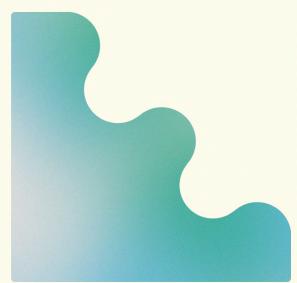


Analisi e Progetto di Algoritmi

879276 * A.A. 2023/2024



haru
879276



Analisi e Progetto di Algoritmi

Appunti del Corso ≈ Anno Accademico 2023-2024 ≈ 879276

25.09.2023

La **programmazione dinamica**, argomento principale del corso, è una tecnica algoritmica per risolvere i problemi basata sulla ricorsione

Ripasso - Notazione

Sia una funzione T che ha come dominio N , il cui singolo valore rappresenta la grandezza dell'input, e dove $T(n)$ rappresenta il costo computazionale dell'algoritmo

Usiamo $T(n)$ per valutare il costo di un algoritmo

$\Theta(T(n))$ è una funzione $f(n)$ tale che da un certo valore in poi, $c_1 g(n) \leq f(n) \leq c_2 g(n)$

Ovvero, $f(n)$ è infinitamente limitata da $g(n)$

$f(n)$ si dice invece $O(g(n))$ se $g(n)$ la limita superiormente

$f(n)$ si dice invece $\Omega(g(n))$ se $g(n)$ la limita inferiormente

Ripasso - Tempo di Calcolo

Il tempo di calcolo di un algoritmo **iterativo** si calcola nel seguente modo:

$x = 1$	1
for $i = 2$ to n	$n - 2 + 1 + 1$
for $j = 1$ to i	$\sum_{i=2}^n [i - 1 + 1 + 1] = \sum_{i=2}^n [i + 1] = (n(n + 1))/2 + (n - 1)$
$x = x + i + j$	$\sum_{i=2}^n = (n(n + 1))/2 - 1$
return x	1

$$T(n) = 1 + n + ((n(n + 1))/2 + (n - 1)) + ((n(n + 1))/2 - 1) + 1 = \Theta(n^2)$$

Il tempo di calcolo di un algoritmo **ricorsivo** si calcola nel seguente modo:

```
fatt(n)                                T(n) = 2           se n = 0  
if n = 0 return 1                         2 + T(n - 1)   se n > 0  
return n * fatt(n - 1)
```

$$T(n) = 2 + T(n - 1) = 2 + (2 + T(n - 2)) = 2 + 2 + 2 + \dots T(0) = 2 + 2 + 2 + \dots + 2 = 2 + 2n$$

26.09.2023

$X = \langle x_1, \dots, x_n \rangle$ sequenza ordinata di n simboli

$Z = \langle z_1, \dots, z_k \rangle$ è una sottosequenza di X se e solo se esiste una sequenza $I \subset \{1, \dots, n\}$ strettamente crescente di indici tali che per ogni j in I , $z_i = x_{I_j}$

$X = \langle a, b, c, b, d, a, b \rangle \quad n = 7$

$Z = \langle b, c, d, b \rangle$ è una sottosequenza di X poiché esiste $I = \langle 2, 3, 5, 7 \rangle$

Programmazione Dinmica - LCS

LCS - Longest Common Subsequence

$X = \langle x_1, \dots, x_m \rangle$ Sequenze di simboli sullo stesso alfabeto

$Y = \langle y_1, \dots, y_n \rangle$

Z sottosequenza sia di X che di Y tale che $|Z| = \max(|W|)$, con W sottosequenza sia di X che di Y

Algoritmo Brute-Force $\Omega(2^m)$

```
for i=1 to  $2^m$   
    genera i-esima sequenza W di X  
    se W è sottosequenza di Y  
        se W è più lunga della sequenza Z più lunga trovata fin ora  
            Z = W
```

Il brute-force è un algoritmo combinatorio dalle basse prestazioni, è necessario scrivere la risoluzione in termini ricorsivi per ottimizzare la complessità dell'algoritmo

Per farlo, è necessario individuare un **sottoproblema** generico

Il **sottoproblema** che rappresenta il passo ricorsivo per l'LCS sarà trovare una sottosequenza Z comune ad un prefisso di X di lunghezza i e un prefisso di Y di lunghezza j . Ogni sottoproblema è quindi caratterizzato dalla coppia (i, j)

