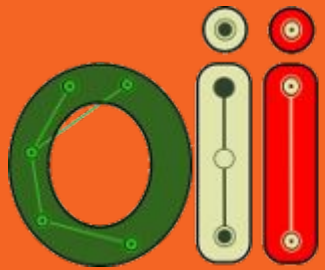
# Palestra di Algoritmi



Olimpiadi Italiane
di Informatica

*Liceo Galilei - Trento*

#3 - 09/12/2021

# 0. Calendario

**Prossime lezioni:** ONLINE 15-17

- ➔ #4 giovedì 16 dicembre
- ➔ #5 giovedì 23 dicembre
- ➔ #6 **giovedì 30 dicembre ?? (vacanze)**
- ➔ #7 giovedì 13 gennaio 2022
- ➔ #8 giovedì 20 gennaio 2022
- ➔ #9 giovedì 27 gennaio 2022
- ➔ #10 giovedì 3 febbraio 2022

# Siete pronti? Partiamo!

# Funzioni

- Sotto-programmi autonomi
- permettono di evitare di duplicare codice
- semplificano lettura del codice, diventa più comprensibile

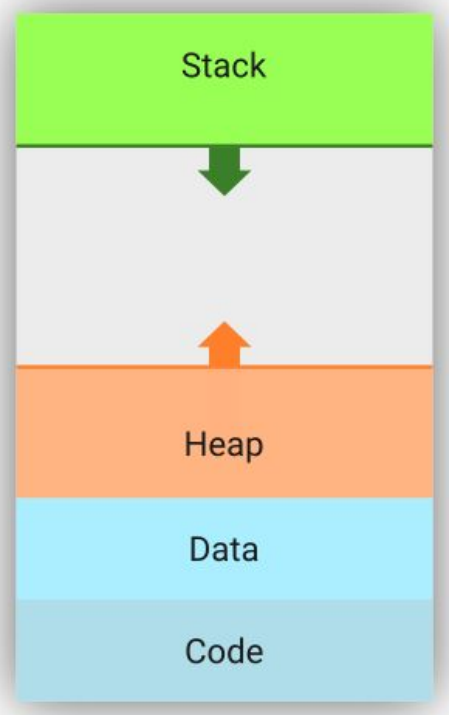
**4 componenti**

- **nome**
- **parametri**
- **valore di ritorno**
- **corpo**

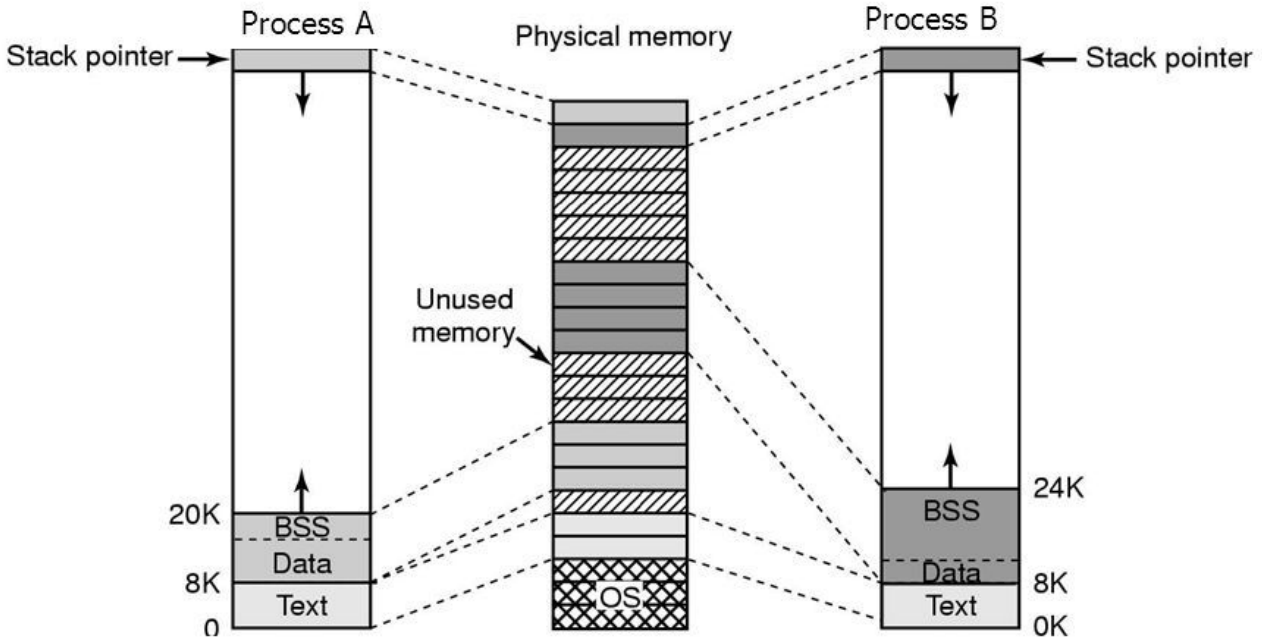
```
tipo_di_ritorno nome_della_funzione(tipo_parametro1 nome_parametro1, tipo2 nome2, ...) {
    tipo_di_ritorno res;
    // Operazioni varie che usano i parametri e assegnano un valore a res
    return res;
}
```

