Università Roma La Sapienza
Corsi di Laurea
Informatica/Tecnologie Informatiche

La Ricorsione

Prof. Stefano Guerrini
guerrini@di.uniroma1.it

Programmazione II (can. P-Z)
A.A. 2005-06

Il principio di induzione

Induzione (semplice)

$$\frac{P(0) \qquad P(n) \Rightarrow P(n+1) \ \text{ per ogni } n}{P(n) \ \text{ per ogni } n}$$

Induzione forte o completa

$$\frac{((P(i) \text{ per } 0 \leq i < n) \Rightarrow P(n)) \text{ per ogni } n}{P(n) \text{ per ogni } n}$$

Il fattoriale: iterativo

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \cdots \cdot n$$

```
int fatt_iter(int n) {
  int i, f = 1;
  for (i = 2; i <= n; i++)
      f *= i;
  return f;
}
```

```
int fatt_iter_2(int n) {
  int f;
  for (f = 1; n > 1; n--)
          f *= n;
  return f;
}
```

Posso anche risparmiare la variabile di iterazione

Il fattoriale: ricorsivo

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \cdot n! \end{aligned}$$

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \cdot n! \\ &= (n+1) \cdot \prod_{i=1}^{n} i \end{aligned}$$

```
int fatt_ric(int n) {
  if (n == 0)
    return 1;
  return n * fatt_ric(n-1);
}
```

Algoritmo di Eulero /1

$$\begin{aligned} \mathrm{mcd}(p,0) &= p \\ \mathrm{mcd}(p,q) &= \mathrm{mcd}(q, p \bmod q) \end{aligned}$$

$$\begin{aligned} r = p \bmod q &= p - l \cdot q \\ &= k \cdot \mathrm{MCD}(p.q) - l \cdot h \cdot \mathrm{MCD}(p,q) \\ &= (k - l \cdot h) \cdot \mathrm{MCD}(p,q) \end{aligned}$$

$$\begin{aligned} p &= l \cdot q + r \\ &= l \cdot k' \cdot \mathrm{MCD}(q,r) + h' \cdot \mathrm{MCD}(q,r) \\ &= (l \cdot k' + h') \cdot \mathrm{MCD}(q,r) \end{aligned}$$

Algoritmo di Eulero /2

```
int euler_ric(int p, int q) {
  if (q == 0)
    return p;
  return euler_ric(q, p % q);
}
```

ricorsiva

```
int euler_iter(int p, int q){
  int r;

  while (q != 0) {
    r = p % q;
    p = q;
    q = r;
  }
  return p;
}
```

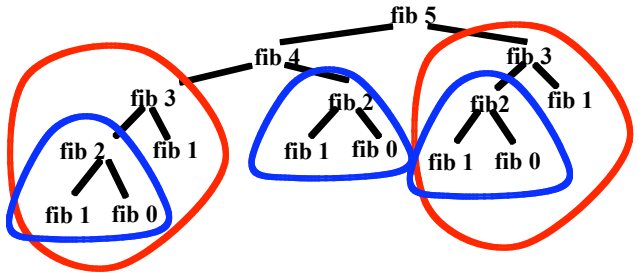
iterativa

Fibonacci

$$\begin{aligned} \mathrm{fib}(n) &= 1 \qquad & n = 0, 1 \\ \mathrm{fib}(n+2) &= \mathrm{fib}(n+1) + \mathrm{fib}(n) \end{aligned}$$

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


Esplosione esponenziale



- numero esponenziale di chiamate ricorsive
- molti calcoli ripetuti

Esponenziale /1

$$\exp(x, 0) = 1$$
$$\exp(x, n+1) = x \cdot \exp(x, n)$$

```
float exp1_ric(float x, int n){
  if (n <= 0)
     return 1;
  return x * exp1_ric(x, n-1);
}
```

```
float exp1_iter(float x, int n) {
  float res = 1.0;

  while (n > 0) {
    res = x * res;
    n--;
  }
  return res;
}
```

