TECNICA *PROGRAMMAZIONE DINAMICA*

1. Introduzione

Come il metodo divide et impera, la programmazione dinamica (DP) risolve problemi mettendo insieme le soluzioni di un certo numero di sottoproblemi. A differenza del dividi et impera, i sottoproblemi da risolvere non sono necessariamente *disgiunti* (vale a dire: uno stesso sottoproblema può essere comune a diversi sottoproblemi). Onde evitare di ricalcolare più volte la soluzione di uno stesso sottoproblema, i sottoproblemi vengono risolti con una strategia *bottom-up* (vale a dire: dal sottoproblema più "piccolo" al sottoproblema più "grande") e le soluzioni a questi sottoproblemi vengono memorizzate in apposite tabelle di modo che siano disponibili se necessari alla soluzione di altri sottoproblemi.
Un buon esempio di come trascurare il fenomeno dell' "overlapping" tra sottoproblemi possa avere un grosso impatto sulla complessità temporale di una procedura è il seguente:

1.1. **Numeri di Fibonacci.**

I numeri di Fibonacci sono definiti dalla seguente ricorrenza:

$$\begin{array}{rcl} F_0 & = & 1 \\ F_1 & = & 1 \\ F_i & = & F_{i-1} + F_{i-2} \quad \text{per } i \geq 2 \end{array}$$

Perciò ogni numero di Fibonacci è la somma dei due precedenti e si ottinene la sequenza $1, 1, 2, 3, 5, 8, 13, \ldots$.

Consideriamo ora il seguente problema:

Dato un naturale n, calcolare il numero di Fibonacci F_n.

Utilizzando direttamente la formula ricorsiva $F_n = F_{n-1} + F_{n-2}$ otteniamo il seguente programma:

FIBONACCI (n)
  IF $n \leq 1$ THEN RETURN 1
  ELSE RETURN FIBONACCI$(n-1)$ + FIBONACCI$(n-2)$

La ricorrenza $T(n)$ per il tempo di esecuzione di FIBONACCI è

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & \text{altrimenti.} \end{cases}$$

sfruttando il fatto che $T(n-1) > T(n-2)$ per $n \geq 2$, possiamo ottenere un limite inferiore alla complessità di FIBONACCI studiando la ricorrenza

$$T'(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2 \cdot T(n-2) + \Theta(1) & \text{altrimenti.} \end{cases}$$

il cui albero di ricorsione è un albero binario completo di altezza $\lfloor \frac{n}{2} \rfloor$, un albero quindi con $2^{\lfloor \frac{n}{2} \rfloor + 1} - 1$ nodi. La complessità di tempo di FIBONACCI è quindi $2^{\Omega(n)}$. Come si spiega questa complessità esponenziale?

Con l'utilizzo diretto della ricorrenza $F_n = F_{n-1} + F_{n-2}$ pur espirmendo il problema del calcolo di F_n in termini di due sottoproblemi più piccoli (F_{n-1} e F_{n-2}) avremo che degli stessi valori verranno ricalcolati più volte (i due sottoproblemi non sono disgiunti). Ad esempio in figura 1 è riportato l'albero di ricorsione che viene fuori dal calcolo di F_5 e tutti i numeri di Fibonacci inferiori a 4 sono ricalcolati più volte.

Tipico della PD è risolvere le formule ricorsive bottom-up. Nel nostro caso significa calcolare i valori F_i dal più piccolo al più grande registrando tutti i risultati. Così facendo, quando bisognerà calcolare F_{i+1} si hanno già pronti i valori di F_i e di F_{i-1}. Ecco lo pseudocodice del nuovo algoritmo:

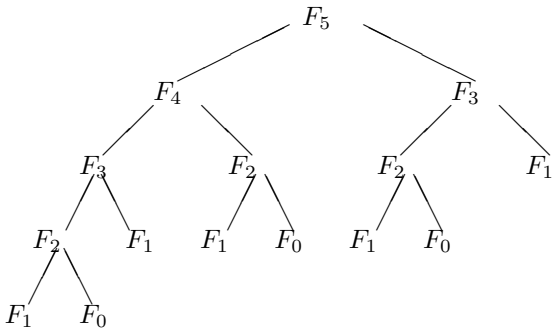

FIGURE 1. Albero di ricorsione per il calcolo di F_5.

```
FIBONACCI1
INPUT un numero naturale n
    F[0] ← 1
    F[1] ← 1
    FOR i ← 2 TO n DO F[i] ← F[i − 1] + F[i − 2]
OUTPUT F[n]
```

Ognuno degli n valori viene calcolato come semplice somma di due interi e l'algoritmo ha complessità $O(n)$ che rappresenta un enorme miglioramento rispetto al tempo esponenziale richiesto dalla procedura FIBONACCI.

Questo primo esempio ha mostrato che, se la soluzione del problema generale si può scomporre nella composizione di sottoproblemi parziali non indipendenti, per poter ridurre i tempi di esecuzione si ha bisogno della tabulazione dei risultati parziali. Così facendo quando si presenta nuovamente il medesimo sottoproblema la sua soluzione viene trovata con un semplice accesso alla tabella (che richiede sempre un tempo costante).

