Tema 0 - Proiectarea algoritmilor

Dumitru Elena Bianca
324CD
email: elena.dumitru0312@stud.acs.upb.ro

Martie - Aprilie 2021

Facultatea de Automatica si Calculatoare
Universitatea Politehnica Bucuresti
Aprilie 2021

Cuprins

1 Tehnica Divide et Impera

1.1 Tiling Problem

1.1.1 Enunț

Se dă o tablă de dimensiune n x n, unde n este un număr de forma 2^k, cu k $\geq$ 1. Deci, n este o putere a lui 2 care poate lua valoarea minimă = 2.

Tabla are o căsuță lipsă (de dimensiune 1 x 1). Se cere să umplem tabla folosind obiecte în formă de L. Un obiect în formă de L este, prin definiție, un pătrat de dimensiune 2 x 2 căruia îi lipsește o căsuță 1 x 1.

1.1.2 Descrierea soluției problemei

Această problemă poate fi rezolvată folosind tehnica Divide et Impera deoarece poate fi împărțită în subprobleme mai mici, iar combinarea lor conduce la soluția finală a problemei.

Așadar, trebuie să tratăm cazul de bază, în care k = 1, adică dimensiunea tablei este de 2 x 2. Având 4 căsuțe disponibile, dintre care una blocată, putem așeza forma într-un singur fel și am rezolvat problema.

În continuare, plasăm un obiect în formă de L în partea centrală a tablei, reprezentată de cele 4 căsuțe centrale astfel încât să nu acopere pătrățelul interzis. Acum, putem să împărțim problema în alte 4 subprobleme (pentru fiecare sfert al tablei), de dimensiune $\frac{n}{2}$ x $\frac{n}{2}$, fiecare având câte o căsuță lipsă (adică o căsuță care nu poate fi ocupată).

Acum problema trebuie rezolvată recursiv pentru fiecare dintre cele 4 cazuri. Fie p_1, p_2, p_3, p_4 pozițiile celor 4 căsuțe lipsă din cele 4 table nou formate. Pentru a rezolva problema trebuie să apelăm recursiv funcția în toate cele 4 cazuri:

func($\frac{n}{2}$, p_1);

func($\frac{n}{2}$, p_2);

func($\frac{n}{2}$, p_3);

func($\frac{n}{2}$, p_4);

1.1.3 Pseudocod

Listing 1: Pseudocod Tiling Problem

```
cnt = 0;
board[size][size];

// Place the cnt-th tile
function place (x1, y1, x2, y2, x3, y3)
  cnt++;
  board[x1][y1] = cnt;
  board[x1][y1] = cnt;
  board[x1][y1] = cnt;
end function

function tile (n, x, y)
  int r, c;
  if n == 2
      cnt++;
      for i = 1 ... n
          for j = 1 ... n
              if arr[x + i][y + j] == 0
                  arr[x + i][y + j] = cnt;
              end if
          end for
      end for
      return 0;
  end if

  // find forbidden square
  for i = x ... x + n
      for j = y ... y + n
          if arr[i][j] != 0
              r = i; c = j;
          end if
      end for
  end for
```

```
    // We occupy the remaining squares corresponding
    // to the other quadrants

    // If the missing tile is in 1st quadrant

    if r < x + n/2 && c < y + n/2
        place(x + n/2, y + n/2 - 1,
              x + n/2, y + n/2,
              x + n/2 - 1, y + n/2);
    end if

    // If the missing tile is in the 2nd quadrant

    if r >= x + n/2 && c < y + n/2
        place(x + n/2 - 1, y + n/2,
              x + n/2, y + n/2,
              x + n/2 - 1, y + n/2 - 1);
    end if

    // If the missing tile is in the 3rd quadrant

    if r < x + n/2 && c >= y + n/2
        place(x + n/2, y + n/2 - 1
              x + n/2, y + n/2,
              x + n/2 - 1, y + n/2 - 1);
    end if

    // If the missing tile is in the 4th quadrant

    if r >= x + n/2 && c >= y + n/2
        place(x + n/2 - 1, y + n/2,
              x + n/2, y + n/2 - 1,
              x + n/2 - 1, y + n/2 - 1);
    end if

    // divide
    tile(n/2, x, y + n/2);
    tile(n/2, x, y);
    tile(n/2, x + n/2, y);
    tile(n/2, x + n/2, y + n/2);

    return 0;

  end function
```

```
main
  a, b;
  arr[a][b] = -1;
  tile(size, 0, 0);
  print arr;

  return 0;
end function
```

1.1.4 Complexitatea algoritmului

Putem sa deducem relatia de recurenta pentru acest algoritm din modul in care il divizam in subprobleme. Asadar, avem:
T(n) = 4T($\frac{n}{2}$) + O(1).
Folosind metoda Master de calculare a complexitatii, obtinem O(n^2).

