Programmazione Dinamica

1

I numeri di Fibonacci

Leonardo da Pisa (detto Fibonacci) era interessato alla dinamica delle popolazioni. Quanto velocemente si espande una popolazione di conigli?

Ipotesi: ogni coppia di conigli genera una coppia (?) di coniglietti ogni anno, i conigli cominciano a riprodursi dal secondo anno di vita. Il numero di coppie di conigli sarà:

- *Anno 1: F(1) = 1 – Si inizia con una coppia di coniglietti*
- *Anno 2: F(2) = 1 – Troppo giovani per riprodursi*
- *Anno 3: F(3) = 2 – prima coppia di figli*
- *Anno 4: F(4) = 3 – altra coppia di figli.*
- *Anno 5: F(5) = 5 – prima coppia di nipoti*

In generale F(n) = F(n-1) + F(n-2): i conigli dell'anno prima sono ancora lì (F(n-1)) e in più ci sono i nuovi figli delle coppie di almeno due anni (F(n-2))

2

I numeri di Fibonacci

Algoritmo 1:

```
int fib1(int n)
{ if (n<=2) return 1
  else return fib1(n-1)+fib(n-2)
}
```

$T(n) = 2 + T(n-1) + T(n-2) = O(2^n)$

n=45 → un miliardo di passi

Perchè?

Algoritmo 2:

```
int fib2(int n)
{ int * f = new int[n+1];
  f[1] = f[2] = 1;
  for (int i=3;i<=n;i++)
    f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

$T(n) = O(n)$ (si può fare di meglio)

n=45 → 90 passi

Più memoria, meno tempo!

Programmazione Dinamica

Divide et impera: si suddivide il problema in sottoproblemi indipendenti, si calcola ricorsivamente una soluzione per i sottoproblemi e poi si fondono le soluzioni così trovate per calcolare la soluzione globale per il problema originale.

Programmazione dinamica: simile al divide et impera, ma tiene traccia (in una tabella) delle soluzioni dei sottoproblemi perchè può capitare di dover risolvere il medesimo sottoproblema per più di una volta.

DP, Passi fondamentali

1. *Verifica* della caratterizzazione della struttura di una soluzione ottima
2. *Definizione* ricorsiva del valore di una soluzione ottima tramite equazioni ricorsive
3. *Calcolo* del valore di una soluzione ottima con strategia bottom-up
4. *Costruzione* di una soluzione ottima a partire dalle informazioni già calcolate.

DP, caratteristiche del problema

Per applicare con successo la programmazione dinamica, è necessario che il problema abbia:

Sottostruttura ottima.

Una soluzione ottima per il problema contiene al suo interno le soluzioni ottime dei sottoproblemi

Sottoproblemi comuni.

Un problema di ottimizzazione ha sottoproblemi comuni quando un algoritmo ricorsivo richiede di risolvere più di una volta lo stesso sottoproblema

max-subarray

Elementi contigui!

Vettore A[1..*n*] di *n* valori reali arbitrari

Vogliamo individuare un sottovettore non vuoto di A in cui la somma degli elementi sia massima

Può essere negativa!

3	-5	10	2	-3	1	4	-8	7	-6	-1

Quanti sono i sottovettori di A?

- 1 sottovettore di lunghezza n
- 2 sottovettori di lunghezza n - 1
- 3 sottovettori di lunghezza n - 2
- ...
- k sottovettori di lunghezza $n - k + 1$
- ...
- n sottovettori di lunghezza 1

Risposta:
$n(n+1)/2 = \Theta(n^2)$ sottovettori

Soluzione forza bruta

```
Algorithm SommaMax1(A[1..n])
smax = A[1];
for (i = 1 to n)    // per ogni possibile inizio
  for (j = i to n)  // per ogni possibile fine
    s = 0;
    for (k = i to j)
      s = s + A[k]; // elementi del sottovettore
    endfor
    if (s > smax) then smax = s;
  endfor
endfor
return smax;
```

$\Theta(n^3)$

Un po' più efficiente

