Progettazione di Algoritmi - lezione 24

Discussione dell'esercizio [cassaforte]

Sapendo che la combinazione di una cassaforte è composta da una sequenza di n cifre decimali la cui somma ha valore k, dobbiamo descrivere un algoritmo che genera tutte le possibili combinazioni con questa proprietà. La complessità deve essere limitata da $O(nC(n, k))$, dove $C(n, k)$ è il numero di combinazioni possibili.

Possiamo usare lo schema di generazione tramite backtracking, ovvero la visita dell'albero implicito di tutti i prefissi delle sequenze da generare. Le combinazioni possibili sono le sequenze S di n cifre decimali la cui somma delle cifre è k. Durante la visita, quando ci troviamo nel prefisso $S[1\cdots i]$ dobbiamo sapere con quali cifre può essere esteso. Per fare ciò ci occorre conoscere la somma delle cifre del prefisso, denotiamola con p. Allora una cifra c può estendere il prefisso $S[1\cdots i]$ di somma p solo se $c \leq k - p$. Però dobbiamo anche garantire che possiamo sempre completare la sequenza con somma finale k. Ne segue che deve essere $p + c + 9(n - i - 1) \geq k$. Vale a dire, $c \geq k - p - 9(n - i - 1)$. Quindi le cifre c che possono estendere un prefisso di somma p sono esattamente quelle tali che $k - p - 9(n - i - 1) \leq c \leq k - p$. Per efficienza conviene che la somma del prefisso sia mantenuta durante la visita.

```
CC(n: lung. comb., k: somma comb., i: lung. prefisso, p: somma prefisso, S: combinazione)
    IF i = n THEN
        OUTPUT S
    ELSE
        minc = max{0, k - p - 9(n - i - 1)}
        maxc = min{9, k - p}
        FOR c = minc TO maxc DO
            S[i+1] <- c
            CC(n, k, i+1, p+c, S)
```

La prima chiamata sarà `CC(n, k, 0, 0, S)`. È garantito che in ogni nodo interno, cioè $i < n$, c'è sempre almeno un'estensione ammissibile e quindi ogni nodo interno porta a una combinazione generata. Possiamo perciò applicare il risultato generale che dà un limite superiore alla complessità. Il tempo in ogni nodo interno è $O(1)$ e nelle foglie è $O(n)$, quindi la complessità è $O(nC(n, k))$.

Sfrondare lo spazio di ricerca

Una volta capito come generare tutte le soluzioni possibili di un problema (difficile), possiamo iniziare a pensare a un algoritmo per trovare una soluzione ottima. L'obbiettivo è di progettare un algoritmo che possa evitare di visitare l'intero spazio di ricerca di tutte le soluzioni. Per cercare di tagliare o sfrondare il più possibile lo spazio di ricerca ci possiamo basare sul prefisso che abbiamo già generato. Come vedremo presto, in molti casi si può fare un semplice test che permette di evitare di visitare il sottoalbero delle soluzioni possibili che estendono il prefisso perché siamo certi che tra queste non ci può essere una soluzione ottima. Questo tipo di test sono strettamente legati allo specifico problema che si vuole risolvere e non ci sono ricette generali se non quelle relative ad alcune classi di problemi che tipicamente derivano da formulazioni tramite la programmazione lineare. Ma ciò esula da questo corso. Vedremo solamente alcuni esempi di sfrondamento dello spazio di ricerca relativamente ad altrettanti problemi specifici. Tuttavia, al di là della specificità delle tecniche, si potranno intravedere alcune idee che possono guidare verso la determinazione di tecniche simili per molti altri problemi.