Nel caso dell'algoritmo FIBONACCI1 la tabella è un vettore di n elementi, la procedura ha quindi una complessità di spazio $O(n)$. Possiamo diminuire lo spazio richiesto notando che, per il calcolo del nuovo valore $F[i]$, l'algoritmo necessita dei soli due valori $F[i-1]$ ed $F[i-2]$ ed è quindi inutile conservare in memoria l'intero vettore $F[]$. Da questa osservazione è facile ottenere il seguente algoritmo con complessità di tempo $O(n)$ e complessità di spazio $O(1)$.

```
FIBONACCI2
INPUT un numero naturale n
    a ← 1
    b ← 1
    FOR i ← 2 TO n DO
        c ← a + b
        a ← b
        b ← c
    ENDFOR
OUTPUT b
```

1.2. **Uno schema.** Il primo passo per lo sviluppo di un algoritmo di PD richiede l'individuazione di una collezione di sottoproblemi derivati dal problema originale dalle cui soluzioni può poi essere facilmente calcolata la soluzione del problema originario (ad esempio per i numeri di Fibonacci la soluzione al problema è data dalla soluzione dell'n-esimo degli n sottoproblemi $F_1, F_2, \ldots F_n$ individuati).
Poiché alla soluzione del problema originario si procede solo dopo aver risolto i vari sottoproblemi, la collezione non deve essere troppo ampia (essere costuita da un numero polinomiale di sottoproblemi è ovviamente ingrediente necessario per sperare in un algoritmo polinomiale).
Perchè un simile approccio abbia successo è necessario anche che il calcolo della soluzione di ciascuno dei sottoproblemi richieda un tempo polinomiale. In particolare tra i problemi della collezione deve esserci un ordinamento naturale dal più "piccolo" al più "grande" ed una formula ricorsiva che permetta di determinare la soluzione per un sottoproblema dalla soluzione di un certo numero di sottoproblemi più piccoli (nel caso dei numeri di Fibonacci la formula era ovviamente

$F_i = F_{i-1} + F_{i-2}$ e l'ordinamento quello sui numeri naturali). In particolare il concetto di problema più "piccolo" dipenderà dal tipo di formula ricorsiva che lega tra loro le soluzioni dei sottoproblemi.
In generale conviene individuare un insieme di sottoproblemi che appaiono naturali e poi cercare la ricorrenza che li lega. A parte casi banali come quello del calcolo dei numeri di Fibonacci (dove la formula ricorsiva è data già nella definizione del problema) non è sempre facile individuare quali sono i sottoproblemi da risolvere (vale a dire definire la collezione) Ed una qualunque collezione non avrà alcuna utilità finchè non si trova una formula ricorsiva che lega i sottoproblemi da cui è composta.
Per i problemi di ottimizzazione, nel processo appena descritto conviene limitarsi a trovare il valore della soluzione ottima piuttosto che la soluzione. Risulta poi facile estendere l'algoritmo in modo da tener straccia di una soluzione ottima oltre che del suo valore o, in alternativa, dopo che il valore ottimo è stato calcolato, ricostruire la soluzione ottima dai valori calcolati per i vari sottoproblemi.
Naturalmente quelle appena descritte sono solo linee guida. Le sessioni che seguono hanno lo scopo di proporre esempi concreti di applicazione di questa tecnica.

2. Il problema del SOTTOVETTORE DI VALORE MASSIMO.

Come primo esempio di applicazione della tecnica DP si vuole risolvere il seguente problema

> *Dato un vettore V di n interi determinare il valore massimo tra i valori dei suoi sottovettori (il valore di un sottovettore è dato dalla somma dei valori dei suoi elementi).*

Cominciamo col definire sottoproblemi dalla composizione delle cui soluzioni sarà poi possibile risolvere il problema di partenza.

> *Per $i \leq n$ sia $T[i]$ il valore massimo per i sottovettori che terminano con l'i-esimo elemento di V.*

Poichè il sottovettore a valore massimo del vettore deve terminare con uno dei valori di V, la soluzione al nostro problema è data ovviamente dal valore

$$\max_{1 \leq i \leq n} \{T[i]\}$$

Stabiliamo ora una regola ricorsiva che permetta di calcolare facilmente i diversi $T[i]$.
Per $i = 1$ ovviamente $T[i] = V[1]$. Per $i > 1$ notiamo che se il sottovettore di valore massimo che termina in i consta di un unico elemento allora il suo valore è $V[i]$. In caso contrario il suo valore deve essere dato da $T[i-1] + V[i]$ in quanto $T[i-1]$ è il massimo che può ottenersi da un sottovettore che termina in $i-1$ ed il sottovettore in questione è formato da un prefisso che termina in $i-1$ seguito dall'elemento in posizione i.
Per quanto detto si ha la seguente formula ricorsiva:

$$T[i] = \max\{V[i], T[i-1] + V[i]\}$$

Grazie all'equazione di ricorrenza appena descritta, otteniamo il seguente algoritmo

```
MAX VALORE DI SOTTOVETTORE
INPUT un vettore V[1,...,n] di n interi
    T[1] ← V[1]
    max ← T[1]
    FOR i ← 2 TO n DO
        T[i] ← V[i]
        IF T[i-1] > 0 THEN T[i] ← T[i] + T[i-1]
        IF max < T[i] THEN max ← T[i]
    ENDFOR
OUTPUT max
```

L'algoritmo riempe le varie caselle della tabella T e tiene traccia del massimo valore via via ottenuto. Riempire ciascuna delle n caselle della tabella richiede tempo costante e la complessità dell'algoritmo è quindi $O(n)$.

