Esercizi Ricorsione

Credits Prof. Campi

Esercizio

- Scrivere un programma C che, dato un numero calcola la somma dei primi N numeri pari positivi in maniera ricorsiva.
- Specifica Liv 1: La somma dei primi N numeri pari è data dalla seguente,

  $S_N = 2*1 + 2*2 + 2*3 + \ldots + 2*i + \ldots + 2*(N-1) + 2*N.$
- Specifica Liv 2:
  - se N =1, $S_N = 2$, **(CASO BASE)**
  - se N >1, $S_N = 2 * N + S_{N-1}$ **(PASSO INDUTTIVO)**

    (somma dell'N-esimo numero pari + la sommatoria dei primi N-1 numeri pari.)

```
int somma_pari(int N) {
        if (N == 1)
        return 2;
        else
        return 2*N + somma_pari(N-1);
}
```

Esercizio

- Calcolo del massimo di un vettore con procedimento ricorsivo.
- Si pensi di assegnare il primo elemento dell'array come massimo e confrontarlo con tutti gli altri cambiando il valore del massimo se questo è minore della cella corrente del vettore.
- Detta N la lunghezza del vettore "array"
  - Se N = 1 **(caso base)**
    - max = array[0]
  - Se N >= 2 **(passo induttivo)**
    - il massimo del vettore di N elementi (max) sarà uguale al risultato del confronto tra array[0] e il massimo degli elementi del sotto-vettore che che va da array[1] a array[N] (lungo N-1).

```
#include <stdio.h>
#define len 10
int maxArray(int *array, int n);

int main() {
        int test[len] = {2, 3, 9, 2, 13, 4, 34, 2, 9, 5};

        printf("\nMax = %d ", maxArray(test,len));

        return 0;
 }
```

```
int maxArray(int *array, int n) {
        int maxsub;
        if (n == 1) return array[0];
        if (n >= 2) {
        maxsub = maxArray(&array[1],n-1);
                if (array[0] > maxsub)
                        return array[0];
                else
                        return maxsub;
  }
  return -1;  /* non raggiungibile */
}
```

Esercizio

- Si progetti la funzione ricorsiva che svolge il compito seguente. Siano dati due vettori V1 e V2, di dimensione N1 e N2, rispettivamente (con $1 \leq N2 \leq N1$). La funzione restituisce il valore 1 in uscita se tutti gli elementi del vettore V2 si trovano nel vettore V1 nell'ordine inverso rispetto a quello in cui essi figurano in V2, ma non necessariamente in posizioni immediatamente consecutive; altrimenti (ovvero se questo non si verifica), la funzione restituisce valore 0.

TRACCIA DI SOLUZIONE

•L'idea "ricorsiva" è di scandire in avanti il vettore V1 dall'elemento 0 all'elemento N1 − 1 e all'indietro il vettore V2 dall'elemento N2 − 1 all'elemento 0, ricorrendo su vettori "accorciati": V1 viene privato dell'elemento iniziale e V2 può (anche se non deve necessariamente) venire privato dell'elemento finale.

•Siccome in C i vettori hanno dimensioni fisse gli "accorciamenti" dei vettori V1 e V2 verranno in realtà codificati tramite due indici, da aggiornare opportunemante di volta in volta. L'indice iniz indica il primo elemento di V1 e viene progressivamente aumentato; l'indice fine indica l'ultimo elemento di V2 e viene progressivamente diminuito.

•BASE (con esito positivo): se il vettore V2 è ormai vuoto termina con esito positivo 1 (un vettore vuoto, senza elementi, è senz'altro contenuto in qualsiasi altro vettore).

•BASE (con esito negativo): se il vettore V1 è ormai vuoto (e il vettore V2 non è anch'esso vuoto) termina con esito negativo 0 (un vettore non vuoto, che contiene almeno un elemento, non può essere contenuto in un vettore vuoto).

•PASSO INDUTTIVO:

•Se l'elemento iniziale di V1 è uguale a quello finale di V2 elimina l'elemento iniziale di V1 e quello finale di V2.

•Altrimenti (ovvero se l'elemento iniziale di V1 è diverso da quello finale di V2) elimina soltanto l'elemento iniziale di V1.

