1. Programmazione dinamica

Una filosofia simile al *divide-et-impera*, ma portata all'estremo, si basa sulla massima:

"Il pesce grosso mangia tutti i pesci più piccoli".

In altri termini, se non si sa con esattezza quali sottoproblemi risolvere, si può provare a risolverli tutti e conservare i risultati ottenuti per poterli usare successivamente nel risolvere il problema di dimensione più grande. Questa tecnica è detta *programmazione dinamica*, perché indica un processo di programmazione (o pianificazione) di un metodo risolutivo che sia applicabile dinamicamente per dimensioni via via crescenti del problema. Rispetto al *divide-et-impera*, la programmazione dinamica presenta tre grandi differenze: è iterativa e non ricorsiva, affronta i sottoproblemi "dal basso verso l'alto" anziché "dall'alto verso il basso", e memorizza i risultati dei sottoproblemi in una tabella. Comunque, mentre il *divide-et-impera* risulta vantaggioso quando i sottoproblemi più piccoli da risolvere sono indipendenti, la programmazione dinamica è conveniente quando i sottoproblemi non sono indipendenti, ma ne condividono altri. Procedendo dal basso verso l'alto, la programmazione dinamica risolve un sottoproblema comune una sola volta, mentre il *divide-et-impera* lo risolverebbe più volte.

Esempio (Coefficiente binomiale) $C(n,k) = n!/(k!(n-k)!)$ rappresenta il numero di modi di scegliere k oggetti da un insieme di n oggetti, con $0 \leq k \leq n$, ed è definibile ricorsivamente come segue:

$$C(n,k) = \begin{cases} 1 & \text{se } k = 0 \vee k = n \\ C(n-1,k-1) + C(n-1,k) & \text{altrimenti} \end{cases}$$

Un algoritmo ricorsivo *divide-et-impera* si ottiene immediatamente dalla precedente definizione ricorsiva:

```
int C(int n,k)
    if n = k or k = 0 then return 1
                      else return C(n−1,k−1)+C(n−1,k)
```

Purtroppo, la complessità dell'algoritmo *divide-et-impera* cresce come il numero di chiamate ricorsive, che è proprio uguale a $C(n,k)$. Questo è dovuto all'elevato numero di sottoproblemi identici che vengono risolti più volte. Per esempio già $C(n-2,k-1)$ è richiamata due volte, sia per calcolare $C(n-1,k)$ che $C(n-1,k-1)$, mentre al decrescere dei valori degli argomenti ciascun sottoproblema è risolto per un numero sempre maggiore di volte. Un algoritmo di programmazione dinamica, invece, risolve i sottoproblemi per valori crescenti degli argomenti secondo il ben noto schema del "triangolo di Tartaglia", memorizzando i risultati una volta per tutte in una matrice intera $C[0\ldots n][0\ldots n]$ in $\Theta(n^2)$ tempo.

```
tartaglia(int n, int[][] C)
    for i = 0 to n do
        C[i,0] = 1
        C[i,i] = 1
    for i = 2 to n do
        for j = 1 to i−1 do
            C[i, j] = C[i−1, j−1]+C[i−1, j]
```

In questo modo, il valore di $C(m,k)$, per ogni coppia di interi m,k tali che $0 \leq k \leq m \leq n$, può essere successivamente letto direttamente dalla matrice C in tempo $O(1)$. ■

La programmazione dinamica è stata introdotta da Bellman (1957) per progettare algoritmi che risolvono problemi di ottimizzazione. Perché sia applicabile, occorre che:

(1) sia possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più grande;
(2) le decisioni prese per risolvere in modo ottimo un sottoproblema rimangano valide quando il sottoproblema diviene un "pezzo" di un problema più grande;
(3) una tecnica *divide-et-impera* richieda di risolvere più volte gli stessi sottoproblemi e sia pertanto inutilizzabile da un punto di vista computazionale.

Le proprietà (1) e (2) prendono il nome di *sottostruttura ottima*. Ma non sono sufficienti; in pratica, affinché un algoritmo basato sulla programmazione dinamica abbia complessità polinomiale, occorre inoltre che:

(4) ci sia un numero polinomiale di sottoproblemi da risolvere;
(5) per evitare di risolvere più di una volta lo stesso sottoproblema, si usi una tabella in cui si memorizzano le soluzioni di tutti i sottoproblemi, senza preoccuparsi se la soluzione di un particolare sottoproblema verrà poi utilizzata oppure no;
(6) il tempo per combinare le soluzioni dei sottoproblemi e trovare la soluzione del problema più grande sia polinomiale.

Questi principi generali sono di difficile comprensione se non vengono calati in esempi concreti. Questo capitolo ne contiene un discreto numero, ognuno dei quali insegna un particolare aspetto della programmazione dinamica.

1.1 String matching approssimato

Nei motori di ricerca e in generale nelle applicazioni che si occupano di ricerca di testo, può capitare di dover risolvere il seguente problema.

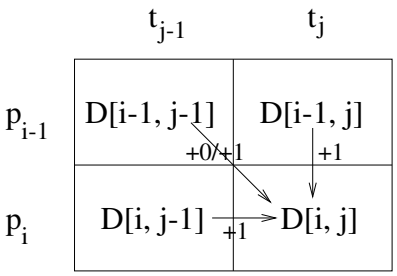

Figura 1.1: Calcolo di $D[i,j] = \min\{D[i-1,j-1]+\delta, D[i-1,j]+1, D[i,j-1]+1\}$, dove $\delta = 0$, se $p_i = t_j$, oppure $\delta = 1$, se $p_i \neq t_j$.

STRING MATCHING APPROSSIMATO (APPROXIMATE STRING MATCHING). *Date una stringa $P = p_1 \cdots p_m$ (pattern) e una stringa $T = t_1 \cdots t_n$ (testo), con $m \leq n$, un'occorrenza k-approssimata di P in T, con $0 \leq k \leq m$, è una copia della stringa P nella stringa T in cui sono ammessi k "errori" (o differenze) tra caratteri di P e caratteri di T, del seguente tipo:*

(1) i corrispondenti caratteri in P e in T sono diversi (sostituzione);
(2) un carattere in P non è incluso in T (inserimento);
(3) un carattere in T non è incluso in P (cancellazione).

Il problema consiste nel trovare un'occorrenza k-approssimata di P in T per cui k sia minimo.

Esempio (Occorrenza 2-approssimata) Siano T = "`questoèunoscempio`" (dove $n = 17$) e P = "`unesempio`" (dove $m = 9$). Un'occorrenza 2-approssimata di P parte dalla posizione 8 di T: `questoèunoscempio`. Infatti, la prima `e` di P corrisponde ad una `o` di T (errore di tipo (1)), mentre la `c` di T è un carattere non incluso in P (errore di tipo (3)). ■

Per comodità, definiamo il *prefisso* $X(k) = x_1 \cdots x_k$ di una stringa $X = x_1 \cdots x_n$ come la sottostringa formata dai primi k caratteri.

Per applicare la programmazione dinamica, dobbiamo trovare una formulazione ricorsiva che abbia la proprietà di sottostruttura ottima. Definiamo $D[i,j]$, con $0 \leq i \leq m$ e $0 \leq j \leq n$, come il minimo valore k per cui esiste un'occorrenza k-approssimata di $P(i)$ in $T(j)$. È possibile calcolare $D[i,j]$, per $1 \leq i \leq m$ e $1 \leq j \leq n$, in funzione dei tre casi seguenti, riassunti in Figura 1.1, che tengono conto del tipo di errore che può occorrere tra p_i e t_j:

(1) $D[i-1,j-1]$, se $p_i = t_j$, oppure $D[i-1,j-1]+1$, se $p_i \neq t_j$, perché "avanzando" di un carattere sia nel pattern che nel testo il numero di errori resta identico se i due caratteri sono uguali mentre aumenta di uno se sono diversi (errore di tipo (1));
(2) $D[i-1,j]+1$, perché "avanzando" di un carattere solo nel pattern il numero di errori aumenta di uno in quanto p_i non compare in T (errore di tipo (2));
(3) $D[i,j-1]+1$, perché "avanzando" di un carattere solo nel testo il numero di errori aumenta di uno in quanto t_j non compare in P (errore di tipo (3)).

