Programarea dinamica I

Principiile programării dinamice

Programarea dinamică, ca și metoda divide et impera, rezolvă problemele combinând soluțiile subproblemelor. După cum am văzut, algoritmii de divide et impera, partiționează problemele în subprobleme independente, rezolvă subproblemele în mod recursiv, iar apoi combină soluțiile lor pentru a rezolva problema inițială. Dacă subproblemele conțin subprobleme comune, în acest caz, este mai bine să folosim tehnica programării dinamice.

Să analizăm pentru început ce se întâmplă cu un algoritm de divide et impera în această situație. Descompunerea recursivă a cazurilor în subcazuri ale aceleiași probleme, care sunt apoi rezolvate în mod independent, poate duce la calcularea de mai multe ori a aceluiași subcaz, și deci la o eficiență scăzută a algoritmului. Să ne amintim de exemplu algoritmul lui Fibonacci expus în primul curs. Sau să calculăm coeficientul binomial

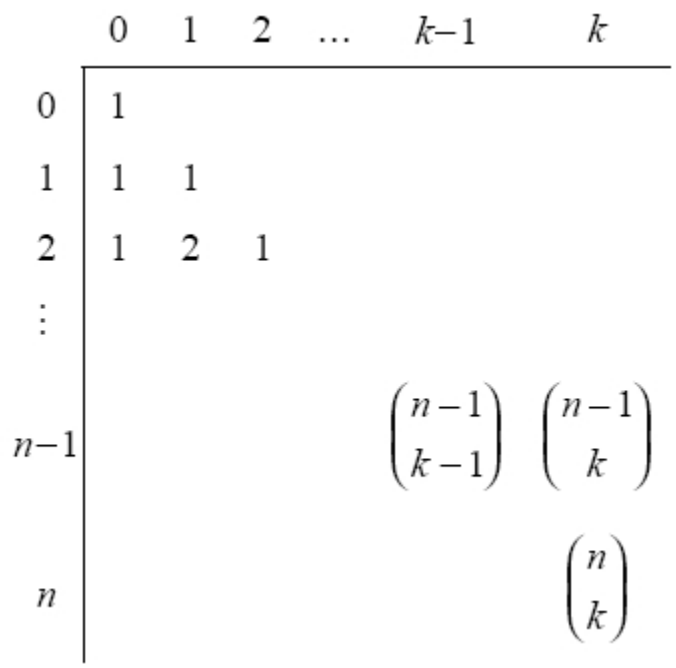
$$\binom{n}{n} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & pentru\ \ 0 < k < n \\ 1 & altfel \end{cases}$$

În mod direct aceasta se realizează astfel:

$FUNCTION\ C(n,k)$

1. $\boldsymbol{if}\ k = 0\ \boldsymbol{or}\ k = n\ \ \boldsymbol{then\ return}\ 1$
2. $\boldsymbol{else\ return}\ C(n-1,k-1) + C(n-1,k)$

Multe din valorile $C(i,j),\ i < n,\ j < k$ sunt calculate în mod repetat. Deoarece rezultatul final este obținut prin adunarea a $\binom{n}{k}$ de 1, rezultă că timpul de execuție pentru un apel $C(n,k)$ este în $\Omega(\binom{n}{k})$. Dacă memorăm aceste valori într-un tablou de forma celui din figura de mai jos, obținem un algoritm mai eficient.



	0	1	2	…	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
$n-1$					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n						$\binom{n}{k}$

Figura 1. Coeficienții binomiali

Acesta este triunghiul lui Pascal. Este suficient să memorăm un vector de lungime k, reprezentând linia curentă din triunghiul lui Pascal din figură. Dacă am completa o matrice de (n, k) putem spune că este vorba de elementele de sub diagonala principală. Noul algoritm necesită un timp de $O(nk)$.

Primul principiu de bază al programării dinamice este evitarea calculării de mai multe ori a aceluiași subcaz, prin memorarea rezultatelor intermediare.

Putem spune că metoda divide et impera operează de sus în jos, descompunând un caz în subcazuri din ce în ce mai mici pe care le rezolvă separat. Al doilea principiu fundamental al programării dinamice este faptul că operează de jos în sus. Se pornește de obicei de la cele mai mici cazuri. Combinând soluțiile lor se obțin soluții pentru subcazuri din ce în ce mai mari, până se ajunge în final la soluția cazului inițial.

Programarea dinamică este folosită de obicei în probleme de optimizare. În acest context, conform celui de-al treilea principiu fundamental, programarea dinamică este utilizată pentru a optimiza o problemă care satisface principiul optimalității: într-o secvență optimă de decizii sau alegeri, fiecare subsecvență trebuie să fie de asemenea, optimă. Cu toate că pare evident, acest principiu nu este întotdeauna valabil și aceasta se întâmplă atunci când subsecvențele nu sunt independente, adică atunci când optimizarea unei secvențe distruge optimizarea altei secvențe.

