CoderFarm - Corso avanzato

Lezione 1

Lorenzo Ferrari, Davide Bartoli

28 aprile 2023

Contenuti

Ricorsione

Introduzione

Spesso risulta comodo formulare un problema in maniera ricorsiva, ossia in funzione di istanze più piccole dello stesso problema.

Spesso risulta comodo formulare un problema in maniera ricorsiva, ossia in funzione di istanze più piccole dello stesso problema. Prendiamo per esempio i numeri di Fibonacci

$$F_n = \begin{cases} 1 & \text{per } 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & \text{per } n \geq 2 \end{cases}$$

Spesso risulta comodo formulare un problema in maniera ricorsiva, ossia in funzione di istanze più piccole dello stesso problema. Prendiamo per esempio i numeri di Fibonacci

$$F_n = \begin{cases} 1 & \text{per } 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & \text{per } n \geq 2 \end{cases}$$

o il fattoriale

$$n! = \begin{cases} 1 & \text{per } n = 0 \\ n \cdot (n-1)! & \text{per } n \geq 1 \end{cases}$$

Spesso risulta comodo formulare un problema in maniera ricorsiva, ossia in funzione di istanze più piccole dello stesso problema. Prendiamo per esempio i numeri di Fibonacci

$$F_n = \begin{cases} 1 & \text{per } 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & \text{per } n \geq 2 \end{cases}$$

o il fattoriale

$$n! = \begin{cases} 1 & \text{per } n = 0 \\ n \cdot (n-1)! & \text{per } n \geq 1 \end{cases}$$

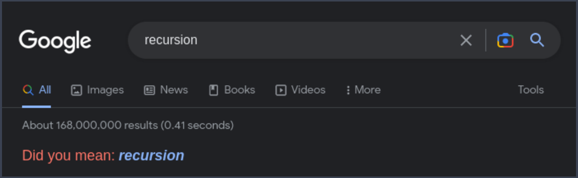
Queste definizioni possono essere scritte in C++ usando delle **funzioni ricorsive**.

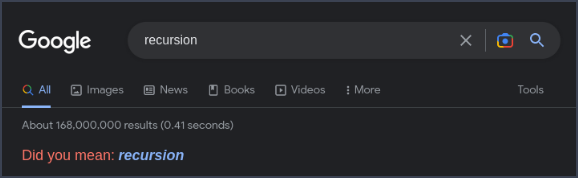
Funzioni ricorsive

Una funzione ricorsiva è una funzione che richiama se stessa.

Funzioni ricorsive

Una funzione ricorsiva è una funzione che richiama se stessa.
Deve sempre essere definita una condizione di terminazione, ossia un caso base in cui la funzione non richiama se stessa, altrimenti si entra in un ciclo infinito.


Google
recursion
All
Images
News
Books
Videos
More
Tools
About 168,000,000 results (0.41 seconds)
Did you mean: recursion


Google
recursion
All
Images
News
Books
Videos
More
Tools
About 168,000,000 results (0.41 seconds)
Did you mean: recursion


Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function
No exit condition

Ricorsione

Funzioni ricorsive

```
// ritorna n!
int fact(n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n-1);
}
```

