Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Le funzioni ricorsive

Catia Prandi - catia.prandi2@unibo.it

Credit: Pietro Di Lena

In order to understand recursion you must first understand recursion.

Introduzione

- La **ricorsione** è un processo che permette di definire qualcosa in termini di se stesso.

Introduzione

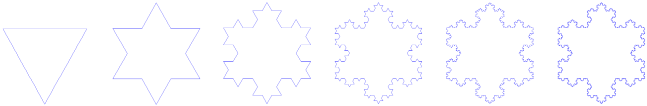
- La **ricorsione** è un processo che permette di definire qualcosa in termini di se stesso.
- Possiamo trovare esempi di applicazione della ricorsione in tutte (o quasi) le discipline umane: dalla matematica, all'arte.
- In informatica la *ricorsione* è una tecnica di programmazione molto potente.
- E' supportata in quasi tutti i linguaggi di programmazione di alto livello.
- Il linguaggio C permette di definire **funzioni ricorsive**, cioè funzioni che *richiamano se stesse*.

La ricorsione in geometria

- Un esempio molto comune di ricorsione in geometria sono i **frattali**: oggetti geometrici che ripetono la propria forma su scale diverse.

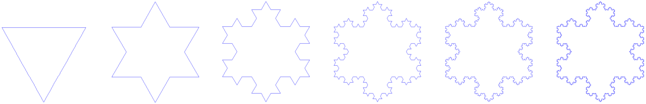
La ricorsione in geometria

- Un esempio molto comune di ricorsione in geometria sono i **frattali**: oggetti geometrici che ripetono la propria forma su scale diverse.
- Il fiocco di neve di Koch (1904) è uno dei primi esempi di *curva frattale* descritta in letteratura.



La ricorsione in geometria

- Un esempio molto comune di ricorsione in geometria sono i **frattali**: oggetti geometrici che ripetono la propria forma su scale diverse.
- Il fiocco di neve di Koch (1904) è uno dei primi esempi di *curva frattale* descritta in letteratura.



- Il triangolo di Sierpisnki (1915) ha diverse proprietà matematiche.

La ricorsione in linguistica

- In linguistica, la ricorsività interviene quando applichiamo una regola al risultato di una precedente applicazione della stessa regola.

La ricorsione in linguistica

- In linguistica, la ricorsività interviene quando applichiamo una regola al risultato di una precedente applicazione della stessa regola.
- Possiamo, ad esempio, definire **acronimi ricorsivi**:

GNU ⇒	**GNU** Is Not UniX	⇒ **GNU Is Not UniX** Is Not UniX
PHP ⇒	**PHP**: Hypertext Processor	⇒ **PHP: Hypertext Processor**: Hypertext Processor

La ricorsione in linguistica

- In linguistica, la ricorsività interviene quando applichiamo una regola al risultato di una precedente applicazione della stessa regola.
- Possiamo, ad esempio, definire **acronimi ricorsivi**:

GNU ⇒ **GNU** Is Not UniX ⇒ **GNU Is Not UniX** Is Not UniX
PHP ⇒ **PHP**: Hypertext Processor ⇒ **PHP: Hypertext Processor**: Hypertext Processor

- Possiamo definire **storie ricorsive** (intuiamo come continuerà la storia):

C'era una volta un Re
seduto su un sofà
che disse alla sua serva
raccontami una storia
e la serva cominciò:
"C'era una volta un Re
seduto sul sofà
che disse alla sua serva
raccontami una storia
e la serva cominciò:
"C'era una volta un Re
...

La ricorsione in linguistica

- In linguistica, la ricorsività interviene quando applichiamo una regola al risultato di una precedente applicazione della stessa regola.
- Possiamo, ad esempio, definire **acronimi ricorsivi**:

GNU ⇒ **GNU** Is Not UniX ⇒ **GNU Is Not UniX** Is Not UniX
PHP ⇒ **PHP**: Hypertext Processor ⇒ **PHP: Hypertext Processor**: Hypertext Processor

- Possiamo definire **storie ricorsive** (intuiamo come continuerà la storia):

C'era una volta un Re
seduto su un sofà
che disse alla sua serva
raccontami una storia
e la serva cominciò:
"C'era una volta un Re
seduto sul sofà
che disse alla sua serva
raccontami una storia
e la serva cominciò:
"C'era una volta un Re
...

- Anche Google ci suggerisce ricorsivamente la ricerca della parola recursion.

La ricorsione in matematica

- In matematica la ricorsione è spesso utilizzata per definire una funzione in termini di se stessa.

La ricorsione in matematica

- In matematica la ricorsione è spesso utilizzata per definire una funzione in termini di se stessa.
- La sequenza di Fibonacci (1202) è un esempio molto noto di successione di numeri interi positivi che ha una definizione ricorsiva.
- Nella sequenza di Fibonacci, ogni numero è dato dalla somma dei due precedenti. I primi due numeri della successione sono $F_0 = 0$ e $F_1 = 1$.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

- Possiamo definire ricorsivamente questa successione di numeri:

$$F_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F_{n-1} + F_{n-2} & \text{altrimenti} \end{cases}$$

La ricorsione in matematica

- In matematica la ricorsione è spesso utilizzata per definire una funzione in termini di se stessa.
- La sequenza di Fibonacci (1202) è un esempio molto noto di successione di numeri interi positivi che ha una definizione ricorsiva.
- Nella sequenza di Fibonacci, ogni numero è dato dalla somma dei due precedenti. I primi due numeri della successione sono $F_0 = 0$ e $F_1 = 1$.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

- Possiamo definire ricorsivamente questa successione di numeri:

$$F_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F_{n-1} + F_{n-2} & \text{altrimenti} \end{cases}$$

- La sequenza di Fibonacci nasce come intento di formalizzare matematicamente il rate di crescita di una popolazione di conigli.
- La sequenza ha legami con numerosi settori, oltre a quelli affini alla matematica, tra cui: botanica, cristallografia, musica, ecc.

