1.LA PROGRAMMAZIONE DINAMICA

1.1.Descrizione della Tecnica

La *Programmazione Dinamica* è una tecnica di realizzazione di algoritmi che risolvono un problema utilizzando le soluzioni di sottoproblemi. Il termine "programmazione" fa riferimento all'uso di tabelle per appuntare le soluzioni dei sottoproblemi durante l'applicazione manuale della tecnica e non alla scrittura di codice.

Come abbiamo visto nell'Unità 5, anche la tecnica Divide et Impera partiziona un problema in sottoproblemi e risolve il problema attraverso un'opportuna combinazione delle sotto-soluzioni. La differenza tra le due tecniche è che Divide et Impera intende preliminarmente individuare solo quei sottoproblemi che sono rilevanti per la soluzione del problema originario (metodo *top-down*, dall'alto verso il basso) mentre la Programmazione Dinamica parte direttamente da tutti i sottoproblemi più piccoli per poi arrivare alla soluzione del problema originario (metodo *bottom-up*, dal basso verso l'alto). Il diverso metodo adoperato comporta un diverso schema di algoritmo: come abbiamo visto nell'Unità 4, l'algoritmo tipico Divide et Impera è ricorsivo mentre, come vedremo in quest'unità, l'algoritmo tipico della Programmazione Dinamica è iterativo. Ma la differenza sostanziale non è tanto nell'uso della ricorsione o meno — sappiamo che la ricorsione può sempre essere sostituita dalla iterazione; la differenza è il diverso numero di sottoproblemi risolti dalle due tecniche.

Nel caso di un problema in cui solo un numero limitato di sottoproblemi è rilevante per determinare la soluzione finale, la tecnica Divide et Impera risulta più conveniente in quanto l'extra-lavoro per individuare i sottoproblemi è ripagato dal minor numero di sottoproblemi da risolvere. D'altra parte, se tutti o quasi tutti i sottoproblemi devono essere comunque risolti e, addirittura, accade che la soluzione di uno stesso sottoproblema debba essere usata più volte, allora la Programmazione Dinamica risulta essere la tecnica più conveniente poiché essa parte direttamente dalla soluzione di tutti i problemi di dimensione atomica per ricomporre via via le soluzioni di tutti i sottoproblemi di dimensioni maggiori, risolvendo ogni sottoproblema solo una volta e conservando la sua soluzione in una tabella. Come esempio riproponiamo il calcolo del numero di Fibonacci di n. L'algoritmo di tipo Divide et Impera ha la seguente struttura:

```
Card fibonacciRic( Card n )
{     if ( n <= 1 )
            return n;
      else
            return fibonacci(n-1)+fibonacci(n-2);
}
```

e l'algoritmo perde un bel pò di tempo per scoprire ciò che è ben noto: tutti i sottoproblemi sono rilevanti, cioè per calcolare il numero di Fibonacci di n è necessario aver calcolato i numeri di Fibonacci da 0 a n-1. Ma quel che è peggio, poiché uno stesso numero di Fibonacci è utilizzato più volte, l'algoritmo è costretto

a calcolarlo un numero di volte che è esponenziale in *n*. L'algoritmo della Programmazione Dinamica invece adotta il seguente schema:

```
Card fibonacciIt( Card n )
{     if ( n <= 1 )
            return n;
      else
      {     Card f1 = 1; Card f2 = 0; Card f;
            for ( Card i = 2; i <= n; i++ )
            {     f = f1 + f2;
                  f2 = f1; f1 = f;
            }
            return f;
      }
}
```

e, poiché non è costretto a ricalcolare lo stesso numero di Fibonacci più di una volta, ha una complessità lineare in *n*.

QUANDO

Sulla base di quanto discusso, possiamo affermare che la tecnica della Programmazione Dinamica va adoperata allorquando la soluzione di un problema è ottenibile dalla combinazione di tutti o quasi tutti i suoi sottoproblemi e/o quando la soluzione di uno stesso sottoproblema debba essere utilizzata più volte. Cioè la Programmazione Dinamica va utilizzata per quei problemi per i quali la risoluzione attraverso i sottoproblemi non è semplice dal punto di vista computazionale. Questo spiega il fatto che gli algoritmi di Programmazione Dinamica hanno tipicamente complessità esponenziale; nonostante l'elevata complessità, essi rimangono validi in quanto il ricorso ad algoritmi di ricerca più o meno esaustivi comporterebbe una complessità ancora più elevata in termini di costanti. Come vedremo tra poco, esistono comunque alcuni algoritmi di Programmazione Dinamica che risolvono problemi significativi con tempi di esecuzione polinomiale.