Ca și în cazul algoritmilor greedy, soluția optimă nu este în mod necesar unică. Dezvoltarea unui algoritm de programare dinamică poate fi descrisă de următoarea succesiune de pași:

- Se caracterizează structura unei soluții optime
- Se definește recursiv valoarea unei soluții optime
- Se calculează de jos în sus valoarea unei soluții optime

  Dacă pe lângă valoarea unei soluții optime se dorește și soluția propriu-zisă atunci se mai efectuează și acest pas:

  - Din informațiile calculate se construiește de sus în jos o soluție optimă.

O competi□ie

Din acest prim exemplu de programare dinamică reiese structura de control și ordinea rezolvării cazurilor. Problema considerată nu este una de optimizare.

Să ne imaginăm o competiție în care doi jucători A, B joacă o serie de cel mult $2n - 1$ partide, câștigător fiind jucătorul care acumulează primul n victorii. Presupunem că nu există partide egale, și că rezultatele sunt independente între ele și că pentru orice partidă există o probabilitate p ca jucătorul A să câștige, și o probabilitate $1 - p$ ca jucătorul B să câștige.

Ne propunem să calculăm $P(i, j)$, probabilitatea ca jucătorul $\boldsymbol{A}$ să câștige competiția, dat fiind că mai are nevoie de i victorii, iar jucătorul B mai are nevoie de j victorii pentru a câștiga. La început evident, probabilitatea este $P(n, n)$ pentru că fiecare jucător mai are nevoie de n victorii.

Pentru $1 \leq i \leq n$ avem $P(0, i) = 1$ implică $P(i, 0) = 0$. Probabilitatea $P(0,0)$ este nedefinită.

Pentru $i, j \geq 1$ se poate calcula $P(i, j)$ după formula:

$$P(i, j) = pP(i - 1, j) + qP(i, j - 1)$$

Algoritmul pentru a calcula această probabilitate este:

$FUNCTION\ P(i, j)$

1. $if\ i = 0\ \boldsymbol{then\ return}\ 0$
2. $if\ j = 0\ \boldsymbol{then\ return}\ 1$
3. $\boldsymbol{return}\ pP(i - 1, j) + qP(i, j - 1)$

Fie $t(k)$ timpul necesar, în cazul cel mai nefavorabil pentru a calcula probabilitatea $P(i, j)$ unde $k = i + j$.

Avem :

$t(1) \leq a$

$t(k) \leq 2t(k-1) + c,\ k > 1$

unde a, c sunt două constante. Prin metoda iterației, obținem $t \in O(2^k)$, iar dacă $i = j = n$ atunci $t \in O(4^n)$. Dacă urmărim modul în care sunt generate apelurile recursive, în figura 2, vom observa că algoritmul este la fel de ineficient ca cel al calculului coeficienților binomiali:

$$C(i+j, j) = C\big((i-1)+j, j\big) + C(i+(j-1), j-1)$$

Pentru a calcula numărul total de apeluri recursive necesare pentru a obține $C(n,k)$, notăm cu $r(n,k)$ numărul de apeluri recursive necesare pentru a-l calcula pe $C(n,k)$. Folosim inducția și anume:

Dacă n este 0, proprietatea ca $C(n,k) = 2\binom{n}{k} - 2$ este adevărată.

Presupunem proprietatea adevărată pentru $n-1$ și demonstrăm pentru n.

Fie $0 < k < n$ atunci avem recurența

$$r(n,k) = r(n-1, k-1) + \#r(n-1,k) + \#2$$

De aici rezultă că

$$r(n,k) = 2\binom{n-1}{k-1} - 2 + 2\binom{n-1}{k} - 2 + 2 = 2\binom{n}{k} - 2$$

În cazul în care k este 0 sau n atunci $r(n,k) = 0$ și $\binom{n}{k} = 1$, deci proprietatea este adevărată.

Revenind la problema de mai sus, rezultă că numărul total de apeluri recursive este $2\binom{i+j}{j} - 2$.

Timpul de execuție pentru un apel în $P(n,n)$ este deci în $\Omega(\binom{2n}{n})$ ceea ce înseamnă că este ineficient. Pentru a îmbunătăți algoritmul procedăm ca în cazul triunghiului lui Pascal. Se va completa un tablou bidimensional însă nu linie cu linie ci pe diagonală. Probabilitatea $P(n,n)$ poate fi calculată prin apelul $serie(n,p)$ al algoritmului de mai jos.

