Programmazione Dinamica

domenica 11 aprile 2021 06:36

Cos'è la DP (dynamic programming)?

Prendiamo l'esempio di Fibonacci. L'approccio ricorsivo ha complessità $O(2^n)$. Però posso farmi furbo e salvare da qualche parte le soluzioni dei sottoproblemi. Questa tecnica è detta "memorizzazione".

KNAPSACK PROBLEM

Ho uno zaino che sopporta un peso C. Inoltre ho N oggetti, l'i-esimo ha peso W_i e valore V_i ($0 \le i \le N-1$).

Voglio mettere alcuni oggetti nello zaino in modo da massimizzare il valore totale.

Problema: Calcolare $ans(K, c)$ = max valore per uno zaino di capacità c avendo soltanto gli oggetti $0, 1, \dots, K-1$

La risposta al problema originale è $ans(N-1, C)$.

Serve una ricorrenza:

$$ans(K, c) = \max\{ans(K-1, c), \underbrace{ans(K-1, c-W_K) + V_K}_{\text{se } W_K \le c}\}$$

E i passi base? $ans(0, c) = \begin{cases} 0 & \text{se } W_0 > c \\ V_0 & \text{altrimenti} \end{cases}$

oppure $ans(-1, c) = 0 \quad \forall c$.

Una volta che ho questa ricorrenza, posso trasformarla in un approccio DP tramite memorizzazione.

APPROCCI BOTTOM-UP E TOP-DOWN

L'approccio top-down procede "dall'alto verso il basso".

In pratica consiste nell'adattare l'algoritmo ricorsivo nel modo ovvio.

Quasi sempre si implementa con una funzione ricorsiva.

```
int solve(int K, int c) {
    if (K == -1) return 0;

    if (ans[K][c] != -1)
```

```
if (ans[k][c] != -1)
    return ans[k][c];

ans[k][c] = solve(k-1, c);
if (w[k] <= c)
    ans[k][c] = max(ans[k][c], solve(k-1, c-w[k]) + v[k]);

return ans[k][c];
}
```

L'approccio bottom-up fa il contrario, e si implementa iterativamente

```
for (int c=0; c<=C; c++)
    ans[0][c] = w[0] > c ? 0 : v[0];
```

Ma quando conviene usare l'uno piuttosto che l'altro?

- top-down è più intuitivo di bottom-up;
- solitamente bottom-up è più efficiente;
- bottom-up consente (a volte) il risparmio di memoria.

COMPLESSITÀ

Quasi sempre la complessità di un algoritmo DP è data da

$$O((\#\text{stati}) \cdot (\text{tempo di transizione}))$$

- numero di stati: "la dimensione di ans[]"

  per lo knapsack problem, #stati = N·C.

  in pratica, uno stato è ciascuno di sottoproblemi da risolvere

- tempo di transizione: la complessità del ricorrersi agli stati più piccoli.

  per lo knapsack problem, transizione = O(1)

⇒ per knapsack problem, ho complessità O(N·C).

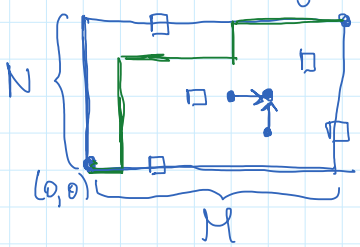
ALTRI ESEMPI

- Longest increasing subsequence (LIS) → $a[0, \dots, N-1]$
  Dato un array di interi, calcolare la lunghezza della più lunga sottosequenza strettamente crescente.

$$ans(k) = \max\left\{ans(k-1),\ 1 + \max_{\substack{0 \le i < k \\ a_i < a_k}} (ans(i))\right\}$$

Complessità: $O((\#\text{stati}) \cdot \text{transizione}) = O(N \cdot N) = O(N^2)$

- DP su array bidimensionali



Quanti percorsi monotoni da $(0,0)$ a $(N-1, M-1)$ che non passano per caselle bloccate?

$$ans(i,j) = \begin{cases} 0 & \text{se } (i,j) \text{ è bloccata} \\ ans(i-1,j) + ans(i,j-1) & \text{altrimenti} \end{cases}$$

(i: riga, j: colonna)

Andate a guardare "trabucco" (train.mg.olinfo.it)

RISPARMIO DI MEMORIA

Esempio: Knapsack problem con $N = C = 10\,000$.

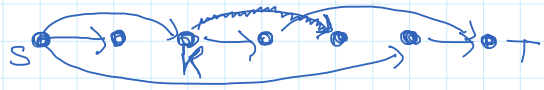
Con queste constraints sto nei tempi ma non nei limiti di memoria.

```
for (int k=1; k<N; k++) {
    for (int c=0; c<=C; c++) {
        ans[k][c] = ans[k-1][c];
        if (w[k] <= c)
            ans[k][c] = max(ans[k][c], ans[k-1][c-w[k]] + v[k]);
    }
}
```

Osservazione: per calcolare $ans[k][c]$ accedo solo ad $ans[k-1][\cdot]$.
Quindi mi basta tenere solo la riga precedente di ans.
A ogni iterazione del ciclo esterno, creo una nuova riga ans_tmp, la riempio con le soluzioni relative a k, e poi rimpiazzo ans con ans_tmp. Qui ans e ans_tmp sono vettori di dimensione C.

DP su DAG

Un DAG può essere linearizzato tramite toposort.



Esempio: quanti sono i cammini distinti da S a T?

come al solito, $cms(K)$ = # cammini da K a T = N-1.

intanto numeriamo i nodi: S=0, 1, ..., N-1 = T.

$$cms(K) = \sum_{v \in adj(K)} cms(v)$$

$cms(N-1) = 1$.

complessità? $\sum_{K=0}^{N-1} transizione(K) = \sum_{K=0}^{N-1} \#adj(K) = \#archi$

in realtà, il toposort non è necessario se uso un approccio top-down (in pratica sto facendo una DFS).