•CHIAMATA RICORSIVA: ricorri sui due vettori V1 e V2 accorciati in uno dei due modi indicati nel passo induttivo.

•CONDIZIONE DI TERMINAZIONE: nel passo induttivo almeno uno dei due vettori V1 o V2 viene sempre accorciato (se non tutti e due) e quindi dopo un numero finito di passi almeno uno dei due vettori deve essere vuoto; l'algoritmo termina sulla base (positiva o negativa) quando almeno uno dei due vettori è vuoto; la terminazione della ricorsione è dunque sempre garantita.

```
int contiene (int V1 [], int V2 [], int iniz, int fine) {
        if (fine < 0) {          /* vettore V2 vuoto */
                return 1;  /* termina con esito positivo */
   }
        if (iniz > N1 – 1) {   /* vettore V1 vuoto */
                return 0;  /* termina con esito negativo */
    }
   /* passo induttivo e chiamata ricorsiva */
   if (V1[iniz] == V2[fine]) {   /* caso 1 */
            return contiene (V1, V2, iniz + 1, fine – 1);
    }
   return contiene (V1, V2, iniz + 1, fine);  /* caso 2          */
}
```

```
//chiamata nel main
// N1 è costante
contiene (V1, V2, 0, N2-1);
```

•Il programma principale chiama la funzione nel modo seguente:

```
#define   N1        ...
#define   N2        ...
int VETT1 [N1], VETT2 [N2], esito;
esito = contiene(VETT1, VETT2, 0, N2 − 1);
```

•In partenza l'argomento iniz indica l'elemento iniziale dell'intero vettore V1 mentre l'argomento fine indica quello finale di V2; i due vettori vengono infatti accorciati dagli estremi opposti.

•Sarebbe possibile evitare di passare come argomenti gli indirizzi dei due vettori usandoli direttamente all'interno della funzione come variabili globali; gli argomenti essenziali sono infatti i due indici iniz e fine (che vengono modificati ad ogni chiamata ricorsiva).

Esercizio (tde 14-11-2008)

Si consideri il seguente programma

```
#include <stdio.h>
…
```

Si dica sinteticamente che funzioni svolgono f1 ed f2.

Le risposte che semplicemente spiegano il codice passo per passo non saranno considerate.

Dire cosa stampano i tre cicli for.

```
int f1(int a,int b){
        if (b==0)
                return 1;
        return a * f1(a, b-1);
}
int f2(int a,int b){
        int i;
        for (i=0;f1(a,i)<=b;i++)
                ;
        return i-1;
}
int main(){
        int i,a[3]={2,3,4},b[3]={8,30,256};
        for (i=0;i<3;i++)
                printf("%i ",f2(a[i],b[i]));
        for (i=0;i<3;i++)
                printf("%i ",f1(a[i],f2(a[i],b[i])));
        for (i=0;i<3;i++)
                printf("%i ",f2(a[i],f1(a[i],b[i])));
}
```

f1: Restituisce a elevato b
f2: Restituisce la parte intera del logaritmo che ha base a e argomento b

Si scriva l'output generato dal primo ciclo for:
3 3 4

Si scriva l'output generato dal secondo ciclo for:
8 27 256

Si scriva l'output generato dal terzo ciclo for:
8 … (stamperebbe 30 256 se non andasse fuori dal range degli interi)

Esercizio (tdeB 29-1-2007)

- La funzione di libreria strlen() potrebbe essere (teoricamente) codificata in modo ricorsivo, come segue…

```
int strlenRic ( const char * s ) {
  if ( *s == '\0' ) return 0;
  else return 1 + strlenRic( s+1 );
}
```

…anche se ciò non accade mai in pratica, per ovvi motivi di efficienza.