Ricorsione

Funzioni ricorsive

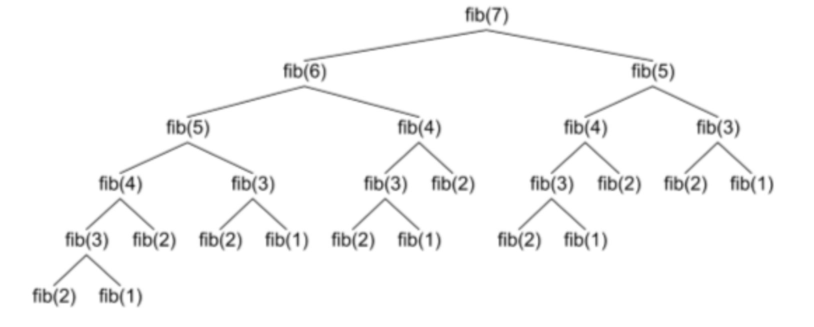
```
// n-esimo numero di fibonacci
int fib(n) {
  if (n <= 1) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

Albero di ricorsione

Le chiamate alla funzione ricorsiva `fib` formano un albero detto **albero di ricorsione**.
Nel caso del fattoriale, l'albero è una linea con N nodi.

Albero di ricorsione

Le chiamate alla funzione ricorsiva `fib` formano un albero detto **albero di ricorsione**.
Nel caso del fattoriale, l'albero è una linea con N nodi.



Problema di riscaldamento

Generare tutte le soluzioni

Piastrelle

Vuoi piastrellare un corridoio di dimensione $1 \times N$ e hai a disposizione solo piastrelle di dimensioni 1×1 e 1×2. Stampa tutte le possibili piastrellature di lunghezza N.

Problema di riscaldamento

Generare tutte le soluzioni

Piastrelle

Vuoi piastrellare un corridoio di dimensione $1 \times N$ e hai a disposizione solo piastrelle di dimensioni 1×1 e 1×2. Stampa tutte le possibili piastrellature di lunghezza N.

Esempio di input/output

File `input.txt`	**File** `output.txt`
4	[O] [O] [O] [O] [O] [O] [OOOO] [O] [OOOO] [O] [OOOO] [O] [O] [OOOO] [OOOO]

https://training.olinfo.it/#/task/piastrelle/statement

Implementazione

```cpp
void solve(int n, string prefix) {
    if (n < 0) return;
    if (n == 0) {
        cout << prefix << "\n";
    } else {
        solve(n - 1, prefix + "[0]");
        solve(n - 2, prefix + "[0000]");
    }
}
```

Ricerca completa

Quando esploriamo tutto lo spazio delle soluzioni eseguiamo una **ricerca completa**.

Ricerca completa

Quando esploriamo tutto lo spazio delle soluzioni eseguiamo una **ricerca completa**.

Spesso è possibile *tagliare* alcuni rami dell'albero di ricorsione sapendo che la risposta migliore non si trova in quel sottoalbero, e in alcuni casi possiamo evitare di ricalcolare più volte la stessa cosa.

La ricerca completa spesso permette di risolvere i primi subtask di un problema!

https://training.olinfo.it/#/task/ois_solitario/statement

Programmazione Dinamica

Fibonacci

Torniamo un momento al calcolo dell'*n*-esimo numero di Fibonacci. Quante chiamate stiamo facendo?

Torniamo un momento al calcolo dell'n-esimo numero di Fibonacci. Quante chiamate stiamo facendo?

- un numero esponenziale (in questo caso $O(\phi^n)$, dove $\phi = 1.618\ldots$), decisamente troppe...

Si può fare meglio di così?

Torniamo un momento al calcolo dell'n-esimo numero di Fibonacci. Quante chiamate stiamo facendo?

- un numero esponenziale (in questo caso $O(\phi^n)$, dove $\phi = 1.618\dots$), decisamente troppe...

Si può fare meglio di così?

- stiamo calcolando più volte F_n per uno stesso n, che è molto inefficente
- in totale dobbiamo calcolare solo $O(n)$ valori, i numeri $F_0, F_1, \dots, F_n$. Possiamo quindi salvarci i valori calcolati e riutilizzarli all'occorrenza. La complessità è quindi $O(n)$.
- Questa tecnica di **memorizzazione** è alla base della **programmazione dinamica**.

Programmazione Dinamica

Memoization

```cpp
const int MAXN = 1e6;
const int MOD = 1e9 + 7;

bool vis[MAXN]; // ho già calcolato fib(i)?
int memo[MAXN]; // i-esimo numero di Fibonacci

int fib(int n) {
  if (vis[n]) {
    return memo[n];
  }
  vis[n] = true;
  if (n <= 1) {
    return memo[n] = 1;
  } else {
    return memo[n] = (fib(n-1) + fib(n-2)) % MOD;
  }
}
```

Programmazione Dinamica

DP Iterativa