I **casi base** della ricorsione sono tutte le coppie (i, j) dove $i = 0$ o $j = 0$

In questi casi $Z = \langle \rangle$ è sempre sottosequenza di almeno o X_i o Y_j , o entrambi

$(i, j) \quad i = 0 \text{ or } j = 0$

$Z = \langle \rangle$

$i = 0$

Ci sono dunque $m + n + 1$ coppie che rientrano nel caso base, le restanti sono $m \cdot n$

In totale le coppie sono $(m+1) \cdot (n+1)$

Per la generica coppia (i, j) con $i, j > 0$, la soluzione Z dipende dal simbolo corrente:

27.09.2023

$(i, j) \quad i, j > 0$

se $X_i = Y_j$

$Z = f(i-1, j-1) \mid X_i$

altrimenti

$Z = \max(f(i-1, j), f(i, j-1))$

Data la coppia di valori (i, j) , in generale, per poterlo risolvere nel passo ricorsivo ho bisogno di avere la soluzione a tre sottoproblemi: $(i-1, j)$ $(i, j-1)$ $(i-1, j-1)$

LCS-Ricorsivo (X, Y, i, j)

```

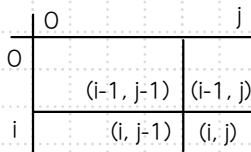
if i = 0 or j = 0
    return <>
if X[i] = Y[j]
    return LCSR(X, Y, i-1, j-1) | Xi
A = LCSR(X, Y, i-1, j)
B = LCSR(X, Y, i, j-1)
if A.len > B.len
    return A
return B

```

Equazione ricorsiva

$$\begin{aligned}
f(i, j) = & \quad f(i-1, j-1) | X_i & \text{se } X_i = Y_j \\
& f(i-1, j) & \text{se } |f(i-1, j)| > f(i, j-1) \\
& f(i, j-1) & \text{se } |f(i, j-1)| \geq f(i-1, j)
\end{aligned}$$

Per costruire la soluzione iterativa dinamica, ragioniamo in modo **bottom-up**, partendo dal sottoproblema più piccolo



I sottoproblemi necessari al problema (i, j) sono quelli a lui adiacenti
I case base si trovano lungo tutta la prima riga e la prima colonna

Le soluzioni sono salvate in memoria in un array bidimensionale S della forma sopra, di dimensioni $m \times n$, dove $S[i, j]$ rappresenta la soluzione $Z(i, j)$

LCS-Iterativo (X, Y)

```

m = X.len
n = Y.len
for j = 0 to n
    S[0, j] = <>
for i = 0 to m
    S[i, 0] = <>

```

```

for i = 1 to m
    for j = 1 to n
        if X[i] = Y[j]
            S[i, j] = S[i-1, j-1] | X[i]
        else
            if S[i-1, j].len > S[i, j-1].len
                S[i, j] = S[i-1, j]
            else
                S[i, j] = S[i, j-1]

```

La proprietà della sottostruttura ottima dell'LCS è:

$$i = 0 \text{ o } j = 0 \quad Z(i, j) = <>$$

$$i > 0 \text{ e } j > 0$$

$$X_i = Y_j \quad Z(i, j) = Z(i-1, j-1) | X_i$$

$$X_i \neq Y_j \quad Z(i, j) = Z(i-1, j) \text{ se } |Z(i-1, j)| > |Z(i, j-1)|$$

$$Z(i, j-1) \text{ se } |Z(i-1, j)| \leq |Z(i, j-1)|$$

Si dice **sottostruttura** in quanto per ogni problema è immediato visualizzare e memorizzare la soluzione dei sottoproblemi ad esso adiacenti

Riformulando il problema e chiedendo come soluzione la **lunghezza** della sottosequenza comune più lunga anziché la sottosequenza stessa, si pesa di meno sulla memoria in quanto nella matrice S vengono memorizzate solo le lunghezze e non le sequenze intere

L-LCS (X, Y, i, j)

```
if i = 0 o j = 0
    return 0
if X[i] = Y[j]
    return L-LCS(X, Y, i-1, j-1) + 1
return max(L-LCS(X, Y, i-1, j), L-LCS(X, Y, i, j-1))
```