- Si chiede tuttavia di proporre una codifica ricorsiva anche per le funzioni indicate in seguito. Si badi a considerare bene i casi base e a progettare bene i "passi induttivi". Si tenga eventualmente presente che il codice ASCII del carattere '\0' (terminatore di stringa) è il numero zero.
-
```
// Ha lo stesso stesso effetto di strcpy() in string.h : s1  riceve tutti i  caratteri
// contenuti in s2 (incluso il '\0')
void strcpyRic( char *s1, char *s2 );
```

-
```
// Come strcmp() in string.h : restituisce 0 se le stringhe sono uguali, n<0 (n>0) se
// s1 precede (segue) alfabeticamente s2
int strcmpRic( char *s1, char *s2 );
```

```
void strcpyRic( char *s1, char *s2 ) {
  *s1 = *s2;
  if ( *s2 != '\0' )
    strcpyRic( s1+1, s2+1 );
  return;
}
```

```
int strcmpRic( char *s1, char *s2 ) {
  if ( *s1 == '\0' && *s2 == '\0' )
    return 0;
  else if ( *s1 < *s2 )
    return -1;
  else if ( *s1 > *s2 )
    return  1;
  else
    return strcmpRic( s1+1, s2+1 );
}
```

Esercizio

- Scrivere un programma C che stampi sullo standard output tutti i valori del triangolo di Tartaglia per un certo ordine N, utilizzando una funzione ricorsiva:

  int cobin(int n, int k);

  1 n = 0

  1 1 n = 1

  1 2 1 n = 2

  1 3 3 1 n = 3

  1 4 6 4 1 n = 4

  1 5 10 10 5 1 n = 5

  1 6 15 20 15 6 1 n = 6

  .....

Leggendo la figura del triangolo di Tartaglia riga per riga, è possibile dedurre come il calcolo di ognuna di esse sia funzione della riga precedente. Il calcolo dei coefficienti binomiali segue dunque le seguenti regole:

con k == 0 e k == n, cobin(n,k) = 1. **(caso base)**

ogni coefficiente è la somma del suo "soprastante" e del predecessore di quest'ultimo. **(passo induttivo)**

```
#include <stdio.h>
#define N 7

int cobin(int n, int k);

int main() {
        int n, k;

        for (n=0; n<=N; n++) {
                for (k=0; k<=n; k++)
                        printf(" %5d", cobin(n, k));
                printf("\n");
        }

        return 0;
}
```

```
int cobin(int n, int k) {
        if  (n<k || n<0 || k<0) {
                printf(”Errore\n");
                return 0;
        }

        if  (k==0 || k==n)
                return 1;
        else
                return cobin(n-1, k-1) + cobin(n-1, k);
}
```

Esercizio

- Scrivere un programma C che stampi a video tutte le possibili N! permutazioni degli elementi di un vettore di N interi.
- Il problema proposto si presta in modo naturale ad una formulazione ricorsiva, infatti:
- Il vettore è lungo “len” e inizialmente dobbiamo costruire le permutazioni di “n = len” elementi:
  - Per generare tutte le possibili permutazioni di n elementi, si può pensare di tenere fisso l’ elemento in prima posizione e stampare l’intera sequenza per ognuna delle permutazioni dei restanti n-1 elementi.
  - Scambiare il primo degli n elementi da permutare con il secondo
  - ripetere considerando n-1 elementi (tranne il primo)
  - scambiare nuovamente il primo degli n elementi con il secondo
  - Scambiare il primo degli n elementi elemento con il terzo
  - ripetere considerando n-1 elementi
  - scambiare nuovamente il primo degli n elementi con il secondo

Vettore V con len elementi, vogliamo costruire le permutazioni dei suoi primi n = len elementi.

caso base

se N=1 allora la stampo il vettore V stesso.

passo induttivo

se N>1

per ogni elemento V[i] tra V[len-n] e V[len-1]

si scambia l'elemento V[len-n] con l'elemento V[len-n+i]

si stampano i vettori contenenti le permutazioni del sotto-vettore tra V[len-n] e V[len-1].

si scambia l'elemento V[len-n] con l'elemento V[len-n+i]

```
#include <stdio.h>

void stampaVett(int *V, int len);
void scambia(int *x, int *y);
void permuta(int *V, int n, int len);

int main( ) {
  int V[] =  {1,2,3,4,5,6};
  int len = 6;

  permuta(V,len,len);  // dopo la chiamata a sottoprogramma l'array V
                       // non risulta essere stato modificato.

  return 0;
} // end main
```

```
void stampaVett(int *V, int len) {
   int i;
   printf("\n");
   for (i=0 ;i<len ;i++)
            printf(" %c ",V[i]);
}

void scambia(int *x, int *y)  {
   int aux;

   aux = *x;
  *x = *y;
   *y = aux;
}
```

```
void permuta(int *V, int n, int len) {
    int i ;

    if (n == 1) {
        stampaVett(V,len);
    } else {
        for (i=0; i<n; i++) {
                scambia(&V[len-n],&V[len-n+i]);
                permuta(V,n-1,len);
                scambia(&V[len-n],&V[len-n+i]);
        }
    }
}
```