3-Colorazione

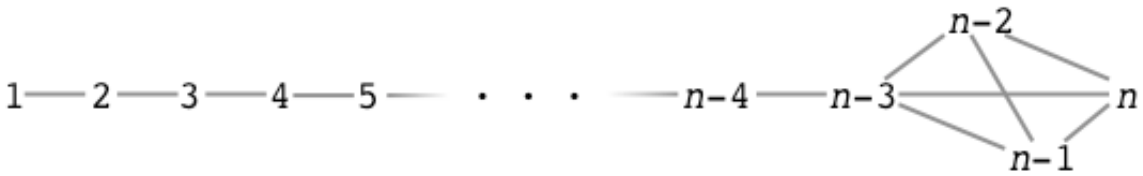
Iniziamo con un problema le cui soluzioni possibili hanno una struttura molto semplice. Le soluzioni possibili per il problema della 3-Colorazione sono le sequenze i cui elementi sono uno dei tre colori { r , g , b }. Per un grafo (non diretto) G, una **soluzione effettiva** è una 3-colorazione C degli n nodi di G tale che, per ogni arco $\{i, j\}$ di G, $C[i] \neq C[j]$. Ovviamente, il problema della 3-Colorazione per G consiste proprio nel determinare se una 3-colorazione effettiva per G esiste oppure no. Sappiamo che il problema della 2-Colorazione (che coincide con il problema di

determinare se un grafo è bipartito o meno) è efficientemente risolvibile. La 3-Colorazione è invece un problema difficile.

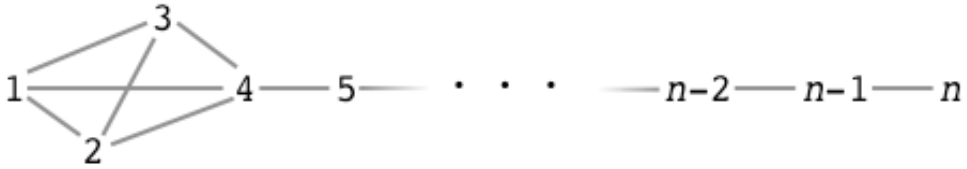
I nodi del grafo sono numerati da 1 a *n* e le colorazioni colorano i nodi a partire dal nodo 1 e procedono seguendo la numerazione. Un prefisso *C*[1⋯*k*] di lunghezza *k* corrisponde ad aver colorato i nodi 1,2…,*k*. Un'idea molto semplice per tagliare lo spazio di ricerca è controllare, ogni volta che cerchiamo di estendere un prefisso di lunghezza *k*, se il colore che assegniamo al nodo *k*+1 è lecito. Per ogni arco {*i*, *k*+1} con $i \leq k$, togliamo *C*[*i*] dall'insieme dei colori disponibili { *r* , *g* , b }, ottenendo alla fine l'insieme *L* dei colori leciti. Il seguente programma, se il grafo di input *G* è 3-colorabile ritorna una effettiva 3-colorazione di *G*, altrimenti ritorna `NULL` :

```
3C(G: grafo non diretto, n: numero nodi, k: lunghezza prefisso, C: colorazione)
    IF k = n THEN
        RETURN C    /* Trovata una 3-colorazione di G */
    ELSE
        L <- {r,g,b}
        FOR ogni adiacente i di k+1 DO
            IF i ≤ k THEN
                 L <- L - C[i]
        FOR ogni c in L DO
            C[k+1] <- c
            R <- 3C(G, n, k+1, C)
            IF R ≠ NULL THEN   /* È stata trovata una 3-colorazione */
                RETURN R
        RETURN NULL
```

Si osservi che rispetto ai programmi di generazione che abbiamo visto finora, i nodi interni dell'albero dei prefissi delle colorazioni visitati da `3C` non portano necessariamente a soluzioni effettive. E questo è dovuto proprio alla potatura dell'albero effettuata in base agli archi del grafo di input. Dare una buona stima della complessità è piuttosto difficile proprio perchè dipende fortemente dal grafo di input. Possiamo però vedere come si comporta su alcuni input. Consideriamo il grafo *G* mostrato nella figura qui sotto (con la relativa numerazione dei nodi):



È chiaro che *G* non è 3-colorabile ma il programma `3C` quanto tempo impiegherà per accorgesene? Per i primi *n*-3 nodi l'algoritmo trova sempre due colori leciti (per il nodo 1 ne ha tre) quindi il programma visita almeno un albero binario completo di profondità *n*-3. Un tale albero contiene almeno 2^{n-3} nodi, per cui la complessità sarà almeno pari a $\Omega(2^n)$ e quindi esponenziale in *n*. Ma se ora modifichiamo la numerazione dei nodi del grafo *G* come nella figura:



La complessità si riduce enormemente perché quando l'algoritmo arriva al nodo 4 non trova mai colori leciti e quindi la complessità è O(1). Quindi la complessità del programma `3C` non solo dipende dal grafo ma può dipendere fortemente anche dalla numerazione dei nodi. Già questo piccolo esempio ci fa pensare che conviene che i nodi siano numerati in ordine di grado decrescente (o meglio, non crescente). Questo eviterebbe l'eplosione esponenziale per il grafo considerato ma non è difficile dare esempi di grafi che anche adottando tale numerazione presentano fenomeni analoghi.

Cicli Hamiltoniani

Dato un grafo non diretto *G* vogliamo determinare se esiste o meno un ciclo Hamiltoniano in *G*, cioè un ciclo che passa per tutti i nodi del grafo. Possiamo considerare come soluzione possibile una qualsiasi permutazione dei nodi del grafo e una permutazione *P* è una soluzione effettiva se esistono nel grafo tutti gli archi {*P*[1],*P*[2]}, {*P*[2],*P*[3]},…, {*P*[*n*-1],*P*[*n*]}, {*P*[*n*],*P*[1]}. Per potare lo spazio di ricerca possiamo imporre che i prefissi rappresentino cammini del grafo. Modificando il programma che genera le permutazioni otteniamo un programma che se il grafo di input *G* ha un ciclo Hamiltoniano ritorna uno di questi, altrimenti ritorna `NULL` :

```
CH(G: grafo non diretto, n: numero nodi, k: lunghezza prefisso, P: permutazione dei nodi)
    IF k = n THEN
        IF {P[n], P[1]} è un arco di G THEN RETURN P
        ELSE RETURN NULL
    ELSE
        E: array di dimensione n, inizializzato a 0
        FOR i <- 1 TO k DO
            E[P[i]] <- 1
        FOR i <- 1 TO n DO
            IF E[i] = 0 AND {P[k], i} è un arco di G THEN
                P[k+1] <- i
                R <- CH(G, n, k+1, P)
                IF R ≠ NULL THEN
                    RETURN R
        RETURN NULL
```

Anche per il programma `CH` la complessità dipende fortemente dal grafo di input. Ci sono grafi per i quali trova subito un ciclo Hamiltoniano (ad es. se il grafo è un ciclo) o determina che non ce ne sono (ad es. se il grafo è un albero). Ma ci sono tantissimi grafi per i quali impiega tempo esponenziale per dare una risposta. Basti pensare che se il grafo di input *G* non ha cicli Hamiltoniani, tutti i cammini dal nodo 1 saranno comunque visitati e se questi sono tanti la complessità sarà molto elevata. Si consideri il semplice grafo nella figura qui sotto:

Il grafo ha $n = 3k+1$ nodi ed è la concatenazione di $k \geq 1$ cicli ognuno di 4 nodi. Ogni ciclo può essere attraversato in due modi diversi quindi il numero di cammini dal nodo 1 al nodo n è 2^k e questi saranno tutti esaminati dal programma `CH` . Quindi impiegherà almeno tempo $\Omega(2^{n/3})$ per determinare che il grafo non ha cicli Hamiltioniani.

Zaino

I precedenti esempi sono problemi decisionali in cui si tratta di decidere se esiste o meno una soluzione effettiva. Nei problemi di ottimizzazione (come ad es. il problema dello Zaino) invece cerchiamo una soluzione ottima, cioè una soluzione che massimizza o minimizza un qualche tipo di misura. Per risolvere un problema di ottimizzazione esaminiamo tutte le soluzioni ammissibili e, nel corso della ricerca, teniamo traccia della miglior soluzione ammissibile trovata, chiamata *ottimo attuale*. Al termine della ricerca, l'ottimo attuale sarà una soluzione ottima. Nel corso della visita dell'albero delle soluzioni, come abbiamo già visto per i problemi decisionali, converrà potare i sottoalberi che non contengono soluzioni ammissibili. Nel contesto dei problemi di ottimizzazione si rivela utile anche un altro tipo di tagli: potare sottoalberi che non contengono soluzioni ammissibili in grado di migliorare l'ottimo attuale.