Non ci sono grosse differenze tra versione ricorsiva e versione iterativa

Esponenziale /2

$$\exp(x, 0) = 1$$
$$\exp(x, 2 \cdot n + 1) = x \cdot \exp(x, n) \cdot \exp(x, n)$$
$$\exp(x, 2 \cdot n) = \exp(x, n) \cdot \exp(x, n)$$

```
float exp2_ric_d(float x, int n) {
  if (n == 0)
    return 1;
  else if (n % 2 == 1)
    return x * exp2_ric_d(x, n/2) * exp2_ric_d(x, n/2);
  else
    return exp2_ric_d(x, n/2) * exp2_ric_d(x, n/2);
}
```


Esponenziale /3

- Stesso problema della implementazione della funzione di Fibonacci data precedentemente
- Più semplice individuare e porre rimedio all'inutile duplicazione di lavoro

```
float exp2_ric(float x, int n) {
  if (n == 0)
    return 1;
  else {
    float y = exp2_ric(x, n/2);
    if (n % 2 == 1)
      return x * y * y;
    else
      return y * y;
    }
}
```


Esponenziale /4

- La profondità dell'albero delle chiamate ricorsive è ~ log n
- In exp2_ric_d si hanno però n chiamate ricorsive
- In exp2_ric le chiamate ricorsive sono pari alla profondità dell'albero delle chiamate ricorsive (~ log n)

Esponenziale /5

Analoga ottimizzazione della implementazione iterativa

```
float exp2_iter(float x, int n) {
  float res = 1.0;
  int i;

  if (n <= 0)
    return 1;
  for (i = 1, res = x; 2*i < n; i *= 2)
    res *= res;
  for (; i < n; i++)
    res *= x;
  return res;
}
```


Fibonacci efficiente /1

- Fibonacci ha una implementazione iterativa efficiente: n iterazioni (ovvero **lineare** in n)
- Mostrato come **esempio di inefficienza** della ricorsione
- **È proprio vero?**
- L'idea base della soluzione iterativa è memorizzare i due valori di fibonacci per n-1 e n-2.
  Come posso farlo con la ricorsione?

```
int fib_iter(int n) {
  int aux, p0 = 0, p1 = 1;
  while (n > 0) {
    aux = p1;
    p1 = p0 + p1;
    p0 = aux;
    n--;
  }
  return p1;
}
```


Fibonacci efficiente /2

$$\begin{aligned} \text{fib} &: N \to N \times N \\ \text{fib}(0) &= \langle p := 0, q := 1 \rangle \\ \text{fib}(n+1) &= \langle p := \text{fib}(n).q, q := \text{fib}(n).p + \text{fib}(n).q \rangle \end{aligned}$$

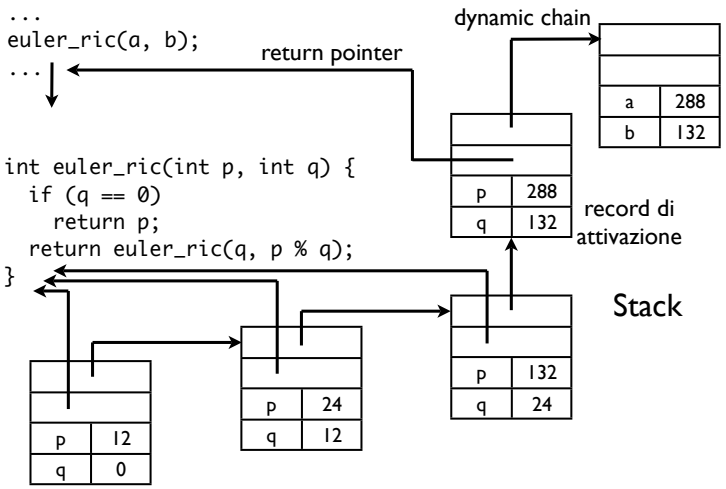
```
struct coppia { int p, q; };

struct coppia fib_lin_p (int n) {
  struct coppia s;
  if (n > 0)  {
    s = fib_lin_p(n-1);
    int aux = s.q;
    s.q = s.p + s.q;
    s.p = aux;
  } else
    s.p = 0, s.q = 1;
  return s;
}
```


Fibonacci efficiente /3

```
void fib_lin (int n, int *p0, int *p1) {
  if (n > 0)  {
    fib_lin(n-1, p0, p1);
    int aux = *p1;
    *p1 = *p0 + *p1;
    *p0 = aux;
  } else
    *p1 = 1, *p0 = 0;
}
```

Come funziona la ricorsione?