La ricorsione nella teoria della calcolabilità

- I teoremi di ricorsione di Kleene (1938), sono due importanti risultati della *teoria della calcolabilità*.
- I due teoremi dimostrano alcune proprietà fondamentali delle *funzioni calcolabili*.

La ricorsione nella teoria della calcolabilità

- I teoremi di ricorsione di Kleene (1938), sono due importanti risultati della *teoria della calcolabilità*.
- I due teoremi dimostrano alcune proprietà fondamentali delle *funzioni calcolabili*.
- Come conseguenza di uno dei due teoremi di ricorsione, abbiamo che con un linguaggio di programmazione sufficientemente potente (i.e. *Turing-completo*), siamo in grado di scrivere programmi che calcolino proprietà del proprio stesso codice:
  - programmi che calcolano/stampano il proprio md5sum;
  - programmi che stampano il proprio codice sorgente (chiamati **quine**);
  - ecc.
- Esempio di quine nel linguaggio C:

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){printf(f,34,f,34,10);}
```

La ricorsione nella teoria della calcolabilità

- I teoremi di ricorsione di Kleene (1938), sono due importanti risultati della *teoria della calcolabilità*.
- I due teoremi dimostrano alcune proprietà fondamentali delle *funzioni calcolabili*.
- Come conseguenza di uno dei due teoremi di ricorsione, abbiamo che con un linguaggio di programmazione sufficientemente potente (i.e. *Turing-completo*), siamo in grado di scrivere programmi che calcolino proprietà del proprio stesso codice:
  - programmi che calcolano/stampano il proprio md5sum;
  - programmi che stampano il proprio codice sorgente (chiamati **quine**);
  - ecc.
- Esempio di quine nel linguaggio C:

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){printf(f,34,f,34,10);}
```

- I teoremi di ricorsione hanno anche (e soprattutto) conseguenze *negative*: non può esistere un algoritmo che verifichi una qualsiasi proprietà (se non banale) degli algoritmi scritti nello stesso linguaggio (se questo è Turing-completo).

Algortimi ricorsivi

- In informatica, un **algoritmo ricorsivo** è un algoritmo espresso in termini di se stesso.
- Più nello specifico, parliamo di
  - **ricorsione diretta**: quando un algoritmo è espresso in termini di se stesso;
  - **ricorsione indiretta** o **mutua ricorsione**: quando due algoritmi si invocano reciprocamente.

Algortimi ricorsivi

- In informatica, un **algoritmo ricorsivo** è un algoritmo espresso in termini di se stesso.
- Più nello specifico, parliamo di
  - **ricorsione diretta**: quando un algoritmo è espresso in termini di se stesso;
  - **ricorsione indiretta** o **mutua ricorsione**: quando due algoritmi si invocano reciprocamente.
- Una tecnica di programmazione molto comune è quella di suddividere un problema in istanze più semplici dello stesso problema, risolvere tali istanze in modo ricorsivo, e combinare i risultati per ottenere la soluzione finale.
- Tale tecnica è solitamente indicata col nome **divide et impera**.

Algortimi ricorsivi

- In informatica, un **algoritmo ricorsivo** è un algoritmo espresso in termini di se stesso.
- Più nello specifico, parliamo di
  - **ricorsione diretta**: quando un algoritmo è espresso in termini di se stesso;
  - **ricorsione indiretta** o **mutua ricorsione**: quando due algoritmi si invocano reciprocamente.
- Una tecnica di programmazione molto comune è quella di suddividere un problema in istanze più semplici dello stesso problema, risolvere tali istanze in modo ricorsivo, e combinare i risultati per ottenere la soluzione finale.
- Tale tecnica è solitamente indicata col nome **divide et impera**.
- Ogni algoritmo ricorsivo è *simulabile* con un algoritmo iterativo, e viceversa.
- Alcuni problemi hanno una soluzione elegante ed intuitiva tramite algoritmi ricorsivi, mentre risultano essere estremamente complessi da risolvere in modo iterativo.

Algortimi ricorsivi

- In informatica, un **algoritmo ricorsivo** è un algoritmo espresso in termini di se stesso.
- Più nello specifico, parliamo di
  - **ricorsione diretta**: quando un algoritmo è espresso in termini di se stesso;
  - **ricorsione indiretta** o **mutua ricorsione**: quando due algoritmi si invocano reciprocamente.
- Una tecnica di programmazione molto comune è quella di suddividere un problema in istanze più semplici dello stesso problema, risolvere tali istanze in modo ricorsivo, e combinare i risultati per ottenere la soluzione finale.
- Tale tecnica è solitamente indicata col nome **divide et impera**.
- Ogni algoritmo ricorsivo è *simulabile* con un algoritmo iterativo, e viceversa.
- Alcuni problemi hanno una soluzione elegante ed intuitiva tramite algoritmi ricorsivi, mentre risultano essere estremamente complessi da risolvere in modo iterativo.
- Per risolvere un problema tramite un algoritmo ricorsivo è necessario:
  1. individuare uno o più **casi base**: insieme di valori in input per cui la funzione termina immediatamente.
  2. individuare uno o più **casi ricorsivi**: insieme di valori in input per cui la funzione richiama se stessa.

