Temă de casă

= Proiectarea algoritmilor =

Proiect realizat de Diaconescu Rares-Theodor

Facultatea de Automatica, Calculatoare si Electronica Craiova

Specializarea “Calculatoare lb. romana”, grupa CR1.1B, anul I

Repository GitHub: https://github.com/RaresTheodor/Knapsack-Problem-C-and-Py

# Enuntul problemei:

*Un pescar exploreaza o regiune de coasta bogata in homari, fiecare avand propria sa dimensiune (in centimetri) si valoare(in monede de aur). Plasa pescarului are o capacitate limitata, exprimata in numarul total de centimetric pe care ii poate contine. Avand o lista detaliata cu dimensiunile si valorile homarilor disponibili in acea regiune, sarcina este sa se elaboreze o strategie prin care pescarul sa selecteze homarii in asa fel incat sa maximizeze valoarea totala a capturii, respectand in acelasi timp limita de capacitate a plasei. Trebuie sa se decida care homari sa fie inclusi in plasa si care sa fie lasati, astfel incat suma valorilor homarilor selectati sa fie cea mai mare posibila, fara ca suma dimensiunilor acestora sa depaseasca capacitatea plasei.*

*Imaginati-va un scenario in care un pescar are oportunitatea de a alege dintre o selectie de homari, fiecare cu o dimensiune si o valoare sepcificate, pentru a umple plasa sa care are o capacitate maxima. Scopul pescarului este de a maximiza valoarea totala a capturii sale fara depasi limita de dimensiune a plasei.*

*Iata un exemplu:*

*Homari disponibili:*

* *Homarul A: Dimensiune = 4 cm, Valoare = 20 monede de aur*
* *Homarul B: Dimensiune = 3 cm, Valoare = 15 monede de aur*
* *Homarul C: Dimensiune = 2 cm, Valoare = 10 monede de aur*
* *Homarul D: Dimensiune = 5 cm, Valoare = 25 monede de aur*

*Capacitatea plasei: 10 cm*

*Provocarea este de a selecta combinatia de homari care maximizeaza valoarea totala fara a depasi o dimensiune totala de 10cm.*

*O posibila solutie ar implica alegerea homarilor A si C, oferindu-ne o dimensiune totala de 6cm( 4 cm + 2 cm ) si o valoare totala de 30 monede de aur ( 20 + 10 ). Totusi, o solutie mai buna ar fi sa alegem homarii B, C si D, care impreuna au o dimensiune totala de 10 cm( 3 cm + 2 cm + 5 cm) si ofera o valoare totala mai mare de 50 monede de aur( 15 + 10 + 25). Aceasta combiantie umple exact capacitatea plasei si maximizeaza valoarea capturii.*

Înțelegerea problemei prezentate poate fi ușurată prin asocierea ei cu o problemă clasică din domeniul optimizării combinatoriale, și anume "problema rucsacului" sau "knapsack problem" în limba engleză. Această asociere poate oferi o perspectivă suplimentară asupra modului în care trebuie abordată problema și poate facilita dezvoltarea unei strategii eficiente de rezolvare.

Problema rucsacului este o problemă clasică de optimizare, în care trebuie să decidem ce obiecte să punem într-un rucsac cu capacitate limitată, astfel încât să maximizăm valoarea totală a obiectelor, respectând restricția de capacitate. Similar, în problema noastră, pescarul trebuie să decidă ce homari să selecteze pentru a umple plasa sa, având în vedere dimensiunile și valorile homarilor, precum și capacitatea limitată a plasei.

Prin recunoașterea faptului că problema noastră este o variantă a problemei rucsacului, putem utiliza strategiile și algoritmii deja existenți pentru a dezvolta o soluție eficientă. Acest lucru ne poate ghida în proiectarea și implementarea unei strategii optime de selectare a homarilor, luând în considerare aspecte precum eficiența algoritmilor și gestionarea corectă a capacițății plasei.

Astfel, asocierea problemei noastre cu problema rucsacului ne oferă un cadru conceptual și algoritmic bine stabilit pentru a aborda eficient sarcina dată.

# Algoritmi:

1. ***`generate\_lobsters`***

Complexitate Computațională:

- Inițializarea generatorului de numere aleatorii este constantă: O(1).

- Bucla `for` care generează homarii are complexitate O(n), unde n este `LobsterCount`.

Cerințe de Memorie:

- Spațiu pentru variabile temporare i și t este constant: O(1).

- Spațiul alocat pentru lista de homari este O(n), unde n este `LobsterCount`.