Record di attivazione

- Ogni chiamata di funzione implica l'allocazione di un record di attivazione.
- Il record di attivazione contiene
  - le locazioni di memoria per i parametri e per le variabili locali della funzione (**ambiente locale**);
  - l'indirizzo dell'istruzione a cui ritornare il controllo quando la funzione termina (**return pointer**);
  - l'indirizzo del record di attivazione da riattivare al termine della funzion (**dynamic chain**).

Costo della ricorsione /1

- Chiamare una funzione ha un **costo in tempo e spazio** legato all'allocazione del record di attivazione.
- Nel confrontare l'efficienza di un'implementazione iterativa con quella di una ricorsiva dello stesso algoritmo devo tenere conto che ogni chiamata ricorsiva introduce un overhead di tempo dovuto al tempo richiesto per l'allocazione del record di attivazione.

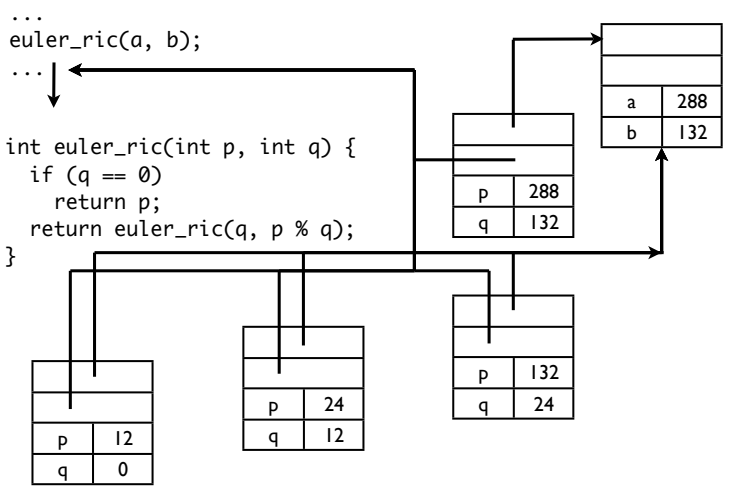
Costo della ricorsione /2

- Dopo una sequenza di n chiamate ricorsive lo stack contiene n record di attivazione della funzione.
- Quindi, lo stack cresce con il numero di chiamate ricorsive non completate.
- Il numero massimo di chiamate ricorsive non completate è detto la **profondità della ricorsione** e nella rappresentazione ad albero delle chiamate ricorsive è pari all'altezza dell'albero.
- Devo garantire che lo stack possa crescere sino a contenere un numero di record di attivazione pari alla profondità della ricorsione.

Tail recursion

- Attenzione! Non sempre è necessario mantenere il record di attivazione corrente al momento di una nuova chiamata ricorsiva.
- Se la chiamata ricorsiva è l'ultima operazione eseguita dalla funzione, **tail recursion** o **ricorsione in coda**, al momento del ritorno della chiamata ricorsiva, anche la funzione chiamante termina.
- Di conseguenza, al momento della chiamata, il record di attivazione della funzione chiamante può essere deallocato, visto che non servirà al ritorno dalla funzione chiamata.

Esempio di tail recursion



La tail recursion e lo stack

- L'ottimizzazione della chiamata di una funzione tail recursive può essere eseguita dal compilatore.
- Tutti i moderni compilatori la fanno, dato che riconoscere se una funzione è tail recursive è molto semplice.
- Nella tail recursion **lo stack non cresce**.

