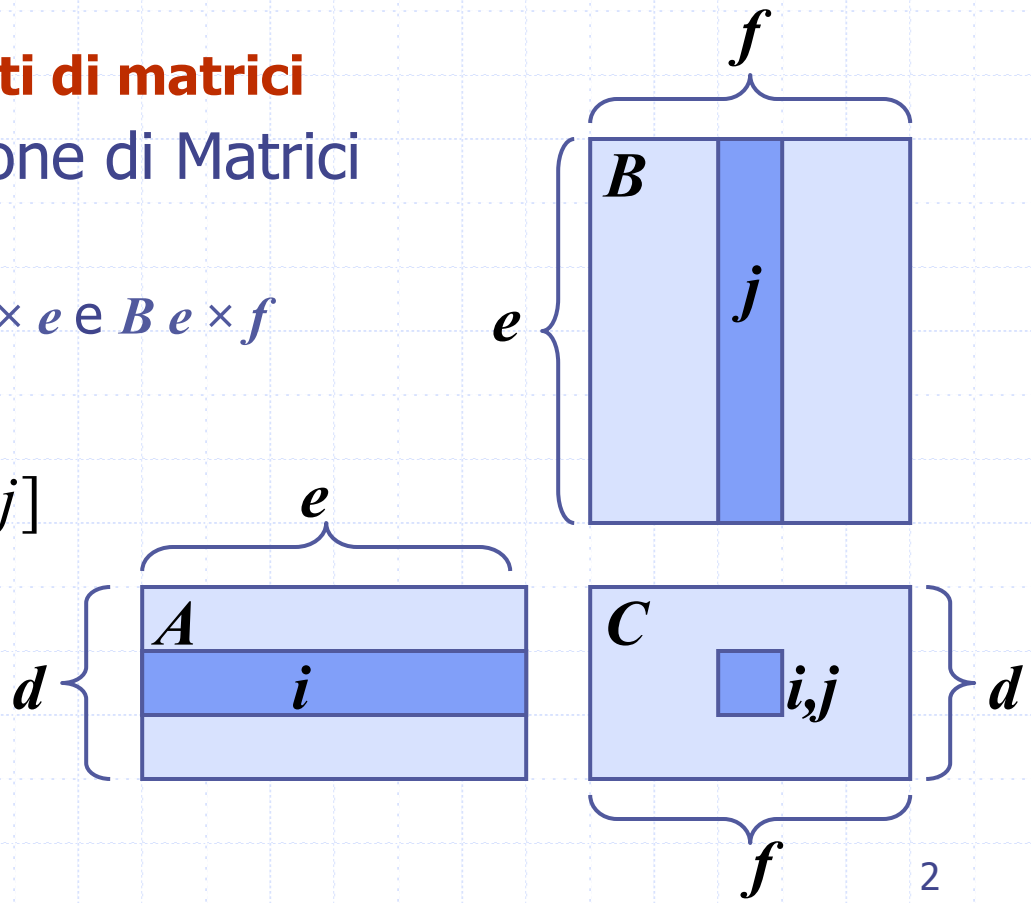
Programmazione dinamica

Sequenza di prodotti di matrici

- La Programmazione Dinamica è una tecnica generale per il progetto di algoritmi . Esempio
  - **Sequenza di prodoti di matrici**
- Ripasso: Moltiplicazione di Matrici
  - $C = A*B$
  - A ha dimensione $d \times e$ e B $e \times f$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

  - *Tempo* $O(def)$



Sequenza di prodotti di matrici

- **Sequenza di prodotti di Matrici:**
  - Calcola $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i è $d_i \times d_{i+1}$
  - Problema: Come parentesizzare? Cioè in quale ordine eseguire i prodotti?
- Esempio
  - B è una matrice 3×100
  - C è una matrice 100×5
  - D è una matrice 5×5
  - (B*C)*D richiede 1500 + 75 = 1575 operazioni
  - B*(C*D) richiede 1500 + 2500 = 4000 operazioni

Un approccio enumerativo

- **Algoritmo Sequenza di prodotti:**
  - Prova tutte le possibili sequenze di parentesi per $A=A_0*A_1*...*A_{n-1}$
  - Per ciascuna calcola il numero di operazioni
  - Scegli la sequenza migliore
- Tempo esecuzione:
  - Il numero di possibili parentesizzazioni di n matrici è uguale al numero di alberi binari con n nodi
  - Questo numero è **esponenziale**!
  - E' noto come il Numero Catalano, ed è quasi 4^n.
  - Algoritmo proposto è molto inefficiente! NON VA BENE