La ricorsione nei linguaggi di programmazione

- La ricorsione è supportata nei linguaggi di programmazione che permettono alle funzioni di richiamare se stesse.
- Attualmente, la ricorsione è supportata da praticamente tutti i linguaggi ad alto livello.
- Vecchie versioni di BASIC e FORTRAN sono esempi di linguaggi che non supportano la ricorsione.

La ricorsione nei linguaggi di programmazione

- La ricorsione è supportata nei linguaggi di programmazione che permettono alle funzioni di richiamare se stesse.
- Attualmente, la ricorsione è supportata da praticamente tutti i linguaggi ad alto livello.
- Vecchie versioni di BASIC e FORTRAN sono esempi di linguaggi che non supportano la ricorsione.
- Tipicamente i *linguaggi funzionali*, come LISP e OCaml, non hanno nessuna struttura di controllo iterativa ma fanno uso unicamente della ricorsione per poter eseguire iterativamente una serie di istruzioni.
- In particolare, i linguaggi funzionali supportano un particolare tipo di ricorsione, chiamato **ricorsione in coda** (**tail recursion**).
- La ricorsione in coda può essere implementata efficientemete (**tail call elimination**) senza dover gestire sullo stack l'intera catena di chiamate ricorsive.

La ricorsione nel linguaggio C

- Le funzioni del linguaggio C supportano la ricorsione.
- Anche la funzione `main()` può essere richiamata ricorsivamente, anche se non è in generale buona norma invocare il `main()` all'interno di un programma C.

La ricorsione nel linguaggio C

- Le funzioni del linguaggio C supportano la ricorsione.
- Anche la funzione `main()` può essere richiamata ricorsivamente, anche se non è in generale buona norma invocare il `main()` all'interno di un programma C.
- Abbiamo visto che in C l'esecuzione di una funzione è gestita tramite lo stack call.
- Ogni chiamata ricorsiva comporta l'allocazione di un record di attivazione sullo stack.
- Le variabili locali e i parametri formali non sono *condivisi* tra le varie chiamate ricorsive: ogni record di attivazione ha la propria copia locale di parametri e variabili della funzione.

La ricorsione nel linguaggio C

- Le funzioni del linguaggio C supportano la ricorsione.
- Anche la funzione `main()` può essere richiamata ricorsivamente, anche se non è in generale buona norma invocare il `main()` all'interno di un programma C.
- Abbiamo visto che in C l'esecuzione di una funzione è gestita tramite lo stack call.
- Ogni chiamata ricorsiva comporta l'allocazione di un record di attivazione sullo stack.
- Le variabili locali e i parametri formali non sono *condivisi* tra le varie chiamate ricorsive: ogni record di attivazione ha la propria copia locale di parametri e variabili della funzione.
- Se la funzione ricorsiva ha molti parametri o molte variabili locali, la gestione di ogni record di attivazione diventa onerosa, in termini di tempo di esecuzione.
- Inoltre, livelli di ricorsione molto profondi possono facilmente saturare lo spazio a disposizione sullo stack, causando uno stack overflow.

La ricorsione nel linguaggio C

- Le funzioni del linguaggio C supportano la ricorsione.
- Anche la funzione `main()` può essere richiamata ricorsivamente, anche se non è in generale buona norma invocare il `main()` all'interno di un programma C.
- Abbiamo visto che in C l'esecuzione di una funzione è gestita tramite lo stack call.
- Ogni chiamata ricorsiva comporta l'allocazione di un record di attivazione sullo stack.
- Le variabili locali e i parametri formali non sono *condivisi* tra le varie chiamate ricorsive: ogni record di attivazione ha la propria copia locale di parametri e variabili della funzione.
- Se la funzione ricorsiva ha molti parametri o molte variabili locali, la gestione di ogni record di attivazione diventa onerosa, in termini di tempo di esecuzione.
- Inoltre, livelli di ricorsione molto profondi possono facilmente saturare lo spazio a disposizione sullo stack, causando uno stack overflow.
- Argomenti a favore e a sfavore della ricorsione in C:
  - PRO: permette di scrivere in poche righe di codice, ed in modo chiaro, algoritmi estremamente complessi.
  - CONTRO: rispetto ad una procedura iterativa, ha un costo in termini di efficienza causato dall'overhead associato alla gestione del record di attivazione.

Calcolo del fattoriale: 1/2

- Abbiamo già visto come calcolare iterativamente il fattoriale di un intero positivo n:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot .... \cdot 1 \quad \text{e assumiamo } 0! = 1$$

- Vediamo come definire un algoritmo ricorsivo che calcoli il fattoriale.

Calcolo del fattoriale: 1/2

- Abbiamo già visto come calcolare iterativamente il fattoriale di un intero positivo n:
$$n! = n \cdot (n-1) \cdot (n-2) \cdot .... \cdot 1 \quad \text{e assumiamo } 0! = 1$$
- Vediamo come definire un algoritmo ricorsivo che calcoli il fattoriale.
- Ricordiamo che per risolvere un problema con un algoritmo ricorsivo è necessario:
  1. individuare uno o più **casi base**: insieme di valori in input per cui la funzione termina immediatamente.
  2. individuare uno o più **casi ricorsivi**: insieme di valori in input per cui la funzione richiama se stessa.