In pratica, un moderno compilatore ottimizzante **trasforma la tail recursion in iterazione!**

Ricorsione: Vantaggi

- I programmi ricorsivi sono più chiari, più semplici, più brevi e più facili da capire delle corrispondenti versioni iterative.
- Il programma riflette fedelmente la strategia di soluzione del problema.
- Se l'efficienza è un punto critico, spesso la soluzione ricorsiva può essere trasformata, più o meno meccanicamente, in una soluzione iterativa equivalente ma più efficiente.

Ricorsione: Svantaggi /1

Spazio

- Ogni chiamata della funzione può richiedere spazio per i parametri e le variabili locali, oltre che per l'indicazione del punto di rientro della chiamata.
- Queste informazioni (record di attivazione) fanno crescere lo stack, sino ad una dimensione massima pari alla profondità della ricorsione.
- Caso particolare: tail recursion. Un compilatore ottimizzante garantisce che lo stack non cresce.

Ricorsione: Svantaggi /2

Tempo.

- Le operazioni coinvolte in una chiamata (allocazione e rilascio della memoria, copia dei valori dei parametri nella memoria locale, rientro dalla chiamata) contribuiscono ad appesantire il tempo di calcolo.
- Questo overhead rimane anche nel caso di tail recursion.

Conclusioni /1

- La ricorsione è uno strumento potente per la progettazione di programmi.
- Ha un costo.
- Attenzione a non sopravvalutare il costo della ricorsione.

Se la ricorsione è **implementata bene**,
i vantaggi sono maggiori degli svantaggi.

Conclusioni /2

- E in alcuni casi la crescita dello stack corrisponde a dati che devono essere comunque memorizzati.
- *Esempio 1*
  Leggere da input una stringa e stamparla in ordine inverso.
- *Esempio 2*
  Verificare se le parentesi in una stringa sono bilanciate, ovvero sono aperte e chiuse correttamente.
  (Visto in dettaglio quando si studierano le pile).

Stampa invertita: ricorsiva

```
int inv_print(int endc) {
  int c = getchar();
  int ret;

  if (c == endc)
    return 0;
  if (c == EOF)
    return -1;
  ret = inv_print(endc);
  putchar(c);
  return ret;
}
```

- Legge l'input sino al carattere endc ricevuto come parametro o a EOF.
- Stampa la sequenza di input letta invertita.
- Ritorna 0 se ha trovato endc o -1 se ha trovato EOF.

Stampa invertita: iterativa

```
int inv_print_iter(int endc) {
  int c, i;
  char s[MAXLEN];

  /* legge */
  for (i = 0, c = getchar();
     c != endc && c != EOF && i < MAXLEN;
     i++, c = getchar())
    s[i] = c;
  /* stampa */
  for (i--; i >= 0; i--)
      putchar(s[i]);
  /* condizione di terminazione */
  if (c == endc)
    return 0;
  return -1;
}
```

- Richiede un buffer per la memorizzazione dell'input letto.
- La dimensione del buffer limita la lunghezza della stringa di input.

Commenti: versione ricorsiva

- Non viene esplicitamente allocata memoria per memorizzare la stringa.
- La stringa non è memorizzata affatto? Come è possibile? Osserviamo che:
  - ogni chiamata ricorsiva legge e memorizza un carattere in una variabile locale;
  - il numero di chiamate ricorsive annidate è pari alla lunghezza della stringa.
- Quindi, la stringa viene memorizzata nello stack della ricorsione.

Commenti: versione iterativa

- Si potrebbe superare il limite sulla lunghezza dell'input usando la memoria dinamica per il buffer, e.g., mantenendo una lista di buffer.
- Però, il programma diviene più pesante (overhead legato alla gestione della memoria dinamica) e complicato.
- Si potrebbe usare una struttura dati opportuna per memorizzare i caratteri letti - uno stack o pila.
- La versione ricorsiva corrisponde alla versione con pila: anziché definire e implementare la pila, sfrutta quella di sistema.