Consideriamo il problema di ottimizzazione dello Zaino. Si hanno *n* oggetti ciascuno con un proprio valore ed un proprio peso ed uno zaino con una fissata capacità. Si vuole il sottoinsieme di oggetti il cui peso complessivo non eccede la capacità dello zaino e il cui valore complessivo sia massimo. Le soluzioni sono quindi i sottoinsiemi di

oggetti, le soluzioni ammissibili sono i sottoinsiemi il cui valore non eccede la capacità dello zaino e la soluzione ottima è la soluzione ammissibile per cui risulta massimo il valore dei suoi oggetti. Un primo tipo di taglio che possiamo fare è di evitare di esaminare sottoinsiemi il cui peso supera la capacità. Inoltre, possiamo cercare di tagliare quei sottoalberi che siamo certi non contengono soluzioni ammissibili migliori dell'ottimo attualmente trovato. Per far ciò calcoliamo una limitazione superiore al valore delle soluzioni ammissibili presenti in un sottoalbero, se questo valore non migliora l'ottimo attuale allora il sottoalbero può essere potato. Quanto più precisa sarà la limitazione, tanto più il taglio sarà efficace. D'altra parte limitazioni superiori molto precise potrebbero richiedere tempi di calcolo tali da rendere preferibile comunque l'esplorazione del sottoalbero. Un modo molto semplice ed efficiente di ottenere una tale limitazione consiste nel sommare al valore degli oggetti del prefisso delle soluzioni il valore di tutti gli oggetti rimanenti.

Per implementare efficientemente il primo tipo di taglio conviene che il peso `pS` della soluzione parziale `S` sia mantenuto durante la visita. Analogamente per un calcolo efficiente del taglio del secondo tipo conviene che il valore `vS` della soluzione parziale e il valore totale degli oggetti ancora non considerati `rS` sia mantenuto durante la visita. Inoltre oltre all'ottimo attuale `A` conviene mantenere anche il valore `vA` di tale soluzione. Sia `A` che `vA` devono essere mantenuti durante l'intera visita e modificati solamente quando si trova una soluzione migliore e quindi conviene che siano delle variabili globali.

```
A <- soluzione iniziale
vA <- valore di A

ZAINO(n: num. oggetti, P: pesi, V: valori, c: capacità, k: lung. prefisso, S: sott., vS, pS, rS)
    IF k = n THEN
        IF vS > vA THEN
             A <- S
             vA <- vS
    ELSE
        IF pS + P[k+1] ≤ c THEN
            S[k+1] <- 1
            ZAINO(n, P, V, c, k+1, S, vS+V[k+1], pS+P[k+1], rS-V[k+1])
        IF vS + rS - V[k+1] > vA THEN
            S[k+1] <- 0
            ZAINO(n, P, V, c, k+1, S, vS, pS, rS-V[k+1])
```

La prima chiamata sarà `ZAINO(n, P, V, c, 0, S, 0, 0, sumv)` dove `sumv` è la somma di tutti i valori. Come soluzione `A`, inizialmente, si può prendere il sottoinsieme vuoto e quindi `vA` = 0. Però, migliore è la soluzione `A` da cui si parte e più i tagli del secondo tipo saranno efficaci e quindi converrebbe partire con un ottimo attuale di una certa qualità. Per fare ciò si può ottenere una buona soluzione con un algoritmo greedy che non garantisce l'ottimalità ma garantisce soluzioni di buona qualità. Partire con un ottimo attuale di buona qualità è una buona strategia non solo per l'algoritmo `ZAINO` ma per tutti gli algoritmi di backtracking progettati per problemi di ottimizzazione.

