Progettazione di Algoritmi - lezione 22

Discussione dell'esercizio [senza spazi]

Data una stringa *t* di *n* caratteri, senza spazi e senza punteggiatura, vogliamo sapere se è la concatenazione di una sequenza di parole, dove le parole lecite sono determinate da una subroutine `DICT` cosicché `DICT(w)` ritorna *true* se e solo se `w` è una parola.

Inizialmente si potrebbe pensare a un algoritmo greedy che cerca il primo prefisso di *t* che è una parola, una volta trovato ripete la ricerca per i prefissi della stringa che rimane togliendo la prima parola e così via finché o rimane la stringa vuota o non trova nessun prefisso che sia una parola. Ma si vede subito che non può funzionare, basta considerare ad esempio *t* = `"lacostanteinerzia"` . L'algoritmo greedy trova le prime parole `"la"` e `"costa"` ma poi non riesce a trovare una terza parola tra `"n"` , `"nt"` , `"nte"` , ecc. Mentre ovviamente *t* è la concatenazione delle parole `"la"` , `"costante"` , `"inerzia"` .

Cerchiamo quindi di trovare un algoritmo con la Programmazione Dinamica. Essendo l'istanza del problema una sequenza, proviamo a considerare come sotto-problemi quelli relativi ai prefissi: per *i* = 1,…,*n*,

$$W[i] = \begin{cases} true & \text{se il prefisso } t[1 \cdots i] \text{ è una concatenazione di parole} \\ false & \text{altrimenti} \end{cases}$$

Il caso base è *W*[1] = `DICT` (t[1]). Vediamo come calcolare *W*[*i*] per *i* ≥ 2. Se *t*[1⋯*i*] è una concatenazione di parole, allora o è una parola o *t*[1⋯*i*] è la concatenazione di un prefisso più corto e una parola *w*, cioè *t*[1⋯*i*] = *t*[1⋯*k*]*w*, dove *k* < *i* e *t*[1⋯*k*] è una concatenazione di parole. Quindi abbiamo che

$$W[i] = \mathrm{DICT}(t[1 \cdots i]) \vee \bigvee_{k=1}^{i-1} (W[k] \wedge \mathrm{DICT}(t[k+1 \cdots i]))$$

Il programma per il calcolo è abbastanza immediato:

```
CON(t: stringa di lunghezza n)
    W: tabella di dimensione n
    W[1] <- DICT(t[1])
    FOR i <- 2 TO n DO
        k <- 1
        WHILE k < i AND NOT (W[k] AND DICT(t[k+1···i])) DO
            k <- k + 1
        IF k < i OR DICT(t[1···i]) THEN
            W[i] <- true
        ELSE
            W[i] <- false
    RETURN W[n]
```

La complessità è O(n^2) dato che l'*i*-esima iterazione del `FOR` può richiedere tempo O(*i*). Si osservi però che se c'è un limite *L* sulla lunghezza delle parole, allora la complessità può essere ridotta a O(*nL*) perché il `WHILE` più interno può evitare di esaminare i valori di *k* tali che *i* - *k* > *L*, cioè *k* può iniziare da max{1, *i* - *L*}. In particolare se *L* è una costante, ad esempio 30, la complessità diventa lineare in *n*, cioè O(*n*).