Esercizio (tdeB 18-2-2008)

Le funzioni *pila(n)* e *torre(n)* sono definite (per n>0) come segue:

pila(n) = $\mathbf{n}^{(\mathbf{n-1})^{(\mathbf{n-2})^{\cdots}}}$ fino a n=1. Esempi: pila(1)=1, pila(2)=2, pila(3)=3^2=9; pila(4)=4^9=262144 ...

torre(n) = $\mathbf{n}^{\mathbf{n}^{\mathbf{n}^{\cdots}}}$ n compare n volte: torre(1)=1, torre(2)=4, torre(3)=3^{27}=7,6·10^{12}, torre(4)=$4^{1,3\cdot 10^{154}}$!

N.B.: l'associatività è dall'alto al basso: pila(4) = $4^{(3^2)}$ = 4^9, torre(3) = $3^{(3^3)}$ = 3^{27} [e ***non*** $(4^3)^2$ *e* $(3^3)^3$]

Si diano opportune definizioni ricorsive di pila(n) e torre(n), indicando chiaramente e i casi base i passi induttivi. Se si ricorre a funzioni ausiliarie, si indichi chiaramente di quali funzioni si tratta.
Si codifichino in C le due funzioni. Si apprezza e si consiglia l'introduzione di funzioni di supporto

Elevamento a Potenza: usiamo la funzione pot(a, b) che implementa il calcolo di a^b per a,b>=0

Diamone una definizione ricorsiva: caso base: se b = 0 → 1

pot(a,0) = 1

passo induttivo: $a^b = a*a^{b-1}$

pot(a,b) = b * pot(a, b-1)

Pila: caso base: pila(1) = 1

passo induttivo: $pila(n) = n^{pila(n-1)} = pot(n, pila(n-1))$

Introduciamo poi la funzione tower(n, x) con n,x>0 tale che tower(n,1)=n, tower(n,2)= n^n, tower(n,3)= n^{n^n}...

Tower(n,x): eleva n a se stesso x-1 volte (cioè n compare x volte nella rappresentazione esponenziale)

caso base: tower(n,1) = n

passo induttivo: tower(n,x) = pot(n, tower(n,x-1))

Torre: caso base: torre(1) = 1

passo induttivo: $torre(n) = tower(n,n) = n^{tower(n,n-1)} = n^{(n^{tower(n,n-2)})}$...

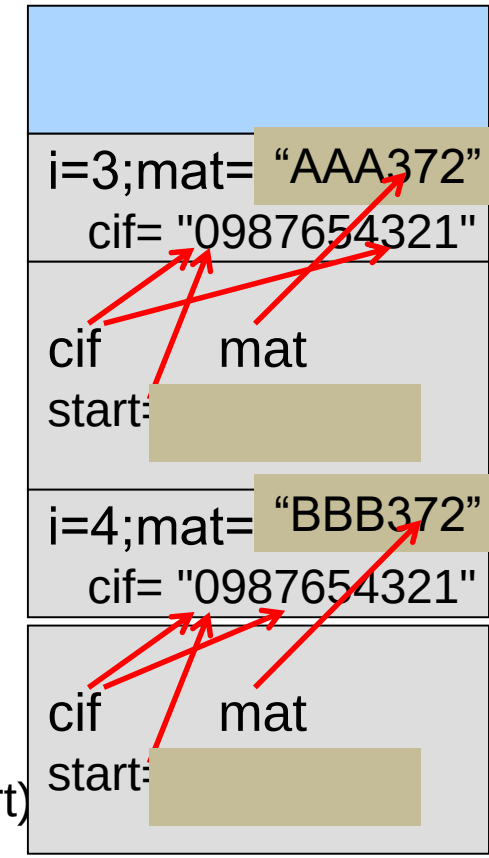
```
int pot( int b, int e ) {
  if( e > 0 )
    return b * pot(b, e-1);
  return 1;
}

int pila( int n ) {
  if( n == 1 )
    return 1;
  return pot(n, pila(n-1));
}

int tower( int n, int x ) {
  if( x == 1 )
    return n;
  return pot( n, tower( n, x-1 ) );
        // attenzione: x-1, NON n-1
}

int torre( int n ) {
  return tower( n, n );
}
```