La ricorsione termina quando una delle due stringhe diventa vuota. Se il pattern è vuoto, il numero di errori è 0, perché la stringa vuota $P(0)$ occorre in ogni posizione di T; se il testo è vuoto, invece, il numero di errori è pari alla lunghezza della stringa pattern, perché questi caratteri devono essere tutti cancellati. Riassumendo, si ottengono pertanto la seguenti relazioni di programmazione

dinamica, per $0 \leq i \leq m$ e $0 \leq j \leq n$:

$$D[i,j] = \begin{cases} 0 & \text{se } i = 0 \\ i & \text{se } j = 0 \\ \min\{D[i-1,j-1] + \delta, D[i-1,j] + 1, D[i,j-1] + 1\} & \text{altrimenti} \end{cases}$$

dove $\delta = 0$, se $p_i = t_j$, oppure $\delta = 1$, se $p_i \neq t_j$. Non abbiamo fornito una dimostrazione formale, ma è chiaro che tale definizione illustra la proprietà di sottostruttura ottima del problema.

Si noti che $D[m,j] = k$ se e solo se c'è un'occorrenza k-approssimata di P in T che termina in t_j e che la soluzione del problema è data dal valore di $D[m,j]$ più piccolo, per $0 \leq j \leq n$. La funzione stringMatching() calcola la matrice D e restituisce la posizione di T in cui termina un'occorrenza approssimata di P col numero minimo di errori. La complessità della funzione è $\Theta(mn)$, poiché sono valutati esplicitamente tutti gli elementi di D.

```
int stringMatching(item[] P, item[] T, int m, int n)
    int[][] D = new int[0...m][0...n]
    for i = 1 to m do D[i,0] = i
    for j = 0 to n do D[0,j] = 0
    for i = 1 to m do
        for j = 1 to n do
            int t = D[i-1,j-1] + iif(P[i] = T[j],0,1)
            t = min(t,D[i-1,j] + 1)
            t = min(t,D[i,j-1] + 1)
            D[i,j] = t
    int min = D[m,0]
    int pos = 0
    for j = 1 to n do
        if D[m,j] < min then
            min = D[m,j]
            pos = j
    return pos
```

Esempio (String matching approssimato) La Figura 1.2 mostra la matrice D ottenuta con le relazioni di programmazione dinamica per $T =$ `ababa` e $P =$ `bab`, dove $m = 3$ ed $n = 5$. Un'occorrenza 0-approssimata di P termina nella posizione 4 di T: `ababa`. ■

Le frecce in Figura 1.2 indicano la provenienza del minimo; la funzione può essere facilmente modificata, senza aumentare la complessità, al fine di ricavare le occorrenze approssimate e i relativi errori; è sufficiente percorrere a ritroso tali frecce. Si noti tuttavia che non è effettivamente necessario memorizzare a parte le frecce; è sufficiente, per ogni posizione $[i,j]$, cercare quale fra le tre posizioni $[i-1,j]$, $[i,j-1]$ e $[i-1,j-1]$ ha dato origine al valore minimo. L'esercizio Esercizio 1.1 chiede di scrivere tale funzione.

Una variante dello *string matching* approssimato prende il nome di *distanza di Levenshtein* o distanza di *editing* (in inglese, *edit distance*): date due stringhe, vogliamo conoscere il numero minimo di operazioni (sostituzione, inserimento, cancellazione) necessarie per trasformare una nell'altra (o viceversa, visto che inserimento e cancellazione sono simmetriche). Per esempio, la distanza di *editing* fra `google` e `yahoo` è pari a 6, perché bisogna cancellare i caratteri `gle`, inserire `ya` e sostituire `g` con `h`. Per adattarsi alla nuova semantica, la relazione di ricorrenza è la

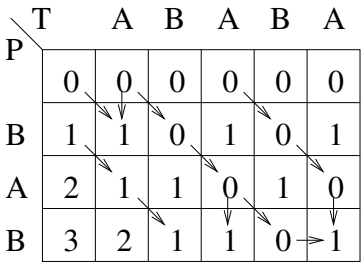

Figura 1.2: Risoluzione dello *string matching* approssimato con la programmazione dinamica.

stessa ma le condizioni iniziali cambiano: come prima, $D[i,0] = i$ per $1 \leq i \leq m$; ma ora anche $D[0,j] = j$, perché $T(j)$ ha j caratteri in più di $P(0)$ (da cancellare o inserire). Al termine, la distanza di *editing* si trova in $D[m,n]$.

Qualora sia richiesto, dato k, di trovare un'occorrenza approssimata di P in T con al più k errori, il problema può essere risolto in tempo $O(kn)$, ma il relativo algoritmo è complicato e non può essere qui riportato. Tale algoritmo, ideato da Landau e Vishkin (1988), riesce a valutare esplicitamente solo quegli elementi di D che provocano un incremento lungo una diagonale, cioè quelli per cui $D[i,j] > D[i-1,j-1]$, ed utilizza una struttura di dati ad albero detta *albero dei suffissi*.

1.2 Insieme indipendente di intervalli pesati

Supponete di essere il manager di un grande albergo. La vostra sala riunioni è molto richiesta, e vi trovate a dover decidere come affittarla nel prossimo mese. Avete ricevuto un certo numero di richieste, ognuna caratterizzata da un orario di inizio e un orario di fine, e da un guadagno associato. Il vostro compito è massimizzare il guadagno dell'albergo. È facile vedere che il vostro problema è un caso particolare di questo problema classico:

INSIEME INDIPENDENTE DI INTERVALLI PESATI. *Siano dati n intervalli distinti $[a_1,b_1[$, ..., $[a_n,b_n[$ della retta reale, aperti a destra, dove all'intervallo i è associato un peso w_i, $1 \leq i \leq n$. Trovare un insieme indipendente di peso massimo, ovvero un sottoinsieme di intervalli tutti disgiunti tra di loro tale che la somma dei pesi degli intervalli nel sottoinsieme sia la più grande possibile.*

Gli intervalli sono aperti a destra in quanto un evento che finisce a mezzogiorno può essere seguito da un evento che inizia alla stessa ora. Per risolvere questo problema, ordiniamo gli intervalli per estremi finali non decrescenti, in modo che $b_1 \leq b_2 \leq \cdots \leq b_n$. Per ogni intervallo i, sia $p_i = j$ il predecessore di i, dove $j < i$ è il massimo indice tale che $[a_j,b_j[$ non interseca $[a_i,b_i[$ (se non esiste j, allora $p_i = 0$).

Sia $P[i]$ il sottoproblema dato dai primi i intervalli e sia $S[i]$ una sua soluzione ottima di peso $D[i]$. Se l'intervallo i-esimo non fa parte di tale soluzione, allora deve valere $D[i] = D[i-1]$, dove si assume $D[0] = 0$; altrimenti, deve essere $D[i] = w_i + D[p_i]$.

Infatti, se $i \notin S[i]$, $S[i]$ è anche soluzione ottima per $P[i-1]$; se così non fosse, esisterebbe una soluzione ottima $S'[i-1]$ per $P[i-1]$ di peso $D'[i-1] > D[i]$. Ma tale soluzione sarebbe ottima anche per il problema $P[i]$, visto che i non è incluso, e questo è assurdo. Se invece $i \in S[i]$, $S[i] - \{i\}$ è una soluzione ottima per $P[p_i]$; se così non fosse, esisterebbe una soluzione ottima $S'[p_i]$ per $P[p_i]$ di peso $D'[p_i] > D[i] - w_i$. Ma allora basterebbe sostituire $S'[p_i]$ a $S[i] - \{i\}$ nella nostra soluzione ottima per il problema $P[i]$ e otterremmo una soluzione con costo $D'[p_i] + w_i > D[i] - w_i + w_i = D[i]$, assurdo.