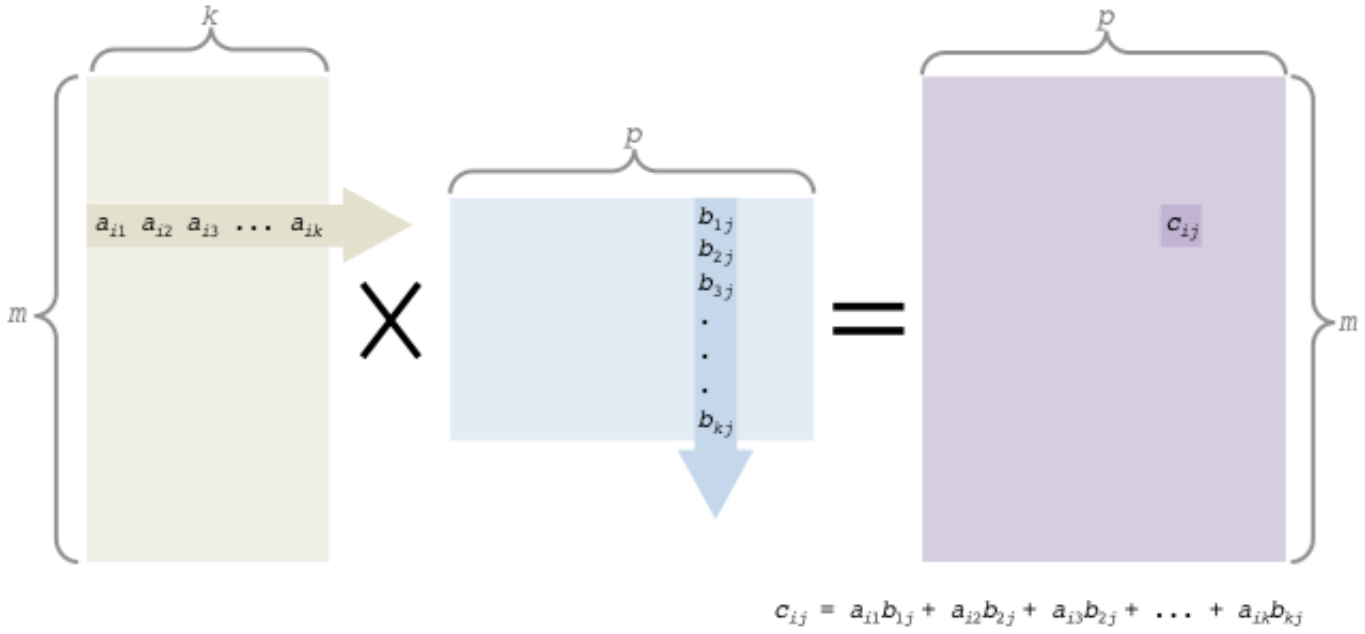
Una volta calcolata la tabella *W*, per ricostruire anche la sequenza di parole basta trovare, procedendo a ritroso dalla stringa *t* verso i suoi prefissi via via più corti, la parola suffisso *w* tale che *t*[1⋯*i*] = *t*[1⋯*k*]*w* e *W*[*k*] = *true* o *k* = 0.

```
CON_SOL(t: stringa di lunghezza n, W: tabella)
    SOL <- lista vuota
    i <- n
    WHILE i > 0 DO
        k <- 1
        WHILE k < i AND NOT (W[k] AND DICT(t[k+1···i])) DO
            k <- k + 1
        IF k < i THEN
            SOL <- t[k+1···i] + SOL   /* Aggiunge in testa alla lista */
            i <- k
        ELSE
            SOL <- t[1···i] + SOL
            i <- 0
    RETURN SOL
```

Assumiamo di chiamare il programma `CON_SOL` solo se `W[n]` è *true*. La complessità della ricostruzione della sequenza di parole è O(*n*) dato che ad ogni iterazione del `WHILE` più interno è come se fosse decrementato anche l'indice *i*.

Prodotti di matrici

Data una sequenza di matrici $M_1,\ldots,M_n$ vogliamo calcolarne il prodotto, cioè la matrice $M_1 \times M_2 \times \cdots \times M_n$, minimizzando il numero totale di moltiplicazioni di scalari. Il prodotto è l'usuale prodotto righe per colonne: se A è una matrice $m \times k$ e B è una matrice $k \times p$, il loro prodotto $C = A \times B$ è una matrice $m \times p$ e l'elemento c_{ij} è il prodotto della i-esima riga di A per la j-esima colonna di B. Il prodotto $A \times B$ richiede quindi mkp moltiplicazioni di scalari.