Per ottenere il sottovettore di valore massimo possiamo procedere come segue. Un sottovettore di V è univocamente individuato dalla coppia (a, b) con $1 \leq a \leq b \leq n$, dove a e b sono gli indici in V del primo e dell'ultimo valore del sottovettore, rispettivamente. Se siamo interessati a trovare il sottovettore di lunghezza massima basterà tener traccia dell'indice che ha causato l'ultima modifica della variabile max. Questo ci permetterà di conoscere dove termina il sottovettore (vale a dire il valore di b). Per calcolare il punto di inzio del sottovettore basterà, scorrere il vettore V verso sinistra a partire dalla

posizione b e ad ogni passo decrementare max del valore dell'elemento di V in quella posizione. Quando max assumerà valore 0 si avrà il punto di inizio del sottovettore. In pseudocodice:

```
MAX SOTTOVETTORE
INPUT un vettore V[1,...,n] di n interi
    T[1] ← V[1]
    max ← T[1]
    b ← 1
    FOR i ← 2 TO n DO
        T[i] ← V[i]
        IF T[i − 1] > 0 THEN T[i] ← T[i] + T[i − 1]
        IF max < T[i] THEN
            max ← T[i]
            b ← i
        ENDIF
    ENDFOR
    a ← b
    WHILE max − V[a] ≠ 0 DO
        max ← max − V[a]
        a ← a − 1
    ENDWHILE
OUTPUT (a, b)
```

Se siamo interessati ad ottimizzare anche la risorsa spazio notiamo che è inutile mantenere in memoria la tabella T di dimensione $O(n)$ in quanto per il calcolo del nuovo valore $T[i]$ l'algoritmo necessita del solo valore $T[i-1]$. Da questa osservazione si ottiene il seguente algoritmo che ha complessità di tempo $O(n)$ e complessità di spazio $O(1)$:

```
MAX SOTTOVETTORE
INPUT un vettore V[1,...,n] di n interi
    t ← V[1]
    max ← t
    b ← 1
    FOR i ← 2 TO n DO
        IF t > 0 THEN t ← t + V[i]
            ELSE t ← V[i]
        IF max < t THEN
            max ← t
            b ← i
        ENDIF
    ENDFOR
    a ← b
    WHILE max − V[a] ≠ 0 DO
        max ← max − V[a]
        a ← a − 1
    ENDWHILE
OUTPUT (a, b)
```

3. Il problema della SOTTOSEQUENZA COMUNE PIÙ LUNGA

Una *sottosequenza* di una certa sequenza è la sequenza originale con l'eventuale esclusione di alcuni elementi. La lunghezza di una sottosequenza è il numero di elementi che la compongono. Consideriamo quindi il seguente problema:

> *Date due sequenze* $X = (x_1, x_2, \ldots x_n)$ *e* $Y = (y_1, y_2, \ldots, y_m)$*, trovare la più lunga sottosequenza comune ad* X *e* Y*.*

Ad esempio, per le due sequenze di interi $X = (9, \underline{15}, 3, \underline{6}, 4, \underline{2}, 5, 10, \underline{3})$ e $Y = (8, \underline{15}, \underline{6}, 7, 9, \underline{2}, 11, \underline{3}, 1)$, la soluzione al problema è la sottosequenza $(15, 6, 2, 3)$.

Per una sequenza di n elementi esistono 2^n distinte sottosequenze. Un algoritmo che tenti di risolvere il problema considerando tutte le sottosequenze possibili della prima sequenza per vedere quale di queste sottosequenze è sottosequenza anche della seconda tenendo traccia della sottosequenza comune più lunga via-via trovata, non può che avere una complessità esponenziale. Quello che descriveremo ora è un algoritmo basato sulla tecnica della programmazione dinamica che risolve il problema in $O(n \cdot m)$ tempo.
Per ora consideremo solamente il calcolo della lunghezza della sottosequenza comune più lunga e successivamente vedremo come modificare l'algoritmo per ottenerla.

> *Per $i \leq n$ e $j \leq m$ sia $T[i,j]$, la lunghezza della sottosequenza più lunga comune alle sequenze $X_i = (x_1, x_2, \ldots x_i)$ e $Y_j = (y_1, y_2, \ldots y_j)$.*

La lunghezza della sottosequenza più lunga comune ad X e Y è data quindi da $T[n,m]$. Come calcolare questo valore?

E' ovvio che $T[i,0] = 0$ per $0 \leq i \leq n$ e $T[0,j] = 0$ per $1 \leq j \leq m$.
mentre per i e j entrambi maggiori di 0 possiamo ragionare come segue distinguendo due casi:

CASO $x_i = y_j$. Sia a il simbolo con cui terminano le due sequenze X_i e Y_j (i.e. $x_i = y_j$). In questo caso la sottosequenza comune ad X_i e Y_j più lunga termina con il simbolo a (infatti una qualunque sottosequenza comune che non termini con a resta comune anche con l'aggiunta del simbolo a). In questo caso quindi si ha $T[i,j] = T[i-1, j-1] + 1$.

CASO $x_i \neq y_j$. In questo caso per la sottosequenza comune S più lunga non possono esserci che le seguenti tre alternative: S termina con il simbolo x_i, S termina con il simbolo y_j, S termina con un simbolo a defferente da x_i e y_j. Nel primo caso S avrà lunghezza $T[i, j-1]$ nel secondo caso S avrà lunghezza $T[i-1, j]$ mentre nel terzo caso la lunghezza di S sarà $T[i-1, j-1]$. Per decidere qual'è la risposta giusta basta quindi porre

$$T[i,j] = \max\{T[i,j-1], T[i-1,j], T[i-1,j-1]\} = \max\{T[i,j-1], T[i-1,j]\}$$

Dove l'ultima uguaglianza segue banalmente dal fatto che $T[i-1,j-1] \leq T[i,j-1]$ e $T[i-1,j-1] \leq T[i-1,j]$.
Da quanto detto si arriva alla seguente formula ricorsiva:

$$T[i,j] = \begin{cases} T[i-1,j-1]+1 & \text{se } x_i = y_j \\ \max\{T[i,j-1], T[i-1,j]\} & \text{altrimenti.} \end{cases}$$