```
Algorithm SommaMax2(A[1..n])
smax = A[1];
for (i = 1 to n)    // per ogni possibile inizio
s = 0;
  for (j = i to n)  // somma fino a ogni possibile fine
    s = s + A[j];
    if (s > smax) then smax = s;
  endfor
endfor
return smax;
```

Per ogni possibile inizio, tutte le possibili fini: $\Theta(n^2)$

Programmazione dinamica

Sia $P(i)$ il problema che consiste nel determinare il valore massimo della somma degli elementi dei sottovettori non vuoti del vettore $A[1..i]$ che hanno $A[i]$ come ultimo elemento

Sia $S[i]$ il valore della soluzione di $P(i)$

- $S[i]$ è la massima somma degli elementi del sottovettori di $A[1..i]$ che hanno $A[i]$ come ultimo elemento

La soluzione S^* al problema di partenza può essere espressa come

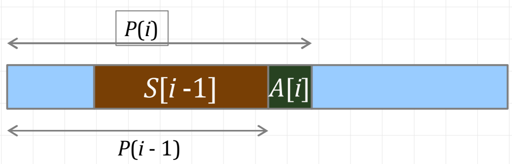
$$S^* = max_{1 \le i \le n} S[i]$$

Programmazione dinamica

$P(1)$ ammette una unica soluzione: $S[1]$ = A[1]

Consideriamo il generico problema $P(i)$, $i > 1$

- Supponiamo di avere già risolto il problema $P(i - 1)$, e quindi di conoscere $S[i -1]$
- Se $S[i\text{-}1]+A[i] \geq A[i]$ allora $S[i] = S[i\text{-}1]+A[i]$
- Se $S[i\text{-}1]+A[i] < A[i]$ allora $S[i] = A[i]$



max-subarray, progr. dinamica

Idea per S[i]: somma massima dei sottoarray che finiscono in i

Inzializzazione:

S[1] = A[1]

Espansione, due casi:

1. Inizio un nuovo max-subarray in i, S[i]= A[i]
2. Continuo il max-subarray che finisce in $i-1$: S[i]= S[$i-1$]+A[i]

Complessivamente: S[i] = max{A[i], S[$i-1$] + A[i]}

max-subarray

max-subarray: algoritmo in O(n)

```
Algorithm maxSubArray(A,n)
new array S
S[1] = A[1]
m = S[1]
for i = 2 to n do
  S[i] = max(A[i], S[i-1] + A[i])
  m = max(m, S[i])
return m
```

$\Theta(n)$

Esempio

A[]	3	-5	10	2	-3	1	4	-8	7	-6	-1
S[]	3	-2	10	12	9	10	14	6	13	7	6

$$S[i] = max\ \{ A[i], A[\mathrm{i}] + S[i - 1] \}$$

Ma qual'é il sottovettore?

Siamo in grado di calcolare il valore della massima somma tra tutti i sottovettori di *A*[1..*n*]

Come facciamo a determinare *quale* sottovettore produce tale somma?

- Abbiamo l'indice dell'elemento finale del sottovettore
- Possiamo ricavare l'indice iniziale procedendo a ritroso:
  - Se *S*[*i*] = *V*[*i*], il sottovettore massimo inizia nella posizione *i*

Esempio

A[]	3	-5	10	2	-3	1	4	-8	7	-6	-1

S[]	3	-2	10	12	9	10	14	6	13	7	6

imax