L-LCS-Iterativo (X, Y)

$m = X.\text{len}$ $n = Y.\text{len}$ for $j = 0$ to n $c[0, j] = 0$ for $i = 0$ to m $c[i, 0] = 0$	for $i = 1$ to m for $j = 1$ to n if $X[i] = Y[j]$ $c[i, j] = c[i-1, j-1] + 1$ else $c[i, j] = \max(c[i-1, j], c[i, j-1])$
--	---

Per ricostruire la sequenza posso salvarmi una lista di puntatoti (i, j) usati per calcolare la lunghezza massima

02.10.2023 - Esercitazione: Weighted Interrupt Scheduling

i	p(i)	v _i
1	0	10
2	0	2
3	1	8
4	2	1
5	1	1
6	4	3

Determinare un sottoinsieme di attività eventualmente compatibili che massimizza il valore totale. Nell'esempio, la soluzione sarebbe {1, 3, 6}

Istanza $n \in N$, $X_n = \{1, \dots, n\}$ insieme di n attività

Per ogni $i \in X_n$	si	tempo di inizio dell'attività i
	fi	tempo di fine dell'attività i
	vi	valore dell'attività i

dove	$v(): P(X_n) \rightarrow R$	$\forall A \subseteq X_n$
	$v(A) = \sum v_i$	$\forall i \in A \quad \text{se } A \neq \emptyset$
	0	$\text{se } A = \emptyset$

$\text{comp}(A) = \text{true}$ se A contiene attività mutualmente **compatibili**
 false altrimenti

compatibili: $\forall i, j \in A \text{ con } i \neq j \quad [s_i, f_i] \cap [s_j, f_j] = \emptyset$

Il sotto problema è trovare una soluzione ad un **sottoinsieme** X_i di X_n

$$X_i = \{1, \dots, i\} \text{ con } 1 \leq i \leq n$$

$p(i)$ è la precedente attività di indice più alto compatibile con quella presa in considerazione attualmente (i)

$$p(i) = \max\{j \mid \text{t.c. } j < i \text{ e } j \text{ compatibile con } i\}$$

$$p(i) = 0 \text{ se } \{j \mid \text{t.c. } j < i \text{ e } j \text{ compatibile con } i\} = \emptyset$$

I **casi base** della ritorsione avvengono quando il sottoinsieme S ha uno o zero elementi

$$X_1 = \{1\}$$

$$X_0 = \emptyset$$

$$S_1 = \{1\}$$

$$S_0 = \emptyset$$

$$\text{opt}(1) = v(S_1) = v_1$$

$$\text{opt}(0) = 0$$

Dato un sottoproblema $i > 1$, il **passo ricorsivo** avviene in questo modo:

$S_i =$	$S_p(i) \cup \{i\}$	se $i \in S_i$	(Questa non è l'equazione di ricorrenza ma la formalizzazione del sotto problema visto che le condizioni non sono algoritmicamente verificabili se non a posteriori)
	S_{i-1}	se $i \notin S_i$	

↓

$S_i =$	$S_p(i) \cup \{i\}$	se $\text{opt}(p(i) + v_i) \geq \text{opt}(i-1)$	Proprietà della sottostruttura ottima
	S_{i-1}	altrimenti	

Per la soluzione iterativa dinamica avrò bisogno di un array mono dimensionale M che contenga i valori di $\text{OPT}()$

OPT-D (n)

$$\begin{aligned} M[0] &= 0 \\ M[1] &= v_1 \quad S[1] = \{1\} \end{aligned}$$

for $i = 2$ to n

$$M[i] = \max (M[p(i)] + v_i, M[i-1])$$

$$S[i] = S[p(i)] \cup S[i]$$

$$S[i-1]$$

È necessario però ordinare le attività per tempo di fine prima di eseguire OPT-D

09.10.2023 - Esercitazione Hateville

Si ha un certo numero di case su una vita, ogni proprietario è disposto a fare una donazione di una certa quantità, dobbiamo trovare un sottoinsieme S di case che massimizza la donazione totale, tenendo conto che il sottoinsieme non deve contendere case adiacenti