Per il calcolo della lunghezza massima per la sottosequenza comune ad X e Y basta procedere al calcolo dei valori di $T[i,j]$ al crescere dei valori di i e j fino ad ottenere il valore $T[n,m]$ Ecco lo pseudo-codice dell'algoritmo:

```
LUNGHEZZA-SOTTOSEQUENZA:
INPUT le due sequenze x_1, ..., x_n e y_1, ..., y_m
    FOR i ← 0 TO n DO T[i,0] ← 0
    FOR j ← 0 TO m DO T[0,j] ← 0
    FOR i ← 1 TO n DO
        FOR j ← 1 TO m DO
            IF x_i = y_j THEN T[i,j] ← T[i-1,j-1] + 1
            ELSE T[i,j] ← max{T[i,j-1], T[i-1,j]}
        ENDFOR
    ENDFOR
OUTPUT T[n,m]
```

Il calcolo di ciascuna delle $O(n \cdot m)$ caselle della tabella T richiede tempo costante. La complessità dell'algoritmo è pertanto $O(n \cdot m)$.

Sappiamo come i valori della matrice T sono stati ottenuti quindi per ottenere la sottosequenza comune di lunghezza massima basta ripercorrere "all'indietro" le varie scelte fatte per ottenere il valore $T[n,m]$.

Più precisamente la seguente procedura ricorsiva permette di ricostruire la sottosequenza comune

```
STAMPA-SOTTOSEQUENZA-MASSIMA (i, j)
    IF i > 0 AND j > 0 THEN
        IF x_i = y_j THEN
            STAMPA − SOTTOSEQUENZA − MASSIMA(i − 1, j − 1)
            STAMPA x_i
        ELSE
            IF T[i − 1, j] ≥ T[i, j − 1] THEN STAMPA-SOTTOSEQUENZA-MASSIMA(i − 1, j)
                ELSE STAMPA-SOTTOSEQUENZA-MASSIMA(i, j − 1)
        ENDIF
    ENDIF
```

Poichè ad ogni livello della ricorsione almeno uno degli indici i e j si decrementa ed ogni livello di ricorsione richiede tempo costante il tempo di esecuzione della procedura è $O(n + m)$.

4. Il problema CAMMINI MINIMI per grafi con pesi anche negativi.

Ricordiamo che in un grafo $G = (V, E)$ diretto e con pesi sugli archi, un *cammino* è una sequenza $v_1, v_2, \ldots, v_k$ di vertici tale che $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ sono archi in E, questi archi sono anche chiamati gli archi del cammino. In questo caso il cammino connette il vertice v_1 al vertice v_k. La *lunghezza* di un cammino è semplicemente il numero di archi del cammino, mentre il *peso* di un cammino è semplicemente la somma dei pesi degli archi del cammino. Il cammino privo di archi che connette un vertice a se stesso ha lunghezza 0 e peso 0. Dati due vertici s e v, si dice che v è *raggiungibile* da s se esiste almeno un cammino da s a v. Una semplice visita del grafo G a partire dal vertice s permette di stabilire se il vertice v è raggiungibile da s e in questo caso trovare un cammino che porta da s a v. Se la visita del grafo è una visita in ampiezza, il cammino trovato sarà anche quello di lunghezza minima. Se v è raggiungibile da s, come procedere alla ricerca di un cammino di peso minimo? In generale la raggiungibilità non assicura che un tale cammino esista. Infatti, la presenza in G di archi con peso negativo può indurre in G la presenza di *cicli* con peso negativo (i.e. cammini chiusi dove la somma dei pesi degli archi del cammino ha valore negativo). E vale la seguente proprietà:

Proprietà 1. *Dato un grafo G diretto e pesato, se dal vertice s sono raggiungibili vertici appartenenti a cicli di peso negativo, allora esiste un vertice v per cui esistono cammini da s a v di peso arbitrariamente piccolo.*

Dimostrazione. Sia v un vertice del ciclo di peso negativo raggiungibile da s. Si consideri il cammino da s a v che comincia con un sotto-cammino che da s porta a v e attraversa poi gli archi del ciclo negativo da v a v un numero arbitrario di volte. Poichè il ciclo che porta da v ad v ha peso negativo, più volte verrà percorso, minor peso avrà il cammino risultante da s a v.

In un grafo dove nessun vertice appartenente a cicli di peso negativo è raggiungibile dal vertice s, il fenomeno appena descritto non può verificarsi e la raggiungibilità di v da parte di s è condizione sufficiente (oltre che necessaria) perchè il cammino minimo esista. Vale infatti la seguente proprietà:

Proprietà 2. *Dato un grafo G diretto e pesato, se dal vertice s non sono raggiungibili vertici appartenenti a cicli di peso negativo, allora i vertici v del grafo raggiungibili da s hanno cammini minimi di lunghezza inferiore a $|V|$*

Dimostrazione. La prova è per assurdo. Assumiamo che il cammino minimo da s a v ha lunghezza almeno $|V|$. In questo caso c'e' almeno un vertice u' che nel cammino compare due volte. Possiamo scomporre il cammino in tre sotto-cammini, il primo che da s raggiunge u' per la prima volta, il secondo che da u' raggiunge u' la seconda volta ed il terzo che da u' raggiunge v. Consideriamo ora il nuovo cammino da s a v che si ottiene concatenando il primo ed il terzo sotto-cammino, questo nuovo cammino evita di percorrere gli archi del ciclo che da u' portano a u', ciclo che per ipotesi ha peso non negativo in quanto il vertice u' è raggiungibile da s grazie al primo sotto-cammino. Il nuovo cammino ha quindi lunghezza inferiore al primo e peso non superiore contraddicendo l'ipotesi.