Esercizi

Esercizio [cricca]

Si consideri un grafo non diretto $G = (V, E)$. Una cricca di G è un sottoinsieme V' di V tale che tutti i nodi di V' sono tra loro adiacenti. Una k-cricca di G, $1 \leq k \leq |V|$, è una cricca con k nodi. Descrivere un algoritmo che, presi in input un grafo non diretto G e un intero k, stampi tutte le k-cricche di G. L'algoritmo deve sfruttare il grafo di input per fare opportuni tagli allo spazio di ricerca.

Esercizio [regine]

Su di una scacchiera $n \times n$ due regine non si contrastano se sono posizionate su caselle appartenenti a righe diverse, colonne diverse e diverse diagonali. Una *configurazione lecita* è un assegnamento di n regine ad n caselle della scacchiera di modo che queste non si contrastino. Descrivere un algoritmo che, preso in input un intero n,

stampi tutte le configurazioni lecite di n regine per la scacchiera $n \times n$. In una configurazione lecita, due qualsiasi delle n regine non possono essere sulla stessa riga della scacchiera, quindi una configurazione lecita si può rappresentare tramite una sequenza C dove $C[i]$ è la colonna in cui si trova la regina posizionata nell'i-esima riga. L'algoritmo deve sfruttare i vincoli di una configurazione lecita per tagliare opportunamente lo spazio di ricerca.

Esercizio [parità]

Descrivere un algoritmo che, preso in input un intero n, stampi tutte le stringhe di lunghezza $2n$ sull'alfabeto $\{a, b\}$ con lo stesso numero di occorrenze dei simboli a e b. La complessità dell'algoritmo deve essere $O(n\boldsymbol{S}(n))$, dove $\boldsymbol{S}(n)$ è il numero di stringhe da stampare.

Esercizio [numero sottosequenze crescenti]

Descrivere un algoritmo che, data una sequenza S di n interi distinti, calcola il numero di sottosequenze crescenti di S in $O(n^2)$ tempo. Ad esempio per $S = (5, 3, 7, 8, 6)$ il numero di sottosequenze crescenti è 14 (le sottosequenze sono: (5), (5, 7), (5, 7, 8), (5, 8), (5, 6), (3), (3, 7), (3, 7, 8), (3, 8), (3, 6), (7), (7, 8), (8), (6)).

Esercizio [cammino]

Una pedina è posizionata sulla casella (1, 1) in alto a sinistra di una scacchiera $n \times n$ e mediante una sequenza di mosse tra caselle adiacenti deve raggiungere la casella (n, n) in basso a destra. Una pedina posizionata sulla generica casella (i, j) ha al più due mosse possibili: spostarsi verso il basso nella casella $(i + 1, j)$, posto che $i < n$, o spostarsi verso destra nella casella $(i, j + 1)$, posto che $j < n$. La sequenza di caselle toccate determina un cammino. Ad ogni casella della scacchiera è associato un valore $v_{i,j}$ ed il valore del cammino è dato dalla somma dei valori delle caselle toccate. Descrivere un algoritmo che trova il cammino di valore massimo in tempo $O(n^2)$.

Esercizio [somma]

Dato un insieme $A = \{a_1, a_2, \ldots, a_n\}$ di n interi positivi e un intero M, vogliamo sapere se c'è un sottoinsieme X di A tale che la somma degli elementi in X è uguale a M. Ad esempio, per $A = \{2,5,7,8,10\}$ se $M = 12$ la risposta è SI (7+5) mentre per $M = 11$ la risposta è NO. Descrivere un algoritmo che, dati A e M, determina se un tale sottoinsieme esiste o meno in tempo $O(nM)$.

Soluzioni

Di seguito presentiamo alcune possibili soluzioni degli esercizi proposti.

