Riduciamo le interazioni 😕

Spegnete microfoni e telecamere

Scrivete **eventuali domande sulla chat**: io le ripeto ad alta voce e rispondo (se ne sono capace).

Speriamo vada un po' meglio.

Informatica Generale
Programmazione C

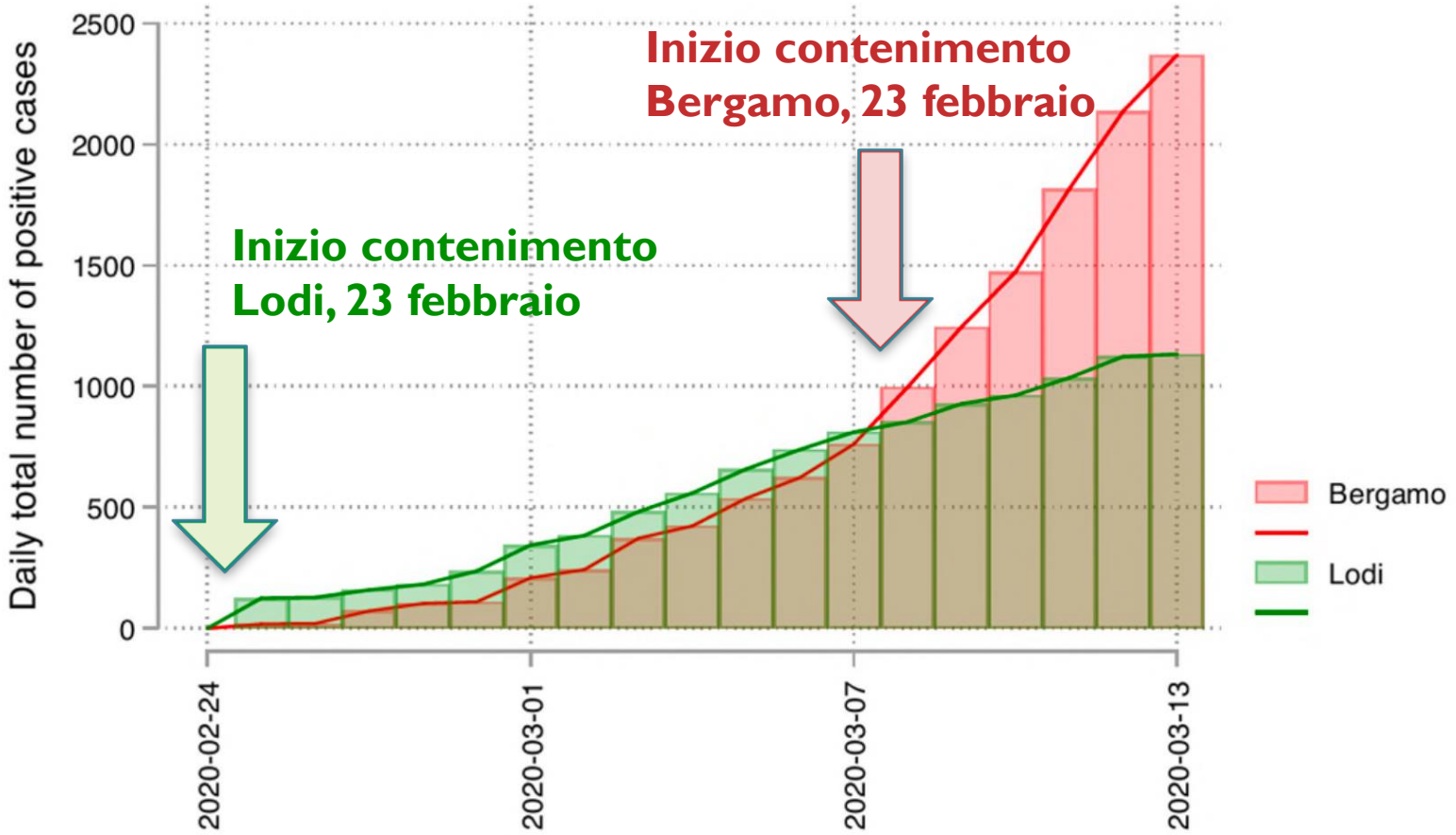
Ivano Salvo

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **4**, 17 marzo 2020

Quarantena sì, Quarantena no?