Salviamo in un vettore $d[]$ i pesi delle varie case; risolviamo i sottoproblemi come sottoinsiemi X di case. Il caso base di formula come segue

$$\begin{aligned} X_1 &= \{1\} & S_1 &= \{1\} & D_1 &= d_1 \\ X_0 &= \emptyset & S_0 &= \emptyset & D_0 &= 0 \end{aligned}$$

Assumendo di conoscere il risultato dei sottoproblemi, il caso passo si formula come segue:

$$S_i = \begin{cases} S_{i-2} \cup \{i\} & \text{se } i \in S_i \\ S_{i-1} & \text{se } i \notin S_i \end{cases}$$

Implementando le condizioni in forma algoritmica, l'equazione di ricorrenza diventa:

$$S_i = \begin{cases} S_{i-2} \cup \{i\} & \text{se } D(S_{i-2} \cup \{i\}) \geq D(S_{i-1}) \\ S_{i-1} & \text{altrimenti} \end{cases}$$

$$D(S_i) = \max(D(S_{i-2}) + d_i, D(S_{i-1})) = \text{opt}(i) = \text{PD-HV}$$

Dove D è una funzione che dato un sottoinsieme X_i ritorna il valore totale del sottoinsieme

$\text{PD-HV}(n)$

$$M[0] = 0$$

$$M[1] = d[1]$$

for $i = 2$ to n

$$M[i] = \max(M[i-2] + d[i], M[i-1])$$

Dimostrazione soluzione

Dato il problema HV, si consideri il sottoproblema i -esimo di HV con $i \geq 2$, siamo S_0, S_1, \dots, S_{i-1} le soluzioni dei sottoproblemi $0, 1, \dots, i-1$, allora vale

$$S_i = \begin{cases} S_{i-2} \cup \{i\} & \text{se } i \in S_i \\ S_{i-1} & \text{se } i \notin S_i \end{cases}$$

Caso $i \notin S_i$

Se per assurdo S_{i-1} non è la soluzione del problema i -esimo, allora $D(S_i) > D(S_{i-1})$

Dunque $S_i \subseteq \{1, \dots, i-1\}$, e $\text{comp}(S_i) = \text{true}$, e quindi S_{i-1} non può essere soluzione del problema $i-1$, contraddizione con le ipotesi

Caso $i \in S_i$

Se per assurdo $S_{i-2} \cup \{i\}$ non è soluzione del problema i -esimo, allora $D(S_i) > D(S_{i-2}) + d_i$

Dunque $S_i = S' \cup \{i\}$, $\text{comp}(S') = \text{true}$ e $\text{comp}(S_i) = \text{true}$, e $S' \subseteq \{1, \dots, i-2\}$, e quindi

$D(S') + d_i > D(S_{i-2}) + d_i$, e quindi $D(S') > D(S_{i-2})$, ma allora $D(S_{i-2})$ non può essere soluzione del problema $i-2$, contraddizione con le ipotesi

16.10.2023

Stringhe Palindroma

Sia S una stringa e $F(S)$ il numero di caratteri minimi per rendere S palindroma, indichiamo con $S(i,j)$ il sottoproblema caratterizzato dalla sottostringa di S che parte dal carattere di indice i al carattere di indice j

$$S = \text{stringa vuota} \quad i > j \quad F(i,j) = 0$$

$$S = aij \quad i = j \quad F(i, j) = 0$$

$$S = ai \dots aj \quad i < j$$

se $a_i = a_j \quad F(i, j) = F(i+1, j-1)$

se $a_i \neq a_j \quad F(i, j) = \min(F(i, j-1), F(i+1, j)) + 1$

Costruendo le soluzioni in modo bottom-up, utilizziamo una matrice quadrata M con indici i e j . In questo caso, le celle che rappresentano i casi base si trovano sulla diagonale principale della matrice ($i = j$) e la metà ad essa sottostante ($i > j$)

PAL(S)

```
n = length(S)
for j = 1 to n
    for i = j to n
        M[i, j] = 0
for i = n-1 down to 1
    for j = i+1 to n
        if S[i] = S[j]
            M[i, j] = M[i+1, j-1]
        else
            M[i, j] = min(M[i+1, j], M[i, j-1])
```