Per la proprietà appena dimostrata il cammino minimo andrà ricercato in un insieme finito di cammini possibili (i.e. tutti quelli di lunghezza al più $|V| - 1$).

Consideriamo il seguente problema.

Dato un grafo $G = (V, E)$ diretto e pesato con pesi anche negativi ed un vertice $s \in V$, che chiameremo sorgente,

- *testare se in G sono presenti cicli di peso negativo i cui vertici sono raggiungibili dalla sorgente.*
- *nel caso non ci siano cicli negativi con vertici raggiungibili dalla sorgente, calcolare i cammini minimi da s ai vertici raggiungibili.*

Il problema è risolvibile in tempo $O(|V| \cdot |E|)$ grazie ad un algoritmo basato sulla tecnica della programmazione dinamica, che ora analizzeremo. L'algoritmo è noto come algoritmo di BELLMAN-FORD dal nome dei due ricercatori che l'hanno indipendentemente proposto.
Per ora consideremo solamente il calcolo del peso dei cammini minimi e successivamente vedremo come modificare l'algoritmo per ottenere i cammini minimi.

Per $l \geq 0$ e $v \in V$ sia $T[l, v]$ il costo minimo per un cammino che porta da s a v utilizzando al più l archi ($+\infty$ se tale cammino non esiste).

Ovviamente $T[0, v] = \begin{cases} 0 & \text{se } v \text{ è il vertice sorgente} \\ +\infty & \text{altrimenti} \end{cases}$

Come calcolare i valori di $T[l, j]$ quando $l > 0$? Per $l > 0$ possono esserci cammini che vanno da s a v utilizzando al più $l-1$ archi e cammini che vanno da s ad un vertice u incidente a v attraversando $l-1$ archi e raggiungono poi v attraversando l'arco (u, v).
Indicando con $Inc(v)$ l'insieme degli archi di G incidenti nel vertice v (i.e. $Inc(v) = \{u|(u, v) \in E\}$) si arriva alla seguente formula ricorsiva:

$$T[l, v] = \min \left\{ T[l-1, v], \min_{u \in Inc(v)} \{T[l-1, u] + c(u, v)\} \right\}$$

Vale la seguente proprietà

$T(|V|-1, v) = T(|V|, v)$ per ogni $v \in V$ se e solo se non ci sono vertici appartenenti a cicli di peso negativo raggiungibili da s.

Dimostrazione. Se non ci sono vertici raggiungibili da s in cicli di peso negativo per la proprietà 2 deve aversi $T[|V|-1, v] = T[|V|, v]$ per ogni $v \in V$. Se, al contrario, ci sono vertici raggiungibili da s che appartengono a cicli di peso negativo, allora per la proprietà 1 c'è un vertice v raggiungibile da s con cammini il cui costo può essere arbitrariamente piccolo. L'esistenza del vertice v implica che esiste un l', con $l' > |V|-1$, per cui $T[|V|-1, v] > T[l', v]$. E da ciò segue che deve aversi $T[|V|-1, v] \neq T[|V|, v]$. Infatti, il valore di $T[l, v]$ dipende unicamente dai valori di $T(l-1, u)$ con $u \in Inc(v)$ quindi, avere $T[|V|-1, v] = T[|V|, v]$ per ogni v, implicherebbe $T[|V|-1, v] = T[l', v]$ per ogni $l' > |V|$.

Alla ricerca delle distanze minime, l'algoritmo di BELLMAN-FORD procede al calcolo dei diversi valori $T(l, v)$ con l via-via crescenti fino ad $l = |V|$. Controlla poi che sia $T(|V|-1, v) = T(|V|, v)$ per ogni $v \in V$. Se la condizione è verificata allora per ogni $v \in V$ restituisce come peso minimo per raggiungere v da s il valore $T(|V|-1, v)$. Se la condizione non è verificata, segnala la presenza di vertici appartenenti a cicli di peso negativo raggiungibili da s.

L'algoritmo è descritto dal seguente pseudocodice:

```
DISTANZE-MINIME
INPUT un grafo G = (V, E) diretto e pesato e un nodo s ∈ V
    FOR v ∈ V DO genera la lista di incidenza Inc(v)
    T[0, s] ← 0
    FOR v ∈ V − {s} DO T[0, v] ← +∞
    flag ← 0
    FOR l ← 1 TO |V| DO
        FOR v ∈ V DO
            D[l, v] ← D[l − 1, v]
            FOR u ∈ Inc(v) DO
                IF T[l − 1, u] + c(u, v) < T[l, v] THEN
                    T[l, v] ← T[l − 1, u] + c(u, v)
                    IF l = |V| THEN flag ← 1
```

```
                ENDIF
            ENDFOR
        ENDFOR
    ENDFOR
OUTPUT flag e la riga |V|-ma della matrice T
```

La procedura restituisce il valore di $flag$ che è 1 qualora in G siano presenti vertici appartenenti a cicli di peso negativo raggiungibili da s, 0 altrimenti. Se il valore di $flag$ è 0, allora i valori $T[|V|, v]$, con $v \in V$, sono i costi minimi per raggiungere v da s. Se il valore di $flag$ è 1, allora i valori $T[|V|, v]$, con $v \in V$, sono i costi minimi per raggiungere v da s con cammini di lunghezza al più $|V|$.