Questo prova che il problema ha sottostruttura ottima; pertanto, la relazione di programmazione dinamica che si ottiene è:

$$D[i] = \max\{D[i-1], w_i + D[p_i]\}, \qquad i = 1, \ldots, n.$$

$D[n]$ è il costo della soluzione ottima del problema originario, partendo dalla quale è facile ricavare a ritroso gli intervalli di tale soluzione. Tutti i $p_1, \ldots, p_n$ si possono precalcolare in tempo $O(n)$ con una singola scansione degli intervalli. Basta ordinare i $2n$ estremi degli intervalli $\{a_1, \ldots, a_n, b_1, \ldots, b_n\}$ in modo non decrescente, utilizzare una variabile i inizializzata a 0 e scandire gli estremi da sinistra verso destra: se l'elemento scandito è un estremo iniziale, per esempio a_j, allora si assegna i a p_j, mentre se l'elemento è un estremo finale, per esempio b_j, allora si assegna j a i. Dopo la computazione dei predecessori, ciascun $D[j]$ si ricava in tempo costante. La procedura maxinterval() ha quindi costo totale $O(n \log n)$, dovuto all'ordinamento.

```
SET maxinterval(int[] a, int[] b, int[]w, int n)
{ ordina gli intervalli per estremi di fine crescenti }
{ calcola p_j }
int[] D = new int[0...n]
D[0] = 0
for i = 1 to n do
    D[i] = max(D[i-1], w[i] + D[p_i])
i = n
SET S = Set()
while i > 0 do
    if D[i-1] > w[i] + D[p_i] then
        i = i - 1
    else
        S.insert(i)
        i = p_i
return S
```

La procedura mostra che, con una preelaborazione non troppo costosa, è possibile risolvere un problema grazie alla programmazione dinamica.

1.3 Moltiplicazione catena di matrici

Date n matrici rettangolari $A_1, A_2, \ldots, A_n$, si consideri il problema di calcolare il prodotto $A_1 \cdot A_2 \cdots A_n$. Questo è possibile se per ogni i, $1 \leq i < n$, le matrici A_i e A_{i+1} sono *compatibili al prodotto*, ovvero il numero di colonne di A_i è uguale al numero di righe di A_{i+1}.

Moltiplicare due matrici A_i di dimensione $c_{i-1} \times c_i$ e A_{i+1} di dimensione $c_i \times c_{i+1}$ produce una matrice di dimensione $c_{i-1} \times c_{i+1}$ e richiede $c_{i-1} \cdot c_i \cdot c_{i+1}$ moltiplicazioni scalari, in quanto ogni elemento della matrice risultante viene calcolato dalla formula seguente:

$$\sum_{k=1}^{c_i} A_i[h,k] \cdot A_{i+1}[k,j], \qquad 1 \leq h \leq c_{i-1}, 1 \leq j \leq c_{i+1}$$

In generale, il numero totale di moltiplicazioni scalari necessarie per moltiplicare n matrici A_i di dimensione $c_{i-1} \times c_i$ dipende dall'ordine in cui vengono moltiplicate le matrici. Il prodotto fra matrici infatti è associativo.

Esempio (Moltiplicazione $A_1 \cdot A_2 \cdot A_3$) Si consideri la moltiplicazione di A_1 di dimensione 30×5, A_2 di dimensione 5×25 e A_3 di dimensione 25×7. Il prodotto delle tre matrici può essere calcolato come $(A_1 \times A_2) \times A_3$ oppure $A_1 \times (A_2 \times A_3)$. Nel primo caso sono necessarie $(30 \cdot 5 \cdot 25 + 30 \cdot 25 \cdot 7) = 9000$ moltiplicazioni, mentre nel secondo ne sono necessarie solo $(30 \cdot 5 \cdot 7 + 5 \cdot 25 \cdot 7) = 1925$. ■

Una domanda importante è la seguente.

MOLTIPLICAZIONE CATENA DI MATRICI (MATRIX CHAIN MULTIPLICATION). *Date n matrici A_1, ..., A_n, trovare l'ordine di moltiplicazione di costo minimo, ovvero che richiede il numero minimo di moltiplicazioni scalari.*

Avere queste informazione in anticipo permette di risparmiare tempo al momento della moltiplicazione effettiva.

Caratterizziamo la soluzione nel modo seguente: una *parentesizzazione* $P_{i,j}$ del prodotto $A_i \cdot A_{i+1} \cdots A_j$ consiste nella matrice A_i, se $i = j$; nel prodotto di due parentesizzazioni $(P_{i,k} \cdot P_{k+1,j})$, altrimenti. Una parentesizzazione $P_{i,j}$ si dice *ottima* se non esiste altra parentesizzazione $P'_{i,j}$ con costo inferiore.

Abbiamo caratterizzato la nostra soluzione in maniera ricorsiva; dobbiamo ora verificare che goda di sottostruttura ottima. Vogliamo dimostrare che nella parentesizzazione ottima $P_{i,j} = (P_{i,k} \cdot P_{k+1,j})$, le parentesizzazioni $P_{i,k}$ e $P_{k+1,j}$ sono anch'esse ottime. Per assurdo, supponiamo che esista una parentesizzazione $P'_{i,k}$ di $A_i \cdots A_k$ con costo inferiore a $P_{i,k}$ (il caso di $P'_{k+1,j}$ con costo inferiore a $P_{k+1,j}$ è simmetrico). Allora la parentesizzazione $(P'_{i,k} \cdot P_{k+1,j})$ avrebbe costo inferiore a $P_{i,j}$, ma questo è assurdo in quanto quest'ultima è ottima.

La definizione ricorsiva ci permetterebbe di scrivere una soluzione basata su *divide-et-impera*; ma molti problemi verrebbero ripetuti inutilmente. Il numero $F(n)$ di possibili parentesizzazioni è infatti definibile ricorsivamente nel modo seguente:

$$F(n) = \begin{cases} 1 & \text{se } n = 1, \\ \sum_{k=1}^{n-1} F(k)F(n-k) & \text{altrimenti,} \end{cases}$$

dove una singola matrice ha un solo modo per essere espressa in parentesi, mentre nel caso di n matrici, per ognuno dei possibili indici k ci sono $F(k)$ modi di parentesizzare $A_1 \cdots A_k$ e $F(n-k)$ modi di parentesizzare $A_{k+1} \cdots A_n$. $F(n)$ è l'$(n-1)$-esimo "numero catalano" e sappiamo che $F(n) = \Omega(4^n/n^{3/2})$, quindi un algoritmo ricorsivo basato su *divide-et-impera* è superpolinomiale.

È possibile esprimere il costo della parentesizzazione ottima $P_{i,j} = (P_{i,k} \cdot P_{k+1,j})$ come la somma del costo delle parentesizzazioni ottime $P_{i,k}$ e $P_{k+1,j}$ più $c_{i-1}c_kc_j$ moltiplicazioni scalari per moltiplicare tra loro le matrici risultanti di dimensione $c_{i-1} \times c_k$ e $c_k \times c_j$. Sia $M[i,j]$ il minimo numero di moltiplicazioni aritmetiche necessarie per calcolare $P_{i,j}$; è possibile esprimere tale costo tramite le seguenti relazioni di programmazione dinamica:

$$M[i,j] = \begin{cases} \min_{i \le k \le j-1} \{M[i,k] + M[k+1,j] + c_{i-1}c_kc_j\} & \text{se } 1 \le i < j \le n, \\ 0 & \text{se } 1 \le i = j \le n \end{cases}$$

La procedura parentesizzazione() calcola la matrice M, che viene riempita considerando il fatto che il costo per moltiplicare le sequenze di $h = j - i + 1$ matrici dipende soltanto dal costo per moltiplicare sequenze con meno di h matrici, per $h = 2, 3, \ldots, n$. Per $h = 1$, invece, il costo è zero ed è rappresentato dalla diagonale principale. Fissato h, gli estremi i e j della sequenza $A_i \cdots A_j$ di h matrici sono tali che $1 \le i \le n - h + 1$, mentre $j = i + h - 1$.

La procedura parentesizzazione() lavora in tempo $O(n^3)$. Ovviamente, oltre a calcolare il costo di una parentesizzazione ottima, è necessario anche produrne una. Questo richiede una tabella aggiuntiva S che contiene, per ogni parentesizzazione $P_{i,j}$, l'indice $S[i,j] = k$ che rappresenta il punto di moltiplicazione fra le due sottoparentesizzazioni. Sfruttando questa informazione, possiamo chiamare la procedura stampaPar$(S,1,n)$ per stampare una rappresentazione della parentesizzazione in tempo $O(n)$.