Calcolo del fattoriale: 1/2

- Abbiamo già visto come calcolare iterativamente il fattoriale di un intero positivo n:
$$n! = n \cdot (n-1) \cdot (n-2) \cdot .... \cdot 1 \quad \text{e assumiamo } 0! = 1$$
- Vediamo come definire un algoritmo ricorsivo che calcoli il fattoriale.
- Ricordiamo che per risolvere un problema con un algoritmo ricorsivo è necessario:
  1. individuare uno o più **casi base**: insieme di valori in input per cui la funzione termina immediatamente.
  2. individuare uno o più **casi ricorsivi**: insieme di valori in input per cui la funzione richiama se stessa.
- Prima di passare all'implementazione, notiamo che la funzione fattoriale può essere facilmente definita in termini di se stessa (caso ricorsivo):
$$n! = n \cdot (n-1)!$$

Calcolo del fattoriale: 1/2

- Abbiamo già visto come calcolare iterativamente il fattoriale di un intero positivo n:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot .... \cdot 1 \quad \text{e assumiamo } 0! = 1$$

- Vediamo come definire un algoritmo ricorsivo che calcoli il fattoriale.
- Ricordiamo che per risolvere un problema con un algoritmo ricorsivo è necessario:
  1. individuare uno o più **casi base**: insieme di valori in input per cui la funzione termina immediatamente.
  2. individuare uno o più **casi ricorsivi**: insieme di valori in input per cui la funzione richiama se stessa.
- Prima di passare all'implementazione, notiamo che la funzione fattoriale può essere facilmente definita in termini di se stessa (caso ricorsivo):

$$n! = n \cdot (n-1)!$$

- Questa definizione ricorsiva funziona fintanto che $n \neq 0$, diversamente dobbiamo valutare l'espressione non definita $(-1)!$. Il caso $n = 0$ è quindi il nostro caso base:

$$\text{fact}(n) = \begin{cases} 1 & \text{se } n = 0 \quad \text{(caso base)} \\ n \cdot \text{fact}(n-1) & \text{altrimenti (caso ricorsivo)} \end{cases}$$

Calcolo del fattoriale: 2/2

- Implementazione ricorsiva:

```c
unsigned long int fact(unsigned int n) {
  if(n==0)            // caso base
    return 1;
  else                // caso ricorsivo
    return n*fact(n-1);
}
```

Calcolo del fattoriale: 2/2

- Implementazione ricorsiva:

```
unsigned long int fact(unsigned int n) {
  if(n==0)             // caso base
    return 1;
  else                 // caso ricorsivo
    return n*fact(n-1);
}
```

- Implementazione iterativa:

```
unsigned long int fact(unsigned int n) {
  unsigned long int res=1;

  for(; n>0; n--) res *= n;
  return res;
}
```

Calcolo della potenza: 1/3

- Vediamo come definire un algoritmo ricorsivo che calcoli la potenza n^m tra due interi.
- Per semplificare il problema, consideriamo per il momento solo il caso in cui $m \geq 0$.
- Come visto precedentemente, dobbiamo individuare un *caso base* e un *caso ricorsivo*.

Calcolo della potenza: 1/3

- Vediamo come definire un algoritmo ricorsivo che calcoli la potenza n^m tra due interi.
- Per semplificare il problema, consideriamo per il momento solo il caso in cui $m \geq 0$.
- Come visto precedentemente, dobbiamo individuare un *caso base* e un *caso ricorsivo*.
- Notiamo che la funzione potenza può essere definita in termini di se stessa (caso ricorsivo):

$$n^m = n \cdot n^{m-1}$$

Calcolo della potenza: 1/3

- Vediamo come definire un algoritmo ricorsivo che calcoli la potenza n^m tra due interi.
- Per semplificare il problema, consideriamo per il momento solo il caso in cui $m \geq 0$.
- Come visto precedentemente, dobbiamo individuare un *caso base* e un *caso ricorsivo*.
- Notiamo che la funzione potenza può essere definita in termini di se stessa (caso ricorsivo):

$$n^m = n \cdot n^{m-1}$$

- L'espressione ricorsiva non è ben definita quando $m = 0$, o meglio, n^{-1} non è ben definito dato che accettiamo solo potenze positive. Questo è il nostro caso base:

$$\text{power}(n, m) = \begin{cases} 1 & \text{se } m = 0 \quad \text{(caso base)} \\ n \cdot \text{power}(n, m-1) & \text{altrimenti (caso ricorsivo)} \end{cases}$$

Calcolo della potenza: 2/3

- Implementazione ricorsiva:

```
double power(int n, unsigned int m) {
  if(m==0)   // Caso base
    return 1;
  else       // Caso ricorsivo
    return n*power(n,m-1);
}
```

Calcolo della potenza: 2/3

- Implementazione ricorsiva:

```c
double power(int n, unsigned int m) {
  if(m==0)    // Caso base
    return 1;
  else        // Caso ricorsivo
    return n*power(n,m-1);
}
```

- Implementazione iterativa:

```c
double power(int n, unsigned int m) {
  double pow = 1;

  for(; m>0; m--) pow *= n;
  return pow;
}
```

Calcolo della potenza: 3/3

- Come definiamo ricorsivamente la funzione potenza n^m se vogliamo ammettere $m \leq 0$?