```
indiceInizio(A[1..n], S[1..n], imax)
i = imax;
while ( S[i] ≠ A[i] ) do
  i = i - 1;
endwhile
return i;
```

knapsack01

Dati:

- Un insieme S con n elementi
  $\rightarrow$ ogni elemento i ha un peso w_i e un valore v_i
- Peso massimo totale W

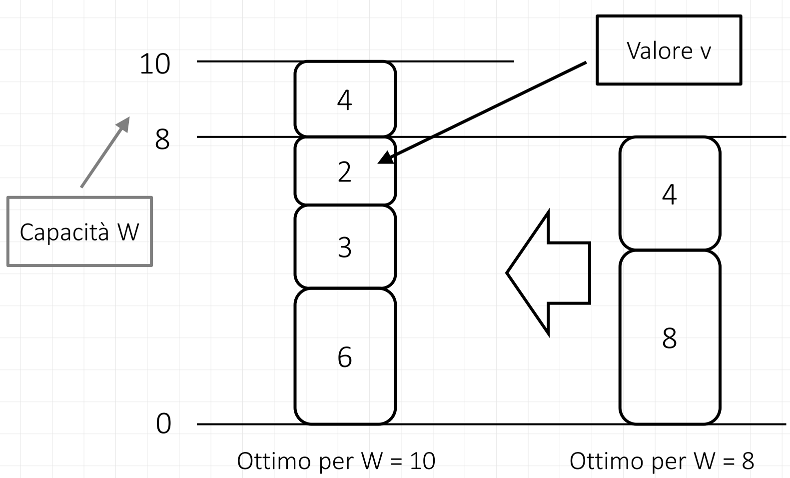
Obiettivo: Scegliere un sottinsieme di elementi $\mathrm{T} \subseteq \mathrm{S}$ che *massimizzi la somma dei valori* degli elementi selezionati $max \sum_{i \in T} v_i$

Vincolo: la somma dei pesi degli elementi selezionati non sia superiore a W, $\sum_{i \in T} w_i \leq W$

Approccio naive: considera tutti i possibili sottinsiemi T (sono 2^n)

Knapsack01, ottimalitá

Per assurdo, assumiamo che non valga.



Knapsack 01, divide et impera

Una soluzione ottima completa è composta da soluzione ottime di sottoproblemi.

Il problema può essere ridotto in sottoproblemi più piccoli sia riducendo la capacità del knapsack sia riducendo l'insieme di alternative fra cui scegliere.

Knap(S,W) -> Knap(S_i,W): scelgo solo fra i primi i elementi dell'insieme S, valore ottimo $f[S_i,W]$

Knap(S,W) -> Knap(S,q): scelgo fra tutti gli elementi di S ma con knapsack di capacità q<W, valore ottimo $f[S,q]$

Knap(S,W) -> Knap(si,q): scelgo solo fra i primi i elementi dell'insieme S e con knapsack di capacità q<W, valore ottimo $f[S_i,q]$

Knapsack01, ricorsione

Idea per l'algoritmo:

- S_i contiene gli elementi $1, \dots, i$
- $f[i,q]$ *valore* della miglior selezione da S_i con peso totale q

Come trovare $f[i,q]$?

- se $w_i > q$: non si può prendere l'elemento i, si mantiene la migliore soluzione con elementi fino a i-1

  quindi $f[i,q] = f[i-1, q]$

- se $w_i \leq q$: si può prendere l'elemento i, lo posso aggiungere alla migliore soluzione che lascia abbastanza spazio ($q - w_i$) e la confronto con la migliore soluzione senza i:

  quindi $f[i,q] = max\{f[i-1, q],\ f[i-1, q - w_i] + v_i\}$

Knapsack, ricorsione

Ricorsione:

Inizializzazione: $f_1(q) = \begin{cases} 0 & q < w_1 \\ max(0, v_1) & q \geq w_1 \end{cases}$ oppure $f_0(q)$=0 $\forall q$

step: $f_i(q) = \begin{cases} f_{i-1}(q) & q < w_i \\ max\{f_{i-1}(q), f_{i-1}(q - w_i) + v_i\} & q \geq w_i \end{cases}$

knapsack01

S consiste di n elementi con v_i e w_i ; W è il peso massimo totale

```
Algoritmo Knapsack01(S,W)
new f[0 ... n, 0 ... W]
for q = 0 to W do
   f[0,q] = 0
for i = 1 to n do
   f[i,0] = 0
for q = 1 to W do
  for i = 1 to n do
    if w[i] ≤ q then
      f[i,q] = max(f[i-1,q], f[i-1, q-w[i]] + v[i])
    else
      f[i,q] = f[i-1,q]
```

Complessità $\Theta(n\text{W})$

Pseudopolinomiale!

Knapsack01

Knapsack01, esempio:

- item 1 con $w_1 = 3$ e $v_1 = 9$
- item 2 con $w_2 = 2$ e $v_2 = 5$
- item 3 con $w_3 = 2$ e $v_3 = 5$
- peso massimo $W = 4$