```
const int MAXN = 1e6;
const int MOD = 1e9 + 7;

int fib[MAXN];

fib[0] = fib[1] = 1;

for (int i = 2; i <= n; ++i) {
  fib[i] = (fib[i - 1] + fib[i - 2]) % MOD;
}
```

Programmazione Dinamica

La programmazione dinamica permette di risolvere problemi apparentemente complessi in modo efficiente grazie alla memorizzazione dei risultati parziali.

Può essere applicata a problemi che possono essere decomposti in sotto-problemi **indipendenti**.

Programmazione Dinamica

La programmazione dinamica permette di risolvere problemi apparentemente complessi in modo efficiente grazie alla memorizzazione dei risultati parziali.

Può essere applicata a problemi che possono essere decomposti in sotto-problemi **indipendenti**.

È caratterizzata da:

- lo **stato**: una configurazione del problema, generalmente rappresentata dai parametri della funzione ricorsiva;

Programmazione Dinamica

La programmazione dinamica permette di risolvere problemi apparentemente complessi in modo efficiente grazie alla memorizzazione dei risultati parziali.

Può essere applicata a problemi che possono essere decomposti in sotto-problemi **indipendenti**.

È caratterizzata da:

- lo **stato**: una configurazione del problema, generalmente rappresentata dai parametri della funzione ricorsiva;
- la **transizione**: il passaggio da uno stato all'altro, generalmente rappresentato dal corpo della funzione ricorsiva.

Programmazione Dinamica

La programmazione dinamica permette di risolvere problemi apparentemente complessi in modo efficiente grazie alla memorizzazione dei risultati parziali.

Può essere applicata a problemi che possono essere decomposti in sotto-problemi **indipendenti**.

È caratterizzata da:

- lo **stato**: una configurazione del problema, generalmente rappresentata dai parametri della funzione ricorsiva;
- la **transizione**: il passaggio da uno stato all'altro, generalmente rappresentato dal corpo della funzione ricorsiva.

La complessità si calcola come $O(\text{numero\_stati} \cdot \text{costo\_transizione})$.

Hateville

Nella città di Hateville ci sono $N \leq 1\,000\,000$ case in fila. Vuoi organizzare una raccolta fondi, e sai che la i-esima casa è disposta a donare V_i €. C'è un problema: ognuno odia i suoi vicini e non parteciperà alla raccolta fondi se uno dei vicini partecipa. Quanto denaro puoi raccogliere al massimo?

Programmazione Dinamica

Problema: Hateville

Hateville

Nella città di Hateville ci sono $N \leq 1\,000\,000$ case in fila. Vuoi organizzare una raccolta fondi, e sai che la i-esima casa è disposta a donare V_i €. C'è un problema: ognuno odia i suoi vicini e non parteciperà alla raccolta fondi se uno dei vicini partecipa. Quanto denaro puoi raccogliere al massimo?

- idee?

Hateville

Nella città di Hateville ci sono $N \leq 1\,000\,000$ case in fila. Vuoi organizzare una raccolta fondi, e sai che la i-esima casa è disposta a donare V_i €. C'è un problema: ognuno odia i suoi vicini e non parteciperà alla raccolta fondi se uno dei vicini partecipa. Quanto denaro puoi raccogliere al massimo?

- idee?
- potremmo controllare ricorsivamente tutti i sottoinsiemi di case validi e prendere quello con somma massima
  - chiaramente è troppo lento: avrebbe complessità $O(2^N)$

Programmazione Dinamica

Problema: Hateville

Hateville

Nella città di Hateville ci sono $N \leq 1\,000\,000$ case in fila. Vuoi organizzare una raccolta fondi, e sai che la i-esima casa è disposta a donare V_i €. C'è un problema: ognuno odia i suoi vicini e non parteciperà alla raccolta fondi se uno dei vicini partecipa. Quanto denaro puoi raccogliere al massimo?

- idee?
- potremmo controllare ricorsivamente tutti i sottoinsiemi di case validi e prendere quello con somma massima
  - chiaramente è troppo lento: avrebbe complessità $O(2^N)$
- neanche prendere o tutte le case pari o tutte le case dispari funziona (considerate il caso $[10, 1, 1, 10]$).

Riusciamo a definire il problema ricorsivamente?

Programmazione Dinamica

Problema: Hateville

Riusciamo a definire il problema ricorsivamente?

- supponiamo di aver risolto il problema per le prime $i - 1$ case.
- siamo capaci di risolvere il problema per le prime i case?

Programmazione Dinamica

Problema: Hateville

Riusciamo a definire il problema ricorsivamente?

- supponiamo di aver risolto il problema per le prime $i - 1$ case.
- siamo capaci di risolvere il problema per le prime i case?

Per ogni i, abbiamo 2 possibilità:

- la casa i non partecipa alla raccolta fondi, quindi la risposta è la stessa del prefisso $i - 1$.
- la casa i partecipa alla raccolta fondi, quindi la casa $i - 1$ non può partecipare.
  In questo caso la risposta è la somma di $V[i]$ e della risposta per il prefisso $i - 2$.

Programmazione Dinamica

Problema

Cerchiamo ora di calcolare la complessità computazionale della soluzione.

Programmazione Dinamica

Problema

Cerchiamo ora di calcolare la complessità computazionale della soluzione.

- Il numeri di stati è $O(N)$, dato che abbiamo N prefissi da calcolare.

Cerchiamo ora di calcolare la complessità computazionale della soluzione.

- Il numeri di stati è $O(N)$, dato che abbiamo N prefissi da calcolare.
- Il costo della transizione è $O(1)$, dato che ogni transizione consiste in esattamente 2 casi.

Programmazione Dinamica

Problema

Cerchiamo ora di calcolare la complessità computazionale della soluzione.

- Il numeri di stati è $O(N)$, dato che abbiamo N prefissi da calcolare.
- Il costo della transizione è $O(1)$, dato che ogni transizione consiste in esattamente 2 casi.

La complessità è quindi $O(N \cdot 1) = O(N)$.

Programmazione Dinamica

Problema