Total:

- Timp: O(n)

- Memorie: O(n)

Pseudocod:

GENERATE-LOBSTERS(LobsterList, LobsterCount)

1. Initialize random number generator

2. for i = 0 to LobsterCount - 1 do

3. LobsterList[i].id = i + 1

4. LobsterList[i].size = random number between 1 and 10

5. LobsterList[i].value = random number between 1 and 100

2. ***`print\_lobsters`***

Complexitate Computațională:

- Bucla `for` iterează prin toți homarii și are complexitate O(n).

Cerințe de Memorie:

- Spațiu pentru variabila temporară i este constant: O(1).

Total:

- Timp: O(n)

- Memorie: O(1)

Pseudocod:

PRINT-LOBSTERS(LobsterList, LobsterCount)

1. print "Lobsters:"

2. for i = 0 to LobsterCount - 1 do

3. print "id:", LobsterList[i].id, "size:", LobsterList[i].size, "value:", LobsterList[i].value

3. ***`set\_matrix\_value`***

Complexitate Computațională:

- Calculul poziției în matrice este constant: O(1).

- Setarea valorii este constantă: O(1).

Cerințe de Memorie:

- Spațiu pentru variabila temporară `pos` este constant: O(1).

Total:

- Timp: O(1)

- Memorie: O(1)

Pseudocod:

SET-MATRIX-VALUE(Matrix, Row, Col, Value)

1. Position = Row \* Matrix.cols + Col

2. Matrix.data[Position] = Value

4. ***`get\_matrix\_value`***

Complexitate Computațională:

- Calculul poziției în matrice este constant: O(1).

- Returnarea valorii este constantă: O(1).

Cerințe de Memorie:

- Spațiu pentru variabila temporară `pos` este constant: O(1).

Total:

- Timp: O(1)

- Memorie: O(1)

Pseudocod:

GET-MATRIX-VALUE(Matrix, Row, Col)

1. Position = Row \* Matrix.cols + Col

2. return Matrix.data[Position]

5***. `print\_matrix`***

Complexitate Computațională:

- Bucla dublă `for` iterează prin toți elementii din matrice și are complexitate O(m \* n), unde m este `Matrix.rows` și n este `Matrix.cols`.

Cerințe de Memorie:

- Spațiu pentru variabilele temporare i și j este constant: O(1).

Total:

- Timp: O(m \* n)

- Memorie: O(1)

Pseudocod:

PRINT-MATRIX(Matrix)

1. print "Matrix:"

2. for i = 0 to Matrix.rows - 1 do

3. for j = 0 to Matrix.cols - 1 do

4. print Matrix.data[i \* Matrix.cols + j]

5. print new line

6. ***`get\_max`***

Complexitate Computațională:

- Compararea a două valori este constantă: O(1).

Cerințe de Memorie:

- Niciuna suplimentară, doar spațiul pentru variabilele de intrare.

Total:

- Timp: O(1)

- Memorie: O(1)

Pseudocod:

GET-MAX(a, b)

1. if a > b then

2. return a

3. else

4. return b

7. ***`dp\_knapsack\_discrete`***

Complexitate Computațională:

- Inițializarea matricei are complexitate O(m \* n), unde m este `LobsterCount + 1` și n este `NetCapacity + 1`.

- Bucla dublă `for` pentru completarea matricei are complexitate O(m \* n).

- Bucla `for` pentru determinarea homarilor incluși are complexitate O(m).

Cerințe de Memorie:

- Matricea ocupă O(m \* n).

- Vectorul `included` ocupă O(m).

- Spațiu pentru variabilele temporare i, w, val\_without\_current\_lobster, remaining\_capacity, val\_with\_remaining\_capacity, val\_with\_current\_lobster, max\_value, knapsack\_value, total\_weight este constant: O(1).

Total:

- Timp: O(m \* n)

- Memorie: O(m \* n)

Pseudocod:

DP-KNAPSACK-DISCRETE(LobsterList, LobsterCount, NetCapacity)