1.1.5 Eficiența algoritmului propus

1.1.6 Exemplificări

2 Tehnica Greedy

2.1 Egyptian Fraction

2.1.1 Enunț

Orice fractie pozitiva poate fi reprezentata ca o suma finita de fractii unitare distincte. O fractie este unitara daca numaratorul este 1 si numitorul este un numar intreg pozitiv, de exemplu $\frac{1}{5}$.
O astfel de reprezentare se numeste Fractie Egipteana deoarece era folosita de egipteni in Antichitate.

2.1.2 Descrierea soluției problemei

Putem genera fractii egiptene folosind tehnica Greedy, algoritmul fiind dezvoltat de Fibonacci. Pentru un numar dat de forma '$\frac{nr}{dr}$' , unde dr $>$ nr, mai intai trebuie sa gasim cea mai mare fractie unitara a carei valoare este mai mica decat acest numar, iar apoi sa apelam algoritmul recursiv pentru diferenta lor.
De exemplu, pentru $\frac{6}{14}$, gasim intai "ceiling-ul" (floor + 1) pentru $\frac{14}{6} = 3$. Asadar, prima functie unitara devine $\frac{1}{3}$. Apoi vom apela functia recursiv pentru EgyptFract($\frac{6}{14}$ - $\frac{1}{3}$) = EgyptFract($\frac{4}{42}$), etc.

2.1.3 Pseudocod

Listing 2: Pseudocod Egyptian Fraction

```
function printEgyptian (int nr, int dr)

    // Daca numaratorul sau numitorul sunt 0
    if dr == 0 || nr == 0
        return;

    // Daca numarul divide numitorul, inseamna ca putem
    // forma fractia cautata printr-o singura impartire
    if dr % nr == 0
        print("1/" + dr / nr);
        return;

    // Daca numaratorul divide numitorul, inseamna ca
    // numarul primit ca input nu este o fractie
    if nr % dr == 0
        print(nr / dr);
```

```
        return;
```

```
    // Daca numaratorul este mai mare decat numitorul
    if nr > dr
        print(nr / dr + " + ");
        printEgyptian(nr % dr, dr);
        return;

    // Daca am ajuns aici, inseamna ca putem descompune in
    // continuare numarul in fractii unitare
    n = dr / nr + 1;
    print("1/" + n + " + ");
    printEgyptian(nr * n - dr, dr * n);

end function


main

    nr = 6; dr = 14;
    printEgyptian(nr, dr);
    return 0;

end fraction
```

2.1.4 Complexitatea algoritmului

Deoarece exista cazuri in care algoritmul poate rula la infinit, nu putem spune exact care este complexitatea algoritmului. Ar trebui stabilita o marja de eroare, de la care sa oprim cautarea.

2.1.5 Posibilitatea de obtinere a optimului global

Tehnica Greedy propusa de Fibonacci nu este mereu solutia cea mai eficienta. Un exemplu relevant ar fi urmatorul:

$$\frac{5}{121} = \frac{1}{25} + \frac{1}{757} + \frac{1}{873960180913} + \frac{1}{1527612795642093418846225}$$

$$\frac{5}{121} = \frac{1}{33} + \frac{1}{121} + \frac{1}{363}$$

Dupa cum se observa din exemplul 2, prin alte metode am putut obtine o extindere mai scurta si mai eficienta.

Asadar, de aici putem extrage concluzia ca acest algoritm nu respecta **proprietatea de substructura optimala** deoarece o solutie a unei subprobleme nu este continuta neaparat in solutia optima. La nivel teoretic, acest fapt se poate demonstra utilizand tehnica reducerii la absurd.

Proprietatea de alegere greedy este indeplinita atunci cand solutia optima a problemei fie este construita printr-o strategie greedy, fie poate fi transformata intr-o alta solutie optima construita pe baza strategiei greedy. Acest fapt poate fi demonstrat prin inductie matematica folosindu-ne de proprietatea de substructura optimala mentionata mai sus, deci nici aceasta proprietatea nu este indeplinita intotdeauna.

