Programmazione Dinamica

A.A. 2016-2017

Tecnica «memoizing»

Programmazione dinamica: osservazioni

Quali sono le caratteristiche dei problemi risolubili con la tecnica della programmazione dinamica?

1. Sottoproblemi comuni

Se un problema di ottimizzazione ha sottoproblemi comuni, un algoritmo ricorsivo richiede di risolvere più di una volta uno stesso sottoproblema. Tipicamente il numero di sottoproblemi distinti è un polinomio nella dimensione dell'input.
Gli algoritmi di programmazione dinamica sfruttano i sottoproblemi ripetuti risolvendo ciascun sottoproblema una sola volta e memorizzano la soluzione in una tabella da cui può essere recuperata in tempo costante.

2. Sottostruttura ottima

Un problema presenta una sottostruttura ottima se una soluzione ottima del problema contiene al suo interno soluzioni ottime dei sottoproblemi.

Programmazione dinamica: osservazioni

La sottostruttura ottima varia in due modi a seconda del tipo di problemi:

a) numero di sottoproblemi.
b) numero di scelte per determinare quali sottoproblemi utilizzare in una soluzione ottima.

Ci sono alcune scelte standard che sembrano presentarsi ripetutamente:

1) L'input è $x_1, x_2, \ldots, x_n$ e un sottoproblema è $x_1, x_2, \ldots, x_i$ $\Rightarrow$
   Il numero di sottoproblemi è $\mathbf{O}(n)$

2) L'input è $x_1, x_2, \ldots, x_n$ e $y_1, y_2, \ldots, y_m$. Un sottoproblema è dato da uno tra $x_1, x_2, \ldots, x_i$ e uno tra $y_1, y_2, \ldots, y_j$ $\Rightarrow$
   Il numero di sottoproblemi è $\mathbf{O}(n.m)$

3) L'input è $x_1, x_2, \ldots, x_n$ e un sottoproblema è $x_i, x_{i+1}, \ldots, x_j$ per qualche $1 \leq i,j \leq n$ $\Rightarrow$ Il numero di sottoproblemi è $\mathbf{O}(n^2)$

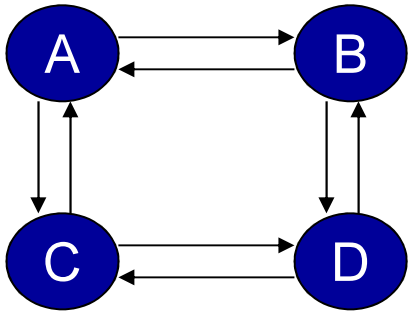
Informalmente il tempo di esecuzione dipende dal prodotto del numero di sottoproblemi da risolvere e dal numero di scelte per ogni sottoproblema.

Programmazione dinamica: osservazioni

Attenzione all'esistenza della sottostruttura ottima

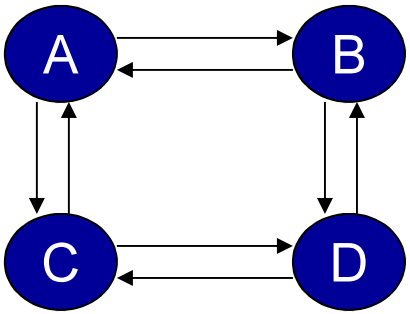
Confrontiamo il problema di trovare un cammino di lunghezza minima tra due vertici in un grafo orientato con quello di trovare un cammino semplice di lunghezza massima.

Un cammino semplice (senza cicli) di lunghezza massima presenta una sottostruttura ottima?



Un cammino massimo tra A e D è ABD

Programmazione dinamica: osservazioni



Il sottocammino AB non è di lunghezza massima perché ACDB è un cammino più lungo tra A e B e in modo analogo tra B e D il cammino di lunghezza massima è BACD. La concatenazione dei due ACDBACD non è un cammino semplice.

Manca una sottostruttura ottima, perciò non è possibile 'assemblare' una soluzione 'valida' dalle soluzioni dei sottoproblemi.

Come mai il problema del cammino massimo è così diverso da quello del cammino minimo?
I sottoproblemi del primo non sono indipendenti, quelli del secondo si.