Sottovettore di Valore Massimo

Supponiamo di avere un vettore V di interi (positivi o negativi), determinare un sotto vettore di valore massimo (la cui somma del valore degli elementi è massima)

Definiamo il sottoproblema i -esimo come il sottovettore V_i di lunghezza i

$$V_1 \quad i = 1$$

$$S_1 = V_1$$

$$OPT(i) = v_1$$

$$V_i \quad i > 1$$

$$S_i = V_i$$

se $v_i < OPT(i-1) + v_i$

$$S_i = S(i-1) \cup V_i$$

se $OPT(i-1) + v_i \geq v_i$

$$OPT(i) = \max(v_i, OPT(i-1) + v_i)$$

PD-OPT(n)

$$M[1] = v[1]$$

$$\max = M[1]$$

for $i = 2$ to n

$$M[i] = \max(M[i-1] + v[i], v[i])$$

if $\max < M[i]$

$$\max = M[i]$$

La soluzione al problema non vincolato è quella il cui valore in $M[]$ è massimo

17.10.2023

Knapsack 0-1

Si ha un numero m di offerti, ogni oggetto ha un valore v e un ingombro w , e uno zaino con capacità C . L'obiettivo è massimizzare il valore degli oggetti da mettere nello zaino

i	v _i	w _i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

La soluzione S per questa istanza del problema con C = 11 è

$$S = \{3, 4\}$$

Istanza	Soluzione	Sottoproblema
$n \in \mathbb{N} \quad X_n = \{1, \dots, n\}$	$S \subseteq X_n$ t.c.	Istanza i, $X_i \subseteq X_n$ t.c. $X_i = \{0, \dots, i\}, i = \{0, \dots, n\}$ $c \in \{0, \dots, C\}$
$\forall i \in X_n \quad v_i > 0 \in \mathbb{N}$ $w_i > 0 \in \mathbb{N}$	$v(S) = \max \{ v(A) \}$ $A \subseteq X_n, w(S) \leq C$	Soluzione $S_i \subseteq X_i$ t.c. $v(S_i) = \max \{ V(A) \}$ $A \subseteq X_i, w(S_i) \leq c$
$C \in \mathbb{N}, C > 0$		
$(i, c) \quad i = 0$ e qualunque valore di c $X_i = \emptyset$	$S_{i,c} = \emptyset$ $w(S_{i,c}) = 0$	$OPT(i, c) = v(S_{i,c}) = 0$
$(i, c) \quad i > 0, c = 0$	$S_{i,c} = \emptyset$ $w(S_{i,c}) = 0$	$OPT(i, c) = 0$

$(i, c) \quad i > 0, c > 0$	$S_{i,c} =$	$S(i-1, c)$	se $w_i > c$
		$S(i-1, c)$	se $w_i \leq c$ e $i \notin S(i, c)$
		$S(i-1, c-w_i) \cup i$	se $w_i \leq c$ e $i \in S(i, c)$
$OPT(i, c) =$		$OPT(i-1, c)$	se $w_i > c$
		$\max(OPT(i-1, c-w_i) + v_i, OPT(i-1, c))$	se $w_i \leq c$

$Sic = S(i-1, c)$	se $w_i > c$
$S(i-1, c)$	se $w_i \leq c$ e $OPT(i-1, c-w_i) + v_i \geq OPT(i-1, c)$
$S(i-1, c-w_i) \cup i$	se $w_i \leq c$ e $OPT(i-1, c-w_i) + v_i < OPT(i-1, c)$

PD-KNAPSACK($v[]$, $w[]$, C , n)

```

for c = 0 to C
    OPT[0, c] = 0
    if i > 0 or c > 0
        PRINT(i, c, OPT, v[], w[])
for i = 1 to n
    OPT[i, 0] = 0
    if w[i] > c
        PRINT(i-1, c, OPT, v[], w[])
    else
        if OPT[i-1, c-w[i]] + v[i] ≥ OPT[i-1, c]
            PRINT(i-1, c-w[i], OPT, v[], w[])
            print(i)
        else
            PRINT(i-1, c-1, OPT, v[], w[])
    OPT[i, c] = max(
        OPT[i-1, c-w[i]] + v[i],
        OPT[i-1, c])

```