Per concludere la descrizione generale della tecnica di Programmazione Dinamica, dobbiamo stabilire sotto quali condizioni essa risolve correttamente un problema. Innanzitutto precisiamo che la Programmazione Dinamica, così come la tecnica golosa, è sostanzialmente usata per risolvere *problemi di ottimizzazione* senza esplorare tutte le soluzioni possibili. Le condizioni di correttezza devono garantire che aver trascurato alcune soluzioni non porti a mancare di raggiungere l'ottimo. Queste condizioni sono meno stringenti del caso della tecnica Golosa in quanto questa volta viene fatta una valutazione esaustiva di tutti i possibili sottoproblemi. Tuttavia, non c'è garanzia che sia possibile costruire l'ottimo sulla base degli ottimi dei sottoproblemi. Affinché ciò avvenga deve valere il *principio di ottimalità*, cioè la soluzione ottima del problema dipende dalla soluzione ottima dei vari suoi sottoproblemi e non da eventuali soluzioni subottime. Ad esempio, dati i seguenti 6 oggetti caratterizzati da un tipo e valore:

codice oggetto	*o1*	*o2*	*o3*	*o4*	*o5*	*o6*
tipo oggetto	*A*	*B*	*C*	*C*	*C*	*C*
valore oggetto	*20*	*19*	*15*	*20*	*10*	*5*

dobbiamo scegliere 3 oggetti con somma dei valori massima ma con la particolarità che se i due oggetti scelti sono dello stesso tipo, i loro valori raddoppiano. L'algoritmo che divide gli oggetti in due gruppi di 3, determina la soluzione ottima per i due sottoproblemi e poi utilizza le due sottosoluzioni per calcolare la soluzione del problema originario non rispetta il principio di ottimalità. Infatti nel primo gruppo vengono scelti gli oggetti *o1* e *o2* e nel secondo gruppo gli oggetti *o4* e *o5*; tuttavia la soluzione ottima è costituita dagli oggetti *o3* e *o4* per quanto *o3* non compaia nella soluzione ottima del primo sottoproblema.

1.2.Moltiplicazione di *n* Matrici

Supponiamo di dover calcolare il prodotto di n matrici:

$$M = M_1 \times M_2 \times ... \times M_i \times ... \times M_n$$

dove ogni matrice M_i ha r_i righe e r_{i+1} colonne per cui le matrici sono compatibili al prodotto; il risultato del prodotto è la matrice M con r_1 righe e r_{n+1} colonne. Poiché il prodotto è associativo, possiamo calcolarlo in diversi modi, ad esempio:

$$M = M_1 \times (M_2 \times (... \times (M_i \times (... \times M_n)...)...)) \text{ oppure}$$

$$M = (...(...(...(M_1 \times M_2) \times ...) \times M_i) \times ...) \times M_n$$

L'ordine di esecuzione non cambia il valore della matrice risultato M ma può comportare l'esecuzione di un diverso numero di operazioni, indipendentemente dal metodo scelto per la moltiplicazione di due matrici (ad esempio quello tradizionale, o il metodo di Strassen o altri metodi ancora più efficienti). Il problema da risolvere è individuare l'ordine di esecuzione del prodotto delle n matrici in modo da minimizzare il numero totale di operazioni.

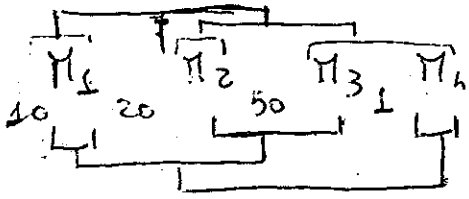
Esempio 1. Consideriamo come esempio il caso: $n = 4$, $r_1 = 10$, $r_2 = 20$, $r_3 = 50$, $r_4 = 1$ e $r_5 = 100$, cioè M_1 ha dimensioni 10×20, M_2 20×50, M_3 50×1, M_4 1×100, e M 10×100. Supponiamo di adoperare il metodo tradizionale per moltiplicare due matrici $M_i \times M_{i+1}$; pertanto il tempo di esecuzione di un prodotto è $\Theta(r_i \cdot r_{i+1} \cdot r_{i+2})$ e l'operazione dominante è la moltiplicazione di due numeri, che è eseguita $r_i \cdot r_{i+1} \cdot r_{i+2}$ volte. Sono possibili 5 modalità differenti per calcolare il prodotto delle 4 matrici. La modalità che richiede il maggior numero di operazioni è la seguente:

(1) $M = M_1 \times (M_2 \times (M_3 \times M_4))$

in cui abbiamo 50×1×100 = 5.000 operazioni per moltiplicare la terza e quarta matrice ottenendo una matrice intermedia di dimensione 50×100; ulteriori 20×50×100 = 100.000 operazioni per moltiplicare la seconda matrice con la matrice intermedia ottenendo una seconda matrice intermedia 20×100; infine 10×20×100 = 20.000 operazioni per moltiplicare la prima matrice con la seconda matrice intermedia ottenendo la matrice M; in totale sono eseguite 125.000 operazioni.

La modalità con il minor numero di operazioni è la seguente

(2) $M = (M_1 \times (M_2 \times M_3)) \times M_4$



PEGGIORE 125.000 op.