$FUNCTION\ serie(n, p)$

1. $\boldsymbol{array}\ P[0..n, 0..n]$
2. $q \leftarrow 1 - p$
3. $\boldsymbol{for}\ s \leftarrow 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
4. $\quad P[0, s] \leftarrow 1;\ P[s, 0] \leftarrow 0$
5. $\quad \boldsymbol{for}\ k \leftarrow 1\ \boldsymbol{to}\ s - 1\ \boldsymbol{do}$
6. $\quad\quad P[k, s - k] \leftarrow pP[k - 1, s - k] + qP[k, s - k - 1]$
7. $\boldsymbol{for}\ s \leftarrow 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
8. $\quad \boldsymbol{for}\ k \leftarrow 0\ \boldsymbol{to}\ n - s\ \boldsymbol{do}$
9. $\quad\quad P[s + k, n - k] \leftarrow pP[s + k - 1, n - k] + qP[s + k, n - k - 1]$
10. $\boldsymbol{return}\ P[n, n]$

Deoarece în esență se completează un tablou de $n \times n$ elemente, timpul de execuție pentru un apel $serie(n, p)$ este în $\Theta(n^2)$. Ca și în cazul coeficienților binomiali nu este nevoie de memorarea întregului tablou P, ci doar diagonala curentă din P.

Multiplicarea matricelor în lan□

Un alt exemplu concret de programare dinamică este un algoritm care rezolvă problema multiplicării unor matrice. Se dă o listă, o secvență de matrice $\langle A_1, A_2, \dots, A_n \rangle$ de n matrice, ce trebuie multiplicate și dorim să calculăm produsul $A_1 A_2 \dots A_n$. Putem evalua această expresie folosind un algoritm clasic de multiplicarea a matricelor, ca o subrutină, odată ce am pus parantezele, pentru a elimina ambiguitatea modului în care matricile sunt multiplicate. Un produs de matrice este complet parantezat dacă fie este o matrice, fie este produsul a două produse de matrice complet parantezate, evident înconjurate de paranteze. Multiplicarea matricelor este asociativă, așa că în orice mod am pune parantezele am obține același produs. De exemplu dacă lanțul de matrice ce trebuie înmulțite ar fi:

$\langle A_1, A_2, A_3, A_4 \rangle$ atunci el poate fi efectuat astfel

$\big(A_1\big(A_2(A_3A_4)\big)\big)$ sau

$\big(A_1\big((A_2A_3)A_4\big)\big)$ sau

$\big((A_1A_2)(A_3A_4)\big)$ sau

$\big(\big((A_1A_2)A_3\big)A_4\big)$

Modul în care efectuăm această ordonare a parantezelor poate avea un impact dramatic asupra eficienței multiplicării totale. Să luăm de exemplu costul multiplicării a două matrice. Algoritmul standard este dat de pseudocodul de mai jos. Atributele *rows* și *columns* sunt numărul de linii și coloane dintr-o matrice.

$MATRIX - MULTIPLY(A, B)$

1. $\boldsymbol{if}\ columns[A] \neq rows[B]\ \boldsymbol{then}\ error\ dimensiuni\ eronate$
2. $\boldsymbol{else}$
3. $\quad \boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ rows[A]\ \boldsymbol{do}$
4. $\quad\quad \boldsymbol{for}\ j \leftarrow 1\ \boldsymbol{to}\ columns[B]\ \boldsymbol{do}$
5. $\quad\quad C[i, j] \leftarrow 0$
6. $\quad\quad \boldsymbol{for}\ k \leftarrow 1\ \boldsymbol{to}\ columns[A]\ \boldsymbol{do}$
7. $\quad\quad\quad C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
8. $\quad \boldsymbol{return}\ C$

Putem înmulți două matrice A și B doar dacă numărul de coloane din A este același cu numărul de rânduri ale lui B. De exemplu dacă A este o matrice de $p \times r$ și B este o matrice de $q \times r$, rezultatul este o matrice C de $p \times r$. Acest algoritm are un ordin în $\theta(n^3)$.

Pentru a ilustra diversele costuri ce apar prin plasarea diferită a parantezelor să considerăm produsul a trei matrice $A_1A_2A_3$.

Să presupunem că matricele au dimensiunile $A_1 - 10 \times 100, A_2 - 100 \times 5$ și $A_3 - 5 \times 50$.

Dacă luăm în considerarea parantezele $\big((A_1A_2)A_3\big)$ vom avea nevoie de un număr de operații de multiplicare de $10 \cdot 100 \cdot 5 = 5000$ pentru a obține produsul A_1A_2. Apoi mai avem nevoie de $10 \cdot 5 \cdot 50 = 2500$ de înmulțiri a elementelor matricelor (A_1A_2) cu A_3. În total avem nevoie de 7500 de produse scalare.

Dacă luăm în considerare al doilea caz și anume $\big(A_1(A_2A_3)\big)$ mai întâi calculăm produsul (A_2A_3), pentru care avem nevoie de $100 \cdot 5 \cdot 50 = 25000$ de operații de înmulțire. Apoi vom avea nevoie de alte $10 \cdot 100 \cdot 50 = 50000$ pentru a calcula produsul dintre A_1 și (A_2A_3). În total obținem 75000 de multiplicări adică de 10 ori mai mult decât cazul anterior.