Lezione 4a:

Ancora sul passaggio di parametri: call-by-value

Passaggio di parametri

Una chiamata di funzione $f(e_1, e_2, \ldots, e_n)$ causa **prima** la valutazione delle espressioni $e_1, e_2, \ldots, e_n$ (**parametri attuali**) e poi l'assegnamento dei valori calcolati alle variabili $x_1, x_2, \ldots, x_n$ (**parametri formali**) definite nel prototipo della funzione $T\ f(T_1\ e_1, T_2\ e_2, \ldots, T_n\ e_n)$ [$T, T, T_2, \ldots, T_n$ sono tipi].

L'**ordine di valutazione** delle espressioni in C **è rilevante**, perché la valutazione di un'espressione può modificare la memoria e quindi altre **valutazioni**.

Ad esempio, potreste essere tentati di scrivere cose come:

```
somma(divRef(5, 3, &q, &r), q)
```

che paiono rischiose, in quanto dipendenti dall'ordine in cui il compilatore valuta i parametri! (`q` viene modificata dalla valutazione di `divRef`).

In C questi fenomeni possono verificarsi anche senza chiamare funzioni. Provate `somma(x++, x)` oppure `somma(x, x++)` oppure `somma(++x, x)`.

Assegnamento parallelo

Nel linguaggio Python è possibile usare il seguente comando:

$$x_1, x_2, \ldots, x_n = e_1, e_2, \ldots, e_n$$

in cui le espressioni vengono valutate nella memoria congelata al momento dell'esecuzione e poi vengono assegnati i valori calcolati alle variabili.

Ad esempio, in Python lo scambio di variabili può essere fatto con l'**assegnazione parallela**:

```
x, y = y, x
```

Sfruttando il `call by value' possiamo ottenere qualcosa di `simile' con la seguente funzione (a meno di side-effects):

```
void assPar(int* x, int* y, int u, int v){
    *x = u;
    *y = v;
}
```

Scambio di variabili revisited

Possiamo fare lo scambio di due variabili **x** e **y**, usando il comando:

```
assPar(&x, &y, y, x)
```

Oppure definire una nuova versione di scambia come segue:

```
void scambiaP(int* x, int* y){
    assPar(x, y, *y, *x);
}
```

Questi esempi possono sembrare un po' artificiosi (e forse lo sono), ma sicuramente se li capite, capirete l'uso degli operatori ***** e **&** in C.

Il problema degli operatori && e ||

La semantica dell'operatore **&&** prevede quanto segue: L'espressione $\mathbf{e_1}$ **&&** $\mathbf{e_2}$ viene valutata prima valutando $\mathbf{e_1}$: se il risultato è FALSE (ad esempio 0 se si tratta di un'espressione intera) il risultato è FALSE. Altrimenti si va a valutare $\mathbf{e_2}$.

A questo punto, potrebbe sembrare che la seguente funzione:

```
int myAnd(int x, int y){
    if (!x) return 0;
    return y;
}
```

Abbia il comportamento dell'operatore **&&**. Tuttavia ciò non è vero. Pensate al comando:

`if (x && resto(y, x))` … **`else`** …

Se **x** vale 0, non si va a valutare **`resto(y, x)`**: per fortuna, perché ovviamente **`resto(y, x)`** **non termina se x vale 0**.

Ma **`myAnd(x,resto(y, x))`** valuta sempre **`resto(y, x)`** a causa della **semantica call-by-value**!