MIGLIORE 2.200 op.

in cui abbiamo 20×50×1 = 1000 operazioni per moltiplicare la seconda e terza matrice ottenendo una matrice intermedia di dimensione 20×1; ulteriori 10×20×1 = 200 operazioni per moltiplicare la prima matrice con la matrice intermedia ottenendo una seconda matrice intermedia 10×1; infine 10×1×100 = 1.000 operazioni per moltiplicare la seconda matrice intermedia con la quarta matrice ottenendo la matrice M; in totale sono eseguite 2.200 operazioni, cioè meno del 20% delle operazioni richieste nel primo caso.

Domanda 1. *Quali sono le altre modalità per calcolare il prodotto delle quattro matrici e quante operazioni sono eseguite in ciascuna di esse?*

Per determinare la modalità ottima di esecuzione del prodotto di n matrici potremmo semplicemente valutare tutte le possibili modalità e quindi scegliere quella migliore. Sfortunatamente questo approccio è estremamente costoso in quanto il numero di modalità di eseguire la moltiplicazione di n matrici (corrispondente al numero di modi con cui si può completamente parentesizzare una stringa di n simboli) è $M(n) = \Omega(2^n)$. Infatti, ovviamente $M(1) = 1$. Consideriamo ora il caso $n > 1$. Abbiamo n-1 sottocasi: per ogni k, $1 \le k < n$, l'ultimo prodotto consiste nel moltiplicare il risultato del prodotto delle prime k matrici con il risultato del prodotto delle ultime $(n-k)$ matrici. A sua volta ogni sottocaso è composto da tanti sottocasi K quanti sono le modalità diverse di moltiplicare le prime k matrici e le ultime $(n-k)$ matrici, cioè $K = M(k) \times M(n-k)$. Quindi

$$M(n) = \begin{cases} 1 & \text{se } n = 1 \\ \sum_{k=1}^{n-1} M(k) \times M(n-k) & \text{se } n > 1 \end{cases}$$

E' noto in letteratura che, risolvendo la ricorrenza, si ottiene che $M(n) = C(n\text{-}1)$ dove

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{2n!}{n!n!(n+1)}$$

è il *numero di Catalan*.

Esercizio 1. *Dimostrare che* $M(n) = \Omega(2^n)$.

Domanda 2. *Quante sono le coppie di parentesi in una modalità di esecuzione del prodotto di n matrici?*

La Programmazione Dinamica in questo caso si comporta molto bene in quanto permette di risolvere il problema in tempo polinomiale nel seguente modo. Sia m_{ij}, $1 \le i \le j \le n$, il minimo costo (misurato in numero di operazioni) per calcolare:

$$M_i \times M_{i+1} \times \ldots \times M_j$$

Nel caso $i = j$, non viene effettuata alcuna moltiplicazione poiché il risultato di essa coincide ovviamente con $M_i = M_j$. Abbiamo dunque:

$$m_{ij} = \begin{cases} 0 & \text{se } i = j \\ \min_{i \le k < j}(m_{ik} + m_{k+1,j} + r_i r_{k+1} r_{j+1}) & \text{se } i < j \end{cases}$$

Il termine m_{ik} è il costo minimo per calcolare il prodotto $M' = M_i \times M_{i+1} \times \ldots \times M_k$ e il termine $m_{k+1,j}$ è il costo minimo per calcolare il prodotto $M'' = M_{k+1} \times M_{k+2} \times \ldots \times M_j$. Il terzo termine è il costo per calcolare $M' \times M''$, poichè M' è una matrice di dimensioni $r_i \times r_{k+1}$ e M'' è una matrice di dimensioni $r_{k+1} \times r_{j+1}$. Il termine m_{ij} rappresenta il minimo di tutte le possibili somme dei tre termini ottenute facendo variare l'indice k da i a j-1 e, quindi, coincide effettivamente con il minimo costo per calcolare il prodotto delle matrici da i a j. Secondo l'approccio della Programmazione Dinamica, i minimi costi m_{ij} vanno calcolati in ordine crescente del valore $v=j-i$: pertanto, dopo aver fissato che $m_{ii} = 0$ per $1 \le i \le n$, vengono calcolati i costi $m_{i,i+1}$ per $1 \le i \le n$-1, poi i costi $m_{i,i+2}$ per $1 \le i \le n$-2, e così via fino a calcolare m_{1n}. Il principio di ottimalità vale in quanto la determinazione dell'esecuzione ottima di un'espressione costituita dal prodotto di n matrici comporta anche la determinazione dell'esecuzione ottima di qualsiasi sottoespressione di essa e, quindi, la correttezza è garantita. L'algoritmo è il seguente:

```
Card minProdottoMatrici( const Vettore<Card>& r, Card n )
{     assert( n > 0 && r.iMin == 1 && r.n == n+1 );
      Matrice<Card> m(n,n);
      // inizializzazione dei costi
      for ( Card i = 1; i <= n; i++ )
            m[i][i] = 0;
      // calcolo costi per valori crescenti di v = i-j
      for ( Card v = 1; v <= n-1; v++ )
            for ( i = 1; i <= n-v; i++ )
            {     Card j = i+v;
                  // calcolo di m_ij come minimo costo
                  m[i][j] = m[i+1][j]+r[i]*r[i+1]*r[j+1]; // k=i
                  for ( Card k = i+1; k < j; k++ )
                  {     Card mTemp = m[i][k]+m[k+1][j]+
                                     r[i]*r[k+1]*r[j+1];
                        m[i][j] = min ( mTemp, m[i][j] );
                  }
            }
      return m[1][n];
}
```