Problema *multiplicării matricelor în lanț* se formulează astfel:

Se dă un lanț de matrice $\langle A_1, A_2, \dots, A_n \rangle$ unde pentru un $i = 1,2, \dots, n$ o matrice A_i are dimensiunea $p_{i-1} \times p_i$. Să se parantezeze complet produsul $A_1 \cdot A_2 \cdot \dots \cdot A_n$ astfel ca numărul de multiplicări să fie minim.

Calculul complexită□ii algoritmului brute-force

Înainte de a rezolva problema prin programare dinamică, va trebui să ne convigem că o căutare exhaustivă este ineficientă. Fie numărul de parantezări alternative a unei secvențe de n matrice notat cu $P(n)$. Din moment ce putem separa secvența de n matrice între primele k matrice și cele de la $k+1$ până la n, pentru orice $k = 1,2, \dots, n-1$atunci parantezând recursiv obținem

$$P(n) = \begin{cases} 1 & dacă\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & dacă\ n \geq 2 \end{cases}$$

Problema se reduce la rezolvarea acestei recurențe folosind numere Catalane:

$P(n) = C(n-1)$ unde $C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(\frac{4^n}{n^{3/2}})$

Astfel se poate observa că metoda brute-force are o complexitate exponențială ceea ce o face ineficientă.

Structura unei parantezări optime

Primul pas din paradigma programării dinamice este de a caracteriza structura unei soluții optime. Pentru o multiplicare a unor matrici putem proceda după cum urmează. Să adoptăm notația $A_{i..j}$ pentru matricea ce rezultă din evaluarea produsului $A_i A_{i+1} \dots A_j$. O aranjare optimă a parantezelor a produsului $A_1 A_2 \dots A_n$ împarte produsul între A_k și A_{k+1} pentru un întreg k din intervalul $[1, n)$. Adică pentru o valoare a lui k, mai întâi calculăm matricele $A_{1..k}$ și $A_{k+1..n}$ și apoi le multiplicăm pentru a produce produsul final $A_{1..n}$. Costul parantezării optime este costul calculului matricei $A_{1..k}$ plus costul calculului matricei $A_{k+1..n}$, plus costul multiplicării finale a celor două matrici.

Observația cheie este că parantezarea sublanțului format din $A_1 A_2 \dots A_k$ în cadrul parantezării optime a lui $A_1 A_2 \dots A_n$ trebuie să fie la rândul ei optimă. De ce? Dacă ar fi un al mod mai bun de a paranteza $A_1 A_2 \dots A_k$, aceasta în cadrul lanțului $A_1 A_2 \dots A_n$, va duce la o altă parantezare mai bună a în locul celei presupuse a fi optime. De aici apare contradicția ce confirmă ipoteza. Asemenea vom spune că $A_{k+1} A_{k+2} \dots A_n$ este parantezarea optimă în cadrul lanțului mare $A_1 A_2 \dots A_n$.

O solutie recursivă

Al doilea pas din paradigma programării dinamice este să definim valoarea unei soluții optime recursiv în raport cu soluțiile optime ale subproblemelor. Pentru problema multiplicării în lanț alegem ca subprobleme determinarea costului minim în a paranteza $A_iA_{i+1} \dots A_j$ pentru $1 \leq i \leq j \leq n$. Fie $m[i,j]$ numărul minim de multiplicări necesare pentru a calcula matricea $A_{i..j}$. Cu aceste considerente costul cel mai mic pentru a calcula $A_{1..n}$ trebuie să fie $m[1,n]$.

Putem defini $m[i,j]$recursiv după cum urmează. Dacă $i = j$, lanțul constă dintr-o singură matrice $A_{i..i} = A_i$ astfel că nici o multiplicare între elementele acestei matrice nu este necesară. De aceea $m[i,j] = 0\ pentru\ \ i = 1,2, \dots, n$. Pentru a calcula $m[i,j]$ când $i < j$, vom profita de structura stabilită la pasul 1. Adică presupunem că parantezarea optimă împarte produsul $A_iA_{i+1} \dots A_j$ între A_k și A_{k+1} unde $i \leq k < j$. Astfel, $m[i,j]$ este egal cu minimul atunci când calculăm subprodusele $A_{i..k}$ și $A_{k+1..j}$, plus costul multiplicării matricelor finale. Din moment ce calculul produsului $A_{i..k}A_{k+1..j}$ ia $p_{i-1}p_kp_j$ multiplicări, vom obține:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