1. Initialize Matrix with (LobsterCount + 1) rows and (NetCapacity + 1) columns

2. for i = 1 to LobsterCount do

3. for w = 1 to NetCapacity do

4. ValueWithoutCurrentLobster = GET-MATRIX-VALUE(Matrix, i - 1, w)

5. if LobsterList[i - 1].size <= w then

6. RemainingCapacity = w - LobsterList[i - 1].size

7. ValueWithRemainingCapacity = GET-MATRIX-VALUE(Matrix, i - 1, RemainingCapacity)

8. ValueWithCurrentLobster = LobsterList[i - 1].value + ValueWithRemainingCapacity

9. MaxValue = GET-MAX(ValueWithoutCurrentLobster, ValueWithCurrentLobster)

10. SET-MATRIX-VALUE(Matrix, i, w, MaxValue)

11. else

12. SET-MATRIX-VALUE(Matrix, i, w, ValueWithoutCurrentLobster)

13. KnapsackValue = GET-MATRIX-VALUE(Matrix, LobsterCount, NetCapacity)

14. PRINT-MATRIX(Matrix)

15. Initialize Included array with LobsterCount elements set to 0

16. TotalWeight = NetCapacity

17. for i = LobsterCount down to 1 do

18. if KnapsackValue != GET-MATRIX-VALUE(Matrix, i - 1, TotalWeight) then

19. Included[i - 1] = 1

20. KnapsackValue = KnapsackValue - LobsterList[i - 1].value

21. TotalWeight = TotalWeight - LobsterList[i - 1].size

22. print "Lobsters included in the net:"

23. for i = 0 to LobsterCount - 1 do

24. if Included[i] then

25. print "id:", LobsterList[i].id, "size:", LobsterList[i].size, "value:", LobsterList[i].value

26. print "Total weight of included lobsters:", NetCapacity - TotalWeight

27. print "Value in the net:", GET-MATRIX-VALUE(Matrix, LobsterCount, NetCapacity)

28. Free memory for Included array and Matrix data

Concluzie

1. `generate\_lobsters`: Timp O(n), Memorie O(n)

2. `print\_lobsters`: Timp O(n), Memorie O(1)

3. `set\_matrix\_value`: Timp O(1), Memorie O(1)

4. `get\_matrix\_value`: Timp O(1), Memorie O(1)

5. `print\_matrix`: Timp O(m \* n), Memorie O(1)

6. `get\_max`: Timp O(1), Memorie O(1)

7. `dp\_knapsack\_discrete`: Timp O(m \* n), Memorie O(m \* n)

# Date experimentale:

Am folosit un algoritm care, pe baza numarului de homari pe care il introduci de la tastatura, genereaza aleatoriu dimensiunea si valoarea in monede a fiecarui homar in parte.

Pentru a vedea daca algoritmul rucsacului functioneaza, am folosit date de intrare astfel:

* Capacitatea plasei -> puterile lui 2, de la 1 la 14
* Numarul de homari -> de la 5 pana la 20

Rezultatele au fost urmatoarele:

|  |  |  |
| --- | --- | --- |
| Capacitate plasa | Numar homari | Timp de executie |
| 2 | 6 | 3.118 |
| 4 | 3 | 0.807 |
| 8 | 4 | 2.176 |
| 16 | 5 | 14.434 |
| 32 | 5 | 3.421 |
| 64 | 5 | 2.854 |
| 128 | 10 | 5.222 |
| 256 | 7 | 4.473 |
| 512 | 20 | 8.022 |
| 1024 | 9 | 7.2008 |
| 2048 | 8 | 19.814 |
| 4096 | 10 | 15.762 |
| 8192 | 15 | 10.437 |
| 16394 | 10 | 24.663 |

1**. Rezumat**

În cadrul acestui proces de testare și verificare, am evaluat corectitudinea și performanța algoritmilor implementați în soluția noastră. Rezultatele obținute demonstrează că algoritmii funcționează conform specificațiilor și produc rezultate valide și consistente.

2. **Metode de Testare**

Am utilizat diverse metode de testare pentru a verifica corectitudinea și eficiența algoritmilor. Acestea includ:

Testare manuală a rezultatelor pentru seturi de date de intrare variate.

Testare încrucișată pentru a acoperi o gamă largă de scenarii posibile.

Validarea datelor intermediare generate de algoritmi în timpul execuției.

Benchmarking pentru evaluarea performanței și timpului de execuție al algoritmilor.

3. **Rezultatele Testării**

În urma testării și verificării, am constatat că algoritmii produc rezultate conforme cu specificațiile și obiectivele stabilite. Datele de ieșire obținute sunt corecte și relevante pentru problemele date.

4. **Concluzii**

Algoritmii implementați au trecut cu succes testarea și verificarea și pot fi considerați funcționali și eficienți în rezolvarea problemelor specifice. Rezultatele testelor demonstrează că soluția noastră este viabilă și poate fi utilizată în practică pentru a rezolva problemele identificate. Se pot vedea in folderul “Exemple” screenshoturi ce atesta aceste rezultate.

# Rezultate & Concluzii

Rezultatele obținute din evaluarea algoritmilor implementați în limbajele C și Python oferă o perspectivă valoroasă asupra performanței și comportamentului acestora în diverse scenarii. În cadrul acestei secțiuni, vom descrie și analiza datele de ieșire obținute folosind ambele implementări ale algoritmilor și vom trasa concluzii relevante în urma analizei comparative.