```
f[0,0] = 0, f[0,1] = 0, f[0,2] = 0, f[0,3] = 0, f[0,4] = 0
f[1,0] = 0, f[1,1] = 0, f[1,2] = 0, f[1,3] = 9, f[1,4] = 9
f[2,0] = 0, f[2,1] = 0, f[2,2] = 5, f[2,3] = 9, f[2,4] = 9
f[3,0] = 0, f[3,1] = 0, f[3,2] = 5, f[3,3] = 9, f[3,4] = 10
```


Una tabella di ricorsione

```
Istanza:
W   20
V   3 2 3 1 2
w   7 2 9 3 1
i   1 2 3 4 5
```

q

Inizializzazione

i

20	0	3	5	8	8	10
19	0	3	5	8	8	10
18	0	3	5	8	8	8
17	0	3	5	6	6	8
16	0	3	5	6	6	8
15	0	3	5	5	6	8
14	0	3	5	5	6	8
13	0	3	5	5	6	8
12	0	3	5	5	6	7
11	0	3	5	5	5	7
10	0	3	5	5	5	7
9	0	3	5	5	5	5
8	0	3	3	3	3	5
7	0	3	3	3	3	5
6	0	0	2	2	3	5
5	0	0	2	2	3	4
4	0	0	2	2	2	4
3	0	0	2	2	2	4
2	0	0	2	2	2	2
1	0	0	0	0	0	2
0	0	0	0	0	0	0
	0	1	2	3	4	5

Ricostruzione della soluzione

Istanza:

W	20				
V	3	2	3	1	2
w	7	2	9	3	1
i	*1*	*2*	*3*	*4*	*5*

20	3	5	8	8	10
19	3	5	8	8	10
18	3	5	8	8	8
17	3	5	6	6	8
16	3	5	6	6	8
15	3	5	5	6	8
14	3	5	5	6	8
13	3	5	5	6	8
12	3	5	5	6	7
11	3	5	5	5	7
10	3	5	5	5	7
9	3	5	5	5	5
8	3	3	3	3	5
7	3	3	3	3	5
6	0	2	2	3	5
5	0	2	2	3	4
4	0	2	2	2	4
3	0	2	2	2	4
2	0	2	2	2	2
1	0	0	0	0	2
0	0	0	0	0	0
	1	2	3	4	5

$f_i(q) = f_{i-1}(q)$

$x_i = 0$

$f_i(q) = f_{i-1}(q - w_i) + v_i$

$x_0 = 1$

$f_0(w_0) = v_0$

Pseudocodice: esecizio

Massima sottosequenza comune

X = ABCDBDAB

Y = BDCABA

Z = BCBA

è LCS(X,Y)

Problema di ottimizzazione.

Possiamo applicare la programmazione dinamica ??

Massima sottosequenza comune

Problema della *massima sottosequenza comune*:

- sono date due sequenze $X = x_1x_2...x_m$ e $Y = y_1y_2...y_n$
- si chiede di trovare la più lunga sequenza $Z = z_1z_2...z_k$ che è sia sottosequenza di X che sottosequenza di Y.

27

Sottostruttura ottima

Siano $X=<x_1,...,x_m>$ e $Y=<y_1,...,y_n>$ *due sequenze*

Sia $Z=<z_1,...,z_k>$ una LCS di X e Y.

1. Se $x_m=y_n$ e $z_k=x_m=y_n$ allora Z_{k-1} è LCS di X_{m-1} e Y_{n-1}

   X = ABBA Y = BABA Z = ABA

2. Se $x_m \neq y_n$ e $z_k \neq x_m$ allora Z_k è LCS di X_{m-1} e Y

   X = ABBAC Y = BABA Z = ABA

3. Se $x_m \neq y_n$ e $z_k \neq y_n$ allora Z_k è LCS di X e Y_{n-1}

   X = ABBA Y = BABAC Z = ABA

28

Dimostrazione sottostruttura

1. Supponiamo $x_m = y_n$.

   Se $z_k \neq x_m = y_n$ potremmo aggiungere il simbolo $x_m = y_n$ in coda a Z ottenendo una sottosequenza comune più lunga contro l'ipotesi che Z sia una LCS.

   Quindi $z_k = x_m = y_n$ e quindi Z_{k-1} è sottosequenza comune di X_{m-1} e Y_{n-1}.

2. se $z_k \neq x_m$ allora Z è sottosequenza di X_{m-1} e Y. Essendo Z una LCS di X e Y essa è anche una LCS di X_{m-1} e Y.

3. il caso $z_k \neq y_n$ è simmetrico.

29

Soluzione ricorsiva

Siano $X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$ le due sequenze di cui vogliamo calcolare una LCS.

Per $i = 0,1,\ldots,m$ e $j = 0,1,\ldots,n$ sia $c_{i,j}$ la lunghezza di una LCS dei due prefissi (sottosequenze iniziali) X_i e Y_j .

Per la proprietà che abbiamo appena visto possiamo scrivere:

$$c_{i,j} = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c_{i-1,j-1} + 1 & \text{se } i,j > 0 \text{ e } x_i = y_j \\ \max(c_{i,j-1}, c_{i-1,j}) & \text{se } i,j > 0 \text{ e } x_i \neq y_j \end{cases}$$

30

LCS, esempio

		j	0	1	2	3	4	5	6
i		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS, pseudocodice