Un altro approccio Greedy

- Idea #2: ripetutamente ad ogni passo scegli di eseguire il prodotto che usa il numero minimo di operazioni
- Contro esempio: esegui ABCD dove
  - A è 101 × 11
  - B è 11 × 9
  - C è 9 × 100
  - D è 100 × 99
  - Greedy idea da A*((B*C)*D)), che richiede 109989+9900+108900=228789 operazioni
  - (A*B)*(C*D) takes 9999+89991+89100=189090 operaz.
- L'approccio greedy non fornisce il valore ottimo
- Quindi: Enumerativo: NO (troppo costoso); Greedy:NO (non da soluionzione ottima)

Programmazione dinamica

- Approccio ricorsivo
- Definisci **sottoproblemi**:
  - Trova la migliore parentesizzazione di $A_i*A_{i+1}*...*A_j$.
  - Sia $N_{i,j}$ il numero di operazioni fatto per questo sottoproblema
  - La soluzione ottima per il problema dato è $N_{0,n-1}$.
- **Principio di ottimalità dei Sottoproblemi**: La soluzione ottima può essere definita in funzione delle soluzioni ottime dei sottoproblemi
  - CI deve essere un prodotto finale per la soluzione ottima
  - Supponi che questo sia all'indice i: $(A_0*...*A_i)*(A_{i+1}*...*A_{n-1})$.
  - Allora la soluzione ottima $N_{0,n-1}$ è data dalla somma dei valori della soluzione ottima dei due sottoproblemi $N_{0,i}$ e $N_{i+1,n-1}$ più il tempo dell'ultima moltiplicazione
  - Se l'ottimo globale non è composto in questo modo allora potremmo ottenere una soluzione migliore

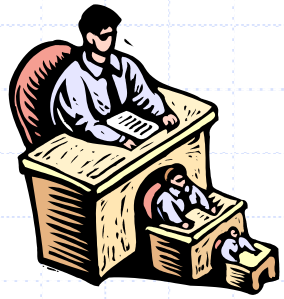
La funzione che caratterizza il costo

- L'ottimo globale deve essere definitio in funzione dei sottoproblemi che variano a seconda dell'ultimo prodotto da eseguire
- Consideriamo tutte le possibili posizioni della moltiplicazione finale:
  - Ricorda che A_i è una matrice $d_i \times d_{i+1}$
  - Quindi una funzione che definisce $N_{i,j}$ è la seguente

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- Nota il terzo termine $d_i\ d_{k+1}\ d_{j+1}$ indica che i sottoproblemi non sono indipendenti

Un algoritmo di programmazione dinamica



- Dato che i sottoproblemi non sono indipendenti NON possiamo usare la ricorsione
- Invece costruiamo le soluzioni ottime dei sottoproblemi "bottom-up."
- $N_{i,i}$= 0 per ogni I
- Quindi eseguiamo per sottoproblemi di lunghezze 2,3,…
- Il tempo di esecuzione è $O(n^3)$

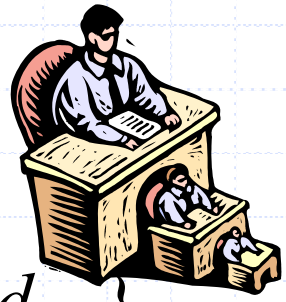
Algorithm *matrixChain*(*S*):
  Input: sequence *S* of *n* matrices to be multiplied
  Output: number of operations in an optimal paranethization of *S*
  for $i \leftarrow 1$ **to** *n-1* **do**
    $N_{i,i} \leftarrow 0$
  for $b \leftarrow 1$ **to** *n-1* **do**
    for $i \leftarrow 0$ **to** *n-b-1* **do**
      $j \leftarrow i+b$
      $N_{i,j} \leftarrow +\infty$
      for $k \leftarrow i$ **to** *j-1* **do**
        $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