Si noti che, a differenza dei problemi precedenti, in questo caso la tabella aggiuntiva riduce il costo di stampaPar(). In assenza della tabella, infatti, avremmo dovuto cercare nuovamente il valore k dell'ultima moltiplicazione per ogni posizione visitata da stampaPar, portando il suo costo ad $O(n^2)$.

```
parentesizzazione(int[] c, int[][] M, int[][] S, int n)
int i, j, h, k, t
for i = 1 to n do
    M[i,i] = 0
for h = 2 to n do
    for i = 1 to n − h + 1 do
        j = i + h − 1
        M[i, j] = +∞
        for k = i to j − 1 do
            t = M[i,k] + M[k + 1, j] + c[i − 1] · c[k] · c[j]
            if t < M[i, j] then
                M[i, j] = t
                S[i, j] = k
```

```
stampaPar(int[][] S, int i, int j)
if i = j then
    print "A["; print i; print "]"
else
    print "("; stampaPar(S, i, S[i, j]); print "·"; stampaPar(S, S[i, j] + 1, j); print ")"
```

1.4 Occupazione di memoria

Negli esempi visti finora, le tabelle di programmazione dinamica hanno una dimensione uguale al numero di possibili sottoproblemi. Una volta completate, le tabelle possono essere utilizzate da procedure come stampaPar() per ricostruire una soluzione ottima. Vi sono tuttavia alcuni casi in cui è possibile evitare di mantenere in memoria tutti i sottoproblemi risolti.

Si consideri come esempio la serie di Fibonacci. Dato un intero $n \geq 0$, l'n-esimo numero di Fibonacci F_n è definito ricorsivamente come segue: $F_0 = 1$, $F_1 = 1$, ed $F_n = F_{n-1} + F_{n-2}$ per $n > 1$. Una soluzione *divide-et-impera* basata sulla definizione ricorsiva tende a risolvere un gran numero di problemi ripetuti. È infatti possibile dimostrare che il valore di F_n è circa $(1.6)^n$ e pertanto la soluzione *divide-et-impera* richiede un numero di chiamate che cresce anch'esso come $(1.6)^n$.

Una soluzione basata su programmazione dinamica memorizza tutti i numeri di Fibonacci fino ad F_n, evitando così problemi ripetuti. Calcolando i valori in maniera iterativa, si ottiene un algoritmo il cui costo è $O(n)$, sia in termini di tempo che di spazio.

```
int fibonacci(int n)
    int[] F = new int[0...max(n,1)]
    F[0] = F[1] = 1
    for i = 2 to n do
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Una domanda importante è la seguente: è veramente necessario un vettore di n posizioni? In realtà, la posizione $F[i]$ smette di essere utilizzata dopo aver calcolato $F[i+2]$. In pratica, non serve un vettore di dimensione n, ma bastano tre variabili F_0, F_1, F_2. Queste variabili vedono scorrere i numeri di Fibonacci in modo che ad ogni iterazione il valore di F_0 viene scartato e sostituito con F_1, F_1 viene sostituito con F_2 e il nuovo termine delle serie di Fibonacci viene calcolato in F_2. La complessità è uguale, lo spazio questa volta è costante.

```
int fibonacci(int n)
    int F0, F1, F2
    F0 = F1 = F2 = 1
    for i = 2 to n do
        F0 = F1
        F1 = F2
        F2 = F0 + F1
    return F2
```

Il problema di Fibonacci può sembrare un caso particolare, non applicabile ad altri problemi. In realtà, sotto certe condizioni, una tecnica simile può essere applicata anche al problema dello *string matching* approssimato.

Supponiamo che non sia necessario produrre in output l'occorrenza con k errori, ma si voglia solamente ottenere il minimo valore k (analogamente, si voglia ottenere solamente la distanza di *editing*). Questo significa che al termine del problema saremmo interessati solo all'ultima riga della tabella, per individuare il valore minimo. Durante l'esecuzione, per calcolare una riga è sufficiente avere a disposizione i soli tre valori illustrati in Figura 1.3. E' quindi possibile memorizzare due sole righe alla volta, come illustrato nella procedura stringMatchingSingleRow(). L'algoritmo si avvale di due vettori, D che contiene la riga corrente (quella che viene attualmente calcolata) e PD che contiene la riga precedente. Ogniqualvolta viene calcolata una nuova riga, si sposta la riga corrente D nella riga precedente PD scambiando i puntatori dei due vettori, per poi utilizzare la formula ricorsiva illustrata in precedenza. Al termine del calcolo di m righe, è sufficiente identificare il valore minimo fra tutti quelli presenti in D. Si noti che è sufficiente utilizzare una sola riga, ma il codice risultante non sarebbe altrettanto chiaro.

```
int stringMatchingSingleRow(item[] P, item[] T, int m, int n)
    int[] PD = new [0...n]
    int[] D = new [0...n]
    for j = 0 to n do D[j] = 0
    for i = 1 to m do
        PD ↔ D
        int D[0] = i                                  % Valore a destra, colonna 0
        for j = 1 to n do
            int D[j] = min(PD[j-1] + iif(P[i] = T[j],0,1), PD[j]+1, D[j-1]+1)
    return min(D)
```

1.5 Cammini minimi tra tutte le coppie

Nel capitolo Capitolo 11 sono stati trattati diversi algoritmi per trovare i cammini minimi tra un dato nodo r e tutti i rimanenti nodi. Per trovare i cammini minimi tra tutte le coppie di nodi, è possibile applicare n volte uno degli algoritmi visti, considerando a turno come nodo "radice" r ciascuno degli n nodi del grafo. Se non ci sono archi con lunghezze negative, tale procedimento è $O(nm\log n)$, utilizzando l'algoritmo di Johnson, e $O(n^3)$ utilizzando quello di Dijkstra. Se ci sono lunghezze negative, invece, la complessità è $O(n^4)$, poiché si deve usare n volte l'algoritmo di Bellman, Ford e Moore.

Vediamo come la programmazione dinamica permetta di progettare un algoritmo di complessità inferiore ad $O(n^4)$ anche per grafi con lunghezze negative degli archi.

Dato G, un *cammino k-vincolato da u a v* è un cammino semplice da u a v che non attraversa i nodi $\{k,k+1,k+2,\ldots,n\}-\{u,v\}$.

Per i cammini minimi k-vincolati, vale la seguente proprietà di sottostruttura ottima: ogni sottocammino c_{ij} di un cammino minimo k-vincolato c_{uv} in G è esso stesso un cammino minimo k-vincolato in G. Per provarlo, scomponiamo il cammino c_{uv} in tre parti:

$$u \xrightarrow{c_{ui}} i \xrightarrow{c_{ij}} j \xrightarrow{c_{jv}} v$$

Il costo del cammino c_{uv} può quindi essere espresso come: $w(c_{uv}) = w(c_{ui}) + w(c_{ij}) + w(c_{jv})$. Ovviamente c_{ij} è k-vincolato. Supponiamo ora per assurdo che esista un cammino k-vincolato c'_{ij} con costo inferiore, ovvero $w(c'_{ij}) < w(c_{ij})$. Possiamo quindi costruire un cammino k-vincolato c'_{uv}, sostituendo c_{ij} con c'_{ij}, il cui costo è

$$w(c'_{uv}) = w(c_{ui}) + w(c'_{ij}) + w(c_{jv}) < w(c_{ui}) + w(c_{ij}) + w(c_{jv}) = w(c_{uv})$$

Ma questo è un assurdo, perché c_{uv} è un cammino minimo k-vincolato.

Sia d^k_{uv} il costo di un cammino minimo k-vincolato fra u e v. Per la proprietà di sottostruttura ottima, è facile osservare che per un cammino minimo $(k+1)$-vincolato sono date due possibilità: o non passa nemmeno per il nodo k, e quindi il suo costo è anche quello di un cammino minimo k-vincolato, oppure include anche il nodo k, e allora è uguale al costo del cammino minimo k-vincolato fra u e k più il costo del cammino minimo k-vincolato fra k e v:

$$d^{k+1}_{uv} = \min\{d^k_{uv}, d^k_{uk} + d^k_{kv}\}, \qquad 1 \le u \le n, 1 \le v \le n,$$

con condizioni iniziali:

$$d^1_{uv} = \begin{cases} w(u,v) & \text{se } (u,v) \in E \\ 0 & \text{se } u = v \\ +\infty & \text{altrimenti.} \end{cases}$$

Per ricavare, oltre alla lunghezza d^k_{uv}, anche il cammino minimo, si definisce in modo analogo p^k_{uv} come il predecessore del nodo v in un cammino minimo k-vincolato da u a v. Per $1 \le k \le n$, si ottiene:

$$p^{k+1}_{uv} = \begin{cases} p^k_{uv} & \text{se } d^{k+1}_{uv} = d^k_{uv} \\ p^k_{kv} & \text{se } d^{k+1}_{uv} = d^k_{uk} + d^k_{kv} \end{cases}$$

con condizioni iniziali:

$$p^1_{uv} = \begin{cases} u & \text{se } (u,v) \in E \\ 0 & \text{altrimenti} \end{cases}$$

È facile convincersi che d_{uv}^{n+1} è la lunghezza di un cammino minimo dal nodo u al nodo v nel grafo G, mentre p_{uv}^{n+1} è il predecessore del nodo v in un cammino siffatto. Si noti che se il nodo v non è raggiungibile dal nodo u, allora $d_{uv}^{n+1} = +\infty$, mentre se esiste un ciclo negativo che passa per il nodo u, allora $d_{uu}^{n+1} < 0$.

La computazione di d_{uv}^{n+1} e p_{uv}^{n+1}, per $1 \leq u \leq n$, $1 \leq v \leq n$, può essere effettuata con la procedura floydWarshall(), proposta da Floyd (1962) basandosi su risultati precedenti di Warshall. La procedura usa iterativamente la riga k e la colonna k delle matrici d e p per aggiornare *in loco* i rimanenti elementi delle matrici stesse; in altre parole, come suggerito dalla sezione precedente, evita di memorizzare inutilmente k matrici. Il tempo richiesto della procedura è $\Theta(n^3)$. La procedura utilizza soltanto $O(n^2)$ spazio di memoria, in quanto i valori d_{uv}^k e p_{uv}^k, per ogni iterazione k, sono sempre riscritti nelle matrici d e p stesse. Si noti che l'inizializzazione della matrice d non è necessaria se il grafo è realizzato con matrice d'adiacenza rappresentata dalle lunghezze degli archi.