Calcolo della potenza: 3/3

- Come definiamo ricorsivamente la funzione potenza n^m se vogliamo ammettere $m \leq 0$?
- Potremmo gestire il caso $m \leq 0$ con una funzione *ad hoc* notando che

$$n^m = \frac{1}{n} \cdot n^{m+1}$$

che ci porta a definire la seguente equazione ricorsiva:

$$\text{neg\_power}(n, m) = \begin{cases} 1 & \text{se } m = 0 \quad \text{(caso base)} \\ (1/n) \cdot \text{neg\_power}(n, m+1) & \text{altrimenti (caso ricorsivo)} \end{cases}$$

Calcolo della potenza: 3/3

- Come definiamo ricorsivamente la funzione potenza n^m se vogliamo ammettere $m \leq 0$?
- Potremmo gestire il caso $m \leq 0$ con una funzione *ad hoc* notando che

$$n^m = \frac{1}{n} \cdot n^{m+1}$$

che ci porta a definire la seguente equazione ricorsiva:

$$\text{neg\_power}(n, m) = \begin{cases} 1 & \text{se } m = 0 \quad \text{(caso base)} \\ (1/n) \cdot \text{neg\_power}(n, m+1) & \text{altrimenti (caso ricorsivo)} \end{cases}$$

- Questa soluzione potrebbe dare problemi di approssimazione numerica, dato che la frazione $1/n$ non è sempre rappresentabile esattamente.
- Una soluzione semplice è quella di gestire separatamente i due casi tramite una funzione che richiama la procedura per il calcolo di potenze con esponente positivo.

```c
double pos_power(int n, unsigned int m) {
  if(m==0)    // Caso base
    return 1;
  else        // Caso ricorsivo
    return n*pos_power(n,m-1);
}

double power(int n, int m) {
  if(m>=0) return pos_power(n,m);
  else     return 1/pos_power(n,-m);
}
```

Pari o dispari con mutua ricorsione

- Un esempio molto semplice e *artificioso* di mutua ricorsione è il calcolo della **parità** di un numero positivo.
- Utilizziamo una coppia di funzioni che si richiamano a vicenda: una gestisce il caso *pari*, `is_even()`, e l'altra il caso *dispari*, `is_odd()`:
  - Decrementiamo di 1 ad ogni chiamata il numero in input n e richiamiamo la funzione *accoppiata*.
  - Indipendentemente dalla funzione di partenza, se quando il numero diventa 0 siamo nella funzione `is_even()`, allora il risultato è true, altrimenti è false.

Pari o dispari con mutua ricorsione

- Un esempio molto semplice e *artificioso* di mutua ricorsione è il calcolo della **parità** di un numero positivo.
- Utilizziamo una coppia di funzioni che si richiamano a vicenda: una gestisce il caso *pari*, `is_even()`, e l'altra il caso *dispari*, `is_odd()`:
  - Decrementiamo di 1 ad ogni chiamata il numero in input n e richiamiamo la funzione *accoppiata*.
  - Indipendentemente dalla funzione di partenza, se quando il numero diventa 0 siamo nella funzione `is_even()`, allora il risultato è true, altrimenti è false.

```c
int is_even(unsigned int n) {
  if(n==0)   // Caso base
    return 1;
  else       // Caso ricorsivo
    return is_odd(n-1);
}

int is_odd(unsigned int n) {
  if(n==0)   // Caso base
    return 0;
  else       // Caso ricorsivo
    return is_even(n-1);
}
```

`is_odd(4)` ⇒ `is_even(3)` ⇒ `is_odd(2)` ⇒ `is_even(1)` ⇒ `is_odd(0)` ⇒ FALSE
`is_even(4)` ⇒ `is_odd(3)` ⇒ `is_even(2)` ⇒ `is_odd(1)` ⇒ `is_even(0)` ⇒ TRUE

Calcolo del massimo comune divisore 1/2

- Il **massimo comune divisore** $\mathrm{MCD}(n, m)$ tra due numeri interi n, m è l'intero più grande per il quale entrambi possono essere divisi.

$$\mathrm{MCD}(2,8) = 2,\ \mathrm{MCD}(9,12) = 3,\ \mathrm{MCD}(9,10) = 1,\ \mathrm{MCD}(9,0) = 9$$

- Per convenzione, se $m = n = 0$ si pone $\mathrm{MCD}(n, m) = 0$.

Calcolo del massimo comune divisore 1/2

- Il **massimo comune divisore** $\mathrm{MCD}(n, m)$ tra due numeri interi n, m è l'intero più grande per il quale entrambi possono essere divisi.

$$\mathrm{MCD}(2, 8) = 2,\ \mathrm{MCD}(9, 12) = 3,\ \mathrm{MCD}(9, 10) = 1,\ \mathrm{MCD}(9, 0) = 9$$

- Per convenzione, se $m = n = 0$ si pone $\mathrm{MCD}(n, m) = 0$.
- Per il calcolo del massimo comune divisore possiamo utilizzare (la versione moderna de) l'**algoritmo di Euclide** (300 a.c. circa). Dati due interi n, m:
  1. Se $m = 0$, allora il risultato è n.
  2. Altrimenti, si calcola il resto r della divisione intera n/m.
  3. Si riparte dal punto 1 con $n = m$ e $m = r$.