La generazione delle $|V|$ liste di adiacenza costa $O(|V|^2)$ tempo. Il calcolo della prima riga della tabella T costa $O(|V|)$. Una singola iterazione del secondo dei tre FOR annidati richiede sostanzialmente di scorrere una delle liste di incidenza del grafo ed ha quindi un costo proporzionale al numero di archi presenti nella lista. Iterazioni distinte di quel FOR fanno riferimento a liste distinte. Poichè il numero totale di archi in tutte le liste è $|E|$, il tempo totale richiesto dal FOR è $O(|E|)$. Il FOR più esterno viene eseguito esattamente $|V|$ volte. Quindi il tempo richiesto dai tre FOR annidati è $O(|V| \cdot |E|)$ che è quindi la complessità dell'algoritmo.

Per ricavare il cammino di peso $T[|V| - 1, v]$ che porta da s a v possiamo definire il vettore $P[]$ dei predecessori inizializzato con

$$P[v] = \begin{cases} v & \text{se } v \text{ è la sorgente} \\ +\infty & \text{altrimenti} \end{cases}$$

e da aggiornare alla l-ma iterazione del FOR in base alla regola

$$P[v] = \begin{cases} P[v] & \text{se } T[l-1, v] = T[l, v] \\ u \text{ dove } u \text{ è il vertice in } Inc(v) \text{ che dà il minimo per } \min_{u \in Inc(v)}\{T[l-1, u] + c(u, v)\} & \text{altrimenti} \end{cases}$$

È facile convincersi che al termine dell'algoritmo, in $P[v]$ troveremo $+\infty$ se il vertice v non è raggiungibile dalla sorgente, in caso contrario $P[v]$ conterrà il predecessore di v nel cammino minimo che dalla sorgente porta a v.

Al fine di risparmiare spazio, possiamo poi osservare che per calcolare la riga $l + 1$ della matrice T è sufficiente conservare in memoria la sola riga l di T. Così facendo la complessità di spazio dell'algoritmo passa da $O(|V|^2)$ a $O(|V|)$.

Quello che segue è lo pseudo-codice che tiene conto delle osservazioni appena fatte.

```
DISTANZE-MINIME
INPUT un grafo diretto G = (V, E) con archi pesati e un nodo s
    FOR v ∈ V DO genera la lista di adiacenza Inc(v)
    T[0, s] ← 0
    P[s] ← s
    FOR v ∈ V − {s} DO
        T[0, v] ← +∞
        P[v] ← +∞
    ENDFOR
    flag ← 0
    x ← 0
    FOR l ← 1 TO |V| DO
        y ← l mod 2
        FOR v ∈ V DO
            T[y, v] ← T[x, y]
            FOR u ∈ Inc(v) DO
                IF T[x, u] + c(u, v) < T[y, v] THEN
                    T[y, v] ← T[x, u] + c(u, v)
                    P[v] ← u
                    IF l = |V| THEN flag ← 1
            ENDFOR
        ENDFOR
        x ← y
```

ENDFOR
OUTPUT il $flag$, la riga l mod 2 della matrice T ed il vettore P

Qualora $flag = 0$ e $P[v] \neq +\infty$ allora v è raggiungibile dalla sorgente s e la seguente procedura ricorsiva permette di ottenere la sequenza dei vertici incontrasti sul cammino minimo da s a v.

```
CAMMINO(v)
    IF P[v] ≠ v THEN
        CAMMINO(P[v])
        STAMPA v
    ENDIF
```

La complessità è lineare nella lunghezza del cammino ed è quindi $O(|V|)$.

4.1. **SISTEMI CON VINCOLI DI DIFFERENZA.** Un *sistema di vincoli di differenza* è un insieme di m vincoli su n incognite, in cui ogni vincolo è una diseguaglianza lineare della forma $x_i - x_j \leq c$ dove x_i ed x_j sono due delle n incognite e c è una qualunque costante. Una soluzione per il sistema è un assegnamento di valori alle n variabili che soddisfa tutti i vincoli del sistema.
Ad esempio, per il sistema nelle 5 incognite $\{x_1, x_2, \ldots, x_5\}$ con i seguenti 5 vincoli di differenza:

$$\begin{array}{rcr} x_1 - x_2 & \leq & 0 \\ x_1 - x_5 & \leq & -1 \\ x_4 - x_3 & \leq & -1 \\ x_5 - x_3 & \leq & -3 \\ x_3 - x_1 & \leq & 5 \end{array}$$

una possibile soluzione è $x = (-5, , -3, 0, -1, -4)$, come si può verificare direttamente controllando ogni diseguaglianza. In effetti vi è più di una soluzione per questo problema: ad esempio $x' = (-2, , 0, 3, 2, -1)$ è un'altra soluzione. Queste due soluzioni sono correlate: ogni componente di x' è più grande della corrispondente componente di x di 3, e questa non è una coincidenza.

Se $x = (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$ è una soluzione di un sistema di vincoli di differenza e d una qualunque costante, allora $x' = (\overline{x}_1 + d, \overline{x}_2 + d, \ldots, \overline{x}_n + d)$ è anch'essa una soluzione del sistema. Infatti per ogni $\overline{x}_i$ ed $\overline{x}_j$ si ha $(\overline{x}_i + d) - (\overline{x}_j + d) = \overline{x}_i - \overline{x}_j$, di conseguenza se la soluzione x soddisfa il generico vincolo $x_i - x_j \leq c$ del sistema anche x' lo fa.

