Ricorsione e Programmazione dinamica

Palestra di algoritmi 2020/2021

Funzioni ricorsive

- In C/C++ (e quasi tutti gli altri linguaggi) una funzione può invocare se stessa (funzione ricorsiva)
- ... o due o più funzioni possono chiamarsi a vicenda (funzioni mutualmente ricorsive)
- Formulare alcuni problemi in maniera ricorsiva risulta naturale:
  - Il **fattoriale**: $0! = 1$; $n! = n * (n-1)!$
  - **pari/dispari**: $\mathrm{even}(n) \Longleftrightarrow \mathrm{odd}(n-1)$; $\mathrm{odd}(n) \Longleftrightarrow \mathrm{even}(n-1)$;
  - **espressioni**: somma = numero; somma = (somma+somma)
- Due componenti:
  - una o più condizioni di terminazione
  - una o più chiamate ricorsive

Rischio di produrre sequenze infinite

Esempio: Fattoriale

factorial(n)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

Possiamo tradurre direttamente la definizione (ricorsiva) in una funzione ricorsiva!

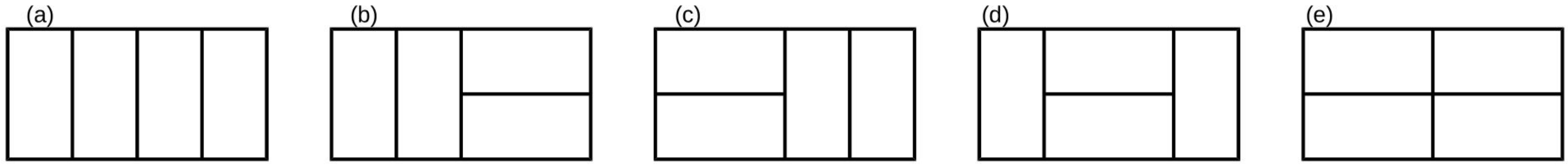
In questo caso è facile anche scrivere una soluzione iterativa ma non sempre è così...

```
using namespace std;
#include <iostream>
/** versione ricorsiva **/
long long factorial_rec (int n) {
  long long res;
  if (n==0)
    res = 1;
  else
    res = n * factorial_rec(n-1);
  return res;
}

int main() {
  int n;
  cout << "n? ";
  cin >> n;
  cout << "fattoriale(" << n << ") = " <<
factorial_rec(n) << endl;
  return 0;
}
```

Esempio: Domino

l gioco del domino è basato su tessere di dimensione 2×1. Scrivere un algoritmo efficiente che prenda in input un intero **n** e restituisca il numero di possibili disposizioni di **n** tessere in un rettangolo 2×n.

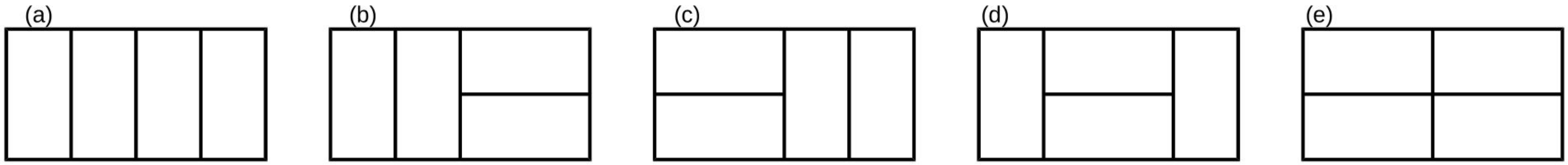


Esempio: Domino

Come possiamo calcolare il risultato?

Casi base:

- $n = 1$: devo riempire un rettangolo $2 \times n$ => 1 solo modo (tessera verticale)
- $n = 0$: ? (0 o 1, vediamo…)



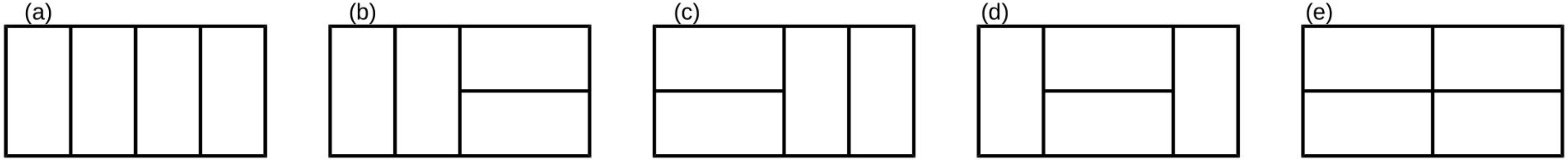
Esempio: Domino

Casi base:

- n = 1: devo riempire un rettangolo 2×n => 1 solo modo (tessera verticale)
- n = 0: ? (0 o 1, vediamo…)

n > 1: Pensiamo di partire dall’ultima tessera. Posso metterla:

- Verticale ➞ poi devo riempire un rettangolo 2×(n-1)
- Orizzontale ➞ sono obbligato a mettere anche quella prima orizzontale. poi devo riempire un rettangolo 2×(n-2)

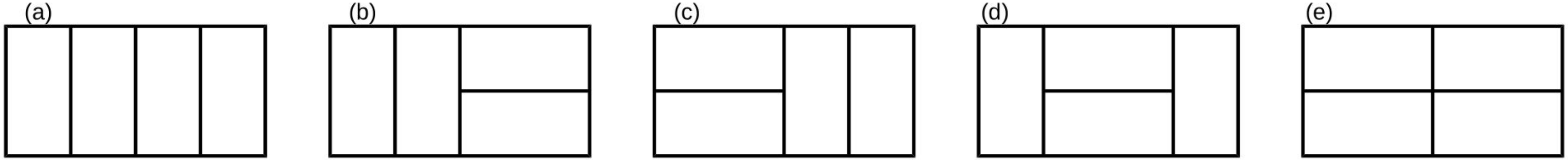


Esempio: Domino

È una definizione ricorsiva!

- Se metto una tessera in verticale, risolverò il problema di dimensione n−1
- Se metto una tessera in orizzontale, ne devo mettere due; risolverò il problema di dimensione n−2. Queste due possibilità si sommano insieme

$$domino[n] = \begin{cases} 1 & \text{if } n \leq 1 \\ domino[n-1] + domino[n-2] & \text{if } n > 1 \end{cases}$$



Esempio: Domino

È la successione di Fibonacci!

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

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

Visualizziamo come viene calcolata…

https://visualgo.net/bn/recursion

$$domino[n] = \begin{cases} 1 & \text{if } n \leq 1 \\ domino[n-1] + domino[n-2] & \text{if } n > 1 \end{cases}$$

Esempio: Domino

È la successione di Fibonacci!

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

Tanti sotto-problemi sono ripetuti! Possiamo memorizzare i risultati già calcolati

```
int domino(int n, int DP[]) {
  if(DP[n] == -1) {
    if(n <= 1)
      DP[n] = 1;
    else
      DP[n] = domino(n-1) + domino(n-2);
  }
  return DP[n];
}
int main() {
  int DP[N];
  for(int i=0; i<N; i++)
    DP[i] = -1;
  cout<<domino(5, DP)<<endl;
  cout<<domino(7, DP)<<endl;
}
```

$$domino[n] = \begin{cases} 1 & \text{if } n \leq 1 \\ domino[n-1] + domino[n-2] & \text{if } n > 1 \end{cases}$$

Top down vs. Bottom up

TOP DOWN (Memoization):

```
int domino_mem(int n, int DP[]) {
  if(DP[n] == -1) {
    if(n <= 1)
      DP[n] = 1;
    else
      DP[n] = domino_mem(n-1) + domino_mem(n-2);
  }
  return DP[n];
}
```

BOTTOM UP (Programmazione dinamica):

```
int domino_dp(int n, int DP[]) {
  DP[0] = DP[1] = 1;
  for (int i = 2; i <= n; i++){
    DP[i] = DP[i-1] + DP[i-2];
  }
  return DP[n];
}
```

Top down vs. Bottom up

TOP DOWN (Memoization):

```
int domino_mem(int n, int DP[]) {
  if(DP[n] == -1) {
    if(n <= 1)
      DP[n] = 1;
    else
      DP[n] = fib_mem(n-1) + fib_mem(n-2);
  }
  return DP[n];
}
```