# Memoria

## Vista dal programma



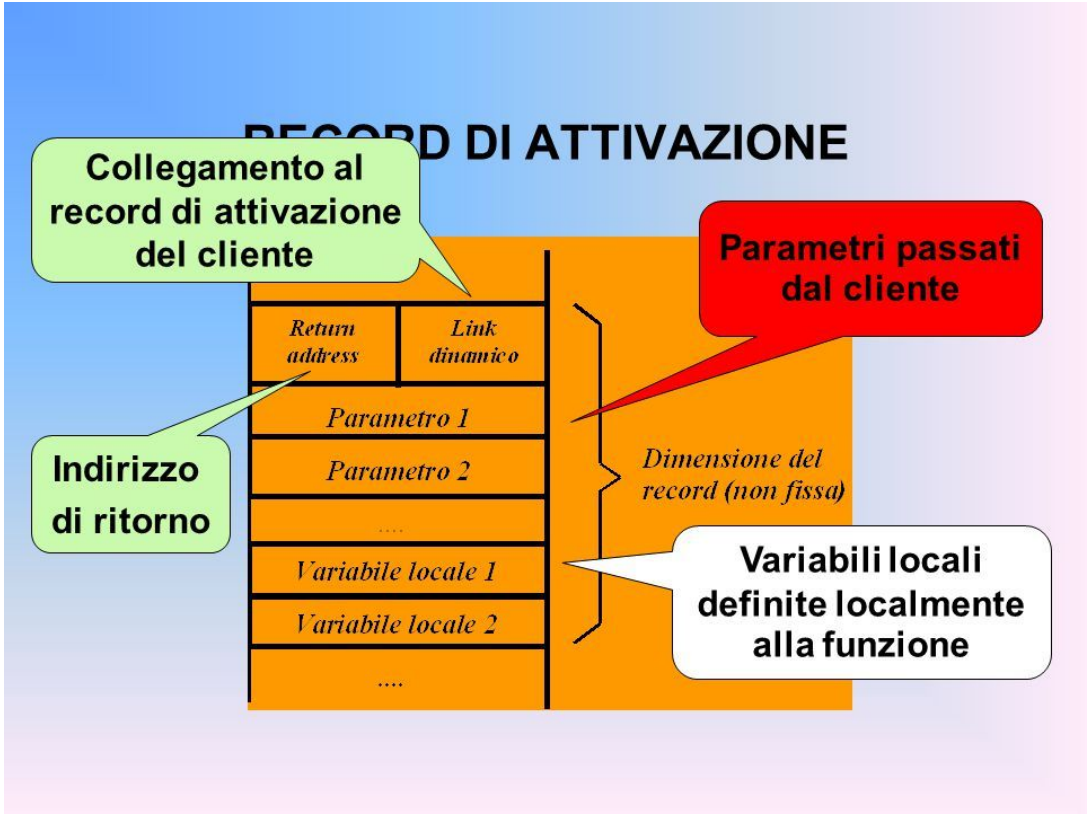

## Vista dal Sistema Operativo



# Memoria

Ad ogni chiamata di funzione, viene allocato un “record di attivazione” nello **stack** (del processo), dove vengono salvati:

- parametri
- variabili della funzione
- registri della CPU

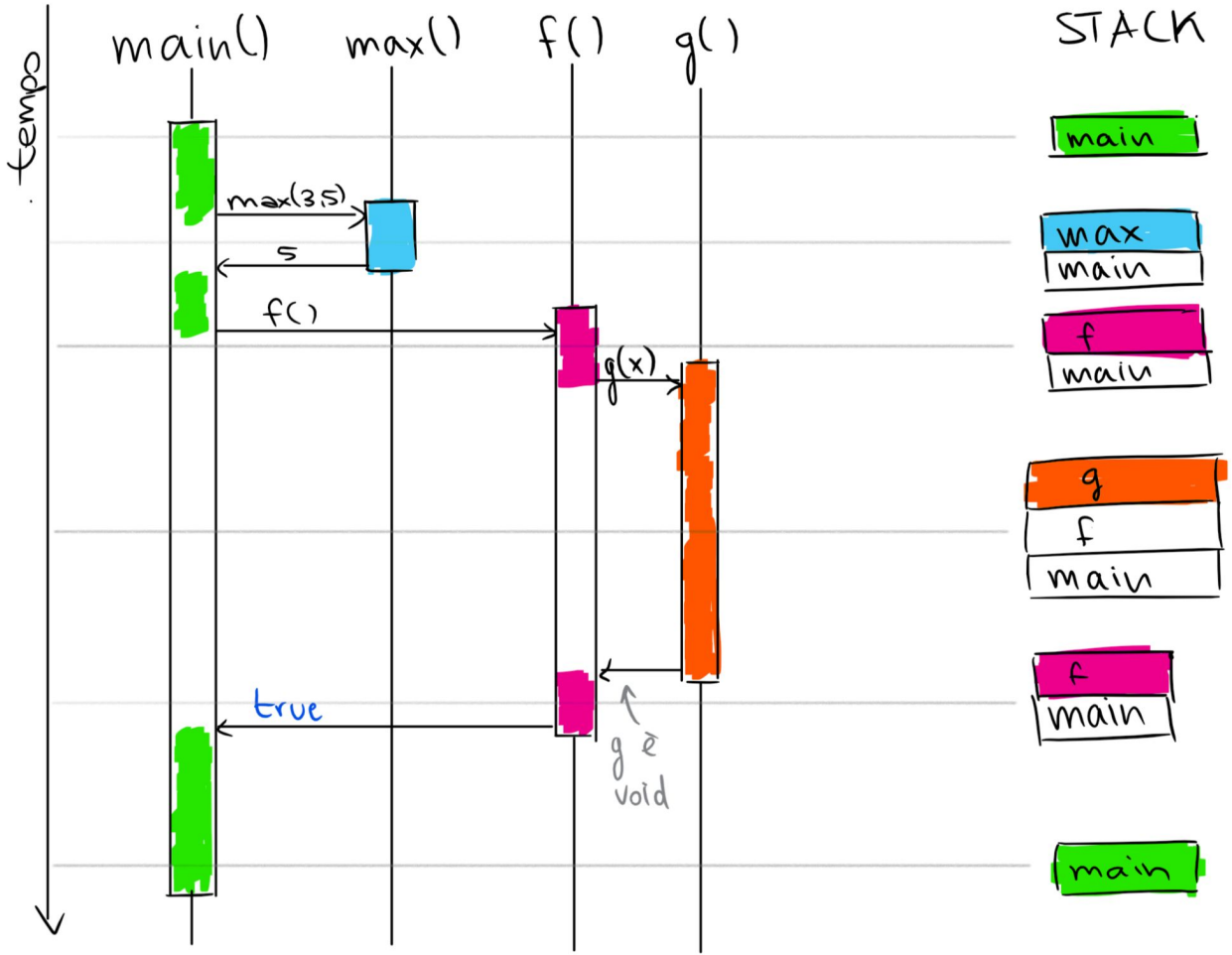



## Scoping delle variabili

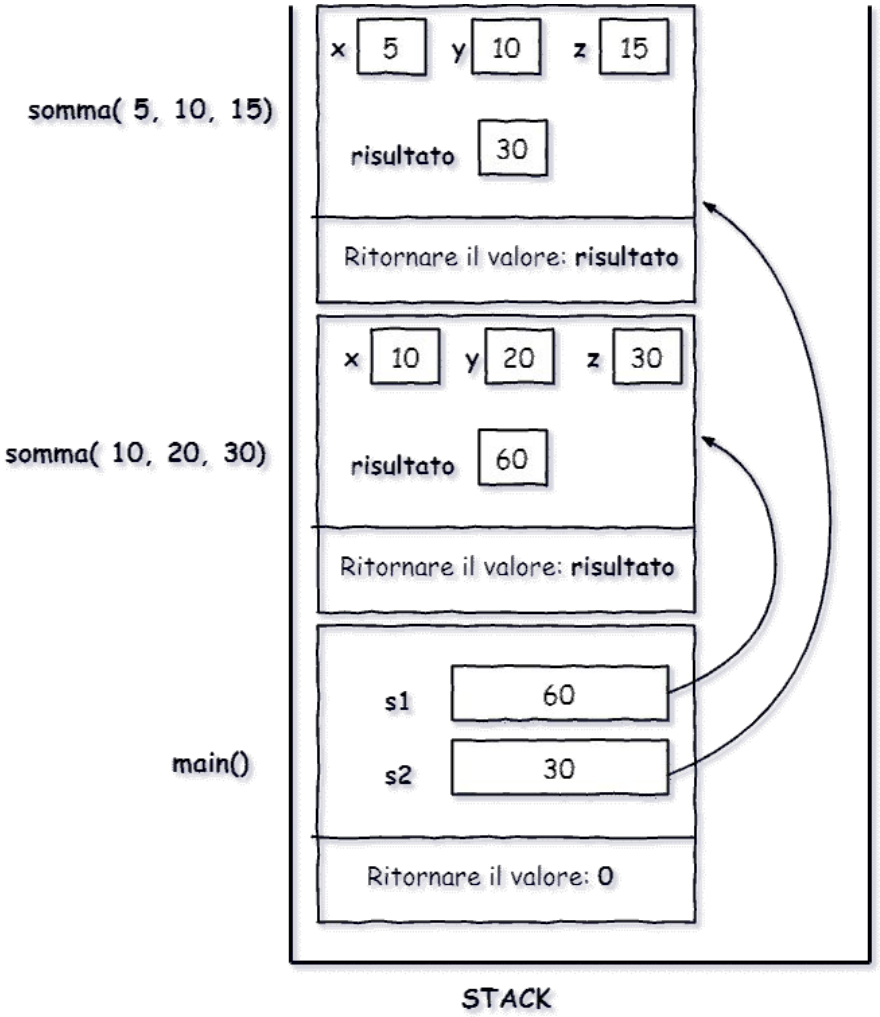
Le variabili dichiarate dentro le funzioni “nascondono” quelle dichiarate fuori.

La risoluzione dei nomi comincia dal corpo della funzione per uscire verso i blocchi esterni oltre le { }

# Memoria


tempo
main()
max()
f()
g()
max(3,5)
5
f()
g(x)
true
g è void
STACK
main
max
main
f
main
g
f
main
f
main
main

# Memoria


somma( 5, 10, 15)
x 5
y 10
z 15
risultato 30
Ritornare il valore: risultato
somma( 10, 20, 30)
x 10
y 20
z 30
risultato 60
Ritornare il valore: risultato
main()
s1 60
s2 30
Ritornare il valore: 0
STACK