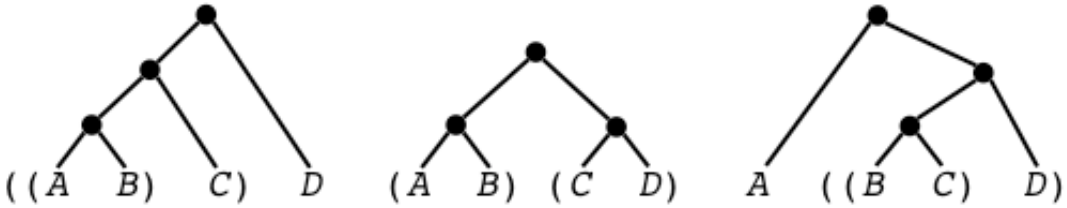


Sappiamo che il prodotto di matrici è un'operazione associativa, cioè $A \times (B \times C) = (A \times B) \times C$ (ma non è commutativa), quindi ci sono molti modi diversi di calcolare il prodotto della sequenza di matrici e ognuno può richiedere un numero diverso di moltiplicazioni. Se le n matrici fossero tutte quadrate, ad es. $m \times m$, allora i diversi modi richiederebbero tutti lo stesso numero di moltiplicazioni, cioè nm^3. Ma se le matrici sono rettangolari ci potrebbero essere modi molto più convenienti di altri.

Consideriamo un esempio con quattro matrici A, B, C e D di dimensioni, rispettivamente, 50×20, 20×1, 1×10, 10×100. Volendo minimizzare il numero di moltiplicazioni per calcolare $A \times B \times C \times D$ potremmo seguire un'approccio greedy e iniziare scegliendo la coppia di matrici il cui prodotto costa di meno. Vediamo, $A \times B$ costa 50×20×1 = 1000, $B \times C$ costa 20×1×10 = 200 e $C \times D$ costa 1×10×100 = 1000. Quindi scegliamo di calcolare prima di tutto $B \times C$, costo 200. Adesso abbiamo le tre matrici A, $(B \times C)$ e D di dimensioni 50×20, 20×10, 10×100 e tra le due coppie

scegliamo di calcolare quella che minimizza il numero di moltiplicazioni che è $A\times(B\times C)$ con costo 50×20×10 = 10000. L'ultimo prodotto $(A\times(B\times C))\times D$ ha costo 50×10×100 = 50000 e in totale abbiamo eseguito 200 + 10000 + 50000 = 60200 moltiplicazioni. Ma c'è un altro modo di fare il prodotto che è molto più conveniente. Infatti, la parentesizzazione $(A\times B)\times(C\times D)$ ha costo 50×20×1 = 1000, 1×10×100 = 1000 e 50×1×100 = 5000, in totale 1000 + 1000 + 5000 = 7000. Quasi un decimo del costo di quella ottenuta seguendo l'approccio greedy.

A questo punto dovrebbe essere abbastanza evidente che trovare il modo di parentesizzare il prodotto della sequenza di matrici che minimizza il numero di moltiplicazioni non è un problema così facile. D'altronde un algoritmo esaustivo dovrebbe prendere in cosiderazione tutte le possibili parentesizzazioni. Quante sono? Ogni parentesizzazione corrisponde a un albero binario pieno (cioè, ogni nodo interno ha due figli) le cui foglie sono le matrici e viceversa.