Această ecuație pornește de la premisa că știm valoarea lui k, ceea ce nu este adevărat. Totuși, există $j - i$ valori posibile pe care k le poate lua și anume $k = i, i+1, .., j-1$. Din moment ce parantezarea optimă este una corespunzătoare uneia din aceste valori ale lui k, le putem verifica pe toate și descoperi pe cea mai bună. De acea definiția recursivă a parantezării optime a produsului $A_iA_{i+1} \dots A_j$ devine:

$$m[i,j] = \begin{cases} 0 & dacă\ i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & dacă\ i < j \end{cases} \quad (1)$$

Valorile $m[i,j]$ dau costul minim pentru o subproblemă *(i,j)*. Pentru a ține evidența construirii soluției optime, fie $s[i,j]$ ce reprezintă valoarea lui k pentru care putem împărți produsul $A_iA_{i+1} \dots A_j$ astfel ca parantezarea să fie optimă. Cu alte cuvinte $s[i,j]$ este acel k pentru care $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

Calculul costului optim

La acest moment, se poate scrie algoritmul recursiv bazat pe recurența (1), algoritm ce calculează costul minim $m[1,n]$ necesar multiplicării lanțului de matrice $A_1, A_2, \dots, A_n$. Se poate intui timpul exponențial al unui astfel de algoritm.

Cea mai importantă observație că avem puține subprobleme: una pentru fiecare alegere a lui i și j ce să fie în intervalul $[1,n]$. În total obținem $\theta(n^2)$ pentru aceste alegeri. Un algoritm recursiv ar putea întâlni fiecare subproblemă de mai multe ori. Această proprietate de a suprapune probleme ce pot apare redundant este marca programării dinamice.

Așa că, în loc să calculăm soluția recurenței (1), vom efectua un al treilea pas și anume vom considera problema de sus în jos. Următorul pseudocod presupune că matricea A_i are dimensiunile $p_{i-1} \times p_i$ pentru $i = 1,2, \dots, n$. Secvența de intrare este $\langle p_0, p_1, \dots, p_n \rangle$ unde $length[p] = n + 1$. Procedura folosește o tabelă auxiliară $m[1..n, 1..n]$ pentru a stoca, costurile $m[i,j]$ și tabela auxiliară $s[1..n, 1..n]$ ce ține indecșii lui k ce au fost obținuți în calculul lui $m[i,j]$.

$MATRIX - CHAIN - ORDER(p)$

1. $n \leftarrow length[p] - 1$
2. **for** $i \leftarrow 1$ **to** n **do**
3. $\quad m[i,i] \leftarrow 0$
4. **for** $l \leftarrow 2$ **to** n **do**
5. $\quad$ **for** i $\leftarrow 1$ **to** $n - l + 1$ **do**
6. $\quad\quad j \leftarrow i + l - 1$
7. $\quad\quad m[i,j] \leftarrow \infty$
8. $\quad\quad$ **for** k $\leftarrow i$ **to** $j - 1$ **do**
9. $\quad\quad\quad q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
10. $\quad\quad\quad$ **if** $q < m[i,j]$ **then**
11. $\quad\quad\quad\quad m[i,j] \leftarrow q$
12. $\quad\quad\quad\quad s[i,j] \leftarrow k$
13. **return** m **and** s

Algoritmul va umple tabela m în așa fel încât să rezolve problema parantezării optime. Ecuația (1) arată că, costul $m[i,j]$ pentru a calcula produsul a $j - i + 1$ matrice, depinde doar de costul optim al

calculului a unui produs de mai puțin de $j - i + 1$ matrice. De aceea pentru $k = i, i + 1, \dots, j - 1$, matricea $A_{i..k}$ este produsul dintre $k - i + 1 < j - i + 1$ matrice, și de asemenea, matricea $A_{k+1..j}$ este produsul a $j - k < j - i + 1$ matrice.

Algoritmul va calcula prima dată (liniile 2-3) $m[i, i] = 0$ pentru $i = 1,2, \dots, n$ adică, costul minim de a calcula produsul dintre A_i și același A_i.

Apoi, pe liniile 4-12 algoritmul folosește formula (1) pentru a calcula $m[i, i + 1]$ pentru $i = 1,2, \dots, n - 1$, adică tocmai costul minim pentru lanțuri de lungime 2. Aceasta se întâmplă la primul pas din for-ul cu variabila l. A doua oară când se trece prin *for*, se vor calcula minimele pentru lanțuri de trei elemente: $m[i, i + 2]$ pentru $i = 1,2, \dots, n - 2$, și tot așa.

La fiecare pas, costul $m[i, j]$ (liniile 9-12) depinde doar de intrările $m[i, k]$ și $m[k + 1, j]$ ce au fost deja calculate la niște pași anteriori.

Figura 2. Ilustrează procedura pentru un lanț de $n = 6$ matrice. Din moment ce am definit $m[i, j]$ doar pentru $i \leq j$, se va folosi doar porțiunea de deasupra diagonalei pentru a calcula costurile minime.