Per quanto detto un sistema di vincoli di differenza se ha una soluzione ne ha in realtà infinite. D'altra parte non è difficile convincersi che possono anche non esserci soluzioni come nel caso del seguente sistema in due incognite e due vincoli:

$$\begin{array}{rcr} x_1 - x_2 & \leq & 1 \\ x_2 - x_1 & \leq & -2 \end{array}$$

I sistemi di vincoli di differenza compaiono in diverse applicazioni. Ad esempio, le incognite possono essere istanti di tempo in cui certi eventi devono occorrere: allora ogni vincolo può stabilire che un certo evento non può occorrere troppo tempo dopo un altro evento.
Si consideri pertanto il seguente problema:

> *Dato un sistema di vincoli di differenza con m vincoli in n incognite, determinare se il sistema ha soluzioni ed in caso affermativo calcolarne una.*

Possiamo ridurre il problema a quello della ricerca dei costi minimi per cammini da sorgente in grafi diretti e pesati. L'idea è quella di costruire, a partire dal sistema di vincoli un grafo diretto e pesato, detto *grafo dei vincoli*, i cui nodi rappresentano le incognite del sistema e con un nodo aggiuntivo detto sorgente. Il grafo dei vincoli avrà la proprietà che, in assenza di cicli di peso negativo, una soluzione per il sistema di vincoli si ottiene assegnando alle varie incognite il valore corrispondente al costo minimo necessario a raggiungere i vertici del grafo a loro corrispondenti, partendo dal vertice sorgente. La presenza nel grafo di cicli di costo negativo implicherà invece la non esistenza di soluzioni per il sistema di vincoli.
Stando cosi le cose per risolvere il problema basterà costruire il grafo dei vincoli G e testare in G l'eventuale presenza di cicli negativi. In mancanza di tali cicli il calcolo dei costi minimi per raggiungere i vari vertici del grafo a partire dalla sorgente produrrà una soluzione al problema.

Il grafo dei vincoli G è così definito: i vertici di G sono le n incognite del sistema $\{x_1, x_2, \ldots, x_n\}$ più un nodo speciale s detto sorgente. Per quanto riguarda gli archi: per ogni vincolo $x_i - x_j \leq c$ del sistema viene introdotto un arco di costo c

dal vertice x_j al vertice x_i. Vengono inoltre introdotti n archi di costo 0 che vanno dal vertice sorgente agli n altri vertici di G.

Definito il grafo dei vincoli, dimostriamo ora le sue proprietà

Dato un sistema di vincoli in n incognite $\{x_1, x_2, \ldots, x_n\}$ ed il corrispondente grafo dei vincoli G,

(1) *Se G contiene un ciclo di peso negativo, allora il sistema non ha soluzioni.*
(2) *Se G non contiene cicli di peso negativo, allora $x = (D[1], D[2], \ldots, D[n])$ è una soluzione per il sistema, dove con $D[i]$, $1 \leq i \leq n$, denotiamo il costo minimo in G per raggiungere il vertice x_i a partire dal vertice sorgente.*

Dimostrazione.

(1) Se G ha un ciclo, allora il vertice sorgente, non avendo archi entranti, non può farne parte. Senza perdere di generalità assumiamo quindi che il ciclo di peso negativo sia $< v_1, v_2, \ldots, v_k >$ dove le v_i, con $1 \leq i \leq k$, corrispondono ad incognite del sistema. Sia inoltre c_i, con $1 \leq i < k$, il peso dell'arco che da v_i va a v_{i+1} e c_k il peso dell'arco che da v_k va a v_1. Per costruzione di G a ciascuno dei k archi del ciclo corrisponde nel sistema un vincolo di differenza. Tra i vincoli di differenza del sistema abbiamo quindi:

$$\begin{array}{rcl} v_2 - v_1 & \leq & c_1 \\ v_3 - v_2 & \leq & c_2 \\ \ldots & \leq & \ldots \\ v_k - v_{k-1} & \leq & c_{k-1} \\ v_1 - v_k & \leq & c_k \end{array}$$

Poichè ogni soluzione del sistema deve soddisfare ognuna di queste k diseguaglianze, qualunque soluzione deve anche soddisfare la diseguaglianza che si ottiene sommandole tutte insieme. Se sommiamo i membri di sinistra, ogni incognita viene addizionata una volta e sottratta una volta, e quindi la somma della parte sinistra è 0. La somma della parte destra è $C = \sum_{i=1}^{n} c_i$ che è il peso del ciclo. Si ottiene quindi $0 \leq C$. Ma siccome il ciclo ha peso negativo deve valere anche $C < 0$. Una qualunque soluzione del sistema deve quindi soddisfare $0 \leq C < 0$ il che è impossibile.

(2) Poichè G non contiene cicli di peso negativo e per costruzione ciascun vertice del grafo è raggiungibile dalla sorgente, i valori $\{D[1], D[2], \ldots, D[n]\}$ sono tutti ben definiti. Sia ora $x_i - x_j \leq c$ uno qualunque dei vincoli di differenza del sistema. Basterà dimostrare che $D[i] - D[j] \leq c$.
Il vincolo $x_i - x_j \leq c$ del sistema implica l'esistenza nel grafo G di un arco di peso c che va da x_j a x_i. In G, posso quindi raggiungere x_i con un cammino che prima mi porta ad x_j e poi arriva ad x_i grazie all'arco di peso c. Il costo di tale cammino è quindi $D[j] + c$. Per quanto detto, per il costo minimo $D[i]$ del cammino che va da s a x_j deve quindi aversi $D[i] \leq D[j] + c$. Da cui ricaviamo $D[i] - D[j] \leq c$.