Si noti che i nodi interni dell'albero corrispondono ai prodotti tra coppie di matrici effettuati secondo la parentesizzazione. Il numero di tutti i possibili modi di parentesizzare il prodotto di n matrici (o equivalentemente il numero di tutti gli alberi binari pieni con n-1 nodi interni) è pari al numero di Catalan C_{n-1}, dove:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Quindi sono tantissimi perché crescono esponenzialmente con n:

$$C_n \geqslant \frac{4^n}{(n+1)^2}$$

Vediamo come possiamo trovare un algoritmo di Programmazione Dinamica che risolve il problema. Ricapitolando, abbiamo una sequenza di matrici $M_1,\ldots,M_n$ e vogliamo calcolarne il prodotto $M_1\times M_2\times\cdots\times M_n$ in modo da minimizzare il numero totale di moltiplicazioni. Il problema è conosciuto con il nome di *Matrix Chain Multiplication*. Per ogni $i = 1,\ldots,n$, sia $m_i\times m_{i+1}$ la dimensione della matrice M_i. Si noti che affinchè il prodotto sia ammissibile il numero di colonne della matrice M_i deve essere uguale al numero di righe della matrice M_{i+1}. Come al solito, inizialmente ci occuperemo di calcolare il numero minimo di moltiplicazioni e poi vedremo come ottenere anche la parentesizzazione ottima. Siccome abbiamo a che fare con una sequenza, viene naturale provare a scegliere come sotto-problemi per la Programmazione Dinamica i prefissi della sequenza di matrici:

$$P[i] = \text{minimo numero di moltiplicazioni per il prodotto } M_1 \times M_2 \times \cdots \times M_i$$

Il caso base non crea problemi. Ma come facciamo a calcolare $P[i]$ conoscendo i valori dei prefissi più piccoli? Conosciamo $P[i-1]$ che è il numero minimo di moltiplicazioni per il prodotto $M_1\times\cdots\times M_{i-1}$. Il prodotto di tale matrice che ha dimensioni $m_1\times m_i$ con la matrice M_i ha costo $m_1 m_i m_{i+1}$. Quindi possiamo sicuramente affermare che

$$P[i] \leqslant P[i-1] + m_1 m_i m_{i+1}$$

Ma non possiamo affermare che vale anche l'uguaglianza, in generale. Infatti, abbiamo considerato un solo modo di parentesizzare il prodotto delle prime i matrici relativamente all'i-esima matrice:

$$(M_1 \times \cdots \times M_{i-1}) \times M_i$$

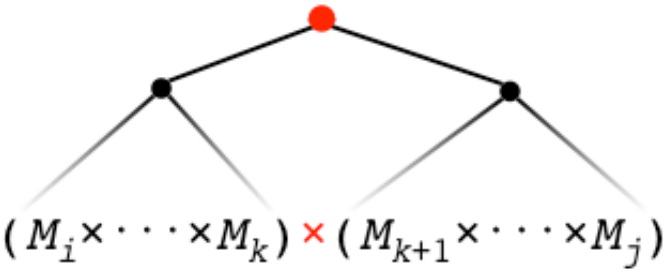
dove la parentesizzazione del prodotto delle prime i-1 matrici è quello ottimo. Ma ci sono molti altri modi:

$$(M_1 \times \cdots \times M_{i-2}) \times (M_{i-1} \times M_i)$$
$$(M_1 \times \cdots \times M_{i-3}) \times (M_{i-2} \times M_{i-1} \times M_i)$$
$$\cdots$$
$$(M_1 \times M_2) \times (M_3 \times \cdots \times M_i)$$
$$M_1 \times (M_2 \times \cdots \times M_i)$$

Per determinare i costi ottimi di tutti questi tipi di parentesizzazione non ci bastano i $P[j]$ che ci danno solamente i costi ottimi per i prodotti $M_1 \times \cdots \times M_j$ per $j = 1,\ldots,i-1$. Mancano i costi ottimi per i prodotti del tipo $M_{j+1} \times \cdots \times M_i$ per $j = 1,\ldots,i-2$. Dobbiamo quindi ampliare l'insieme dei sotto-problemi comprendendo i prodotti relativi a tutte le sottosequenze di matrici consecutive: per $1 \le i \le j \le n$,

$$P[i,j] = \text{minimo numero di moltiplicazioni per il prodotto } M_i \times \cdots \times M_j$$