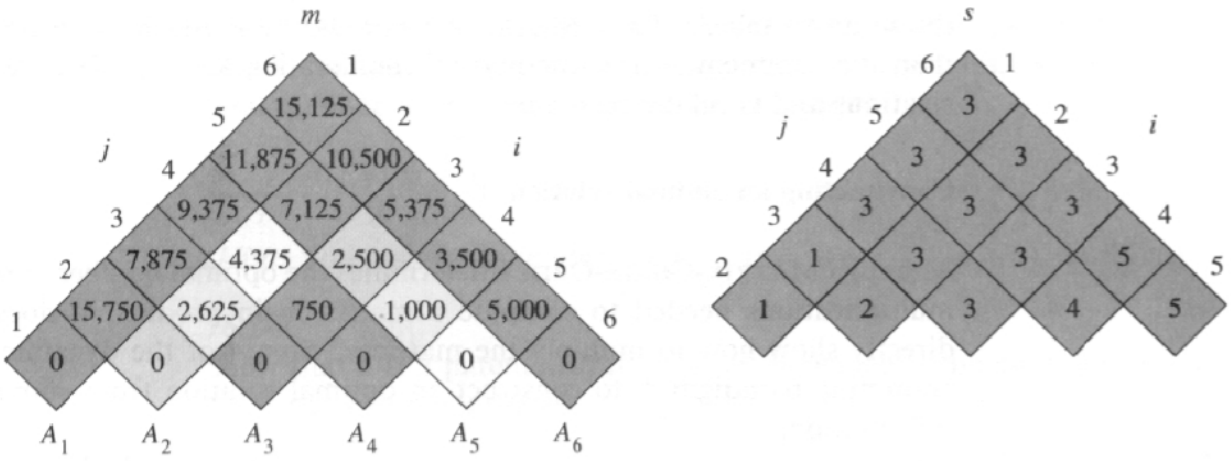


Figura 2. Maticele m și s calculate pentru n=6, rotite cu 45°

Problema inițială este aceasta: Se dau matricile de dimensiunile:

matricea	Dimensiunea
$\boldsymbol{A_1}$	30×35
$\boldsymbol{A_2}$	35×15
$\boldsymbol{A_3}$	15×5
$\boldsymbol{A_4}$	5×10
$\boldsymbol{A_5}$	10×20
$\boldsymbol{A_6}$	20×25

Matricele sunt rotite astfel ca diagonala principală să fie pe orizontală. Cum doar elementele de peste diagonala principală sunt folosite în matricea m rezultă aceste două triunghiuri. Numărul minim de multiplicări este $m[1,6] = 15125$. Pentru elementul $m[2,5]$ calculele ce se fac pentru aflarea minimului sunt:

$$m[2,5] = min\begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 \\ m[2,3] + m[4,5] + p_1p_3p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 \\ m[2,4] + m[5,5] + p_1p_4p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$

Așa că $m[2,5] = 7125$ fapt marcat și în matricea din imagine.

În imaginea 2, fiecare rând orizontal conține elementele din lanțul de matrice de aceeași lungime. Funcția $MATRIX - CHAIN - ORDER$ calculează rândurile de la baza triunghiului la vârf, adică de sus în jos. Orice $m[i,j]$ este calculat folosind produsele $p_{i-1}p_kp_j$ pentru $k = i, i+1, \dots, j-1$ și toate elementele din sud-estul și sud-vestul lui $m[i,j]$. O simplă analiză a acestui algoritm va da timpul total de $\theta(n^3)$ din cauza celor trei for-uri imbricate.

Cea mai lungă subsecventă

Următoarea problemă, denumită și găsirea celei mai lungi subsecvențe comune poate fi formulată după cum urmează. O subsecvență dintr-o secvență dată este o bucată din acea secvență în care unele elemente au fost eventual îndepărtate. Dat fiind o secvență $X = \langle x_1, x_2, \dots, x_m \rangle$, o altă secvență $Z = \langle z_1, z_2, \dots, z_k \rangle$ este o subsecvență a lui X dacă există o secvență $\langle i_1, i_2, \dots, i_k \rangle$ de indici ai lui X astfel ca pentru toate $j = 1,2, \dots, k$ să avem $x_{i_j} = z_j$. De exemplu, $Z = \langle B, C, D, B \rangle$ este subsecvența lui $X = \langle A, B, C, B, D, A, B \rangle$ corespunzător indicilor $\langle 2,3,5,7 \rangle$.