L'algoritmo ha complessità temporale $\Theta(n^3)$.

Esempio 2. Applichiamo l'algoritmo per determinare la modalità ottima per calcolare il prodotto delle 4 matrici dell'Esempio 1. Utilizziamo la tabella di Figura 1 per memorizzare i costi minimi. Nella fase di inizializzazione azzeriamo la riga con v=0. Quindi calcoliamo m_{12} = 10000, m_{23} = 1000 e m_{34} = 5000; quindi m_{13} = 1200 e m_{24} = 3000; infine m_{14} = 2200. Come esempio di uso della tabella, mostriamo come viene calcolato m_{14}. Per k =1, valutiamo il costo $m_{11} + m_{24} + 10\times20\times100 = 0+3000+20000 = 23000$; per k =2, valutiamo il costo $m_{12} + m_{34} + 10\times 50\times100 = 10000+5000+50000 = 65000$; infine, per k =3, valutiamo il costo $m_{13} + m_{44} + 10\times1\times100 = 1200+0+1000 = 2200$ che risulta essere il costo ottimo. Notiamo che nel calcolo di un costo corrente vengono utilizzati solo i costi che stanno nelle due diagonali che passano per esso.

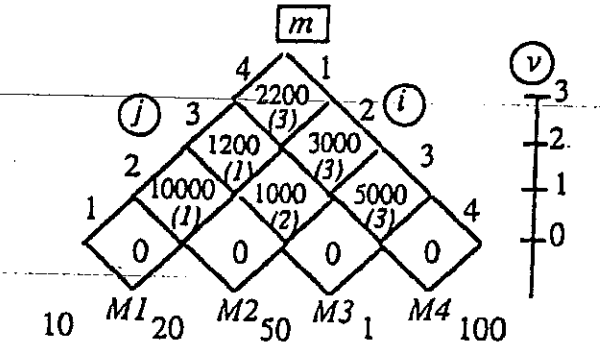


Figura 1. *Tabella per il calcolo dei costi minimi*

L'algoritmo restituisce il costo ottimo ma non la modalità ottima di esecuzione del prodotto. Occorre quindi modificarlo per registrare una informazione opportuna da cui ricavare la modalità di esecuzione. A tale scopo, così come indicati in Figura 1 tra parentesi, possiamo registrare i vari k per cui si sono ottenuti i minori costi. Allora possiamo risalire alla modalità ottima nel seguente modo; poichè nella cella (1,4) k=3, va effettuato il prodotto $(M_1 \times M_2 \times M_3) \times M_4$; poichè nella cella (1,3) k=1, il prodotto diventa $(M_1 \times (M_2 \times M_3)) \times M_4$, che rappresenta la modalità ottima di eseguire il prodotto.

Esercizio 2. *Date le matrici* M_1 30×35, M_2 35×15, M_3 15×5, M_4 5×10, M_5 10×20, *e* M_6 20×25, *determinare in maniera tabellare la modalità ottima di esecuzione del prodotto.*

Esercizio 3. *Modificare la funzione* minProdottoMatrici *in modo che esso restituisca anche gli indici k che permettano di individuare la modalità ottima di esecuzione e valutarne la complessità.*

Esercizio 4. *Scrivere un programma che legga le informazioni sulle dimensioni delle n matrici da moltiplicare e stampi la modalità ottima di esecuzione.*

1.3.Soluzione Esatta del Problema della Bisaccia 0-1

Affrontiamo nuovamente il problema della bisaccia 0-1, per il quale abbiamo visto nel paragrafo 2.1 dell'Unità 6 che la tecnica golosa fornisce solo una soluzione approssimata. Mostriamo ora come sia possibile ottenere una soluzione esatta utilizzando la tecnica della programmazione dinamica.