Discussione dell'esercizio [cricca]

Essendo una k-cricca formata da un sottoinsieme di k nodi del grafo di input G possiamo basare il nostro algoritmo sulla generazione di tutti i k-sottoinsiemi di $\{1,\ldots,n\}$, cioè l'insieme dei nodi di G. Inoltre possiamo operare dei tagli tenendo conto che quando aggiungiamo un nuovo nodo al sottoinsieme che stiamo costruendo, questo deve essere adiacente a tutti i nodi che sono già nel sottoinsieme.

```
CRICCA(G: grafo di n nodi, k: card.cricca, ℓ: card. prefisso, h: lung. prefisso, C: k-cricca)
    IF h = n THEN
        OUTPUT C
    ELSE
        IF ℓ ≥ k - n + h + 1 THEN
            C[h+1] <- 0
            CRICCA(G, k, ℓ, h+1, C)
        IF ℓ < k THEN
            i <- 1
            WHILE i ≤ h AND (C[i] = 0 OR {i, h+1} è un arco di G) DO
                i <- i + 1
            IF i = h+1 THEN  /* h+1 è adiacente a tutti i nodi in C[1…h] */
                C[h+1] <- 1
                CRICCA(G, k, ℓ+1, h+1, C)
```

La prima chiamata sarà `CRICCA(G, k, 0, 0, C)`.

Discussione dell'esercizio [regine]

Rappresentiamo le possibili posizioni delle *n* regine tramite una sequenza (o array) *C* tale che *C*[*i*] è la colonna della casella in cui è posizionata l'*i*-esima regina. Questo tipo di rappresentazione garantisce automaticamente che non ci sono due regine sulla stessa riga. Per garantire che non ci siano due regine sulla stessa colonna, è sufficiente che i valori di *C* siano una permutazione delle colonne (1,…,*n*). Dobbiamo anche garantire che due regine non si trovino sulla stessa diagonale. È facile vedere che due caselle (*r*, *c*) e (*r*', *c*') sono sulla stessa diagonale se e solo se la differenza delle righe è uguale alla differenza delle colonne, cioè $|r - r'| = |c - c'|$. Quindi per generare tutte le configurazioni lecite possiamo basare l'algoritmo su quello che genera le permutazioni (nel nostro caso le permutazioni delle colonne) e operare dei tagli relativamente al divieto di avere due regine sulla stessa diagonale. Più precisamente, durante la visita delle permutazioni *C* quando scegliamo la colonna *c* da mettere in *C*[*h*+1] dobbiamo verificare che per ogni posizione (*i*, *C*[*i*]), con $i \le h$, (*i*, *C*[*i*]) e (*h*+1, *c*) non siano sulla stessa diagonale, ovvero $|i - (h+1)| \neq |C[i] - c|$.

```
REGINE(n: numero regine, h: lunghezza prefisso, C: configurazione)
    IF h = n THEN
        OUTPUT C
    ELSE
        E: array di dimensione n, inizializzato a 0
        FOR i <- 1 TO h DO
            E[C[i]] <- 1
        FOR c <- 1 TO n DO
            IF E[c] = 0 THEN
                i <- 1
                WHILE i ≤ h AND |i - (h+1)| ≠ |C[i] - c| DO
                    i <- i + 1
                IF i = h+1 THEN    /* La posizione (h+1, c) è lecita */
                    C[h+1] <- c
                    REGINE(n, h+1, C)
```

La prima chiamata sarà `REGINE(n, 0, C)`.

Discussione dell'esercizio [parità]

Dobbiamo generare tutte le sequenze di lunghezza 2*n* i cui elementi appartengono a {*a*, *b*} e che contengono esattamente *n* simboli *a* (e quindi anche *n* simboli *b*). L'algoritmo è essenzialmente uguale quello che genera tutti gli *n*-sottoinsiemi di un insieme di cardinalità 2*n*.

```
PARITÀ(n, ℓ: numero di a nel prefisso, h: lung. prefisso, S: sequenza)
    IF h = 2*n THEN
        OUTPUT S
    ELSE
        IF ℓ ≥ h + 1 - n THEN
           S[h+1] <- b
           PARITÀ(n, ℓ, h+1, S)
       IF ℓ < n THEN
           S[h+1] <- a
           PARITÀ(n, ℓ+1, h+1, S)
```

La prima chiamata sarà `PARITÀ(n, 0, 0, S)`. Siccome la visita di ogni nodo dell'albero dei prefisssi costa O(1), la complessità è O($n\boldsymbol{S}(n)$) dove $\boldsymbol{S}(n)$ è il numero delle sequenze da generare che è pari al numero degli n-sottoinsiemi di un insieme di cardinalità $2n$.