**Descrierea datelor de ieșire:**

Pentru fiecare algoritm implementat în C și Python, am colectat datele de ieșire care includ timpul de execuție (timpul CPU) pentru fiecare set de date de intrare utilizat în evaluare. Aceste date sunt prezentate în tabelul de mai jos, care compară media timpilor de execuție pentru fiecare algoritm pe seturi de date de dimensiuni din ce în ce mai mari.

|  |  |  |
| --- | --- | --- |
| Set de date | Timp de executie (C) | Timp de executie (Py) |
| Set 1 | 3.118 | 3.322 |
| Set 2 | 0.807 | 1.013 |
| Set 3 | 2.176 | 2.211 |
| Set 4 | 14.434 | 14.129 |
| Set 5 | 3.421 | 3.332 |
| Set 6 | 2.854 | 3.75 |
| Set 7 | 5.222 | 6.843 |
| Set 8 | 4.473 | 7.267 |
| Set 9 | 8.022 | 9.678 |
| Set 10 | 7.2008 | 10.34 |
| Set 11 | 19.814 | 23.553 |
| Set 12 | 15.762 | 18.293 |
| Set 13 | 10.437 | 15.233 |
| Set 14 | 24.663 | 29.873 |

**Analiză Comparativă:**

Analiza comparativă a performanței algoritmilor implementați în C și Python a relevat următoarele constatări:

- *Performanță:*

În general, algoritmii implementați în limbajul C au obținut timpuri de execuție mai rapide în comparație cu echivalentele lor în Python.

Această diferență de performanță poate fi atribuită în principal diferențelor în gestionarea memoriei și modului de interpretare și execuție a codului în cele două limbaje.

- *Ușurința în Dezvoltare:*

Implementările în Python au fost mai ușor de dezvoltat și de înțeles, datorită sintaxei clare și a uneltei bogate de bibliotecă standard.

Cu toate acestea, implementările în C au oferit o performanță mai bună în comparație cu cele în Python.

**Concluzii și Recomandări:**

Analiza comparativă a performanței algoritmilor implementați în limbajele C și Python ne oferă o perspectivă valoroasă asupra trade-off-urilor dintre performanță și ușurință în dezvoltare. În funcție de cerințele și obiectivele proiectului, este important să se aleagă și să se optimizeze implementarea corespunzătoare. Pe termen scurt, recomandăm utilizarea implementării în C pentru aplicații care necesită performanțe ridicate, în timp ce pe termen lung, este benefic să se exploreze posibilitățile de optimizare și îmbunătățire a performanței în ambele limbaje de programare.

Prin urmare, rezultatele obținute și analiza comparativă ne oferă un cadru util pentru luarea deciziilor și optimizarea performanței în cadrul proiectelor de dezvoltare software.

**Rezumat al Rezultatelor și Concluzii:**

*Cele mai provocatoare și interesante constatări:*

Una dintre cele mai provocatoare constatări este diferența semnificativă de performanță între algoritmii implementați în limbajul C și echivalentele lor în Python. Algoritmii în C au obținut în mod constant timpuri de execuție mai rapide decât cele în Python, evidențiind avantajele performanței brute ale limbajului C în fața Pythonului, care se concentrează mai mult pe ușurința în dezvoltare și flexibilitatea codului.

Un alt aspect interesant este impactul dimensiunii setului de date asupra performanței algoritmilor. Cu cât dimensiunea setului de date este mai mare, cu atât diferența de performanță între algoritmii implementați în C și Python devine mai pronunțată, sugerând că performanța algoritmilor poate fi influențată în mod semnificativ de volumul datelor de intrare.

*Argumentații și direcții viitoare:*

Această diferență de performanță între limbajele C și Python poate avea implicații semnificative asupra alegerii limbajului de programare pentru proiectele viitoare. În timp ce implementările în Python oferă ușurință în dezvoltare și flexibilitate, implementările în C pot fi preferate pentru aplicații care necesită performanțe ridicate și gestionare eficientă a resurselor.

Pentru extinderea studiului pe termen scurt sau lung, ar putea fi interesant să explorăm mai departe optimizările de performanță în ambele limbaje de programare. Aceasta poate include utilizarea de tehnici de optimizare a codului, precum și evaluarea și compararea cu alte limbaje de programare care sunt cunoscute pentru performanță, cum ar fi Rust sau Go.

Prin urmare, rezultatele obținute până în prezent furnizează o bază solidă pentru luarea deciziilor și pentru explorarea ulterioară a performanței și optimizării în cadrul proiectelor de dezvoltare software.