Lezione 4b:

Ricorsione: vizi e virtù

Moltiplicazione Egiziana

Problema 1: *Scrivere una funzione che calcoli la moltiplicazione sfruttando le seguenti uguaglianze:*

$$\begin{aligned} m \times 2n &= (m+m) \times n \quad (n > 1) \\ m \times (2n+1) &= m \times 2n + m \\ m \times 0 &= 0 \end{aligned}$$

Abbiamo visto la volta scorsa una soluzione iterativa, facile ma non completamente banale. Usando tuttavia la possibilità che una funzione chiami sé stessa (**ricorsione**) si può scrivere un programma che semplicemente "traduce" la definizione induttiva:

prima sempre il caso base!!!

```
int multEgyptRec(int m, int n){
    /* PREC: m,n>=0. POST: torna m*n */
    if (n==0) return 0;
    if (restoRec(n,2))  return sommaRec(m,multEgyptRec(m,predRec(n)));
    return multEgyptRec(sommaRec(m,m),div(n,2));
}
```

Somma Ricorsiva

Definizione **induttiva** della somma(si possono definire per induzione **praticamente tutte le funzioni sui naturali**):

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

Può sembrare il gioco delle 3 carte e una banale riscrittura usando la proprietà associativa: ma a destra dell'uguale ho:

- il successore (+1)
- la somma applicata ad argomenti più `piccoli'.

Usando l'if per distinguere i casi, riesco a scrivere un programma **ricorsivo** che traduce questa def **induttiva**:

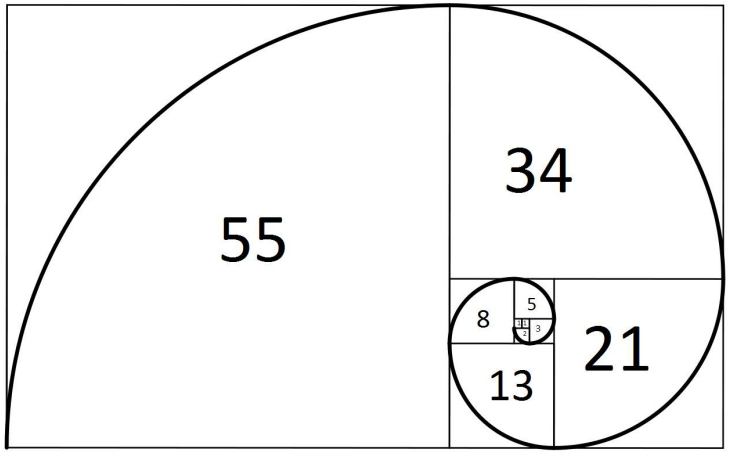
```
int sommaRec(int m, int n){
/* PREC: m,n>=0 */
    if (n==0) return m;
    return sommaRec(m+1, predRec(n));
}
```

Funzione di Fibonacci

La *successione di fibonacci*[1] viene induttivamente definita come segue:

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

Essa definisce la successione di interi, $0, 1, 1, 2, 3, 5, 8, 13, 21, 23, 55, \ldots$. Di questa successione non è altrettanto evidente dare una definizione informale.



[1]La successione prende il nome dal matematico italiano Leonardo Pisano detto FIBONACCI (Pisa ∼1170-∼ 1250) che l'ha introdotta per studiare la riproduzione dei conigli, dando risposta alla seguente domanda: partendo da una coppia di conigli e supponendo che ogni coppia di conigli produca una nuova coppia ogni mese, a partire dal secondo mese di vita, quante coppie di conigli ci sono dopo n mesi? La successione di Fibonacci gode di numerose proprietà interessanti ed ha trovato successivamente moltre altre applicazioni in matematica.

Funzione Ricorsiva per Fibonacci

Si può facilmente scrivere una funzione ricorsiva che calcola i numeri di Fibonacci semplicemente traducendo le clausole della definizione induttiva:

```
int fibRec(int n){
    /* PREC: n>= 0
     * POST: ritorna fib(n)
     */
    if (n<2) return n;
    return sommaRec(fibRec(n-1),fibRec(n-2));
}
```

Sfortunatamente, questa funzione ha un comportamento computazionale anomalo.