Esercizio

- Si consideri il seguente programma C, completando la definizione di **MATR** con la propria matricola

```
#define MATR "....."          /* Ad esempio "623372" */
#define CIFRE "0987654321"

void f( int );
void g( char *, char * );

int main() {  f( 3 );
                  return 0;  }

void f( int i ) {
  char mat[7]  = MATR;
  char cif[11] = CIFRE;
  mat[0]=mat[1]=mat[2]='A'+i-3;

  if ( mat[i] != '\0' )
    g( mat+i, cif );

  return;
}
```

```
void g( char * mat, char * cif ) {
  char * start=NULL;
  if ( *mat != '\0' ) {
    start = cif;
    while ( *cif != *mat )
      cif++;
    printf("+ %c%d +",*mat,cif-start);
    f( 7 - strlen(mat) );
  }
  return;
}
```



+ 37 ++ 73 +

Soluzione

- *Nel caso di 623372:*

  + 37 ++ 73 ++ 28 +

Esercizio (tdeB 18-9-2006)

- Si consideri il seguente programma C, completando la definizione di **MATR** con la propria matricola.

  #define MATR "....." ** Ad esempio "623372" **

- Si spieghi brevemente il comportamento del programma; per comprenderlo, si suggerisce di simularne l'esecuzione (prestando attenzione a "impilare" ed eventualmente "disimpilare" bene le chiamate ricorsive e a prestare attenzione a quali istruzioni sono eseguite e quali no).
- Indicare anche l'output stampato a video dal programma (attenzione alla posizione delle chiamate a printf).

```
void f( char * p ) {
  if ( p == NULL )
    return;
  else {
    printf("%c", *p);
    g( p+1 );        /* +1 !! */
    return;
  }
}


int main() {
  char m[7] = MATR;
  f( m+4 );
  printf("SE ARRIVO QUI HO FINITO\n");
  return 0;
}
```

```
void g( char * p ) {
  if ( strlen(p) == 0 )
    return;
  else {
    f( p-1 );        /* -1 !! */
    printf("%c", *p);
    return;
  }
}
```

Il programma inizia invocando f() con argomento pari a un puntatore alla quinta cifra della matricola, stampa tale cifra (il carattere '7' nel caso di Daniele Braga) e invoca g() sul carattere successivo, che però non viene stampato, perché prima viene invocata nuovamente f() sul carattere precedente… e così si instaura un ciclo ***potenzialmente*** infinito in cui g ed f continuano a invocarsi mutuamente in sempre nuove attivazioni, puntando sempre agli stessi due caratteri. La sesta cifra della matricola quindi non sarà mai stampata, come nemmeno la stringa "SE ARRIVO QUI HO FINITO".

Il programma, tuttavia, termina, seppure in modo anomalo e difficilmente prevedibile, quando lo stack (su cui sono impilati i record di attivazione) arriva a saturare la memoria disponibile. Le chiamate mutue saranno tipicamente in numero molto grande, perché i record di attivazione sono di dimensioni ridotte (circa 700.000 record di attivazione sul sistema su cui ho effettuato la prova - NdProf).

L'output è:

*77777777777777777777777777.....777 → segnalazione "stack full" o chiusura improvvisa (**crash**)*

Esercizio (tdeB 16/9/2009)

- Dire cosa stampa il seguente programma immaginando che un utente lanci il programma, digiti la parola **SELFIR** e prema invio:

  ……

```
void f( ) {
  char c;
  scanf("%c",&c);
  if ( c != '\n' )
    f();
  printf("%c", c);
}

int main() {
  char c[12] = "SPECCHIO\n";
  int *cc = c;
  f( );
  printf("SO\n");
  return 0;
}
```

Stampa:

RIFLESSO

Esercizio (tdeB 29-1-2007)