BOTTOM UP (Programmazione dinamica):

```
int domino_dp(int n, int DP[]) {
  DP[0] = DP[1] = 1;
  for (int i = 2; i <= n; i++){
    DP[i] = DP[i-1] + DP[i-2];
  }
  return DP[n];
}
/** Non serve memorizzare tutto... **/
int domino_dp_opt(int n) {
  if (n <= 1) return 1;
  int n_2 = 1, n_1 = 1;
  for (int i = 2; i <= n; i++){
    int tmp = n_1 + n_2;
    n_2 = n_1;
    n_1 = tmp;
  }
  return n_1;
}
```

Esempio: Hateville

- Hateville è un villaggio particolare, composto da n case, numerate da **1** a **n** lungo una singola strada.
- Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati. Quindi, il vicino **i** odia i vicini **i − 1** e **i + 1** (se esistenti). Hateville vuole organizzare una sagra e vi ha affidato il compito di raccogliere i fondi. Ogni abitante **i** ha intenzione di donare una quantità **D[i]**, ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini.
- **Problema**:
  Scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta

Esempio: Hateville

Chiamiamo HV(i) il valore della donazione ottimale dalle prime i case di Hateville.

Consideriamo il vicino i-esimo.

- Cosa succede se non accetto la sua donazione?
  - $HV(i) = HV(i-1)$
- Cosa succede se accetto la sua donazione?
  - $HV(i) = D[\ i\ ] + HV(i-2)$
- Come faccio a decidere se accettare o meno?

Esempio: Hateville

Chiamiamo HV(i) il valore della donazione ottimale dalle prime i case di Hateville.

Consideriamo il vicino i-esimo.

- Cosa succede se non accetto la sua donazione?
  - $HV(i) = HV(i-1)$
- Cosa succede se accetto la sua donazione?
  - $HV(i) = D[\ i\ ] + HV(i-2)$
- Come faccio a decidere se accettare o meno?
  - Prendo il massimo!
  - $HV(i) = \max(HV(i-1), D[\ i\ ] + HV(i-2))$

Esempio: Hateville

- Il profitto massimo che ottengo dalle prime i case è il massimo tra:
  - non prendo la donazione dell’i-esima casa, guardo la soluzione per le prime i-1.
  - prendo la donazione dell’i-esima casa + il massimo di donazioni tra le prime i-2.

$$HV[i] = \begin{cases} 0 & \text{if } i = 0 \\ D[0] & \text{if } i = 1 \\ \max\left(HV[i-1], D[i-1] + HV[i-2]\right) & \text{if } i > 1 \end{cases}$$

Esempio: Hateville

Versione bottom-up iterativa

```
int hateville(int D[], int n) {
  int DP[MAX];
  DP[0] = 0;
  DP[1] = D[0];
  for (int i=2; i <= n; i++) {
    DP[i] = max(DP[i-1],DP[i-2]+D[i-1]);
  }
  return DP[n];
}
```

$$HV[i] = \begin{cases} 0 & \text{if } i = 0 \\ D[0] & \text{if } i = 1 \\ \max\left(HV[i-1], D[i-1] + HV[i-2]\right) & \text{if } i > 1 \end{cases}$$

Esempio: Hateville

Come viene riempita la tabella DP?

$$HV[i] = \begin{cases} 0 & \text{if } i = 0 \\ D[0] & \text{if } i = 1 \\ \max\left(HV[i-1], D[i-1] + HV[i-2]\right) & \text{if } i > 1 \end{cases}$$

i	0	1	2	3	4	5	6	7
D	10	5	5	8	4	7	12	
DP	0	10	10	15	18	19	25	**31**

Esempio: Hateville

Possiamo risalire alla soluzione ottimale

$$HV[i] = \begin{cases} 0 & \text{if } i = 0 \\ D[0] & \text{if } i = 1 \\ \max\left(HV[i-1], D[i-1] + HV[i-2]\right) & \text{if } i > 1 \end{cases}$$

i	0	1	2	3	4	5	6	7
D	**10**	5	**5**	8	**4**	7	**12**	
DP	0	**10**	10	**15**	18	**19**	25	**31**

Esercizi

- Day 2

    Provate a risolverli con la ricorsione.

    Quando riescono, aggiungete memoization (o DP) per renderli più efficienti!

References

- Funzioni ricorsive: http://disi.unitn.it/~rseba/DIDATTICA/prog1_2020/SLIDES_HANDOUTS/05_FUNZIONI_HANDOUTS.pdf
- Programmazione dinamica: http://disi.unitn.it/~montreso/asd/slides/13-pd1.pdf
- Guida alle olimpiadi di informatica Prof. Bugatti