Ovviamente il valore che ci interessa è $P[1, n]$. I casi base sono $P[i, i] = 0$, per $i = 1,\ldots,n$. Vediamo come calcolare $P[i, j]$. Ogni parentesizzazione del prodotto $M_i \times \cdots \times M_j$ ha un ultimo prodotto, cioè quello che corrisponde alla radice dell'albero:



Relativamente alla figura, il costo ottimo per questo tipo di parentesizzazione è dato da $P[i, k] + P[k+1, j] + m_i m_{k+1} m_{j+1}$. Per calcolare $P[i, j]$ è quindi sufficiente prendere il minimo dei costi di questi modi di parentesizzare il prodotto della sottosequenza da i a j:

$$P[i,j] = \min\{P[i,k] + P[k+1,j] + m_i m_{k+1} m_{j+1} \mid k = i,\ldots,j-1\}$$

Possiamo quindi calcolare $P[i, j]$ in funzione dei valori ottimali per sottosequenze più corte. Il programma che esegue il calcolo della tabella P deve fare attenzione a calcolare i valori in ordine di lunghezza delle corrispondenti sottosequenze, cioè prima le sottosequenze di lunghezza 1, poi quelle di lunghezza 2 e così via.

```
MCM(m: array delle dimensioni delle n matrici)
    P: tabella nxn
    FOR i <- 1 TO n DO P[i, i] = 0
    FOR s <- 1 TO n DO
       FOR i <- 1 TO n - s DO
           j <- i + s
           P[i, j] <- P[i, i] + P[i+1, j] + m[i]*m[i+1]*m[j+1]
           FOR k <- i+1 TO j-1 DO
               IF P[i, k] + P[k+1, j] + m[i]*m[k+1]*m[j+1] < P[i, j] THEN
                   P[i, j] <- P[i, k] + P[k+1, j] + m[i]*m[k+1]*m[j+1]
    RETURN P[1, n]
```

Il calcolo di ogni elemento delle tabella richiede tempo proporzionale alla lunghezza della relativa sottosequenza. Siccome ci sono ordine di n^2 sottosequenze di lunghezza almeno $n/2$, la complessità è $\Theta(n^3)$.

Una volta calcolata la tabella P si può effettuare il prodotto delle matrici secondo la parentesizzazione ottima data dalla tabella. Conviene farlo tramite un algoritmo ricorsivo che essenzialmente fa una visita in preordine dell'albero della parentesizzazione. L'algoritmo qui sotto descritto ritorna la matrice prodotto delle matrici della sottosequenza di input. Usa una subroutine `PROD(A, B)` che ritorna la matrice prodotto delle matrici di input `A` e `B`.

```
MCM_SOL(M: array delle matrici, m: array delle dimensioni, P: tabella, i,j: indici)
    IF i = j THEN
        RETURN M[i]
    ELSE
        k <- i
        WHILE P[i, j] < P[i, k] + P[k+1, j] + m[i]*m[k+1]*m[j+1] DO
            k <- k + 1
        A <- MCM_SOL(M, m, P, i, k)   /* Matrice prodotto delle matrici da i a k
        B <- MCM_SOL(M, m, P, k+1, j) /* Matrice prodotto delle matrici da k+1 a j */
        C <- PROD(A, B)     /* Matrice prodotto delle matrici da i a j */
        RETURN C
```

La prima chiamata è `MCM_SOL(M, m, P, 1, n)` che ritornerà la matrice prodotto della sequenza delle n matrici.

La complessità, escluso il costo dei prodotti delle matrici, è $O(n^2)$ perché l'algoritmo visita un albero binario di n - 1 nodi interni e la chiamata per ogni nodo interno impiega tempo al più $O(n)$.

Esercizi

Esercizio [massima sottosequenza crescente]

Dato un array di n interi A, vogliamo trovare una sottosequenza di A (quindi elementi anche non consecutivi) di valori crescenti di lunghezza massima. In altre parole, vogliamo trovare una massima sequenza di indici $1 \le i_1 < i_2 < \cdots < i_k \le n$ tale che

$$A[i_1] < A[i_2] < \cdots < A[i_k]$$

Descrivere un algoritmo che in tempo $O(n^2)$ trova una massima sottosequenza crescente.