```cpp
int memo[MAXN];
bool vis[MAXN];
//stato: N = numero di case del prefisso
int solve(int N) {
    if (N < 0) return 0;
    if (vis[N]) return memo[N];
    vis[N] = true;

    //transizione
    int prendo = V[N] + solve(N - 2);
    int nonPrendo = solve(N - 1);

    return memo[N] = max(prendo, nonPrendo);
}
```

Programmazione Dinamica

Problema: Hateville

Riusciamo a risolvere il problema anche in maniera iterativa?

Programmazione Dinamica

Problema: Hateville

Riusciamo a risolvere il problema anche in maniera iterativa?

- supponiamo di aver processato le prime $i - 1$ case.
- possiamo trovare la soluzione per le prime i case?

Programmazione Dinamica

Problema: Hateville

Riusciamo a risolvere il problema anche in maniera iterativa?

- supponiamo di aver processato le prime $i-1$ case.
- possiamo trovare la soluzione per le prime i case?

Per ogni i, calcoliamo due valori:

- `prendi[i]`: somma massima considerando le prime i case e prendendo la casa i
- `lascia[i]`: somma massima considerando le prime i case e <u>non</u> prendendo la casa i

Programmazione Dinamica

Problema: Hateville

Se stiamo considerando 0 case, chiaramente `prendi[0] = lascia[0] = 0`.

Programmazione Dinamica

Problema: Hateville

Se stiamo considerando 0 case, chiaramente
`prendi[0] = lascia[0] = 0`.

Altrimenti

- `prendi[i] = lascia[i-1] + v[i]`
- `lascia[i] = max(prendi[i-1], lascia[i-1])`

Programmazione Dinamica

Problema: Hateville

Se stiamo considerando 0 case, chiaramente `prendi[0] = lascia[0] = 0`.

Altrimenti

- `prendi[i] = lascia[i-1] + v[i]`
- `lascia[i] = max(prendi[i-1], lascia[i-1])`

La risposta è `max(prendi[n], lascia[n])`.

Programmazione Dinamica

Problema: Hateville

Se stiamo considerando 0 case, chiaramente `prendi[0] = lascia[0] = 0`.

Altrimenti

- `prendi[i] = lascia[i-1] + v[i]`
- `lascia[i] = max(prendi[i-1], lascia[i-1])`

La risposta è `max(prendi[n], lascia[n])`.

La complessità di tempo è sempre $O(N)$, anche questa soluzione è veloce!

Programmazione Dinamica

Implementazione di Hateville