Calcolo del massimo comune divisore 1/2

- Il **massimo comune divisore** $MCD(n, m)$ tra due numeri interi n, m è l'intero più grande per il quale entrambi possono essere divisi.

$$MCD(2, 8) = 2,\ MCD(9, 12) = 3,\ MCD(9, 10) = 1,\ MCD(9, 0) = 9$$

- Per convenzione, se $m = n = 0$ si pone $MCD(n, m) = 0$.
- Per il calcolo del massimo comune divisore possiamo utilizzare (la versione moderna de) l'**algoritmo di Euclide** (300 a.c. circa). Dati due interi n, m:
  1. Se $m = 0$, allora il risultato è n.
  2. Altrimenti, si calcola il resto r della divisione intera n/m.
  3. Si riparte dal punto 1 con $n = m$ e $m = r$.
- L'algoritmo di Euclide ha una forte connotazione ricorsiva

$$MCD(n, m) = \begin{cases} n & \text{se } m = 0 \quad \text{(caso base)} \\ MCD(m, n \mod m) & \text{altrimenti (caso ricorsivo)} \end{cases}$$

Calcolo del massimo comune divisore 1/2

- Il **massimo comune divisore** $\mathrm{MCD}(n, m)$ tra due numeri interi n, m è l'intero più grande per il quale entrambi possono essere divisi.

$$\mathrm{MCD}(2,8) = 2,\ \mathrm{MCD}(9,12) = 3,\ \mathrm{MCD}(9,10) = 1,\ \mathrm{MCD}(9,0) = 9$$

- Per convenzione, se $m = n = 0$ si pone $\mathrm{MCD}(n, m) = 0$.
- Per il calcolo del massimo comune divisore possiamo utilizzare (la versione moderna de) l'**algoritmo di Euclide** (300 a.c. circa). Dati due interi n, m:
  1. Se $m = 0$, allora il risultato è n.
  2. Altrimenti, si calcola il resto r della divisione intera n/m.
  3. Si riparte dal punto 1 con $n = m$ e $m = r$.
- L'algoritmo di Euclide ha una forte connotazione ricorsiva

$$\mathrm{MCD}(n, m) = \begin{cases} n & \text{se } m = 0 \quad \text{(caso base)} \\ \mathrm{MCD}(m, n \mod m) & \text{altrimenti (caso ricorsivo)} \end{cases}$$

- Nota1: non dobbiamo preoccuparci se i due interi n, m hanno segno negativo se l'operazione di resto tra interi negativi è ben definita (i.e. operatore % in C).
- Nota2: l'algoritmo termina molto velocemente. I valori in input che richiedono il maggior numero di operazioni sono coppie successive di numeri di Fibonacci.

Calcolo del massimo comune divisore 2/2

- Implementazione ricorsiva:

```c
int mcd(int n, int m) {
  if(m==0)    // caso base
    return n;
  else        // caso ricorsivo
    return mcd(m,n%m);
}
```

- Possiamo notare come il caso ricorsivo presenta alcune differenze rispetto agli esempi precedenti: questo è un esempio di ricorsione in coda.
  - Il valore di ritorno non è un'espressione che utilizza **anche** il valore calcolato con la chiamata ricorsiva ma è **unicamente** il valore restituito dalla chiamata ricorsiva.
  - I record di attivazione di ogni chiamata sono superflui e potrebbero essere eliminati dallo stack: ci interessa solo il valore calcolato dall'ultima chiamata ricorsiva.
  - Questa ottimizzazione (**tail call elimination**) è supportata da alcuni compilatori.

Calcolo del massimo comune divisore 2/2

- Implementazione ricorsiva:

```c
int mcd(int n, int m) {
  if(m==0)    // caso base
    return n;
  else        // caso ricorsivo
    return mcd(m,n%m);
}
```

- Possiamo notare come il caso ricorsivo presenta alcune differenze rispetto agli esempi precedenti: questo è un esempio di ricorsione in coda.
  - Il valore di ritorno non è un'espressione che utilizza **anche** il valore calcolato con la chiamata ricorsiva ma è **unicamente** il valore restituito dalla chiamata ricorsiva.
  - I record di attivazione di ogni chiamata sono superflui e potrebbero essere eliminati dallo stack: ci interessa solo il valore calcolato dall'ultima chiamata ricorsiva.
  - Questa ottimizzazione (**tail call elimination**) è supportata da alcuni compilatori.
- Implementazione iterativa:

```c
int mcd(int n, int m) {
  while(m) {
    int r = n%m;
    n=m;
    m=r;
  }
  return n;
}
```

Ricorsione in coda

- Vediamo come possiamo implementare il calcolo del fattoriale e la potenza con la ricorsione in coda.
- Implementazione con ricorsione in coda del fattoriale

```
unsigned long int tail_fact(unsigned int n, unsigned int long acc) {
  if(n==0)             // caso base
    return acc;
  else                 // caso ricorsivo
    return tail_fact(n-1,n*acc);
}

unsigned long int fact(unsigned int n) { return tail_fact(n,1);}
```

Il parametro `acc` in `tail_fact()` ha la funzione di *accumulare* il risultato finale.

Ricorsione in coda

- Vediamo come possiamo implementare il calcolo del fattoriale e la potenza con la ricorsione in coda.
- Implementazione con ricorsione in coda del fattoriale

```
unsigned long int tail_fact(unsigned int n, unsigned int long acc) {
  if(n==0)          // caso base
    return acc;
  else              // caso ricorsivo
    return tail_fact(n-1,n*acc);
}

unsigned long int fact(unsigned int n) { return tail_fact(n,1);}
```

  Il parametro acc in `tail_fact()` ha la funzione di *accumulare* il risultato finale.
- Implementazione con ricorsione in coda della potenza

```
double tail_power(int n, unsigned int m, double acc) {
  if(m==0)  // Caso base
    return acc;
  else      // Caso ricorsivo
    return tail_power(n,m-1,n*acc);
}
double power(int n, int m) {
  return m<0? tail_power(n,m,1) : 1/tail_power(n,-m,1);
}
```

  Anche in questo caso, il parametro acc ha la funzione di accumulatore.