Il problema risiede nel fatto che questa funzione **ripete** molte volte le stesse **computazioni**.

Esecuzione di Fibonacci ricorsivo

Nella figura la sequenza di chiamate.

Le chiamate lungo un cammino convivono (sono attive) nello stesso momento. Implementare correttamente la ricorsione è il motivo per cui la chiamata di una funzione genera un record di attivazione sullo stack di sistema.

fibRec(1) viene ad esempio chiamata 3 volte!!!

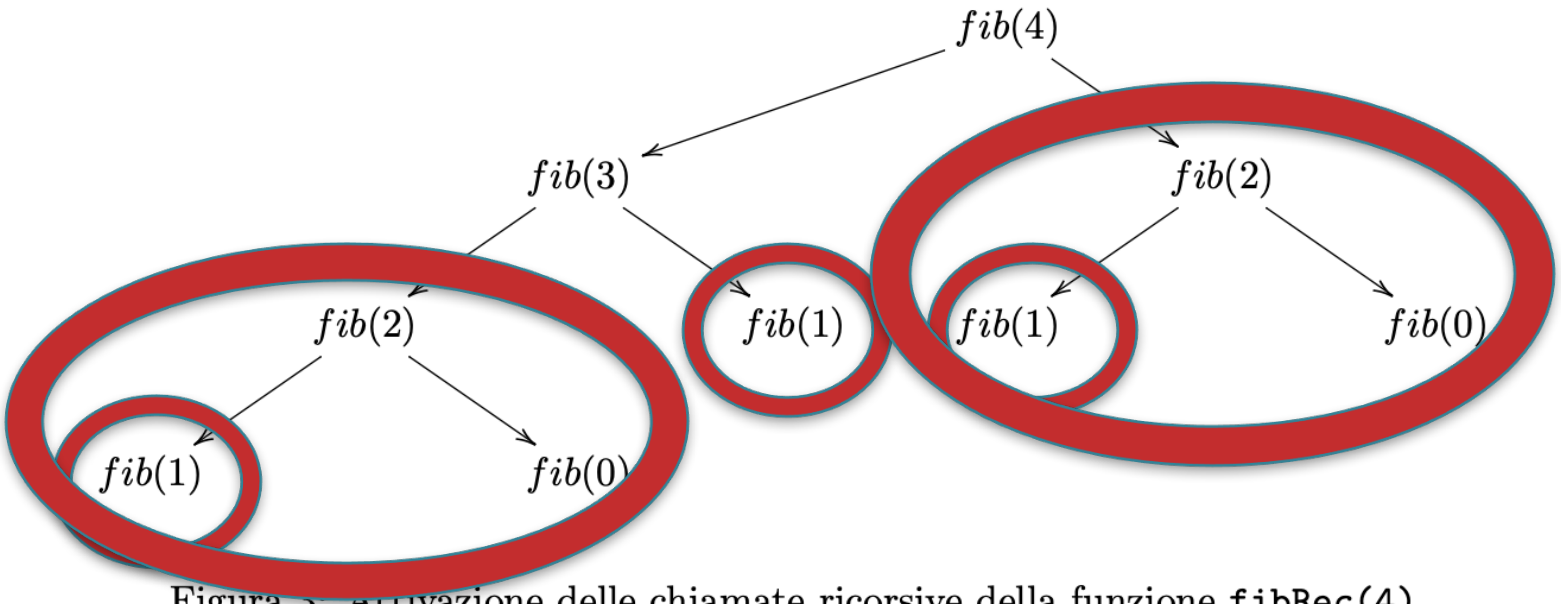


Figura 3: Attivazione delle chiamate ricorsive della funzione `fibRec(4)`

Esecuzione di Fibonacci ricorsivo

Si può dimostrare (per **induzione**!) che *fibRec*(1) viene chiamata esattamente *fib*(*n*) volte che è un numero che cresce esponenzialmente (con base radice aurea, cioè $\varphi = \frac{1+\sqrt{5}}{2}$)

$$fib(n) \approx \varphi^n$$

Informalmente, possiamo osservare che la funzione *fibRec* deve raggiungere un caso base per aggiungere un 1, quindi dovrà eseguire *fibRec*(1) un numero di volte uguale al numero che calcola.