Esercizio [gettoni]

Consideriamno il seguente gioco. Su un tavolo vi sono n gettoni. Due giocatori, a turno, possono prelevare dal tavolo 1, 3 o 4 gettoni. Perde il giocatore che è costretto a prelevare l'ultimo gettone dal tavolo. Vogliamo sapere se il giocatore che comincia il gioco ha una strategia vincente, cioè può vincere qualunque siano le mosse dell'avversario. Ad esempio, per $n = 5$ il primo giocatore ha una strategia vincente: preleva 4 gettoni, il suo avversario sarà costretto poi a prelevare l'unico gettone rimasto facendolo vincere. Per $n = 3$ il primo giocatore non ha una strategia vincente. Può infatti prelevare 3 o 1 gettone nel primo caso fa vincere l'avversario, nel secondo lascia all'avversario 2 gettoni, l'avversario ne preleva 1 e vince in quanto nella mossa successiva il primo giocatore non può che prelevare l'ultimo gettone rimasto.

Descrivere un algoritmo che, dato n, in tempo $O(n)$ ci dice se il primo giocatore ha o meno una strategia vincente.

Esercizio per casa [palindroma]

Descrivere un algoritmo efficiente che data una stringa s di n caratteri trova la più lunga sottostringa di s che sia palindroma. Si ricorda che una stringa è palindroma se rimane la stessa se letta da sinistra verso destra o da destra verso sinistra.

Soluzioni

Di seguito presentiamo alcune possibili soluzioni degli esercizi proposti.

Discussione dell'esercizio [massima sottosequenza crescente]

Usando la Programmazione Dinamica e considerando come sotto-problemi quelli relativi ai prefissi dell'array A,

proviamo a definire:

$$S[i] = \text{massima lunghezza di una sottosequenza crescente di } A[1 \cdots i]$$

Il caso base è facile, $S[1] = 1$. Per il calcolo di $S[i]$ dobbiamo considerare due casi o la massima sottosequenza crescente non comprende $A[i]$ o lo comprende. Nel primo caso $S[i] = S[i\text{-}1]$. Ma nel secondo caso dovremmo considerare la più lunga sottosequenza crescente di $A[1 \cdots i\text{-}1]$ il cui ultimo elemento è minore di $A[i]$ e questa informazione non è fornita dalla tabella che abbiamo definito. Dobbiamo allora cambiare la definizione in modo analogo a come abbiamo fatto in atri problemi relativi alle sequenze:

$$S[i] = \text{massima lunghezza di una sottosequenza crescente di } A[1 \cdots i] \text{ il cui ultimo elemento è } A[i]$$

La lunghezza della massima sottosequenza crescente A sarà data da

$$\max\{S[i] \mid i = 1, \ldots, n\}$$

Il caso base è sempre $S[1] = 1$. Il calcolo di $S[i]$ ora è facile: per $i \geq 2$

$$S[i] = \begin{cases} 1 & \text{se } A[i] \text{ è il minimo valore di } A[1 \cdots i] \\ \max\{S[j] \mid 1 \leqslant j < i \wedge A[j] < A[i]\} + 1 & \text{altrimenti} \end{cases}$$

È parimenti facile scrivere il programma che calcola la lunghezza della massima sottosequenza crescente:

```
MSC(A: array di n interi)
    S: tabella di dimensione n
    S[1] <- 1
    max <- 1
    FOR i <- 2 TO n DO
        S[i] <- 1
        FOR j <- 1 TO i-1 DO
            IF A[j] < A[i] AND S[j] + 1 > S[i] THEN
                S[i] <- S[j] + 1
        IF S[i] > max THEN max <- S[i]
    RETURN max
```