La funzione di Fibonacci 1/2

- Ricordiamo come è definita ricorsivamente la funzione di Fibonacci

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{altrimenti} \end{cases}$$

La funzione di Fibonacci 1/2

- Ricordiamo come è definita ricorsivamente la funzione di Fibonacci

$$\text{fib}(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{altrimenti} \end{cases}$$

- Implementazione ricorsiva

```c
unsigned long int fib(unsigned int n) {
  if(n<2)        // casi base
    return n;
  else           // caso ricorsivo
    return fib(n-1)+fib(n-2);
}
```

La funzione di Fibonacci 1/2

- Ricordiamo come è definita ricorsivamente la funzione di Fibonacci

$$\text{fib}(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{altrimenti} \end{cases}$$

- Implementazione ricorsiva

```c
unsigned long int fib(unsigned int n) {
  if(n<2)          // casi base
    return n;
  else             // caso ricorsivo
    return fib(n-1)+fib(n-2);
}
```

- Implementazione iterativa

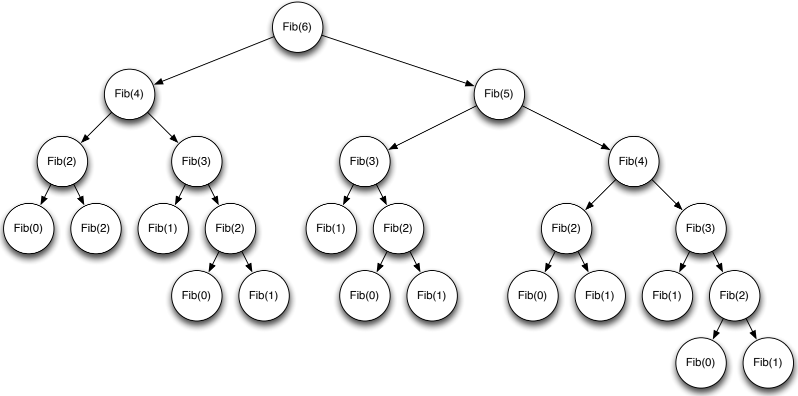
```c
unsigned long int fib(unsigned int n) {
  unsigned long n1=0, n2=1, result, i;

  if(n<2) return n;
  for(i=2; i<=n; i++) {
    result = n1+n2;
    n1     = n2;
    n2     = result;
  }
  return result;
}
```

- Quale delle due implementazioni preferiamo?

La funzione di Fibonacci 2/2

- Implementazione ricorsiva: la chiamata `fib(6)` comporta 24 chiamate ricorsive. Più in generale, una chiamata con input n **grande** comporta circa 1.6^n chiamate ricorsive.



- Implementazione iterativa: la chiamata `fib(6)` comporta l'esecuzione di 5 iterazioni. Più in generale una chiamata con input n comporta n iterazioni.

Le torri di Hanoi 1/5

- Le **Torri di Hanoi** sono un *gioco matematico* inventato dal matematico francese Edouard Lucas (1883):

  Narra una leggenda indiana che all'inizio dei tempi, Brahma portò nel grande tempio di Benares, sotto la cupola d'oro che si trova nel centro del mondo, **tre colonnine** *di diamante e* **sessantaquattro dischi** *d'oro, collocati su una di queste colonnine e* **ordinati dal più grande in basso al più piccolo in alto***. E' la sacra torre di Brahma che vede impiegati, giorno e notte, i sacerdoti del tempio a* **trasferire la torre di dischi dalla prima alla terza colonnina***. Essi devono seguire* **regole** *precise, dettate dallo stesso Brahma, che richiedono di* **spostare un disco alla volta, facendo in modo che nessun disco sia mai posato su uno di diametro inferiore***.*

  Quando i sacerdoti avranno completato il loro lavoro e l'intera torre sarà trasferita sulla terza colonnina, la torre e il tempio crolleranno e il mondo avrà fine.

Le torri di Hanoi 1/5

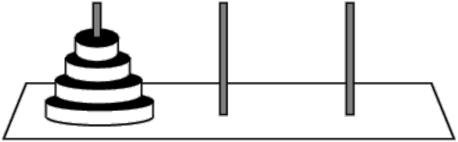
- Le **Torri di Hanoi** sono un *gioco matematico* inventato dal matematico francese Edouard Lucas (1883):

  Narra una leggenda indiana che all'inizio dei tempi, Brahma portò nel grande tempio di Benares, sotto la cupola d'oro che si trova nel centro del mondo, **tre colonnine** *di diamante e* **sessantaquattro dischi** *d'oro, collocati su una di queste colonnine e* **ordinati dal più grande in basso al più piccolo in alto***. E' la sacra torre di Brahma che vede impiegati, giorno e notte, i sacerdoti del tempio a* **trasferire la torre di dischi dalla prima alla terza colonnina***. Essi devono seguire* **regole** *precise, dettate dallo stesso Brahma, che richiedono di* **spostare un disco alla volta, facendo in modo che nessun disco sia mai posato su uno di diametro inferiore***.*
  Quando i sacerdoti avranno completato il loro lavoro e l'intera torre sarà trasferita sulla terza colonnina, la torre e il tempio crolleranno e il mondo avrà fine.