Dat fiind două secvențe X și Y, spunem că o secvență Z este o subsecvență comună a lui X și Y dacă Z este o subsecvență atât a lui X cât și a lui Y. De exemplu, dacă $X = \langle A, B, C, B, D, A, B \rangle$ și $Y = \langle B, D, C, A, B, A \rangle$, secvența $\langle B, C, A \rangle$ are trei elemente și nu este cea mai lungă subsecvență (LCS) a lui X și Y, deoarece există o altă subsecvență comună cu 4 elemente și anume $\langle B, C, B, A \rangle$. Deoarece nu există nici o altă secvență comună cu 5 elemente spunem că $\langle B, C, B, A \rangle$ este si cea mai lungă.

Putem formula problema celei mai *lungi subsecvențe comune* astfel: se dau secvențele $X = \langle x_1, x_2, \dots, x_m \rangle$ și $Y = \langle y_1, y_2, \dots, y_n \rangle$ și se dorește aflarea celei mai lungi subsecvențe a lui X și Y.

Caracterizarea celei mai lungi subsecvente

O abordare brute-force a acestei probleme este de a enumera toate subsecvențele ale lui X și de a verifica fiecare subsecvență pentru a vedea dacă este de asemenea și în Y. Fiecare subsecvență corespunde unui subset de indici $\{1,2, \dots, m\}$ ai lui X. Există 2^m subsecvențe a lui X, deci această abordare este total ineficientă.

Rezolvarea problemei derivă de la o proprietate. După cum vom vedea, clasa naturală de subprobleme corespunde unor perechi de *prefixe*, ale celor două secvențe de intrare. Mai exact, dat fiind o secvență $X = \langle x_{1,} x_2, \dots, x_m \rangle$ definim al i-lea prefix pentru $i = 0,1, \dots, m$ ca fiind $X_i = \langle x_1, x_2, \dots, x_i \rangle$. De exemplu dacă $X = \langle A, B, C, B, D, A, B \rangle$ atunci $X_4 = \langle A, B, C, B \rangle$ și X_0 este secvența vidă.

Teorema 1.

Fie $X = \langle x_1, x_2, \dots, x_m \rangle$ și $Y = \langle y, y_2, \dots, y_n \rangle$ secvențele și fie $Z = \langle z_1, z_{2,} \dots, z_k \rangle$ orice LCS a lui X și Y.

1. Dacă $x_m = y_n$ atunci $z_k = x_m = y_n$ și z_{k-1} este o LCS a lui X_{m-1} și Y_{n-1}.
2. Dacă $x_m \neq y_n$ atunci $z_k \neq x_m$ implică faptul că Z este o LCS a lui X_{m-1} și Y_n
3. Dacă $x_m \neq y_n$ atunci $z_k \neq y_n$ implică faptul că Z este o LCS a lui X_m și Y_{n-1}

Demonstrația

1) Dacă $z_k \neq x_m$ atunci putem atașa $x_m = y_n$ lui Z pentru a obține LCS a lui X, Y de lungime $k + 1$. Contrazicem astfel presupunerea că Z este cea mai lungă subsecvență a lui X și Y. De aceea trebuie să avem $z_k = x_m = y_n$. Acum prefixul Z_{k-1} este de lungime $(k - 1)$ și este LCS pentru X_{m-1}, Y_{n-1}.
2) Dacă $z_k \neq x_m$ atunci Z este subsecvența comună lui X_{m-1} și Y. Dacă ar fi o subsecvență comună W a lui X_{m-1} și Y cu o lungime mai mare decât k, atunci W ar fi de asemenea o subsecvență comună pentru X_m și Y, cotrazicând presupunerea $z_k \neq x_m$.
3) Demonstrația este simetrică lui 2).

O solutie recursivă

Teorema 1, implică faptul că fie există una sau două subprobleme de examinat când căutăm LCS a lui $X = \langle x_1, x_2, \dots, x_m \rangle$ și $Y = \langle y_1, y_2, \dots, y_n \rangle$. Dacă $x_m = y_n$ atunci trebuie să găsim LCS al lui X_{m-1} și Y_{n-1}. Dacă adăugăm $x_m = y_n$ la această soluție de LCS găsită, atunci avem chiar LCS al lui X și Y. Al doilea caz este când $x_m \neq y_n$ și atunci trebuie să găsim fie LCS a lui X_{m-1} și Y_n , fie LCS a lui X_m și Y_{n-1}.

Ca și problema înlănțuirii înmulțirii matricelor, soluția recursivă la problema LCS implică stabilirea unei recurențe de cost optim. Să definim $c[i,j]$ lungimea unei LCS între secvența X_i și Y_i. Dacă ori $i = 0$ ori $j = 0$, una din secvențe are lungimea 0 deci LCS este de lungime 0. Structura optimă a problemei LCS este dată de formula:

$$c[i,j] = \begin{cases} 0 & dacă\ i = 0\ și\ j = 0 \\ c[i-1, j-1] + 1 & dacă\ i, j > 0\ \ și\ \ x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & dacă\ i, j > 0\ și\ x_i \neq y_j \end{cases} \quad (2)$$

Calculul lungimii unui LCS

Pe baza ecuației (2) putem ușor scrie un algoritm recursiv cu un ordin de timp exponențial pentru a calcula lungimea LCS a celor două secvențe. deoarece sunt doar $\theta(mn)$ subprobleme distincte, putem rezolva prin aplicarea programării dinamică.