```cpp
const int MAXN = 1e6 + 1;

int prendi[MAXN];
int lascia[MAXN];

prendi[0] = lascia[0] = 0;
for (int i = 1; i <= n; ++i) {
    prendi[i] = lascia[i - 1] + v[i];
    lascia[i] = max(prendi[i - 1], lascia[i - 1]);
}

cout << max(prendi[n], lascia[n]) << "\n";
```

Frog 1

Ci sono N pietre numerate da 1 a N, ognuna delle quali ha un altezza h_i. Una rana è inizialmente sulla pietra 1 e vuole raggiungere la pietra N. Se la rana si trova su i, può saltare su $i+1$ o su $i+2$ ad un costo di $|h_i - h_j|$, dove j è la pietra su cui atterra. Trova il minimo costo per raggiungere N.

Frog 1

Ci sono N pietre numerate da 1 a N, ognuna delle quali ha un altezza h_i. Una rana è inizialmente sulla pietra 1 e vuole raggiungere la pietra N. Se la rana si trova su i, può saltare su $i+1$ o su $i+2$ ad un costo di $|h_i - h_j|$, dove j è la pietra su cui atterra. Trova il minimo costo per raggiungere N.

Anche qui possiamo definire il problema ricorsivamente:

$$dp[i] = \begin{cases} 0 & \text{per } i = N \\ \begin{aligned} \min(&dp[i+1] + |h_i - h_{i+1}|, \\ &dp[i+2] + |h_i - h_{i+2}|) \end{aligned} & \text{per } i < N \end{cases}$$

Frog 1

Ci sono N pietre numerate da 1 a N, ognuna delle quali ha un altezza h_i. Una rana è inizialmente sulla pietra 1 e vuole raggiungere la pietra N. Se la rana si trova su i, può saltare su $i+1$ o su $i+2$ ad un costo di $|h_i - h_j|$, dove j è la pietra su cui atterra. Trova il minimo costo per raggiungere N.

Anche qui possiamo definire il problema ricorsivamente:

$$dp[i] = \begin{cases} 0 & \text{per } i = N \\ \min(dp[i+1] + |h_i - h_{i+1}|, \\ \qquad dp[i+2] + |h_i - h_{i+2}|) & \text{per } i < N \end{cases}$$

https://atcoder.jp/contests/dp/tasks/dp_a

Programmazione Dinamica

Problema

```cpp
int memo[MAXN];
bool vis[MAXN];
int solve(int pos) {
    if (pos == N) return 0;
    if (vis[pos]) return memo[pos];
    vis[pos] = true;

    int salto1 = solve(pos + 1) + abs(h[pos] - h[pos + 1]);
    if (pos + 1 == N) return memo[pos] = salto1;
    int salto2 = solve(pos + 2) + abs(h[pos] - h[pos + 2]);
    return memo[pos] = min(salto1, salto2);
}
```

Problemi addizionali

https://training.olinfo.it/#/task/ois_monopoly/statement
https://training.olinfo.it/#/task/preoii_treni/statement
https://training.olinfo.it/#/task/luiss_suppli/statement
https://training.olinfo.it/#/task/sommelier/statement
https://training.olinfo.it/#/task/poldo/statement
https://training.olinfo.it/#/task/luiss_laurea/statement