La complessità è chiaramente $O(n^2)$. Per ricostruire la massima sottosequenza tramite la tabella S, basta come al solito seguire a ritroso le scelte che hanno portato a calcolare la lunghezza massima:

```
MSC_SOL(A: array di n interi, m: lunghezza massima, S: tabella)
    SOL <- lista vuota   /* Conterrà la lista degli indici della sottosequenza */
    i <- 1
    WHILE S[i] < m DO
        i <- i + 1
    WHILE i > 0 DO
        SOL <- i + SOL   /* Aggiunge i in testa alla lista */
        j <- i - 1
        WHILE j > 0 AND NOT (A[j] < A[i] AND S[j] + 1 = S[i]) DO
            j <- j - 1
        i <- j
    RETURN SOL
```

La complessità è $O(n)$.

Discussione dell'esercizio [gettoni]

Se a un certo punto del gioco ci sono k gettoni sul tavolo, il primo giocatore ne può prelevare 1, 3 o 4. Dopo la

mossa, sul tavolo ci saranno *k*-1, *k*-3 o *k*-4 gettoni e tocca al secondo giocatore. Se in almeno una delle tre situazioni, *k*-1, *k*-3 o *k*-4 gettoni, il secondo giocatore non ha una strategia vincente, allora il primo giocatore ha una strategia vincente che inizia facendo una mossa che porta in una delle situazioni che è perdente per il secondo giocatore. Se invece per ognuna delle tre situazioni di gioco il secondo giocatore ha una strategia vincente, significa che il primo giocatore dalla situazione con *k* gettoni non ha una strategia vincente. Ciò detto è facile capire quali sono i sotto-problemi da considerare:

$$V[k] = \begin{cases} true & \text{se con } k \text{ gettoni il giocatore che inizia ha un strategia vincente} \\ false & \text{altrimenti} \end{cases}$$

I casi base sono:

$$\begin{aligned} V[1] &= false && \text{(perde, essendo costretto a prelevare 1)} \\ V[2] &= true && \text{(preleva 1 e l'avversario perde)} \\ V[3] &= false && \text{(perde sia che ne preleva 1 che 3)} \\ V[4] &= true && \text{(preleva 1 e lascia l'avversario nella situazione perdente da 3)} \end{aligned}$$

Per $k \geq 5$, abbiamo:

$$V[k] = \neg V[k-1] \vee \neg V[k-3] \vee \neg V[k-4]$$

Il programma che determina se il primo giocatore ha una strategia vincente è immediato:

```
VINCE(n: intero maggiore di 4)
    V: tabella di dimensione n
    V[1] <- false
    V[2] <- true
    V[3] <- false
    V[4] <- true
    FOR k <- 5 TO n DO
        IF NOT V[k-1] THEN V[k] <- true
        ELSE IF NOT V[k-3] THEN V[k] <- true
        ELSE IF NOT V[k-4] THEN V[k] <- true
        ELSE V[k] <- false
    RETURN V[n]
```

La complessità è chiaramente O(*n*) e se ci interessa solamente sapere se c'è una strategia vincente possiamo risparmiare la memoria della tabella *V* usando solo memoria costante anziché O(*n*).

Conoscendo la tabella *V* possiamo seguire la strategia vincente per ogni situazione con $k \leq n$ gettoni tale che *V*[*k*] = *true*. Infatti, supponendo che *V*[*n*] = *true*, ci deve essere uno tra *V*[*n*-1], *V*[*n*-3] o *V*[*n*-4] che deve essere *false*. Se questo è *V*[*n*-*i*], preleviamo *i* gettoni. Non importa quale mossa farà l'avversario, quando ritoccherà a noi, sul tavolo ci saranno *k* gettoni con *V*[*k*] = *true* e potremmo così ripetere quello che abbiamo fatto nella prima mossa vedendo quale tra *V*[*k*-1], *V*[*k*-3] o *V*[*k*-4] è *false* e fare la mossa conseguente. E così via fino a che l'avversario sarà costretto a prelevare l'ultimo gettone e noi vinciamo.