Implementazione Java

```
1   public static int[ ][ ] matrixChain(int[ ] d) {
2     int n = d.length − 1;                          // number of matrices
3     int[ ][ ] N = new int[n][n];                   // n-by-n matrix; initially zeros
4     for (int b=1; b < n; b++)                      // number of products in subchain
5       for (int i=0; i < n − b; i++) {              // start of subchain
6         int j = i + b;                             // end of subchain
7         N[i][j] = Integer.MAX_VALUE;               // used as 'infinity'
8         for (int k=i; k < j; k++)
9           N[i][j] = Math.min(N[i][j], N[i][k] + N[k+1][j] + d[i]*d[k+1]*d[j+1]);
10      }
11    return N;
12  }
```

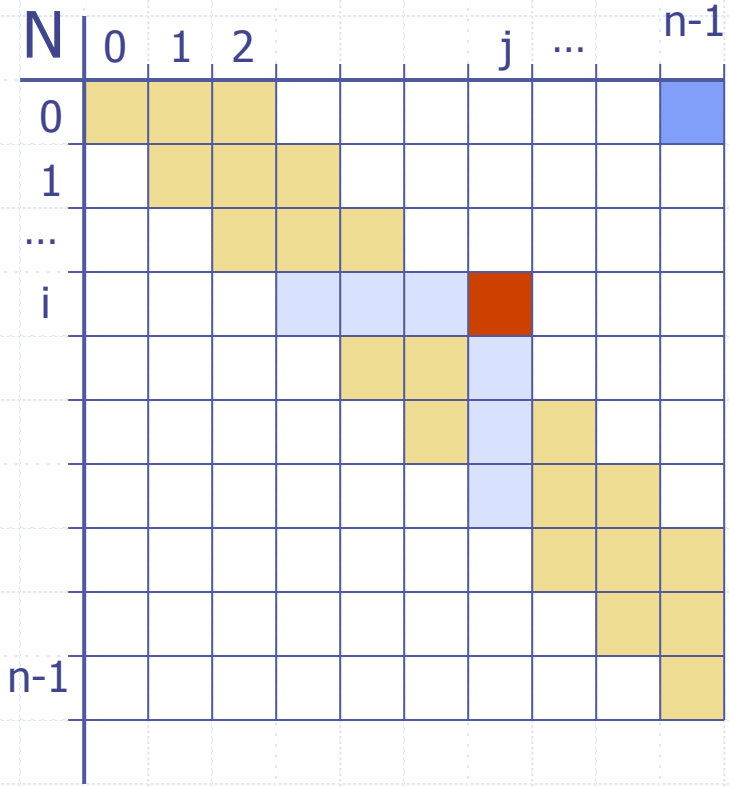
$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

Una visualizzazione dell'algoritmo



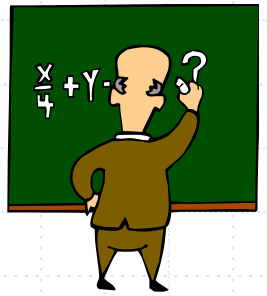
$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- L'approccio costruisce la matrice N per diagonali
- $N_{i,j}$ prende il valore delle entrate precedenti nella riga I e colonna j
- Riempire la tabella in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Per ottenere la sequenza effettiva di parentesi richiede di memorizzare "k" per ciascuna dei N possinbili valori



risposta

La tecnica della Programmazione Dinamica



- Si applica ad un problema che a prima vista richiede un tempo di risoluzione molto alto (anche esponenziale) e richiede che:
  - **Definizione di semplici sottoproblemi:** I sottoproblemi possono essere definiti in funzione di poche variabili come ad esempio j, k, l, m.
  - **Ottimalità dei sottoproblemi:** l'ottimo globale può essere definito in funzione delle soluzioni ottime dei sottoproblemi
  - **Correlazione fra sottoproblemi:** I sottoproblemi non sono indipendenti ma sono correlati (questo motiva approccio bottom-up).

Sottosequenze

- Una sottosequenza di una stringa di caratteri $x_0x_1x_2...x_{n-1}$ è una stringa del tipo $x_{i_1}x_{i_2}...x_{i_k}$, dove $i_j < i_{j+1}$.
- Nota: NON è una sottostringa!
- Esempio Data la stringa: ABCDEFGHIJK
  - sottosequenza: ACEGJIK
  - sottosequenza: DFGHK
  - Non è una sottosequenza: DAGH

Problema della più lunga sottosequenza (LCS)

- Date due stringhe X e Y, la più lunga sottosequenza (LCS) richiede di trovare la più lunga sottosequenza comune a X e Y
- Ha applicazioni nel test di similarità di sequenze DNA (alfabeto è {A,C,G,T})
- Esempio: ABCDEFG e XZACKDFWGH hanno ACDFG come una sottosequenza comune

Un approccio costoso

- Soluzione algoritmo forza bruta
  - Enumera tutte le sottosequenze di X
  - Verifica quali sono anche sottosequenze di Y
  - Scegli la più lunga
- Analisi:
  - Se X è di lunghezza n, allora ha 2^n sottosequenze
  - Algoritmo esponenziale!