```
floydWarshall(GRAPH G, int[][] d, int[][] p)
foreach u,v ∈ G.V() do
    d[u,v] = +∞
    p[u,v] = 0
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        d[u,v] = w(u,v)
        p[u,v] = u
for k = 1 to G.n do
    foreach u ∈ G.V() do
        foreach v ∈ G.V() do
            if d[u,k] + d[k,v] < d[u,v] then
                d[u,v] = d[u,k] + d[k,v]
                p[u,v] = p[k,v]
```

Esempio (Algoritmo di Floyd-Warshall) La Figura 1.3 mostra l'esecuzione dell'algoritmo di Floyd-Warshall su un grafo G avente 4 nodi e 6 archi. Per ogni iterazione k, sono evidenziate in grassetto la k-esima riga e la k-esima colonna delle matrici d e p. Gli elementi delle matrici all'iterazione $(k+1)$-esima sono ottenuti applicando la cosiddetta regola del rettangolo rispetto alla la k-esima riga e alla k-esima colonna. Per esempio, per $k+1 = 3$, l'elemento d_{14}^3 (Figura 1.3c) è ottenuto da $\min\{d_{14}^2, d_{12}^2 + d_{24}^2\} = \min\{+\infty, 5+4\} = 9$ (Figura 1.3b), mentre p_{14}^3 è ottenuto da $p_{24}^2 = 2$. L'u-esima riga della matrice p finale contiene l'albero dei cammini minimi di radice u, rappresentato come vettore dei padri, dove un nodo non raggiungibile da u ha padre uguale a 0 e distanza uguale ad $+\infty$. ■

1.6 Memoization

Nelle sezioni precedenti, abbiamo visto che non è sempre necessario memorizzare le soluzioni di tutti i sottoproblemi. Una domanda correlata è la seguente: è necessario calcolare le soluzioni di tutti i sottoproblemi? Anche in questo caso la risposta può essere negativa.

Si consideri il seguente problema: state svaligiando una casa piena di beni preziosi; volete sottrarre oggetti del maggior valore possibile, ma siete limitati dal fatto che lo zaino che avete

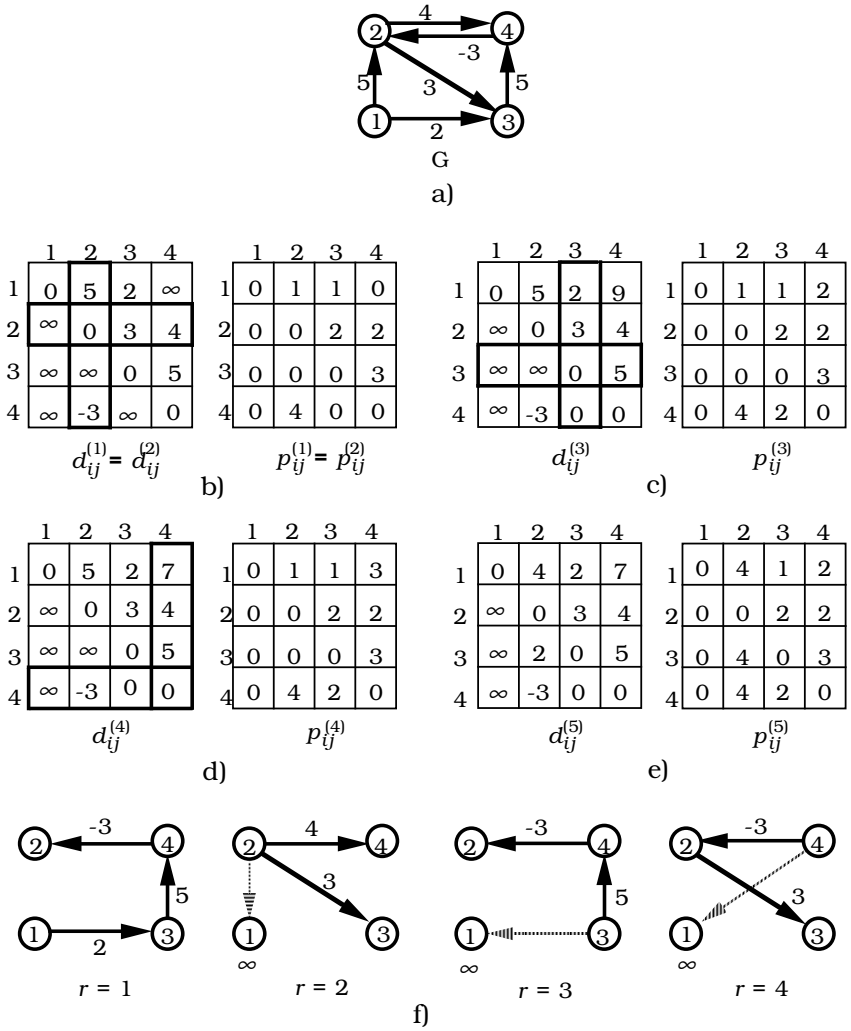

Figura 1.3: Algoritmo di Floyd-Warshall. a) Un grafo G di 4 nodi; b)-e) matrici d e p per $k = 1,2,3,4,5$; f) alberi dei cammini minimi per radice $r = 1,2,3,4$. **TODO**: Nelle tabelle togliere le parentesi dagli apici di d e p; esempio $d_{ij}^{(1)} \rightarrow d_{ij}^{1}$

portato con voi ha una certa capacità. Quali oggetti scegliete? Più formalmente, dovete risolvere il seguente problema:

ZAINO (KNAPSACK). *Dati un intero positivo C (la capacità dello zaino) e un insieme di n "oggetti", tali che l'oggetto i-esimo è caratterizzato da un "profitto" p_i e da un "volume" v_i, entrambi interi positivi, trovare un sottoinsieme S di $\{1,\ldots,n\}$ tale che il volume totale $v(S) = \sum_{i\in S} v_i \leq C$ e il profitto totale $p(S) = \sum_{i\in S} p_i$ sia massimo.*

È facile dimostrare che questo problema gode di sottostruttura ottima. Definiamo con $P(i,c)$ il sottoproblema dato dai primi i oggetti da inserire in uno zaino di capacità c. Il problema originale corrisponde a $P(n,C)$. Sia $S(i,c)$ una soluzione ottima per $P(i,c)$; possono darsi due casi:

– Se $i \in S(i,c)$, allora $S(i,c) - \{i\}$ è una soluzione ottima per il sottoproblema $P(i-1,c-v_i)$. Supponiamo per assurdo che ciò non sia vero. Allora è possibile trovare una soluzione S' per $P(i-1,c-v_i)$ tale per cui $p(S') > p(S(i,c) - \{i\})$ e $V(S') < c - v_i$. Se così fosse, potremmo ottenere una soluzione $S' \cup \{i\}$ al problema $P(i,c)$ tale che $p(S' \cup \{i\}) =$

$p(S') + p_i > p(S(i,c) - \{i\}) + p_i = p(S(i,c))$ e $v(S' \cup \{i\}) \leq c - v_i + v_i = c$; ma questa è una contraddizione, visto che $S(i,c)$ è ottima.

– Se $i \notin S(i,c)$, allora $S(i,c)$ è una soluzione ottima per $P(i-1,c)$. Come sopra, assumiamo per assurdo che ciò non sia vero; allora esiste un insieme S' soluzione ottima di $P(i,c)$ tale che $p(S') > p(S(i,c))$ e $v(S') \leq c$, contraddicendo l'ipotesi di ottimalità di $S(i,c)$.

Detto $D[i,c]$ il profitto massimo ottenuto per il problema $P(i,c)$, la proprietà di sottostruttura ottima ci suggerisce la seguente definizione ricorsiva:

$$D[i,c] = \begin{cases} -\infty & \text{se } c < 0 \\ 0 & \text{se } i = 0 \vee c = 0 \\ \max\{D[i-1,c], D[i-1,c-v_i] + p[i]\} & \text{altrimenti} \end{cases}$$

Possiamo facilmente ottenere una procedura basata sulla programmazione dinamica con costo $O(nC)$. Questo problema, tuttavia, differisce da quelli precedenti in quanto non è detto che sia necessario risolvere tutti i sottoproblemi. Dipende in realtà dai particolari valori dei volumi e dalla capacità.

L'approccio iterativo, "dal basso verso l'alto" della programmazione dinamica risolve obbligatoriamente tutti i problemi. Esiste un approccio alternativo alla programmazione dinamica, ricorsivo e "dall'alto verso il basso", che prende il nome di *memoization* (o *annotazione*). L'idea è scrivere una procedura che utilizza la tabella come una cache per memorizzare le soluzioni dei problemi, controllando ogni volta se un problema è stato risolto; se sì, si riutilizza il risultato; se no, lo risolve (chiamando ricorsivamente la procedura sui sottoproblemi), ne memorizza il risultato nella tabella e lo restituisce al chiamante. La procedura zaino() risolve il problema dello zaino utilizzando *memoization*; prende in input due vettori p e v contenenti i profitti e i volumi, un indice i e la capacità residua c. Inoltre, prende in input anche un puntatore alla tabella D. La prima invocazione della procedura prende la forma $\mathsf{zaino}(p,v,n,C,D)$, con tutte le posizioni della matrice D inizializzate a $\perp$, un valore speciale che indica che il sottoproblema non è stato ancora risolto.