Osservazione: in generale, in TinyC questo è sempre vero. Per calcolare un qualsiasi valore potendo fare solo +1, dovrò eseguire un'operazione di successore (**almeno!**) tante volte quant'è il valore calcolato.

Ma in questo caso, fibRec sarebbe esponenziale anche se somma fosse di complessità costante!

Funzione Iterativa per Fibonacci

Possiamo facilmente scrivere una funzione che calcola i numeri di Fibonacci con un numero di somme lineare in *n*, semplicemente generando un nuovo numero di Fibonacci ad ogni iterazione.

E' sufficiente aver cura di mantenere gli ultimi due numeri di Fibonacci calcolati (e farli scivolare: a ogni iterazione quello nuovo diventa l'ultimo e l'ultimo il penultimo e così via…)

```
int fibIt(int n){
    /* PREC: n>=0 */
        int f2 = 1;
        int f1 = 1;
        int i = 2;
        if (n<2) return n;
        while (i<n){
            /* INV: f1=fib(i) & f2=fib(i-1) */
            asPar(&f1,&f2,f1+f2,f1);
            i++;
        }
        return f1;
    }
```

Fibonacci ricorsivo efficiente

E' possibile scrivere una funzione ricorsiva che calcola i numeri di fibonacci con lo stesso comportamento computazionale della versione iterativa?

La risposta è sì, e spero suggerisca a tutti voi che **qualsiasi ciclo** può essere riprodotto da una funzione ricorsiva, utilizzando **i parametri per memorizzare lo stato della computazione di un ciclo**.

L'idea è quella di avere una funzione ricorsiva con parametri variabili che mimano il comportamento delle variabili **`i`**, **`f1`** ed **`f2`** nella funzione iterativa.

Questo porterebbe a una funzione di 4 parametri: occorre mantenere il prototipo originale: **`fib`** deve essere una funzione di 1 parametro. Scriveremo quindi una funzione ausiliaria.

Fibonacci ricorsivo efficiente

```
int fibRecEffAux(int n, int i, int f1, int f2){
/* PREC: fib_1=fib(i) & fib_2=fib(i-1) & 2<=i<=n
 * POST: ritorna fib(n)
 */
if (i==n) return f1;
return fibRecEffAux(n, i+1, f1+f2, f1);
}

int fibRecEff(int n){
/* PREC: n>=0, POST: ritorna fib(n)
 */
  if (n<2) return n;
  return fibRecEffAux(n,2,1,1);
 }
```

La funzione **`fibRecEff`** risolve i casi facili e prepara la prima chiamata in modo da soddisfare le precondizioni della funzione ausiliaria **`fibRecEffAux`**.

Correttezza di funzioni ricorsive

Occorre dimostrare che, sotto le precondizioni:

1. la funzione tratta correttamente i casi base;
2. sono valide le post-condizioni, assumendo induttivamente che le chiamate ricorsive siano corrette;
3. le chiamate ricorsive soddisfano le precondizioni;
4. la sequenza delle chiamate ricorsive termini (è sufficiente far vedere che i valori delle chiamate ricorsive sono più piccoli).

Nel nostro caso 1. è banale. 2. equivale a dimostrare che si conserva un invariante. 3. equivale a trovare una funzione di terminazione.