```
Algorithm LCS-length(X,Y)
m=length(X)
n=length(Y)
for i=0 to m do c[i,0]=0
for j=1 to n do c[0,j]=0
for i=1 to m do
  for j=1 to n
    if xi=yj
    then c[i,j]=c[i-1,j-1] + 1
      b[i,j]= ↖
    else if c[i-1,j]≥c[i,j-1]
            then c[i,j]=c[i-1,j]
                 b[i,j]= ↑
            else c[i,j]=c[i,j-1]
                 b[i,j]= ←
return b,c
```

Ricostruzione di una LCS

```
Print-LCS(b,X,i,j)
if i=0 or j=0 then return
if b[i,j] = ↖
   then Print-LCS(b,X,i-1,j-1)
        print x_i
    else if b[i,j] = ↑
        then Print-LCS(b,X,i-1,j)
        else Print-LCS(b,X,i,j-1)
```

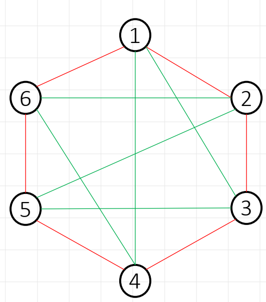

Travelling Salesman Problem

Algoritmo forza bruta– prova tutte le possibilità

Per ogni permutazione π di {1,...,n} visita i nodi nell'ordine di π, e calcolane il costo del percorso,

Ritorna la soluzione migliore

Complessità O(n!)



π = {1,2,3,4,5,6}

π = {1,3,5,2,6,4}

Travelling Salesman Problem

Per ogni sottinsieme S di nodi con $|S| \geq 2$ e per ogni coppia di nodi $u,v \in S$

Sia $f[S,u,v]$ la lunghezza del cammino minimo che

- inizia in u
- termina in v
- visita tutti i nodi in S

Dimensione di $f \approx 2^n n^2$ (cardinalità di S per num coppie u,v)

Sottostruttura ottima: solito, per assurdo. A parità di nodo inizio e fine e di sottinsieme, il cammino da u a v passante per tutti I nodi in S.

Convenzione: dato che il ciclo può iniziare in un nodo qualsiasi, assumiamo u=1 e quindi calcoliamo solo $f[S,v]$

TSP: ricorsione

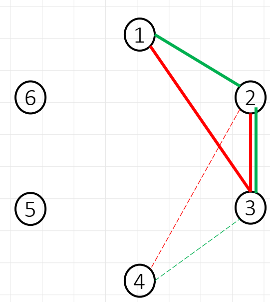
Equazione ricorsiva:

Se $|S| = 2$ (quindi $S=\{1,v\}$)