- **Obiettivo del gioco**.
  - Abbiamo tre colonnine e 64 dischi di diametro differente.
  - Il gioco inizia con tutti i dischi incolonnati sulla prima colonnina, ordinati dal più grande (in basso) al più piccolo (in alto).
  - Dobbiamo spostare tutti i dischi dalla prima alla terza colonnina.
- **Regole del gioco**.
  - Deve essere spostato un disco alla volta.
  - Non è possibile posizionare un disco su un altro di diametro inferiore.
- Per poter provare a giocare, vedere qui https://www.frasi.net/giochionline/torre-di-hanoi/default

Le torri di Hanoi 2/5

- Possiamo generalizzare il gioco ad un numero generico n di dischi.



- E' possibile dimostrare che per spostare n dischi da una colonna ad un'altra, seguendo le regole del gioco, è necessario effettuare un numero minimo di $2^n - 1$ mosse.
- I sacerdoti dovranno quindi effettuare almeno $2^{64} - 1 = 18.446.744.073.709.551.615$ mosse per poter completare il loro compito.
- Assumendo una media di una mossa al secondo, i sacerdoti impiegheranno circa 6 miliardi di secoli prima di provocare la fine del mondo.

Le torri di Hanoi 3/5

- Un modo per risolvere il problema è quello di analizzare la sua natura ricorsiva.
- Chiamiamo le colonne `A`, `B`, `C` e numeriamo gli n dischi da `1` a `n`, dove `1` indica il disco più piccolo e `n` il più grande.
- Per poter spostare gli n dischi dalla colonna `A` alla colonna `C` dobbiamo:
  1. spostare i dischi `1, .., n-1` dalla colonna `A` alla colonna `B`;
  2. spostare il disco `n` dalla colonna `A` alla colonna `C`;
  3. spostare i dischi `1, .., n-1` dalla colonna `B` alla colonna `C`.

Le torri di Hanoi 3/5

- Un modo per risolvere il problema è quello di analizzare la sua natura ricorsiva.
- Chiamiamo le colonne `A`, `B`, `C` e numeriamo gli n dischi da `1` a `n`, dove `1` indica il disco più piccolo e `n` il più grande.
- Per poter spostare gli n dischi dalla colonna `A` alla colonna `C` dobbiamo:
  1. spostare i dischi `1, .., n-1` dalla colonna `A` alla colonna `B`;
  2. spostare il disco `n` dalla colonna `A` alla colonna `C`;
  3. spostare i dischi `1, .., n-1` dalla colonna `B` alla colonna `C`.
- Per poter spostare i dischi `1, .., n-1` dalla colonna `A` alla colonna `B` possiamo applicare la stessa procedura ricorsiva, dove questa volta la colonna `A` è il punto di partenza, la colonna `B` il punto di arrivo e la colonna `C` viene utilizzata come *deposito temporaneo*.
- Il caso base dell'algoritmo ricorsivo è banale: ci fermiamo quando non abbiamo altri dischi da spostare.

Le torri di Hanoi 4/5

- Implementazione ricorsiva

```c
void hanoi(unsigned int n, char from, char to, char tmp) {
  if(n>0) {
    // Sposta n-1 dischi da "from" a "tmp"
    hanoi(n-1,from,tmp,to);
    printf("Move disk %d from %c to %c\n",n,from,to);
    // Sposta n-1 dischi da "tmp" a "to"
    hanoi(n-1,tmp,to,from);
  }
}
```

Le torri di Hanoi 4/5

- Implementazione ricorsiva

```c
void hanoi(unsigned int n, char from, char to, char tmp) {
  if(n>0) {
    // Sposta n-1 dischi da "from" a "tmp"
    hanoi(n-1,from,tmp,to);
    printf("Move disk %d from %c to %c\n",n,from,to);
    // Sposta n-1 dischi da "tmp" a "to"
    hanoi(n-1,tmp,to,from);
  }
}
```

- Implementazione iterativa che fa uso di un approccio binario*.

```c
void hanoi(unsigned int n) {
  unsigned int x;
  char from, to;

  for (x=1; x < (1 << n); x++) {
    from =  (x & (x-1)) % 3;
    to   = ((x | (x-1)) + 1) % 3;
    printf( "Move disk from %c to %c.\n",from+65,to+65);
  }
}
```

* L'approccio binario sfrutta due proprietà assolutamente non banali:

1. esiste una corrispondenza biunivoca tra un numero binario a n cifre (dove n è il numero di dischi) e una configurazione *legale* del gioco.
2. ogni mossa può essere effettuata incrementando di 1 il numero binario.

Le torri di Hanoi 5/5

- Albero delle chiamate ricorsive per numero di dischi $n = 3$.

```
hanoi(3,'A','C','B');
+-hanoi(2,'A','B','C');
| +-hanoi(1,'A','C','B');
| | +-hanoi(0,'A','B','C');
| | +-printf("Move disk %d from %c to %c\n",1,'A','C');
| | +-hanoi(0,'B','C','A');
| +-printf("Move disk %d from %c to %c\n",2,'A','B');
| +-hanoi(1,'C','B','A');
|   +-hanoi(0,'C','A','B');
|   +-printf("Move disk %d from %c to %c\n",1,'C','B');
|   +-hanoi(0,'A','B','C');
+-printf("Move disk %d from %c to %c\n",3,'A','C');
+-hanoi(2,'B','C','A');
  +-hanoi(1,'B','A','C');
  | +-hanoi(0,'B','C','A');
  | +-printf("Move disk %d from %c to %c\n",1,'B','A');
  | +-hanoi(0,'C','A','B');
  +-printf("Move disk %d from %c to %c\n",2,'B','C');
  +-hanoi(1,'A','C','B');
    +-hanoi(0,'A','B','C');
    +-printf("Move disk %d from %c to %c\n",1,'A','C');
    +-hanoi(0,'B','C','A');
```