2.1.6 Exemplificari

Reprezentarea in fractii egiptene folosind tehnica Greedy pentru:

- $\frac{2}{3} = \frac{1}{2} + \frac{1}{6}$
- $\frac{6}{14} = \frac{1}{3} + \frac{1}{11} + \frac{1}{231}$
- $\frac{12}{13} = \frac{1}{2} + \frac{1}{3} + \frac{1}{12} + \frac{1}{156}$

3 Tehnica Programarii Dinamice

3.1 Wine Selling Problem

3.1.1 Enunț

Se dau preturile pentru N vinuri pe un rand. In fiecare an, putem vinde ori primul vin din rand, ori ultimul. Insa, preturile vinurilor cresc in timp. Fie profitul initial de la fiecare vin
P_1, P_2, P_3,..., P_n.
In anul Y, profitul de la vinul i va fi Y * P_i. Pentru fiecare an, trebuie sa hotaram daca vindem primul sau ultimul vin pentru a maximiza profiturile de la toate vinurile.
Se cere profitul maxim.

3.1.2 Descrierea soluției problemei

Daca am incerca toate solutiile posibile, apoi sa pastram solutia care ne da raspunsul maxim, am avea o complexitate exponentiala: $O(2^n)$. Asa, am alege de fiecare data primul sau ultimul vin si ii multiplicam pretul cu anul curent si ne mutam in mod recursiv la urmatoarea parte. In final, am selecta maximul dintre cele 2 probleme.

Daca observam arborele de recursivitate, observam ca intalnim aceleasi subprobleme in mai multe ramuri. Putem eficientiza acest proces de recalculare folosind memoizare si programare dinamica.

In acest sens, vom lua o matrice dp[N][N] unde vom tine minte rezultatele partiale, N reprezentand numarul de vinuri.
Initial, tot profitul din vanzari va fi 0. Pentru memoizare, vom tine minte punctul de start si punctul final.
Daca dp[start][end] este egal cu 0 inseamna ca nu am aflat inca rezultatul pentru acea perioada si trebuie calculat recursiv. Daca este diferit de 0, inseamna ca acel calcul a fost deja efectuat si putem intoarce rezultatul.

3.1.3 Pseudocod

Listing 3: Wine Selling Problem

```
function maxProfit (start, end, year, prices)

    // Daca intervalul nu mai are sens se termina cautarea
    if start > end
        return 0;

    // Daca valoarea de pe aceasta pozitie a fost deja
    // calculata
    if dp[start][end] != 0
        return dp[start][end];

    // Calculam valoarea de pe pozitia (start, end)
    else if start <= end
        return dp[start][end] = max (
            prices[start] * year +
            maxProfit(start + 1, end, year + 1, prices,
            prices[end] * year +
            maxProfit(start, end - 1, year + 1, prices);

    else
        return 0;

end function
```

3.1.4 Complexitatea algoritmului

Complexitatea algoritmului este O(n^2), cat dureaza pana completam matricea dp.

3.1.5 Relatia de recurenta

Cum am mai explicat, la fiecare pas care trebuie calculat trebuie sa ne raportam la profitul pe care il putem face anul acesta si influenta lui asupra profitului din anii urmatori. Practic matricea cu recurentele va fi completata de la final spre inceput, de la cazurile cele mai simple la cazul cel mai complex.

Asadar, de fiecare data vom alege maximul dintre primul sau ultimul vin * anul curent + profitul obtinut in mod recursiv de la pasi anteriori mai simpli.

Raspunsul problemei se va afla in matrice pe pozitia dp[0][n-1], care ia in calcul deci toate elementele initiale.

3.1.6 Exemplificari

Pentru exemplul: 2 4 6 2 5 procedam in felul urmator: Construim matricea dp:

–	0	1	2	3	4
0	-1	28	52	56	64
1	-1	-1	46	52	62
2	-1	-1	-1	38	53
3	-1	-1	-1	-1	33
4	-1	-1	-1	-1	-1

Obs! Daca dorim sa stim si ordinea in care au fost alese sticlele de vin, vom avea nevoie de o matrice suplimentara in care sa retinem pentru fiecare pozitie ce sticla a fost aleasa, spre exemplu 0 pentru prima sticla si 1 pentru ultima.