Indipendenti: la soluzione di un sottoproblema non influisce sulla soluzione di un altro

Programmazione dinamica: osservazioni

Quando si cerca la sottostruttura ottima di un problema si segue uno schema comune:

- dimostrare che una soluzione consiste nel fare una scelta che lascia uno o più sottoproblemi da risolvere
- supporre di conoscere la scelta che porta a una soluzione ottima dei sottoproblemi (la tecnica della ricorsione).
- determinare i sottoproblemi da considerare e il modo migliore di rappresentare lo spazio dei sottoproblemi risultante
- dimostrare che la soluzione dei sottoproblemi deve essere ottima (supporre non ottima la soluzione ai sottoproblemi e arrivare a una contraddizione)

Programmazione dinamica: osservazioni

Le soluzioni di programmazione dinamica possono essere viste come il risultato di una sequenza di decisioni; possono essere generate molte sequenze, ma quelle che contengono sottosequenze non ottimali non possono fornire soluzioni ottimali (per il principio di ottimalità) e così non vengono generate, per quanto possibile.

Programmazione dinamica: osservazioni

Nella programmazione dinamica si descrive il problema con una formula di ricorrenza che esprime problemi più “grandi” in funzione di problemi più piccoli; tali problemi vengono poi risolti in modo bottom-up, dai più piccoli ai più grandi e viene riempita una tabella con i valori “soluzione”.
La formula suggerisce un algoritmo ricorsivo, ma la ricorsione può essere molto inefficiente perché risolve più e più volte gli stessi sottoproblemi.

Abbiamo già visto che si può fare una ricorsione “più intelligente”, che ricordi le sue precedenti invocazioni ed eviti di ripeterle.

La **tecnica di ricorsione memoized** o **annotazione**, pur adottando una strategia di tipo top-down, spesso presenta la stessa efficienza delle soluzioni di programmazione dinamica.

Tecnica di annotazione

Un algoritmo **ricorsivo con annotazione** (***memoized***) memorizza le soluzioni dei sottoproblemi in una tabella, ma la strategia di controllo della costruzione della tabella è sostanzialmente quella definita dall'algoritmo ricorsivo.

- Ogni elemento della tabella inizialmente assume un valore speciale per indicare che il valore dell'argomento non è ancora stato calcolato.
- Quando si deve risolvere per la prima volta un sottoproblema, si calcola la soluzione e la si memorizza nella tabella.
- Quando viene richiesto di risolvere nuovamente lo stesso sottoproblema, si accede alla tabella e si restituisce il valore memorizzato.

Tecnica di annotazione: moltiplicazione di matrici

```
Memoized_Matrix_chain (c)
    n ← length [c] -1
    for i = 1 to  n
        for j = i to  n
            m[i, j] ← ∞
    return Lookup_chain (c, m, 1, n)
```

Inizializzazione della matrice m

Chiamata alla procedura ricorsiva

```
Lookup_chain (c, m, i, j)
    if (m[i, j] < ∞) return m[i, j]
    if (i = j) m[i, j] ← 0
    else for k = i to  j-1
            num ← Lookup_chain (c, m, i, k) +
                + Lookup_chain (c, m, k+1, j)
                + c[i-1]·c[k]·c[j]
            if (num < m[i, j])   m[i, j] ← num
    return m[i, j]
```

Se il valore è già stato calcolato, recuperalo

Calcola un nuovo valore usando quando possibile valori già calcolati e memorizzalo

Tecnica di annotazione: zaino con ripetizione

```
Memoized_zaino (C, P, V, n)
    for k ← 1 to C
            Val[k] ← -1
    Val[0] ← 0
    return Lookup_zaino (k, P, V, n)

Lookup_zaino (k, P, V, n)
    if (Val[k] > -1) return Val[k]
    valore ← Lookup_zaino (k – 1, P, V, n)
    for i ← 1 to n
            if (k ≥ P[i])
                h ← Lookup_zaino (k – P[i], P, V, n)
                if (h + V[i] > valore)    valore ← h + V[i]
    Val[k] ← valore
    return Val[k]
```

Inizializzazione del vettore Val

Chiamata alla procedura ricorsiva

Se il valore è già stato calcolato, recuperalo

Calcola un nuovo valore usando quando possibile valori già calcolati e memorizzalo

Tecnica di annotazione

L'algoritmo memoized non risolve mai più di una volta un sottoproblema, come nella programmazione dinamica, comunque i fattori costanti nella notazione $\mathbf{O}$ sono sostanzialmente superiori a causa dell'overhead della ricorsione.

In alcuni casi tuttavia la tecnica di annotazione è conveniente.
La programmazione iterativa risolve ogni sottoproblema che potrebbe essere necessario, mentre la ricorsione memoized risolve solo quelli effettivamente usati.

Per esempio, per il problema zaino se C e tutti i pesi p_i sono multipli di 100 un sottoproblema Val[k] è inutile se 100 non divide k.
L'algoritmo ricorsivo con annotazione non risolverà mai questi sottoproblemi.