Passiamo ora alla valutazione della complessità dell'algoritmo proposto. Un sistema di vincoli di differenza con m vincoli in n incognite produce un grafo dei vincoli $G = (V, E)$ con $n + 1$ vertici ed $m + n$ archi. La costruzione del grafo richiede $O(n + m)$. Quindi usando l'algoritmo di Bellman-Ford si può risolvere il problema in $O(|V| \cdot |E|) = O(n^2 + m \cdot n)$ tempo.

5. Il problema dello ZAINO a variabili intere.

Consideriamo il seguente problema:

*Dato un insieme di n oggetti caraterizzati da n valori $v_1, v_2, \ldots, v_n$ ed n pesi $p_1, p_2, \ldots p_n$ interi positivi, ed un intero positivo C (la "*capacità*" dello zaino), trovare un sottoinsieme degli n oggetti il cui peso complessivo non ecceda la capacità dello zaino e il cui valore sia massimo (i.e. trovare un sottoinsieme S di $\{1, 2, \ldots n\}$ tale che $\sum_{i \in S} p_i \leq C$ e $\sum_{i \in S} v_i$ sia massimo).*

Descriveremo un algoritmo basato sulla programmazione dinamica che permette di risolvere il problema in $O(n \cdot C)$. Per cominciare procediamo al calcolo del valore della soluzione, successivamente vedremo come modificare l'algoritmo per ottenere la soluzione.

Per $0 \leq i \leq n$ e $0 \leq j \leq C$ sia $T[i, j]$ pari al valore massimo che si può ottenere, disponendo dei soli primi i oggetti e di uno zaino con capacità j.

È ovvio che il valore della soluzione al problema dello zaino è dato da $T[n, C]$. Come calcolare questo valore ?
Ovviamente $T[0, j] = 0$ per $0 \leq j \leq C$. Sia dunque $i > 0$. La soluzione ottima per uno zaino di dimensioni j quando si dispone dei soli oggetti $1, \ldots i$ non può contenere l'oggetto i se questo ha un peso superiore alla capacità dello zaino,

in questo caso quindi il valore della soluzione sarà dato da $T[i-1,j]$. Se l'oggetto i ha peso non superiore a quello dello zaino allora non si può escludere a priori che l'oggetto i appartenga alla soluzione, se non vi appartiene la soluzione varrà $T[i-1,j]$ se vi appartiene allora contribuisce per un valore v_i e, una volta inserito nello zaino, lascia agli altri $i-1$ oggetti una capacità pari a $C-p_i$ in tale situazione i rimanenti oggetti contribuiscono per un valore $T[i-1,C-p_i]$.
Per quanto detto si arriva alla seguente formula ricorsiva

$$T[i,j] = \begin{cases} T[i-1,j] & \text{se } j < p_i \\ \max\{T[i-1,j], T[i-1,j-p_i]+v_i\} & \text{altrimenti.} \end{cases}$$

Ecco lo pseudo-codice dell'algoritmo che si ottiene grazie all'equazione di ricorrenza descritta:

```
VALORE-ZAINO:  INPUT i pesi p_1,...,p_n i valori v_1,...,v_n e la capacità C.
    FOR j ← 0 TO C DO T[0,j] ← 0
    FOR i ← 1 TO n DO
        FOR j ← 0 TO C DO
            IF p_i > j THEN T[i,j] ← T[i-1,j]
            ELSE T[i,j] ← max{T[i-1,j], T[i-1,j-p_i] + v_i}
        ENDFOR
    ENDFOR
OUTPUT T[n,C]
```

Riempire ciscuna delle $O(n \cdot C)$ caselle della tabella T richiede tempo costante. La complessità dell'algoritmo per il calcolo del valore della soluzione al problema dello zaino è quindi $O(n \cdot C)$.

La tabella T può essere utilizzata per costruire direttamente il sottoinsieme di oggetti che costituiscono la soluzione ottima di valore $T[n,C]$. L'idea è piuttosto semplice. Si parte dall'elemento $T[n,C]$ nell'ultima riga della tabella e si percorrono le righe della tabella all'indietro fino ad arrivare alla prima riga. Alla generica riga i ritrovandosi nella colonna j, per determinare se la soluzione ha tra i suoi elementi anche l'elemento i basterà controllare il valore $T[i-1,j]$. Infatti l'i-esimo oggetto appartiene alla soluzione se e solo se $T[i,j] \neq T[i-1,j]$. Se l'oggetto non appartiene alla soluzione ci si sposta quindi nella riga precedente restando nella colonna j, in caso contrario ci si sposta nella riga precedente e nella colonna $j-p_i$.
Per quanto detto, si può ottenere il vettore SOL caratteristico del sottoinsieme di oggetti della soluzione ottima aggiungendo allo pseudocodice di VALORE-ZAINO le seguenti istruzioni:

```
j ← C
FOR i ← n DOWNTO 1 DO
    IF T[i,j] = T[i-1,j] THEN
        SOL[i] ← 0
    ELSE
        SOL[i] ← 1
        j ← j - p_i
    ENDIF
ENDFOR
```

e dare in output il vettore SOL.
L'esecuzione delle istruzioni aggiuntive determina una complessità addittiva $O(n)$ che non cambia la complessità asintotica $O(n \cdot C)$ dell'algoritmo.

È importante osservare che la complessità $O(n \cdot C)$ dell'algoritmo presentato non è polinomiale nella dimensione del problema. In altri termini, la complessià di VALORE-ZAINO non dipende soltanto dal numero di dati in ingresso, ma anche dal valore di uno di questi. Tale complessità risulta polinomiale o meno a seconda che il valore dell'elemento sia piccolo oppure grande rispetto al numero totale di elementi. In situazioni come quella appena descritta si parla di complessità *pseudo-polinomiale*.