```
int zaino(int[] p, int[] v, int i, int c, int[][] D)
if c < 0 then
    return −∞
if i = 0 or c = 0 then
    return 0
if D[i,c] = ⊥ then
    D[i,c] = max(zaino(p,v,i−1,c,D), zaino(p,v,i−1,c−v[i],D) + p[i])
return D[i,c]
```

Il codice risultante è molto elegante e simile alla definizione ricorsiva; l'unica differenza è la presenza del controllo sul valore $\perp$, per evitare di risolvere sottoproblemi già risolti. Vi sono tuttavia alcune considerazioni da fare per valutare l'efficacia della tecnica di *memoization*.

La complessità della procedura nel caso pessimo è comunque $O(nC)$; ma a seconda di volumi e capacità, il numero di sottoproblemi da risolvere potrebbe essere molto inferiore, anche di diversi ordini di grandezza.

Una possibile obiezione riguarda l'inizializzazione della tabella D, che non è inclusa nella procedura ma costa $O(nC)$ in ogni caso. Se il costo di ricombinare i risultati per riempire una posizione non è costante, il costo dell'inizializzazione è asintoticamente inferiore al costo per riempire la tabella e l'obiezione non si pone. Se questo non è il caso (come nel problema dello zaino), si ha una tipica situazione in cui può essere indicato usare una tabella hash: un universo delle possibili chiavi molto grande – $O(nC)$ – e un numero basso di chiavi attese.

Infine, una considerazione sulla complessità: $O(nC)$ non è polinomiale. Infatti, C non è la dimensione dell'input, è un parametro che fa parte dell'input. La dimensione effettiva di C (in numero di bit) è $d = \lceil \log C \rceil$, il che significa che il costo è esponenziale in d. Tuttavia, se $C = O(n^k)$ per qualche k, la soluzione proposta è praticabile. Algoritmi di questo tipo vengono detti *pseudopolinomiali* [cfr. Capitolo 19].

1.7 Reality check

Consideriamo un'ulteriore variante dello *string matching*:

SOTTOSEQUENZA COMUNE MASSIMALE (LONGEST COMMON SUBSEQUENCE). *Data una sequenza X, è possibile ottenere una sottosequenza da X rimuovendone uno o più caratteri (non usiamo il termine sottostringa perché normalmente associato all'idea di una sottosequenza di caratteri contigui). Date due sequenze P e T, X è una sottosequenza comune se è sottosequenza sia di P che di T. Una sequenza X è una sottosequenza comune massimale (in inglese, longest common subsequence o LCS) di P e T se non esistono sottosequenze comuni a P e T più lunghe di X. Il problema consiste nel calcolare la lunghezza delle LCS ed eventualmente produrne una.*

La lunghezza della LCS può essere utilizzata come una misura della similitudine fra due sequenze.

Il problema può essere risolto in modo molto simile al problema della distanza di *editing*. Sia $D[i,j]$ la lunghezza della LCS di $P(i)$ e $T(j)$. Ovviamente, se $i = 0$ oppure $j = 0$, tale valore è pari a 0 (non esiste infatti alcun carattere in comune con la sequenza vuota). Altrimenti, possono darsi due casi:

(1) se $p_i = t_j$, la lunghezza della LCS di $P(i)$ e $T(j)$ è pari alla lunghezza della LCS di $P(i-1)$ e $T(j-1)$ più 1. Questo perché l'ultimo carattere fa sicuramente parte della LCS di $P(i)$ e $T(j)$.

(2) se $p_i \neq t_j$, vi sono due sottocasi: o l'ultimo carattere di $P(i)$ non fa parte della LCS, oppure l'ultimo carattere di $T(j)$ non ne fa parte. Nel primo caso, possiamo ricondurci a calcolare la LCS di $P(i-1)$ e $T(j)$; nel secondo caso, possiamo ricondurci a calcolare la LCS di $P(i)$ e $T(j-1)$.

Queste considerazioni possono essere riassunte dalle seguenti relazioni di ricorrenza:

$$D[i,j] = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ D[i-1,j-1]+1 & \text{se } i > 0 \wedge j > 0 \wedge p_i = t_j \\ \max\{D[i-1,j], D[i,j-1]\} & \text{se } i > 0 \wedge j > 0 \wedge p_i \neq t_j \end{cases}$$

La prima invocazione della procedura prende la forma $\mathsf{lcs}(D,P,T,m,n)$, con tutte le posizioni della matrice D inizializzate a **nil**, un valore speciale che indica che il sottoproblema non è stato ancora risolto. Una versione basata su *memoization* è la seguente:

```
int lcs(int[][] D, item[] P, item[] T, int i, int j)
    if i = 0 or j = 0 then
        return 0
    if D[i, j] = ⊥ then
        if P[i] = T[j] then
            D[i, j] = lcs(D, P, T, i − 1, j − 1) + 1
        else
            D[i, j] = max(lcs(D, P, T, i − 1, j), lcs(D, P, T, i, j − 1))
    return D[i, j]
```

In molti testi di algoritmi si menziona il fatto che questo è un tipico problema di bio-informatica, applicato a sequenze di basi del DNA tratte dall'alfabeto $\{\mathtt{A},\mathtt{T},\mathtt{G},\mathtt{C}\}$ – adenina, timina, guanina e citosina. Ma esiste un'applicazione informatica molto più comune e vicina all'uso quotidiano: il comando Unix `diff`, che esamina due file di testo, e ne evidenzia le differenze a livello di riga. Si noti che lavorare a livello di riga significa che i confronti fra caratteri sono in realtà fra righe, e che n ed m sono il numero di righe dei due file. L'output di `diff` può essere generato in tempo lineare a partire dalla tabella D, come richiesto nell'Esercizio 1.10.

Quando è applicato a due file di testo lunghi migliaia di righe, la complessità $O(mn)$ potrebbe essere troppo elevata, soprattutto per l'occupazione di memoria. È tuttavia possibile applicare numerose ottimizzazioni:

- `diff` è utilizzato per sorgenti di codice, in cui avvengono pochi cambiamenti per volta, raramente all'inizio del file, e molto spesso nemmeno alla fine del file. Conviene quindi scartare dal confronto tutte le prime s righe uguali all'inizio del file, e tutte le t righe uguali al termine; la complessità diventerebbe quindi $O(s+t+(n-s-t)\cdot(m-s-t))$.
- Confrontare righe è costoso; potrebbe essere utile calcolare, per ogni riga, un valore hash e confrontare questi ultimi. Se due valori hash sono diversi, siamo sicuri che le righe sono differenti; se sono uguali, dobbiamo spendere un po' di tempo in più per controllare che le righe siano effettivamente uguali e non abbiano dato origine ad una collisione.
- Utilizzando una tabella hash, possiamo anche scoprire che alcune righe compaiono in un solo file. È possibile rimuovere queste righe dal nostro problema, ricordando di aggiungerle all'output come aggiunte o rimozioni.

1.8 Esercizi

Esercizio 1.1 (k-occorrenza)**.** Data la tabella D calcolata da stringMatching(), scrivere una procedura che stampa l'occorrenza k-approssimata di P che compare in T e termina in t_j.

Esercizio 1.2 (Massima sottostringa comune)**.** Date due stringhe $P = p_1p_2\cdots p_m$ e $T = t_1t_2\cdots t_n$, si progetti un algoritmo di programmazione dinamica per individuare la più lunga sottostringa (sottosequenza contigua) comune tra P e T.

Esercizio 1.3 (Algoritmo di Bellman, Ford e Moore)**.** Si consideri il problema dei cammini minimi da un singolo nodo r a tutti i rimanenti $n-1$ nodi in un grafo orientato pesato (con eventuali lunghezze negative degli archi). Si definisca d_v^k come la lunghezza di un cammino minimo dal nodo r al nodo v che contiene al più k archi. Si dimostri che l'algoritmo di Bellman-Ford-Moore [cfr. Capitolo 11] equivale a risolvere le seguenti relazioni di programmazione dinamica:

$$d_v^{k+1} = \min\{d_v^k, \min_{h\neq v}\{d_h^k + w(h,v)\}\}.$$

Qual è la complessità dell'algoritmo nella versione di programmazione dinamica?

Esercizio 1.4 (Cammini minimi su grafi aciclici)**.** Si consideri il problema dei cammini minimi da un singolo nodo r a tutti i rimanenti nodi in un grafo (con eventuali lunghezze negative degli archi) nel quale esiste l'arco (u,v) solo se $u < v$. Si dimostri che le relazioni di programmazione dinamica dell'Esercizio 1.3 diventano semplicemente

$$d_v = \min_{h<v}\{d_h + w(h,v)\},$$

dove d_v è la lunghezza di un cammino minimo dal nodo r al nodo v. Qual è la complessità dell'algoritmo?

Esercizio 1.5 (Cammini minimi dinamici)**.** Un algoritmo dinamico è un algoritmo che è in grado di aggiornare il risultato di una funzione a fronte di modifiche dei dati di ingresso senza dover ricalcolare tutto da capo. Sia $G = (V,E)$ un grafo orientato con funzione costo $w : V \times V \to \mathbf{R}$. Progettare un algoritmo dinamico che è in grado di aggiornare le distanze fra tutte le coppie di vertici del grafo a fronte dell'inserimento di un nuovo arco nel grafo. Se l'inserimento dell'arco introduce un ciclo negativo nel grafo, l'algoritmo deve essere in grado di lanciare un'eccezione. Quali difficoltà sorgerebbero se volessimo cancellare archi invece di inserirli?

Esercizio 1.6 (Massima sottosequenza crescente)**.** Sia data una sequenza L di n numeri interi distinti. Si scriva una procedura efficiente basata sulla programmazione dinamica per trovare la più lunga sottosequenza crescente di L (per esempio, se $L = 9,15,3,6,4,2,5,10,3$, allora la più lunga sottosequenza crescente è: $3,4,5,10$).

Esercizio 1.7 (Allineamento globale)**.** Date due stringhe $P = p_1 p_2 \cdots p_m$ e $T = t_1 t_2 \cdots t_n$, all'incirca della stessa lunghezza, un *allineamento globale* consiste nell'inserzione di spazi in P o in T in modo da ottenere la stessa lunghezza, col vincolo che uno spazio in P non possa mai essere allineato a uno spazio in T. Ad un allineamento globale si associa un punteggio ottenuto valutando $+1$ se $p_i = t_j$, valutando -1 se $p_i \neq t_j$, e -2 quando p_i oppure t_j è uno spazio. Per esempio, se P = `GACGGATTAG` e T = `GATCGGAATAG`, l'allineamento globale tra `GA␣CGGAT␣TAG` e `GAT␣CGGAA␣TAG` ha un punteggio di $9-1-2=6$. Si progetti un algoritmo di programmazione dinamica per trovare l'allineamento globale di punteggio massimo.

Esercizio 1.8 (Allineamento semiglobale)**.** Il problema dell'allineamento semiglobale è simile a quello dell'allineamento globale dell'Esercizio 1.7, con l'unica differenza che non si penalizzano gli eventuali spazi inseriti prima dell'inizio e/o dopo la fine di una stringa. Si indichi come modificare le relazioni di programmazione dinamica in modo da risolvere il problema.

Esercizio 1.9 (Allineamento locale)**.** Il problema dell'allineamento locale è anch'esso una variante dell'allineamento globale dell'Esercizio 1.7, dove però si vuole trovare l'allineamento di punteggio massimo tra *sottostringhe* di P e T. Si mostri come modificare le relazioni di programmazione dinamica in modo da risolvere il problema.

Esercizio 1.10 (`diff`)**.** Scrivere una procedura printdiff() che, data la tabella D calcolata dalla procedura lcs(), produca in output il file delle differenze come illustrato in Figura 1.4.