Discussione dell'esercizio [numero sottosequenze crescenti]

Volendo applicare la tecnica della Programmazione Dinamica ed essendo l'input una sequenza S, la prima scelta per i sotto-problemi sono i prefissi. Un po' di riflessione ci porta a raffinare questa scelta nel seguente modo:

$$N[i] = \text{numero delle sottosequenze crescenti di } S[1 \cdots i] \text{ che terminano in } i$$

Il valore che a noi interessa, cioè il numero di tutte le sottosequenze crescenti di S, è

$$\sum_{i=1}^{n} N[i]$$

Chiaramente $N[1] = 1$. Vediamo come calcolare $N[i]$. Una sottosequenza crescente che termina in i o è composta solamente da $S[i]$ oppure è composta da una sottosequenza crescente che termina in qualche $j < i$ e da $S[i]$ e soddisfa $S[j] < S[i]$. Quindi, per ogni $i > 1$,

$$N[i] = 1 + \sum_{j:1 \leqslant j < i \wedge S[j] < S[i]} N[j]$$

A questo punto, il programma che calcola il numero di tutte le sottosequenze crescenti è facile da scrivere:

```
NSC(S: sequenza di interi distinti lunga n)
    N: tabella di dim. n
    N[1] <- 1
    nsc <- 1
    FOR i <- 2 TO n DO
        N[i] <- 1
        FOR j <- 1 TO i - 1 DO
            IF S[j] < S[i] THEN
                N[i] <- N[i] + N[j]
        nsc <- nsc + N[i]
    RETURN nsc
```

È chiaro che il programma ha complessità $O(n^2)$.

Discussione dell'esercizio [cammino]

Il problema è simile a trovare un cammino di massimo peso in un DAG, l'unica differenza è che nella rappresentazione più naturale i pesi sarebbero sui nodi invece che sugli archi. Però questa analogia ci può comunque guidare nella scelta dei sotto-problemi. Nel caso del cammino di massimo peso in un DAG un sotto-

problema è relativo al cammino di peso masssimo dal nodo di partenza verso un nodo intermedio. Possiamo definire dei sotto-problemi analoghi anche in questo caso: per ogni casella (i, j) della scacchiera,

$$M[i,j] = \text{massimo valore di un cammino da } (1,1) \text{ a } (i,j)$$

Il caso base è $M[1, 1] = v_{1,1}$. Vediamo come calcolare $M[i, j]$. Un qualsiasi cammino che parte da (1,1) e arriva alla casella (i, j) deve necessariamente passare per le due (al più) caselle adiacenti a (i, j) che con una mossa possono portare nella casella (i, j). Queste sono: (i-1, j) se $i > 1$ e (i, j-1) se $j > 1$. Quindi abbiamo che, per ogni casella (i, j) $\neq$ (1, 1),

$$M[i,j] = v_{i,j} + \begin{cases} M[i,j-1] & \text{se } i = 1 \\ M[i-1,j] & \text{se } j = 1 \\ \max\{M[i,j-1],\ M[i-1,j]\} & \text{se } i > 1 \text{ e } j > 1 \end{cases}$$

Adesso possiamo scrivere il programma che calcola la tabella M:

```
CAM(n: dim. scacchiera, V: valori delle caselle)
    M: tabella nxn
    FOR i <- 1 TO n DO
        FOR j <- 1 TO n DO
            M[i, j] <- V[i, j]
            IF i = 1 AND j > 1 THEN
                M[i, j] <- M[i, j] + M[i, j-1]
            ELSE IF j = 1 AND i > 1 THEN
                M[i, j] <- M[i, j] + M[i-1, j]
            ELSE IF i > 1 AND j > 1 THEN
                M[i, j] <- M[i, j] + max{M[i-1, j], M[i, j-1]}
    RETURN M
```