Un algoritmo di Programmazione Dinamica

- Definisci L[i,j] come la lunghezza della più lunga sottosequenza comune fra X[0..i] e Y[0..j].
- Assumi che -1 sia un indice e che L[-1,k] = 0 e L[k,-1]=0, per indicare che la parte nulla di X o Y non hanno corrispondenza comune
- Possiamo definire L[i,j] come segue:
  1. Se xi=yj, allora L[i,j] = L[i-1,j-1] + 1 (aggiungiamo il match)
  2. Se xi≠yj, allora L[i,j] = max{L[i-1,j], L[i,j-1]} (no match in questo caso)

Caso 1:

Caso 2:

Algoritmo LCS

Algorithm LCS(X,Y):

Input: Stringhe X eY con n e m elementi,

Output: Per i = 0,…,n-1, j = 0,…,m-1, la lunghezza L[i, j] della più lunga stringa che è comune sia alla stringa X[0..i] = $x_0x_1x_2…x_i$ che alla stringa Y [0.. j] = $y_0y_1y_2…y_j$

for i =1 to n-1 **do**

L[i,-1] = 0

for j =0 to m-1 **do**

L[-1,j] = 0

for i =0 to n-1 **do**

for j =0 to m-1 **do**

if $x_i = y_j$ **then**

L[i, j] = L[i-1, j-1] + 1

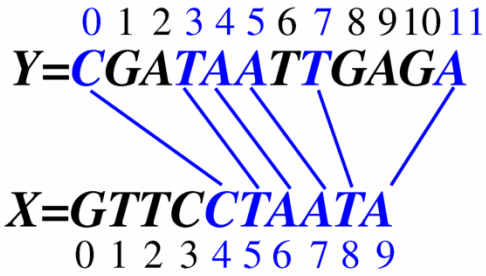
else

L[i, j] = max{L[i-1, j] , L[i, j-1]}

return array L

Visualizzazione dell'algoritmo

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**



Analisi dell'algoritmo LCS

- Abbiamo due cicli annidati
  - L'esterno eseguito *n* volte
  - L'interno eseguito *m* volte
  - All'interno del ciclo si esegue un lavoro costante
  - Quindi il tempo totale di esecuzione è O(*nm*)
- La risposta effettiva è contenuta L[n,m] (e la sottosequenza può essere recuperata da L).

Implementazione Java

```
1   /** Returns table such that L[j][k] is length of LCS for X[0..j−1] and Y[0..k−1]. */
2   public static int[ ][ ] LCS(char[ ] X, char[ ] Y) {
3     int n = X.length;
4     int m = Y.length;
5     int[ ][ ] L = new int[n+1][m+1];
6     for (int j=0; j < n; j++)
7       for (int k=0; k < m; k++)
8         if (X[j] == Y[k])              // align this match
9           L[j+1][k+1] = L[j][k] + 1;
10        else                           // choose to ignore one character
11          L[j+1][k+1] = Math.max(L[j][k+1], L[j+1][k]);
12    return L;
13  }
```

Implementazione Java Output della Soluzione

```
1  /** Returns the longest common substring of X and Y, given LCS table L. */
2  public static char[ ] reconstructLCS(char[ ] X, char[ ] Y, int[ ][ ] L) {
3    StringBuilder solution = new StringBuilder( );
4    int j = X.length;
5    int k = Y.length;
6    while (L[j][k] > 0)                          // common characters remain
7      if (X[j−1] == Y[k−1]) {
8        solution.append(X[j−1]);
9        j−−;
10       k−−;
11     } else if (L[j−1][k] >= L[j][k−1])
12       j−−;
13     else
14       k−−;
15   // return left-to-right version, as char array
16   return solution.reverse( ).toString( ).toCharArray( );
17 }
```