Lezione 4c:

Problemi inerentemente ricorsivi

Il problema della Torre di Hanoi

"narra la leggenda che in un tempio Indù alcuni monaci siano costantemente impegnati a spostare su tre colonne di diamante 64 dischi d'oro di diversi diamentri secondo le regole della Torre di Hanoi (a volte chiamata Torre di Brahma): ogni disco può essere spostato da una colonna all'altra senza mai che un disco di diametro maggiore sia posto sopra un disco di diametro inferiore. L'obiettivo è spostare i 64 dischi dalla prima alla terza colonna, facendoli passare eventualmente sulla seconda. I monaci spostano un disco ogni giorno e quando completeranno il lavoro, il mondo finirà."

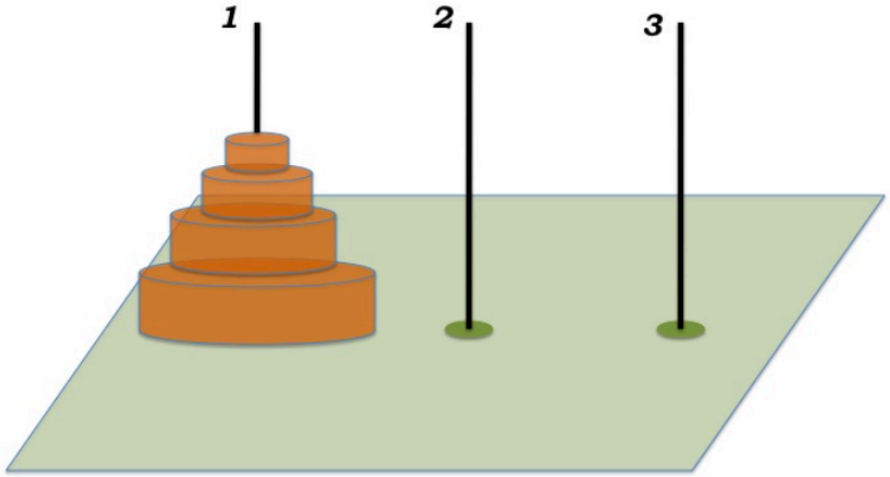


Figura 4: Il problema della Torre di Hanoi con 4 dischi

Soluzione Induttiva

Il problema è banale con un solo disco. Altrettanto facile con 2 dischi. Da 3 dischi in su occorre avere una strategia…