Abbiamo n oggetti, ciascuno con valore v_i e costo c_i, e abbiamo una disponibilità massima C per coprire i costi. Dobbiamo:

```
void migliorOrdineDinamico(Vettore<int> r, Matrice<int>& s, Matrice<int>& m)
{
	int n = r.n() - 1;
	for (int i = 1; i <= n; i++)
		m(i,i) = 0;
	for (int l = 2; l <= n; l++)
		for (int i = 1; i <= n - l + 1; i++)
		{
			int j = i + l - 1;
			// prova tutti i possibili valori di k da i fino a j-1
			// la prima iterazione viene fatta fuori dal for, per avere un primo valore
			// minimo anzichè impostare m(i,j) = +infinito.
			int k = i;
			m(i,j) = m(i,k) + m(k+1,j) + r[i]*r[k+1]*r[j+1];
			s(i,j)=k;
			for (k = i+1; k <= j-1; k++)
			{
				int q = m(i,k) + m(k+1,j) + r[i]*r[k+1]*r[j+1];
				if ( q < m(i,j) )
				{
					m(i,j) = q;
					s(i,j) = k;
				}
			} // for k
		} // for i, for l
}

template <class T>
Matrice<T> produttoria(const Vettore< Matrice<T> > v)
{
	Matrice<int> s(1,v.n(),1,v.n());
	Matrice<int> m(1,v.n(),1,v.n());

	// riempie il vettore p con le dimensioni su cui calcolare
	// l'ordine ottimale
	Vettore<int> p(1,v.n()+1);
	p[1] = v[1].n();
	for (int i = 2; i <= v.n()+1; i++)
		p[i] = v[i].m();

	migliorOrdineDinamico(p,s,m);
	return moltiplicaTanteMatrici(v,s,1,v.n());
}


template <class T>
Matrice<T> moltiplicaTanteMatrici(const Vettore< Matrice<T> >& v, const Matrice<int>& s, const int i, const
int j)

{
	if (j > i) {
		Matrice<T> X = moltiplicaTanteMatrici(v,s,i,s(i,j));
		Matrice<T> Y = moltiplicaTanteMatrici(v,s,s(i,j)+1,j);
		return X*Y;
	}
	else return v[i];
}
```

(2) All'inizio, definiamo i valori iniziali di alcuni elementi della tabella, corrispondenti ai sottoproblemi più semplici.
(3) Al generico passo, avanziamo in modo opportuno sulla tabella calcolando il valore della soluzione di un sottoproblema (corrispondente ad un dato elemento della tabella) in base alla soluzione dei sottoproblemi precedentemente risolti (corrispondenti ad elementi della tabella precedentemente calcolati).
(4) Alla fine, restituiamo la soluzione del problema originario, che è stata memorizzata in un particolare elemento della tabella.

Vedremo ora due ulteriori esempi di utilizzo della tecnica di programmazione dinamica.

10.2.1 La distanza tra due stringhe di caratteri

Date due stringhe x ed y, desideriamo calcolare la "distanza" tra x ed y, misurata opportunamente in termini delle differenze tra le due stringhe. Ad esempio, potremmo essere interessati a trasformare la stringa x nella stringa y. Questo è un problema che nasce nelle correzioni ortografiche automatiche (*spell checking*) dei documenti: in tal caso, i correttori ortografici (*spell checker*) non riconoscono soltanto un vocabolo errato in un documento, ma propongono anche delle possibili correzioni o sostituzioni: la scelta di queste sostituzioni si basa sulla ricerca di tutte le parole che hanno una distanza limitata da quella scritta in modo errato.

Cerchiamo di definire meglio cosa intendiamo per "distanza" tra due stringhe. Siano $X = x_1 \cdot x_2 \cdot \cdots \cdot x_m$ e $Y = y_1 \cdot y_2 \cdot \cdots \cdot y_n$ due stringhe di caratteri, di lunghezza rispettivamente m e n. Possiamo definire il costo della trasformazione di X in Y come il numero di cambiamenti che dobbiamo apportare alla stringa X per ottenere Y. I cambiamenti (od operazioni) che possiamo compiere mentre stiamo esaminando una stringa sono:

`inserisci`(a):	Inserisci il carattere a nella posizione corrente della stringa.
`cancella`(a):	Cancella il carattere a dalla posizione corrente della stringa.
`sostituisci`(a,b):	Sostituisci il carattere a con il carattere b nella posizione corrente della stringa.

Se assumiamo che il costo di ognuna di queste operazioni sia 1, possiamo definire il *costo della trasformazione tra X e Y* come la somma di tutti i costi che abbiamo pagato per compiere le operazioni di trasformazione da X in Y. Potremmo definire quindi la *distanza tra due stringhe X ed Y* come il costo minimo di trasformazione di X in Y.