# Passaggio dei parametri

- per **valore**/copia: viene copiato il valore della variabile, le modifiche dentro la funzione non hanno effetto al di fuori di essa. Evitiamo di passare per copia parametri "pesanti" (es. vettori di 10000 elementi) per non rallentare l'esecuzione

```
int prova(int a, int b){
    a = 25;
    b = 30;
    return a+b;
}
```

```
int a=5, b=9;
prova(a,b); //ritorna 55
cout << a << " " << b; //stampa "5 9"
```

# Passaggio dei parametri

- per **riferimento**: viene passato l'indirizzo di memoria della variabile: le modifiche dentro la funzione si ripercuotono anche fuori da essa

```
int prova(int &a, int &b){
    a = 25;
    b = 30;
    return a+b;
}
```

```
int a=5, b=9;
prova(a,b); //ritorna 55
cout << a << " " << b; //stampa "25 30"
```

# Passaggio dei parametri

- per **puntatore**: viene copiato l'indirizzo di memoria della variabile

```
int prova(int *a, int *b){
    a = 25;
    b = 30;
    return a+b;
}
```

```
int a=5, b=9;
prova(&a,&b); //ritorna 55
cout << a << " " << b; //stampa "25 30"
```

```
int *a = new int, *b = new int;
*a = 5;
*b = 9;
prova(a,b); //ritorna 55
cout << *a << " " << *b; //stampa "25 30"
```

Non abbiamo bisogno di usare puntatori nei nostri esercizi

# Funzioni ricorsive

- una funzione che richiama se stessa si dice **ricorsiva**

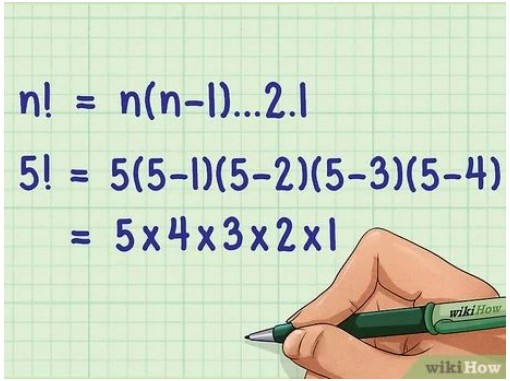
- **DIVIDE ET IMPERA**: spesso riusciamo facilmente a prendere un problema grande e ridurlo in uno o più problemi più piccoli