```
Questo è il testo originale
alcune linee non dovrebbero
cambiare mai
altre invece vengono
rimosse
altre vengono aggiunte
```

```
Questo è il testo nuovo
alcune linee non dovrebbero
cambiare mai
altre invece vengono
cancellate
altre vengono aggiunte
come questa
```

```
- Questo è il testo originale
+ Questo è il testo nuovo
  alcune linee non dovrebbero
  cambiare mai
  altre invece vengono
- rimosse
+ cancellate
  altre vengono aggiunte
+ come questa
```

Figura 1.4: Il file `original.txt` (a sinistra); il file `new.txt` (al centro); l'output di `diff original.txt new.txt` (a destra).

Esercizio 1.11 (Harry Potter)**.** Al celebre maghetto Harry Potter è stata regalata una scopa volante modello Nimbus3000 e tutti i suoi compagni del Grifondoro gli chiedono di poterla provare. Il buon Harry ha promesso che nei giorni a venire soddisferà le richieste di tutti, ma ogni ragazzo è impaziente e vuole provare la scopa il giorno stesso. Ognuno propone ad Harry un intervallo di tempo della giornata durante il quale, essendo libero da lezioni di magia, può fare un giro sulla scopa, e per convincerlo gli offre una certa quantità di caramelle Tuttigusti+1. Allora Harry decide di soddisfare, tra tutte le richieste dei ragazzi, quelle che gli procureranno la massima quantità di

caramelle (che poi spartirà coi suoi amici Ron e Hermione). Aiutalo a trovare la migliore soluzione possibile.

Esercizio 1.12 (Massima somma di sottovettori)**.** Riproponiamo l'Esercizio 12.2: Progettare un algoritmo che, preso un vettore A di n interi (positivi o negativi), trovi un sottovettore (una sequenza di elementi consecutivi del vettore) la somma dei cui elementi sia massima.

Esercizio 1.13 (Ma è italiano?)**.** Vi viene data una stringa di caratteri $s[1...n]$; avete il dubbio che sia una stringa di testo italiano, in cui sono stati tolti tutti gli spazi e i segni di interpunzione. Ad esempio, "eraunanottebuiaetempestosa". Avete a disposizione una struttura dati dizionario h che contiene... un dizionario di italiano! $h.\mathsf{lookup}(w)$ ritorna vero se la parola w è una parola italiana. Progettare un algoritmo che stabilisca se il testo che riceve in input è un test in italiano e calcolarne la complessità, misurata come numero di chiamate a $h.\mathsf{lookup}()$.

1.9 Soluzioni

Soluzione Esercizio 1.2

Si definisce una matrice $D[i,j]$, con $0 \leq i \leq m$ e $0 \leq j \leq n$, come segue: $D[i,j]$ è il numero dei caratteri della sottostringa comune al prefisso i-esimo di P e al prefisso j-esimo di T.

Chiaramente, $D[0,j] = 0$, per $0 \leq j \leq n$, poiché il prefisso 0-esimo di T è vuoto e non ci sono caratteri in comune; anche $D[i,0] = 0$, per $0 \leq i \leq m$, perché il prefisso 0-esimo di P è vuoto. Valgono le seguenti relazioni di programmazione dinamica:

$$D[i,j] = \begin{cases} 0 & \text{se } p_i \neq t_j \\ D[i-1,j-1]+1 & \text{se } p_i = t_j \end{cases}$$

Il massimo elemento $D[i,j]$ della matrice così ottenuta individua la sottostringa più lunga comune a P e T. L'algoritmo ha complessità $\Theta(mn)$.

Soluzione Esercizio 1.7

Assumendo che P e T inizino con un carattere "vuoto" λ, che si trova in posizione 0, si definisce una matrice D tale che $D[i,j]$ è il punteggio del migliore allineamento globale tra $p_0 p_1 \cdots p_i$ e $t_0 t_1 \cdots t_j$. Valgono le seguenti relazioni di programmazione dinamica:

$$D[i,j] = \begin{cases} -2i & \text{se } j = 0 \\ -2j & \text{se } i = 0 \\ \max\{D[i,j-1]-2, D[i-1,j]-2, D[i-1,j-1]+\delta_{ij}\} & \text{altrimenti} \end{cases}$$

dove $\delta_{ij} = 1$ se $p_i = t_j$, e $\delta_{ij} = -1$ se $p_i \neq t_j$. Il valore $D[m,n]$ individua il punteggio dell'allineamento globale ottimo tra P e T e può essere calcolato in tempo $\Theta(mn)$ con una funzione la cui struttura è analoga a quella vista per lo *string matching*. Ovviamente, si può agevolmente modificare l'algoritmo in modo da ricavare, oltre al punteggio dell'allineamento ottimo, anche l'allineamento stesso. L'allineamento globale è un problema importante nella bioinformatica.

Soluzione Esercizio 1.8

Le relazioni di programmazione dinamica sono analoghe a quelle viste nella soluzione dell'Esercizio 1.7, con le seguenti differenze. Per trascurare gli spazi inseriti prima dell'inizio di P basta inizializzare $D[0,j] = 0$ per $0 \leq j \leq n$, mentre per trascurare quelli prima di T si inizializza $D[i,0] = 0$ per $0 \leq i \leq m$. Se invece non si vogliono penalizzare gli spazi inseriti dopo la fine di P occorre produrre come risultato il valore massimo della riga m di D, mentre per non penalizzare gli spazi dopo T si prende come risultato il massimo della colonna n di D.

Soluzione Esercizio 1.9

Le relazioni di programmazione dinamica sono simili a quelle viste nella soluzione dell'Esercizio 1.7, ma l'interpretazione del vettore D è diversa. Nell'allineamento locale, $D[i,j]$ dà il punteggio del migliore allineamento tra un suffisso di $p_1 \cdots p_i$ ed un suffisso di $t_1 \cdots t_j$, dove tali suffissi possono essere anche vuoti. Per questo, l'inizializzazione prevede $D[i,0] = 0$ per $0 \leq i \leq m$ e $D[0,j] = 0$ per $0 \leq j \leq n$. Siccome i punteggi devono tenere conto anche dell'allineamento tra suffissi vuoti, che ha punteggio nullo, la relazione di programmazione dinamica diventa:

$$D[i,j] = \max\{D[i,j-1]-2, D[i-1,j]-2, D[i-1,j-1]+\delta_{ij}, 0\}$$

dove δ_{ij} ha lo stesso significato dell'Esercizio 1.7. Si noti che adesso ogni $D[i,j]$ è non negativa. Al termine, il risultato del problema si trova nella casella $D[i,j]$ che contiene il massimo tra tutte le caselle di D.

Soluzione Esercizio 1.10

La procedura printdiff() prende in input la tabella D, le due stringhe P e T e gli indici i, j della posizione sotto esame. La procedura viene chiamata con $i = m$ e $j = n$. Se i caratteri corrispondenti sono uguali, si chiama ricorsivamente printdiff() sulla posizione $i-1, j-1$, corrispondente all'unico sottoproblema risolto ricorsivamente da lcs(). Se sono diverse, la procedura lcs() ha scelto il massimo fra i due sottoproblemi $i, j-1$ e $i-1, j$; si verifica quindi quale sia il valore massimo per muoversi di conseguenza. Se sono uguali, viene arbitrariamente scelto il primo.

```
printdiff(int[][] D, item[] P, item[] T, int i, int j)
  if i > 0 and j > 0 and P[i] = T[j] then
      printdiff(D,P,T,i-1,j-1)
      print " " P[i]
  else if j > 0 and (i = 0 or D[i,j-1] ≥ D[i-1,j]) then
      printdiff(D,P,T,i,j-1)
      print "–" T[j]
  else if i > 0 and (j = 0 or D[i,j-1] < D[i-1,j]) then
      printdiff(D,P,T,i-1,j)
      print "+" P[i]
```

Soluzione Esercizio 1.12

Come accennato nella soluzione dell'esercizio 12.2, è possibile risolvere questo problema tramite programmazione dinamica. Sia t_i il valore del massimo sottovettore che termina in i. Se conosco t_{i-1}, posso calcolare t_i osservando due casi: se $A[i] + t_{i-1}$ ha un valore positivo, questo è il valore del massimo sottovettore che termina in i; se invece ha valore negativo, il massimo sottovettore che termina in i ha valore 0.

$$t_i = max(t_{i-1} + A[i], 0)$$

Nel codice seguente, questo viene calcolato passo passo dalla variabile *here*, che assume il valore $\max(here + A[i], 0)$. Poiché il sottovettore di valore massimo deve terminare in qualche posizione, calcolando nella variabile m il valore massimo fra tutti i valori *here*, si ottiene il valore massimo per l'intero vettore. Questo algoritmo ha costo $O(n)$.

```
int maxsum(int[] A, int n)
    int max = 0                                   % Massimo valore trovato
    int here = 0                                  % Massimo valore che termina nella posizione attuale
    for i = 1 to n do
        here = max(here + A[i], 0)
        max = max(here, max)
    return max
```