimalSolution(p, solution);
    profit := profits[p.n][p.c];
}

// the case when no object is considered, optimal will be empty solutions with gain 0
method solves0Objects(p: Problem, profits: seq<seq<int>>, solutions : seq<seq<seq<int>>>, i: int)
                  returns (partialProfits: seq<int>, partialSolutions: seq<seq<int>>)
  requires isValidProblem(p)
  requires |profits| == |solutions| == i == 0
  ensures |partialProfits| == p.c + 1
  ensures |partialSolutions| == p.c + 1
  ensures forall k :: 0 <= k < |partialSolutions| ==>
                isOptimalPartialSolution(p, partialSolutions[k], i, k)
  ensures forall k :: 0 <= k < |partialSolutions| ==> (isValidPartialSolution(p, partialSolutions[k])
                    && gain(p, partialSolutions[k]) == partialProfits[k])
{
        partialProfits := [];
        var j := 0;
        partialSolutions := [];
        while j <= p.c
```

```
          invariant 0 <= j <= p.c + 1
          invariant |partialProfits| == j
          invariant |partialSolutions| == j
          invariant forall k :: 0 <= k < |partialSolutions| ==>
            isOptimalPartialSolution(p, partialSolutions[k], i, k)
          invariant forall k :: 0 <= k < |partialSolutions| ==>
            gain(p, partialSolutions[k]) == partialProfits[k]
        {
          partialProfits := partialProfits + [0];
          var currentSolution := [];
          emptySolOptimal(p, currentSolution, i, j);
          assert isOptimalPartialSolution(p, currentSolution, i, j);
          partialSolutions := partialSolutions + [currentSolution];
          j := j + 1;
        }
}

method getPartialProfits(p: Problem, profits: seq<seq<int>>, solutions: seq<seq<seq<int>>>, i: int)
                            returns (partialProfits: seq<int>, partialSolutions: seq<seq<int>>)
  requires isValidProblem(p)
  requires 0 < i < p.n + 1
  requires i == |profits| == |solutions|
  requires forall k :: 0 <= k < i ==> |profits[k]| == p.c + 1
  requires forall k :: 0 <= k < i ==> |solutions[k]| == p.c + 1
  requires forall k :: 0 <= k < |solutions| ==> forall q :: 0 <= q < |solutions[k]| ==>
    isOptimalPartialSolution(p, solutions[k][q], k, q)
  requires forall k :: 0 <= k < |solutions| ==> forall q :: 0 <= q < |solutions[k]| ==>
    isValidPartialSolution(p, solutions[k][q]) && gain(p, solutions[k][q]) == profits[k][q]
  ensures p.c + 1 == |partialSolutions| == |partialProfits|
  ensures 0 <= |profits| <= p.n + 1
  ensures forall k :: 0 <= k < |partialSolutions| ==>
                            isOptimalPartialSolution(p, partialSolutions[k], i, k)
  ensures forall k :: 0 <= k < |partialSolutions| ==> (isValidPartialSolution(p, partialSolutions[k])
                            && gain(p, partialSolutions[k]) == partialProfits[k])
{
        var j := 0;
        partialProfits := [];
        partialSolutions := [];
        while j <= p.c
          invariant 0 <= j <= p.c + 1
          invariant 0 <= |profits| <= p.n + 1
          invariant j == |partialProfits| == |partialSolutions|
          invariant forall k :: 0 <= k < |partialSolutions| ==>
            isOptimalPartialSolution(p, partialSolutions[k], i, k)
          invariant forall k :: 0 <= k < |partialSolutions| ==>
            gain(p, partialSolutions[k]) == partialProfits[k]
        {
          if j == 0 {
              var currentProfit, currentSolution := solvesCapacity0(p, i, j);
              partialProfits := partialProfits + [currentProfit];
              partialSolutions := partialSolutions + [currentSolution];
```

```
              assert isOptimalPartialSolution(p, currentSolution, i, j);
          } else {
            if p.weights[i - 1] <= j {
              if p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]] > profits[i - 1][j] {
                var currentProfit, currentSolution :=
                  solvesAdd1BetterProfit(p, profits, solutions, partialProfits, partialSolutions, i, j);
                partialProfits := partialProfits + [currentProfit];
                partialSolutions := partialSolutions + [currentSolution];
                assert isOptimalPartialSolution(p, currentSolution, i, j);
              } else {
                var currentProfit, currentSolution :=
                  solvesAdd0BetterProfit(p, profits, solutions, partialProfits, partialSolutions, i, j);
                partialProfits := partialProfits + [currentProfit];
                partialSolutions := partialSolutions + [currentSolution];
                assert isOptimalPartialSolution(p, currentSolution, i, j);
              }
            } else {
                var currentProfit, currentSolution :=
                  solvesAdd0TooBig(p, profits, solutions, partialProfits, partialSolutions, i, j);
                partialProfits := partialProfits + [currentProfit];
                partialSolutions := partialSolutions + [currentSolution];
                assert isOptimalPartialSolution(p, currentSolution, i, j);
            }
          }
          j := j + 1;
        }
}