Si consideri il seguente programma C, completando la definizione di **MATR** con la propria matricola. Il programma (che ha una funzione ricorsiva) elabora i vettori mat e p, e alla fine stampa a video una linea.

```
#define MATR "......"        /* Ad esempio "623372"  */


void f( char * x, char ** y )
{
  *y = x++;
  if ( *x != '\0' )
      f( x++, ++y );
  return;
}
```

```
int main() {
  char mat[7] = MATR;
  char * p[7];
  char ** d = p;
  f( mat+1, d );
  for( d=d+4; d>p; d-- )
    (*d)++;
  printf("%s %s %c\n", mat+2, p[2], **p);
  return 0;
}
```

Riportare qui di seguito la riga di output generata dal programma su stdout

3372 72 2

Esercizio (tdeB 21/9/2007)

- Si consideri il seguente programma C.
- ……
- Si indichi la funzionalità realizzata dalla funzione ricorsiva f(). **Non** si chiede di *descrivere* il codice, ma di dire sinteticamente *a che cosa serve* (esempio: *"Visualizza tutti i divisori pari dei parametri"*).

```
void f( int a, int *b ) {
  if ( *b < 2 || *b > 10 )
    return;
  if ( a / *b > 0 )
    f( a / *b, b );
  printf ( "%i", a % *b );
  return;
}

void main() {
  int a[4] = { 5, 200, 100, 4565204 };
  int b[4] = { 2, 2, 5, 10 };
  int i;
  for ( i = 0; i < 4; i++ ) {
    f( *(a+i), b+i );
    printf("\n");
  }
}
```

… …

Esercizio (tdeB 21-7-2006)

- Si consideri il seguente programma C, completando la definizione di **MATR** con la propria matricola.

  #define MATR "....." ** Ad esempio, "623372" **

  ……

- Indicare le quattro righe di output stampate dal programma, simulandone l'esecuzione (si presti attenzione a "impilare" e "disimpilare" bene le chiamate ricorsive e a considerare tutte le chiamate alla funzione printf). Si tenga anche presente che i caratteri '0', '1', …, '9' hanno codici ASCII consecutivi, crescenti in quest'ordine, ed eventualmente che il carattere successivo a '9' è il carattere ':'.
- Per ogni linea si dia una breve giustificazione, spiegando come opera la funzione ricorsiva corrispondente.

```
void f( char * p ) {
  if ( *p != '\0' )
    f(p+1);
  printf(".%s", p);
}


char * f3( char * p ) {
  if ( strlen(p) != 0 )
    if ( strlen(f3(p+1)) == 2 )
      printf("%s", p);
  return p;
}


int main() {
  char m[7] = MATR;
  char *x = m;
  f( m );
  printf("\n%d - %s\n", f2(m), m);
  printf(" - %s\n", f3(m));
  f4( &x );
  return 0;
}
```

```
int f2( char * p ) {
  if ( strcmp(p,"") == 0 )
    return 0;
  else
    return f2(p+1) + ((*p)-'0');
}

/* ATT: d è un doppio puntatore */
void f4( char ** d ) {
  if ( strlen(*d) > 0 ) {
    printf("%c", ((**d)+1) );
    ++(*d);
    f4(d);
  }
}
```

Linea 1

[con “623372”] > **..2.72.372.3372.23372.623372**

La funzione f() “impila” sette chiamate ricorsive avanzando nella scansione del vettore di caratteri (p+1 è un incremento del puntatore pari alla memoria occupata da un elemento del vettore) e stampa quindi in ordine inverso le stringhe che iniziano dai caratteri via via puntati.

Linea 2

[con “623372”] > **23 - 623372**

La funzione f2() scandisce il vettore finché il puntatore non punta a una stringa la cui lunghezza è 0 (cioè ha raggiunto il '\0' in fondo al vettore) e somma le cifre della matricola, restituendo il totale. La somma è ottenuta sfruttando la codifica ASCII.

Linea 3

[con “623372”] > **372 – 623372**

La funzione f3() scandisce il vettore controllando via via la lunghezza della “coda” di stringa restante. Quando la lunghezza della “coda successiva” è 2, stampa la “coda corrente”, composta quindi dai 3 ultimi caratteri della matricola.

Linea 4

[con “623372”] > **734483**

La funzione f4() scandisce il vettore stampando a video via via il carattere successivo (nella tabella ASCII) al carattere correntemente puntato.