Esempio 10.1 Supponiamo di voler trasformare la stringa $X =$ RISOTTO nella stringa $Y =$ PRESTO. La soluzione più elementare è cancellare tutti i caratteri di X e poi aggiungere tutti quelli di Y, come illustrato in Tabella 10.1. In questo modo il costo totale per la trasformazione di X in Y è 13, dato da 7 cancellazioni

Azione	Costo	Stringa ottenuta
Cancello R	1	|*ISOTTO*
Cancello I	1	|*SOTTO*
Cancello S	1	|*OTTO*
Cancello O	1	|*TTO*
Cancello T	1	|*TO*
Cancello T	1	|*O*
Cancello O	1	|
Inserisco P	1	P|
Inserisco R	1	PR|
Inserisco E	1	PRE|
Inserisco S	1	PRES|
Inserisco T	1	PREST|
Inserisco O	1	PRESTO|

Tabella 10.1 Possibile trasformazione della stringa RISOTTO nella stringa PRESTO, il cui costo totale è pari a 13.

seguite da 6 inserimenti. Se desideriamo ottenere un costo totale inferiore a 13, potremmo analizzare la stringa PRESTO, carattere per carattere, e compiere delle opportune modifiche, come illustrato nella Tabella 10.2. Questa trasformazione riduce il costo da 13 a 4. È possibile verificare che non esiste una trasformazione da RISOTTO a PRESTO di costo inferiore a 4, e quindi la distanza tra RISOTTO e PRESTO è proprio 4. □

Date due stringhe X ed Y, desideriamo progettare un algoritmo per calcolare la minima distanza tra X e Y, ovvero il numero minimo di operazioni sufficienti a trasformare X in Y. Indichiamo con $\delta(X, Y)$ la distanza tra le stringhe X ed Y. Data una stringa $X = x_1 \cdot x_2 \cdot \cdots \cdot x_m$, per $0 \leq i \leq m$, definiamo il *prefisso di X fino al carattere i-esimo* come la stringa $X_i = x_1 \cdot x_2 \cdot \cdots \cdot x_i$ se $i \geq 1$, e come la stringa vuota $X_0 = \emptyset$ se $i = 0$. Anzichè risolvere il problema generale $\mathcal{P}$, ovvero trovare la distanza $\delta(X, Y)$ tra la stringa X e la stringa Y, proviamo a considerare i sottoproblemi $\mathcal{P}(i, j)$, ovvero trovare la distanza $\delta(X_i, Y_j)$ tra il prefisso X_i ed il prefisso Y_j. Notiamo che:

(1) Alcuni sottoproblemi $\mathcal{P}(i, j)$ sono particolarmente semplici. Ad esempio la soluzione del sottoproblema $\mathcal{P}(0, j)$ consiste nel partire dalla stringa vuota $X_0 = \emptyset$ e nell'inserire uno dopo l'altro i j caratteri di Y_j. La distanza $\delta(X_0, Y_j)$ è dunque data da j. In modo completamente analogo, la soluzione del sottoproblema $\mathcal{P}(i, 0)$ consiste nel partire dalla stringa X_i contenente i caratteri, e nel cancellare uno dopo l'altro gli i caratteri di X_i per ottenere la stringa vuota Y_0. La distanza $\delta(X_i, Y_0)$ è dunque data da i.

Azione	Costo	Stringa ottenuta
Inserisco P	1	P|*RISOTTO*
Mantengo R	0	PR|*ISOTTO*
Sostituisco I con E	1	PRE|*SOTTO*
Mantengo S	0	PRES|*OTTO*
Cancello O	1	PRES|*TTO*
Mantengo T	0	PREST|*TO*
Cancello T	1	PREST|*O*
Mantengo O	0	PRESTO|

Tabella 10.2 Possibile trasformazione della stringa RISOTTO nella stringa PRESTO, il cui costo totale è pari a 4.

(2) $\mathcal{P} = \mathcal{P}(m, n)$, ovvero $\delta(X, Y) = \delta(X_m, Y_n)$.

Queste proprietà ci suggeriscono di provare a progettare un algoritmo di programmazione dinamica per il problema della distanza tra stringhe. A tale proposito, osserviamo che finora abbiamo già individuato tre punti importanti.

1. *I sottoproblemi da risolvere* $\mathcal{P}(i, j)$. Questa scelta ci consiglia di utilizzare una tabella bidimensionale D, ovvero una matrice $m \times n$, per memorizzare i risultati intermedi. In particolare, sembra opportuno memorizzare in $D[i, j]$ la soluzione al problema $\mathcal{P}(i, j)$, ovvero la distanza minima $\delta(X_i, Y_j)$ tra il prefisso X_i ed il prefisso Y_j.
2. *I valori iniziali di alcuni elementi della tabella*, corrispondenti ai sottoproblemi più semplici. In particolare, per qualsiasi j, $0 \leq j \leq n$, avremo che $D[0, j] = j$, e per qualsiasi i, $0 \leq i \leq m$, avremo che $D[i, 0] = i$. Questo implica che la colonna e la riga 0 della matrice D sono facilmente calcolabili.
3. *Il punto in cui è memorizzata nella tabella* D *la soluzione del problema originale*. In particolare, la soluzione del problema $\mathcal{P} = \mathcal{P}(m, n)$ sarà disponibile nell'elemento $D[m, n]$.