// the case when allowed capacity is 0, so no object can be
// taken since every object's weight is bigger than 0
method solvesCapacity0(p: Problem, i: int, j: int)
        returns (currentProfit: int, currentSolution: Solution)
  requires isValidProblem(p)
  requires 1 <= i <= p.n
  requires j == 0
  ensures isOptimalPartialSolution(p, currentSolution, i, j)
  ensures currentProfit == gain(p, currentSolution)
{
    currentProfit := 0;
    currentSolution := seq(i, y => 0);
    computeWeightAllZeros(p, currentSolution, |currentSolution| - 1);
    optimalSolCapacity0(p, currentSolution, i);
    gainCapacity0(p, currentSolution, i);
    assert isOptimalPartialSolution(p, currentSolution, i, j);
}

// the case when a better profit is obtained if the object is taken into the knapsack
method solvesAdd1BetterProfit(p: Problem, profits: seq<seq<int>>, solutions: seq<seq<seq<int>>>,
    partialProfits: seq<int>, partialSolutions: seq<seq<int>>, i: int, j: int)
        returns (currentProfit: int, currentSolution: seq<int>)
  requires isValidSubproblem(p, i, j)
```

```
  requires areValidSolutions(p, profits, solutions, i)
  requires areValidPartialSolutions(p, profits, solutions, partialProfits, partialSolutions, i, j)
  requires p.weights[i - 1] <= j
  requires p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]] > profits[i - 1][j]
  ensures isOptimalPartialSolution(p, currentSolution, i, j)
  ensures currentProfit == gain(p, currentSolution)
{
    currentProfit := p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]];
    currentSolution := solutions[i - 1][j - p.weights[i - 1]];
    computeWeightFits1(p, currentSolution, i - 1, j);
    optimalSolAdd1(p, profits[i - 1][j - p.weights[i - 1]], profits[i - 1][j],
      currentSolution, solutions[i - 1][j], i, j);
    currentSolution := currentSolution + [1];
    gainAdd1(p, currentSolution);
    assert isOptimalPartialSolution(p, currentSolution, i, j);
}

// the case when a better profit is not obtained if the object is taken into the knapsack
method solvesAdd0BetterProfit(p: Problem, profits: seq<seq<int>>, solutions: seq<seq<seq<int>>>,
    partialProfits: seq<int>, partialSolutions: seq<seq<int>>, i: int, j: int)
        returns (currentProfit: int, currentSolution: seq<int>)
  requires isValidSubproblem(p, i, j)
  requires areValidSolutions(p, profits, solutions, i)
  requires areValidPartialSolutions(p, profits, solutions, partialProfits, partialSolutions, i, j)
  requires p.weights[i - 1] <= j
  requires p.gains[i - 1] + profits[i - 1][j - p.weights[i - 1]] <= profits[i - 1][j]
  ensures isOptimalPartialSolution(p, currentSolution, i, j)
  ensures currentProfit == gain(p, currentSolution)
{
    currentProfit := profits[i - 1][j];
    currentSolution := solutions[i - 1][j];
    computeWeightFits0(p, currentSolution, j);
    optimalSolAdd0(p, profits[i - 1][j - p.weights[i - 1]], profits[i - 1][j],
      solutions[i - 1][j - p.weights[i - 1]], currentSolution, i, j);
    currentSolution := currentSolution + [0];
    gainAdd0(p, currentSolution);
    assert isOptimalPartialSolution(p, currentSolution, i, j);
}

// the case when the weight of the object will exceed the knapsack capacity j
method solvesAdd0TooBig(p: Problem, profits: seq<seq<int>>, solutions: seq<seq<seq<int>>>,
    partialProfits: seq<int>, partialSolutions: seq<seq<int>>, i: int, j: int)
        returns (currentProfit: int, currentSolution: seq<int>)
  requires isValidSubproblem(p, i, j)
  requires areValidSolutions(p, profits, solutions, i)
  requires areValidPartialSolutions(p, profits, solutions, partialProfits, partialSolutions, i, j)
  requires p.weights[i - 1] > j
  ensures isOptimalPartialSolution(p, currentSolution, i, j)
  ensures currentProfit == gain(p, currentSolution)
{
    currentProfit := profits[i - 1][j];
```

```
    currentSolution := solutions[i - 1][j];
    computeWeightFits0(p, currentSolution, j);
    optimalSolAdd0TooBig(p, currentSolution, i, j);
    currentSolution := currentSolution + [0];
    gainAdd0(p, currentSolution);
    assert isOptimalPartialSolution(p, currentSolution, i, j);
}

method Main()
{
    var p: Problem := Problem(n := 4, c := 8,
                                    gains := [1, 2, 5, 6], weights := [2, 3, 4, 5]);
    var maximProfit, finalSolution := solve(p);
    print "\n Maxim profit is: ";
    print maximProfit;
    print "\n Optimal solution is: ";
    print finalSolution;
}
```