$$f[S,v] = d(1,v)$$

Se $|S|>2$

$$f(S, v) = \min \{ f(S-\{v\}, w) + d(w, v)\},\ w\in S,\ w\neq v,\ w\neq 1$$



$S = \{1,2,3,4\}$

$v = 4$

$f(S,4) = \min \{ f(\{1,2,3\},2)+d(2,4), f(\{1,2,3\},3)+d(3,4) \}$

TSP: algoritmo DP

Algorithm TSP_dynprogr

f **[{1,v}, v] = d[1, v], v∈N\{1}**

foreach S, **|S| = 3** to ***n*** do

  foreach **v ∈ S\{1}**

    f **[S, v] =** ***min*** **{** ***f*** **[S\{v}, w]+d[w,v] for w ∈ S, w ≠ v}**

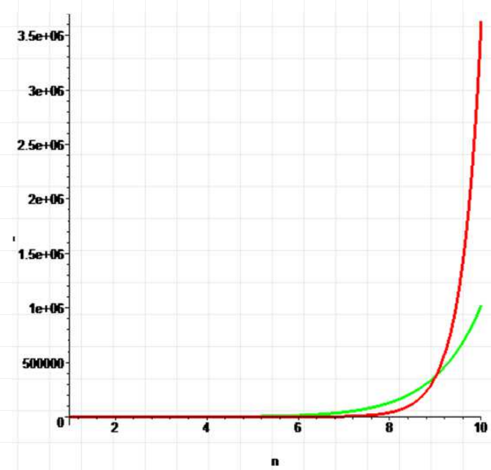
Return min { ***f*****({1, 2, 3, …,** ***n*****},** ***v*****) + d(*****v*****,** ***1*****), j ∈ N\{1}}**

Complessità $O(n^2\ 2^n)$

TSP: commplessitá DP

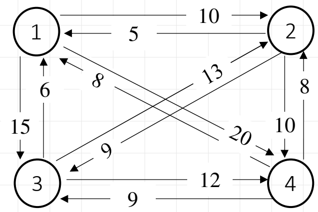
Programmazione dinamica= $O(n^2\ 2^n)$

Forza bruta= $O(n!)$



TSP: esempio

	1	*2*	*3*	*4*
1	**0**	**10**	**15**	**20**
2	**5**	**0**	**9**	**10**
3	**6**	**13**	**0**	**12**
4	**8**	**8**	**9**	**0**



Inizializzazione

f[{1,2}, 2] = d[1,2] = 10, f[{1,3}, 3] = d[1,3] = 15, f[{1,4}, 4] = d[1,4] = 20

S=3

f[{1,2,3}, 2] = f[{1,3}, 3] + d[3,2] = 28, f[{1,2,3}, 3] = f[{1,2}, 2] + d[2,3] = 19
f[{1,2,4}, 2] = f[{1,4}, 4] + d[4,2] = 28, f[{1,2,4}, 4] = f[{1,2}, 2] + d[2,4] = 20
f[{1,3,4}, 3] = f[{1,4}, 4] + d[4,3] = 29, f[{1,3,4}, 4] = f[{1,3}, 3] + d[3,4] = 27

S=4

f[{1,2,3,4}, 2] = *min* { f[{1,3,4}, 3] + d[3,2]=42, f[{1,3,4}, 4] + d[4,2]=35 } = 35
f[{1,2,3,4}, 3] = *min* { f[{1,2,4}, 2] + d[2,3]=37, f[{1,2,4}, 4] + d[4,3]=29 } = 29
f[{1,2,3,4}, 4] = *min* { f[{1,2,3}, 2] + d[2,4]=38, f[{1,2,3}, 3] + d[3,4]=29 } = 31

Risultato

min{35+5=40, 29+6=35, 31+8=39} = 35 tour minimo: [1,2,4,3]