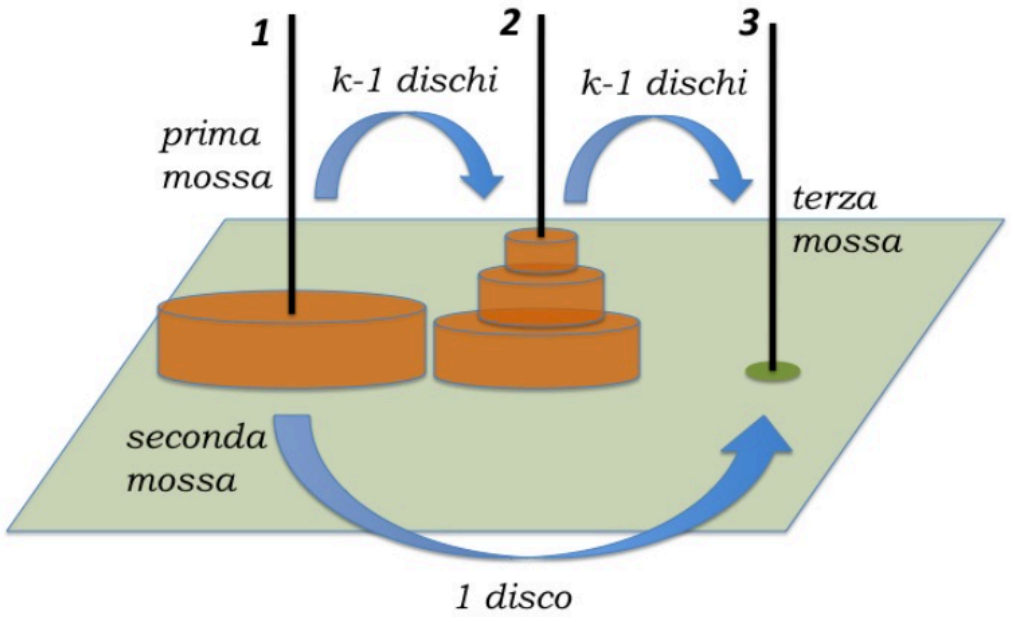


Figura 5: Soluzione del problema della Torre di Hanoi

Soluzione Induttiva

Supponiamo di avere una funzione **`muovi`** che dà in output le mosse… (o comunica con un'appropriata interfaccia grafica)

```
void hanoi(int sorg, int aux, int dest, int n){
   /* sposta n dischi dal piolo sorg al piolo dest,
    * usando aux come appoggio
    */
    if (n==1) muovi(sorg, dest);
       else { hanoi(sorg, dest, aux, n-1);
              muovi(sorg, dest);
              hanoi(aux, sorg, dest, n-1);
              }
}
```

Il problema del Massimo Primo

[vedi dispensa **S4**]

Correzione Primo Esonero 2014, Esercizio 2

Esercizio 2 *Si supponga di avere a disposizione un esecutore la cui unica abilità aritmetica sia la seguente funzione:*

```
int scomponi(int n, int *f1, int *f2)
```

che ha il sequente comportamento: sotto la precondizione che il parametro `n` *sia un numero intero strettamente positivo restituisce come risultato 0 se* `n` *contiene un numero primo, e* `1` *se* `n` *contiene un numero composto. In tal caso, carica nei parametri* `f1` *ed* `f2` *due fattori (maggiori strettamente di 1) il cui prodotto è il valore passato nel parametro* `n`.

Osservate che non possiamo fare **nessuna assunzione sui valori caricati in `f1` ed `f2`**. Ad esempio, se n=24, la risposta è 1 ma potrebbero essere caricati sia 6 e 4, sia 3 e 8, sia 2 e 12 etc.

Ovviamente, non 1 e 24.

Soluzione ricorsiva

Se n è primo allora è anche il massimo primo nella sua scomposizione (**caso base**)

Altrimenti, se $n = f_1 \cdot f_2$ allora il suo massimo fattore primo è il massimo tra il massimo fattore primo nella scomposizione di f_1 e il massimo fattore primo nella scomposizione di f_2 (passo **induttivo**).

Il corrispondente programma ricorsivo è semplicissimo:

```
int maxPrimo(int n){
    int f1, f2;
    if (scomponi(n, &f1, &f2)) return max(maxPrimo(f1), maxPrimo(f2));
       else return n;
}
```

Errore interessante

Errori Tipici L'errore più interessante, è scrivere questa funzione, che, innocentemente, sembra del tutto equivalente a quella scritta sopra:

```
int maxPrimoE(int n){
    int *f1, *f2;
    if (scomponi(n, f1, f2)) return max(maxPrimoE(*f1), maxPrimoE(*f2));
        else return n;
}
```

E invece no! Il problema è che le dichiarazioni `int *f1;` e `int *f2;` dichiarano due puntatori, ma *non allocano memoria*! Viceversa `int f1;` e `int f2;` allocano spazio per due variabili intere. In questo caso quindi, `f1` ed `f2` possono essere puntatori qualsiasi, e un errore di `Segmentation Fault` è il risultato più probabile di questa funzione.

Soluzione iterativa

La soluzione iterativa, viceversa, ha una certa complessità (vedi dispensa **S4** – forse la facciamo usando vettori più in la).

Occorre memorizzare in una struttura dati i fattori (non ancora noti essere primi o meno) in una qualche struttura dati e continuare ad analizzarli finché non si scompongono tutti in fattori primi.

Domanda: ma perché nella versione ricorsiva non ho bisogno di strutture dati?

Perché tali fattori sono memorizzati **nelle variabili `f1` e `f2` nei record di attivazione** delle molte chiamate di **`maxPrimo`** contemporaneamente attive durante l'esecuzione.

Lezione 4

That's all Folks…

…Domande?