L'unica tessera che sembra ancora mancare dal nostro mosaico è come realizzare il generico passo di avanzamento nella tabella di programmazione dinamica, ovvero come calcolare il valore della soluzione del sottoproblema $\mathcal{P}(i, j)$ in funzione della soluzione dei sottoproblemi precedentemente risolti. Notiamo che ci sono varie possibilità, a seconda del valore dei caratteri x_i ed y_j. In particolare, se $x_i = y_j$, allora il minimo costo per trasformare X_i in Y_j sarà dato dal minimo costo per trasformare X_{i-1} in Y_{j-1}. In tal caso, avremo dunque che $D[i, j] = D[i-1, j-1]$. Se invece $x_i \neq y_j$, dovremo distinguire in base all'ultima operazione utilizzata per trasformare il prefisso X_i nel prefisso Y_j in una sequenza ottima di operazioni. In maggior dettaglio, se l'ultima operazione è una:

`inserisci`(y_j): Allora il costo minimo per trasformare X_i in Y_j sarà induttivamente dato dal costo minimo per trasformare

X_i in Y_{j-1}, più 1 per l'inserimento del carattere y_j. In tal caso, avremo dunque che

$$D[i,j] = D[i,j-1] + 1$$

`cancella`(x_i): Allora il costo minimo per trasformare X_i in Y_j sarà induttivamente dato dal costo minimo per trasformare X_{i-1} in Y_j, più 1 per la cancellazione del carattere x_i. In tal caso, avremo dunque che

$$D[i,j] = D[i-1,j] + 1$$

`sostituisci`(x_i, y_j): Allora il costo minimo per trasformare X_i in Y_j sarà induttivamente dato dal costo minimo per trasformare X_{i-1} in Y_{j-1}, più 1 per la sostituzione del carattere $x_i$in y_j. In tal caso, avremo dunque che

$$D[i,j] = D[i-1,j-1] + 1$$

Osserviamo che le precedenti relazioni di ricorrenza assumono che l'ultima operazione utilizzata per trasformare il prefisso X_i nel prefisso Y_j in una sequenza ottima di operazioni sia nota a priori. Questa ipotesi ovviamente non è verificata, dato che la sequenza ottima è proprio ciò che vogliamo calcolare. Tuttavia, sono possibili solo tre tipi di operazioni, inserimento, cancellazione o sostituzione dell'ultimo carattere, e la sequenza ottima deve necessariamente utilizzare una di queste tre operazioni. Quindi, per trovare la scelta effettuata dalla sequenza ottima, possiamo calcolare i tre valori relativi all'ipotesi di inserimento, cancellazione e sostituzione dell'ultimo carattere, e scegliere il valore migliore tra i tre. Queste considerazioni ci conducono alla seguente relazione, che specifica come calcolare il valore della soluzione ottima del sottoproblema $\mathcal{P}(i,j)$ in funzione della soluzione di alcuni dei sottoproblemi precedentemente risolti:

$$D[i,j] = \begin{cases} D[i-1, j-1] & \text{se } x_i = y_j \\ 1 + \min\{D[i,j-1], D[i-1,j], D[i-1,j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

Osserviamo che questa relazione ci fornisce proprio l'informazione che ci mancava per definire completamente il passo di avanzamento nella tabella di programmazione dinamica per il nostro problema. Prima di specificare l'algoritmo, evidenziamone le principali differenze con l'algoritmo `fibonacci3` del Capitolo 1.

- Nel caso dei numeri di Fibonacci, i sottoproblemi considerati generano una tabella unidimensionale, ovvero un array. Nel caso della distanza tra stringhe, i sottoproblemi sono caratterizzati da due indici, e quindi generano una tabella bidimensionale, ovvero una matrice.
- Nel caso dei numeri di Fibonacci, i valori da inizializzare nell'array sono soltanto due: $F[0]$ e $F[1]$. Nel caso della distanza tra stringhe, i valori da inizializzare nella matrice sono $(m+n+1)$: per $0 \leq j \leq n$, inizializzeremo la riga 0 con $D[0,j] = j$, e per $0 \leq i \leq m$, inizializzeremo la colonna 0 con $D[i,0] = i$.

```
algoritmo distanzaStringhe(stringa X, stringa Y) → intero
1.  matrice D di (m + 1) × (n + 1) interi
2.  for i = 0 to m do D[i, 0] ← i
3.  for j = 1 to n do D[0, j] ← j
4.  for i = 1 to m do
5.      for j = 1 to n do
6.          if (x_i ≠ y_j) then
7.              D[i, j] ← 1 + min{D[i, j − 1], D[i − 1, j], D[i − 1, j − 1]}
8.          else D[i, j] ← D[i − 1, j − 1]
9.  return D[m, n]
```

Figura 10.3 Algoritmo di calcolo della distanza tra due stringhe.