2 4 6 2 5 $\Longrightarrow$ gain = 0
2 4 6 2 5 $\Longrightarrow$ gain = 1 * 2 = 2
2 4 6 2 5 $\Longrightarrow$ gain = 2 * 5 = 10
2 4 6 2 5 $\Longrightarrow$ gain = 3 * 2 = 6
2 4 6 2 5 $\Longrightarrow$ gain = 4 * 4 = 16
2 4 6 2 5 $\Longrightarrow$ gain = 5 * 6 = 30
2 4 6 2 5 $\Longrightarrow$ total gain = 2 + 10 + 6 + 16 + 30 = 64

4 Tehnica Backtracking

4.1 Prime numbers after prime P with sum S

4.1.1 Enunț

Se dau 3 numere: suma S, numarul prim P si N. Gasiti toate variantele de N numere prime mai mari decat P a caror suma sa fie S.

4.1.2 Descrierea soluției problemei

Se genereaza toate numerele prime din intervalul [P + 1 ... S]. Pentru fiecare numar din aceasta multime, alegem daca il adaugam sau nu la solutie. Daca da, exploram in continuare in mod asemanator pana ajungem la suma dorita alegand N numere. Daca depasim totalul sau am epuizat numerele din multimea primes, dam return, adica facem backtrack si exploram cazul in care nu am fi adaugat numarul respectiv.

Pasii sunt prezentati mai in detaliu in pseudocodul de mai jos:

4.1.3 Pseudocod

Listing 4: Prime numbers problem

```
vector set, prime;

// Given function
function isPrime(x);

function primeSum(total, N, S, index)

    // We found the N primes with sum S
    if total == S && set.size() == N
        printSolution();
        return;

    // Total is greater than sum S
    // or if index reached the last element
    if total > S || index == prime.size()
        return;
```

```
    // Include the (index)th prime to total
    set.push(prime[index]);

    primeSum(total + prime[index], N, S, index + 1);

    // Remove the (index)th prime from solution vector
    // and from total
    set.pop(); // Backtrack

    primeSum(total, N, S, index + 1);

end function

function generatePrimes(N, S, P)

    // The primes have to be greater than P & less than S
    for i = P + 1 ... S
        if isPrime(i)
            prime.push(i);

    // Not enough numbers
    if prime.size() < N
        return;

    primeSum(0, N, S, 0);

end function
```

4.1.4 Complexitatea algoritmului

Deoarece alegem mai intai numerele prime din intervalul [P + 1 ... S], vom avea o complexitate de O(S - P) pentru aceasta operatie.

In continuare, va trebui sa alegem cate N combinatii de numere care sa dea suma S dintr-o multime cu cardinalul $\leq$ |S - P|. Asadar, complexitatea algoritmului va fi cel mult:

$$C\binom{N}{S-P} = \frac{(S-P)!}{N!(S-P-N)!}$$

Deoarece folosim backtracking, complexitatea problemei o sa fie putin mai buna de atat, insa ea ramane exponentiala.

4.1.5 Analiza eficientei

Solutia prezentata mai sus poate fi optimizata daca folosim **preprocesare**. Putem precalcula multimea numerelor prime din care putem alege folosind, de exemplu, Ciurul lui Eratostene.
In ceea ce priveste tehnica Backtracking, aceasta este aplicata in mod canonic: pentru fiecare adaugare, exploreaza toate optiunile ce deriva din aceasta decizie si se intoarce pas cu pas, daca nu se poate continua explorarea sau daca a gasit o solutie (o printeaza si ne intoarcem pentru a genera si toate combinatiile posibile, deoarece asa era indicat in cerinta).

4.1.6 Exemplificari

- Exemplul 1

  Input: N = 2, P = 7, S = 28
  Output: 11 17
  Explicatie: 11 si 17 sunt numere prime mai mari decat 7, iar suma lor este 28.

- Exemplul 2

  Input: N = 3, P = 2, S = 23
  Output: (3 7 13), (5, 7, 11)
  Explicatie: 3, 5, 7, 11 sunt numere prime mai mari decat 2 si 3 + 7 + 13 = 5 + 7 + 11 = 23.

- Exemplul 3

  Input: N = 4, P = 3, S = 54
  Output: (5, 7, 11, 31), (5, 7, 13, 29), (5, 7, 19, 23), (5, 13, 17, 19),
  (7, 11, 13, 23), (7, 11, 17, 19)
  Explicatie: toate aceste tupluri sunt formate din numere prime a caror suma este 54.

5 Analiza comparativa

Bibliografie

[1] https://www.geeksforgeeks.org/greedy-algorithm-egyptian-fraction/.

[2] https://www.geeksforgeeks.org/greedy-algorithm-egyptian-fraction/.

[3] https://www.geeksforgeeks.org/maximum-profit-sale-wines/.

[4] https://www.geeksforgeeks.org/prime-numbers-after-prime-p-with-sum-s/.