La complessità di questo algoritmo è chiaramente O(n^2). Il massimo valore di un cammino da (1, 1) a (n, n) è in M[n, n]. Per trovare un cammino di valore massimo possiamo seguire a ritroso le scelte che hanno portato al calcolo di M[n, n]:

```
CAM_SOL(n: di. scacchiera, V: valori caselle, M: tabella)
    SOL <- (n, n)   /* Lista che conterrà il cammino */
    i <- n
    j <- n
    DO
         IF i = 1 AND j > 1 THEN
             j <- j - 1
         ELSE IF j = 1 AND i > 1 THEN
             i <- i - 1
         ELSE
            IF M[i, j] = M[i-1, j] + V[i, j] THEN
                i <- i - 1
            ELSE
                j <- j - 1
         SOL <- (i, j) + SOL
    WHILE (i, j) ≠ (1, 1)
    RETURN SOL
```

L'algoritmo `CAM_SOL` ha complessità O(n) dato che ad ogni iterazione del `WHILE` uno dei due indici viene sempre decrementato. In totale la complessità dell'algoritmo (cioè prima `CAM` e poi `CAM_SOL`) per trovare il cammino di valore massimo è quindi O(n^2).

Discussione dell'esercizio [somma]

Il problema è simile al problema File o al problema dello Zaino. Bisogna infatti trovare un sottoinsieme di elementi che ottimizza una certa proprietà. La soluzione in quei casi, usando la Programmazione Dinamica, è basata su sotto-problemi relativi ai primi elementi dell'insieme. Potremmo anche derivare una soluzione passando per la memoizzazione. Consideriamo che gli elementi dell'insieme *A* siano dati come una sequenza (arbitraria) *A* = *A*[1],*A*[2],…,A[*n*]. Definiamo i sotto-problemi nel modo seguente: per ogni *i* = 1,…,*n* e *m* = 1,…,*M*,

$$T[i,m] = \begin{cases} true & \text{se esiste un sottoinsieme degli elementi } A[1 \cdots i] \text{ con somma } m \\ false & \text{altrimenti} \end{cases}$$

Il valore che ci interessa è ovviamente *T*[*n*, *M*]. I casi base sono *T*[1, *m*] = *true* se *m* = *A*[1] e *false* altrimenti. Per il calcolo di *T*[*i*, *m*] osserviamo che *T*[*i*, *m*] è *true* se e solo se uno dei seguenti tre casi accade:

- *A*[*i*] = *m*;
- esiste un sottoinsieme di *A*[1⋯*i*-1] la cui somma è *m* e questo è possibile se e solo se *T*[*i*-1, *m*] = *true*;
- esiste un sottoinsieme di *A*[1⋯*i*] con somma *m* che contiene *A*[*i*] e questo è possibile se e solo se *A*[*i*] < *m* e T[*i*-1, *m* - *A*[*i*]] = *true*.

Quindi, per ogni *i* = 2,…,*n* e *m* = 1,…,*M*,

$$T[i,m] = \begin{cases} true & \text{se } A[i] = m \\ T[i-1,m] & \text{se } A[i] > m \\ T[i-1,m] \vee T[i-1,m-A[i]] & \text{se } A[i] < m \end{cases}$$

Un programma che calcola la tabella *T* è facile (assumiamo che tutti valori di *A* siano minori od uguali a *M*):

```
SOMMA(A: array di n interi positivi, M)
    T: tabella nxM
    FOR m <- 1 TO M DO T[1, m] <- false
    T[1, A[1]] <- true
    FOR i <- 2 TO n DO
        FOR m <-1 TO M DO
            IF A[i] = m THEN
                T[i, m] <- true
            ELSE IF A[i] > m THEN
                T[i, m] <- T[i-1, m]
            ELSE
                T[i, m] <- T[i-1, m] OR T[i-1, m - A[i]]
    RETURN T[n, M]
```

La complessità è O(*nM*). La memoria potrebbe essere risparmiata perché è sufficiente mantenersi solamente le ultime due righe della tabella passando così da memoria O(*nM*) a O(*M*).