Procedura LCS_LENGTH va lua două secvențe $X = \langle x_1, x_2, \dots, x_m \rangle$ și $Y = \langle y_1, y_2, \dots, y_n \rangle$ ca intrare. Va stoca în matricea $c[0..m, 0..n]$ valorile $c[i,j]$ corespunzătoare secvențelor. Primul rând din c este umplut de la stânga la dreapta, apoi al doilea rând este umplut de la stânga la dreapta și așa mai departe. În matricea $b[1..m, 1..n]$ se mențin indicații pentru a reconstrui ușor soluția finală. Practic $b[i,j]$ indică subproblema optimă corespunzătoare calculului $c[i,j]$. Procedura va returna ambele matrice: $c[m,n]$ conține lungimea LCS a secvențelor X, Y.

Procedura are un ordin de timp $O(mn)$ deoarece calculul fiecărui element din matrice este în $O(1)$.

$LCS\_LENGTH(X, Y)$

1) $m \leftarrow length[X]$
2) $n \leftarrow length[Y]$
3) $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ m\ \boldsymbol{do}$
4) $\quad c[i, 0] \leftarrow 0$
5) $\boldsymbol{for}\ j \leftarrow 0\ \boldsymbol{to}\ n\ \boldsymbol{do}$
6) $\quad c[0, j] \leftarrow 0$
7) $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ m\ \boldsymbol{do}$
8) $\quad \boldsymbol{for}\ j \leftarrow 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
9) $\quad\quad \boldsymbol{if}\ x_i = y_j\ \boldsymbol{then}$
10) $\quad\quad\quad c[i, j] \leftarrow c[i-1, j-1] + 1$
11) $\quad\quad\quad b[i, j] \leftarrow$ " ↖ "
12) $\quad\quad \boldsymbol{else}$
13) $\quad\quad\quad \boldsymbol{if}\ c[i-1, j] \geq c[i, j-1]\ \boldsymbol{then}$
14) $\quad\quad\quad\quad c[i, j] \leftarrow c[i-1, j]$
15) $\quad\quad\quad\quad b[i, j] \leftarrow$ " ↑ "
16) $\quad\quad\quad \boldsymbol{else}$
17) $\quad\quad\quad\quad c[i, j] \leftarrow c[i, j-1]$
18) $\quad\quad\quad\quad b[i, j] \leftarrow$ " ← "
19) $\boldsymbol{return}$ *c şi b*

		j	0	1	2	3	4	5	6
i			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figura 3. Matricile c şi b calculate pentru $X = \langle A, B, C, B, D, A, B \rangle$ şi $Y = \langle B, D, C, A, B, A \rangle$.

Cum construim o LCS

Tabelul b returnat de procedura LCS_LENGTH poate fi folosit ușor pentru a construi o secvență optimă. Începem cu $b[m,n]$ și urmărim *drumul* din matrice indicat de săgeți. Atunci când întâlnim un ↖ în locația $b[i,j]$ înseamnă că $x_i = y_j$ adică am găsit un element al lui LCS. Elementele sunt descoperite în ordinea inversă a lor. În figura de mai sus, intrarea 4 din $c[7,6]$ este lungimea unei secvențe $LCS\langle B,C,B,A\rangle$ a lui X,Y. Pentru $i,j > 0$ $c[i,j]$depinde doar de egalitatea dintre x_i și y_j și de valorile $c[i-1,j]$ și $c[i,j-1]$ și $c[i-1,j-1]$ care sunt calculate înainte de $c[i,j]$.

Soluția recursivă de mai jos afișează LCS a lui X și Y în ordinea corectă. Primul apel va fi $PRINT\_LCS(b,X,length[X],length[Y])$.

$PRINT\_LCS(b,X,i,j)$

1) $\boldsymbol{if}\ i = 0\ sau\ j = 0\ \boldsymbol{then\ return}$
2) $\boldsymbol{if}\ b[i,j] =$ " ↖ " $\boldsymbol{then}$
3) $PRINT\_LCS(b,X,i-1,j-1)$
4) $print\ x_i$
5) $\boldsymbol{else\ if}\ b[i,j] =$ " ↑ "
6) $\boldsymbol{then}\ PRINT\_LCS(b,X,i-1,j)$
7) $\boldsymbol{else}\ PRINT\_LCS(b,X,i,j-1)$

Pentru b din figura 3 această procedură afișează $BCBA$. Procedura are un ordin de timp $O(m+n)$ deoarece la fiecare pas al recursivității, fie i, fie j este decrementat.