- Nel passo generico, nel caso dei numeri di Fibonacci per calcolare il valore attuale ($F[i]$) è sufficiente disporre dei due valori precedenti ($F[i-1]$ e $F[i-2]$). Quindi l'array potrà essere riempito da sinistra a destra, secondo i valori crescenti dell'indice. Nel caso della distanza tra stringhe, per calcolare il valore attuale ($D[i, j]$) è sufficiente disporre dei tre valori $D[i-1, j-1]$, $D[i-1, j]$ e $D[i, j-1]$. Per assicurarci che tali valori siano disponibili quando servono, possiamo indifferentemente riempire la matrice D per righe oppure per colonne.
- Alla fine, la soluzione del problema dei numeri di Fibonacci si trova nell'ultimo elemento dell'array ($F[n]$). La soluzione del problema della distanza tra stringhe si trova all'incrocio tra l'ultima riga e l'ultima colonna della matrice ($D[m, n]$).

Siamo ora in grado di presentare l'algoritmo `distanzaStringhe`, illustrato nella Figura 10.3.

Esempio 10.2 Nella tabella 10.3 illustriamo la matrice di programmazione dinamica costruita dall'algoritmo `distanzaStringhe` sulle stringhe RISOTTO e PRESTO. Nella tabella viene indicata in grassetto una sequenza che consente di ottenere la distanza tra le due stringhe, da cui è possibile ricavare una sequenza di operazioni di costo minimo in grado di trasformare la stringa RISOTTO nella stringa PRESTO. Notiamo che alla riga 5, corrispondente alla prima T di RISOTTO abbiamo due possibilità: provenendo da $D[4, 4]$ con un costo di 3, possiamo cancellare subito la T andando in $D[5, 4]$ con un costo di 4, oppure tenere la T andando in $D[5, 5]$ con un costo di 3; in questo secondo caso, quando andremo in $D[6, 5]$ dovremo cancellare la seconda T. È facile verificare che la sequenza corrispondente a questa seconda scelta è esattamente quella già illustrata nella Tabella 10.2. □

Il seguente teorema caratterizza la complessità di spazio e di tempo dell'algoritmo `distanzaStringhe`.

Teorema 10.1 *L'algoritmo* `distanzaStringhe` *richiede un tempo di esecuzione* $O(mn)$*, dove* X *ed* Y *sono le due stringhe in ingresso, con* $|X| = m$ *e* $|Y| = n$*. L'occupazione di memoria è* $O(mn)$*.*

		P	R	E	S	T	O
	0	**1**	2	3	4	5	6
R	1	1	**1**	2	3	4	5
I	2	2	2	**2**	3	4	5
S	3	3	3	3	**2**	3	4
O	4	4	4	4	**3**	3	3
T	5	5	5	5	**4**	**3**	4
T	6	6	6	6	5	**4**	4
O	7	7	7	7	6	5	**4**

Tabella 10.3 Tabella di programmazione dinamica costruita dall'algoritmo `distanzaStringhe` sulle stringhe `RISOTTO` e `PRESTO`. In grassetto vengono indicate due sequenze di operazioni che consentono di ottenere la distanza tra le due stringhe.

Dimostrazione. Consideriamo l'algoritmo `distanzaStringhe` illustrato nella Figura 10.3. La fase di inizializzazione (righe 2–3) può essere chiaramente implementata in tempo $O(m + n)$, mentre invece il ciclo `for` esterno (righe 4–8) implica l'esame dell'intera matrice D, e quindi un tempo totale $O(mn)$. Per quanto riguarda l'occupazione di memoria, la matrice D ha dimensione $(m+1) \cdot (n+1)$ e quindi anche lo spazio totale richiesto dall'algoritmo è $O(mn)$. □

Concludiamo la nostra analisi sul problema della distanza tra due stringhe, osservando che lo spazio utilizzato dall'algoritmo può essere sensibilmente ridotto. Possiamo infatti fare un ragionamento molto simile a quello fatto nel caso dei numeri di Fibonacci nel Capitolo 1, in cui nel passare dall'algoritmo `fibonacci3` all'algoritmo `fibonacci4`, abbiamo osservato che per calcolare l'i-esimo numero di Fibonacci non serviva avere a disposizione tutto l'array, ma soltanto i due valori precedenti. Anche nel caso della distanza tra le stringhe, non è necessario mantenere tutta la matrice D: per calcolare un generico elemento $D[i, j]$, è sufficiente avere a disposizione la riga $(i - 1)$, e la colonna $(j - 1)$: in particolare, è possibile riempire la matrice D, riga per riga o colonna per colonna, mantenendo durante tutta l'esecuzione dell'algoritmo solamente una riga ed una colonna. Questa semplice osservazione è in grado di ridurre lo spazio richiesto dall'algoritmo `distanzaStringhe` da $O(mn)$ a $O(m + n)$.

10.2.2 Associatività del prodotto tra matrici

Introduciamo il problema con un esempio concreto. Supponiamo di dover eseguire il prodotto di tre matrici

$$M = M_1 \cdot M_2 \cdot M_3$$