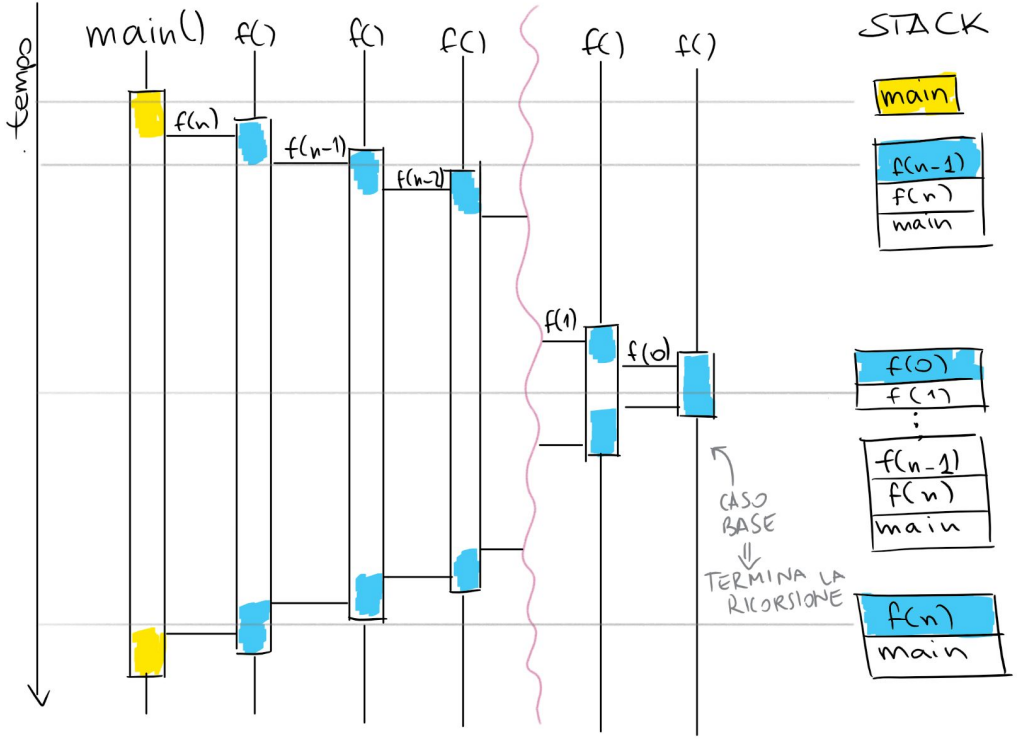
**2 componenti**

- *condizioni di terminazione*: in alcuni casi (*casi base*), la funzione deve restituire un valore
- *chiamate ricorsive*: nei casi non-base, la funzione richiama se stessa

# Esempio: fattoriale

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



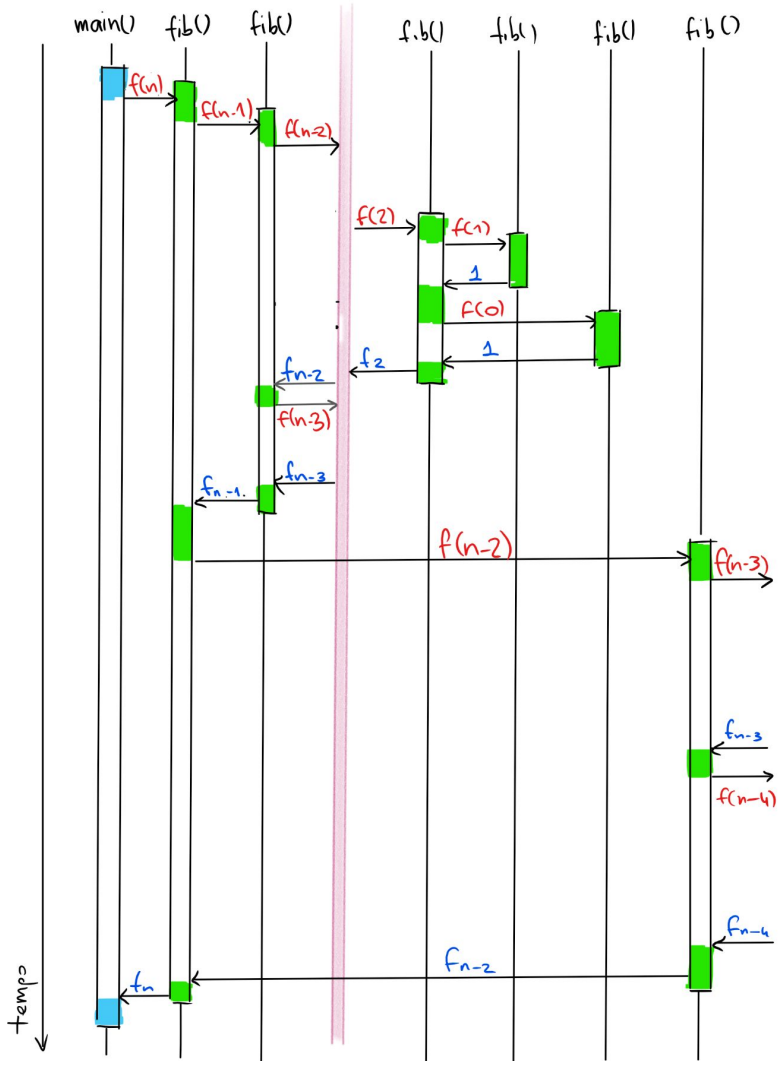
# Esempio: fibonacci

```
int fibonacci(int n) {
  if (n==0) { //caso base
    return 0;
  } else if (n == 1) { //altro caso base
    return 1;
  } else { //chiamate ricorsive
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Ogni numero è dato dalla somma dei due precedenti

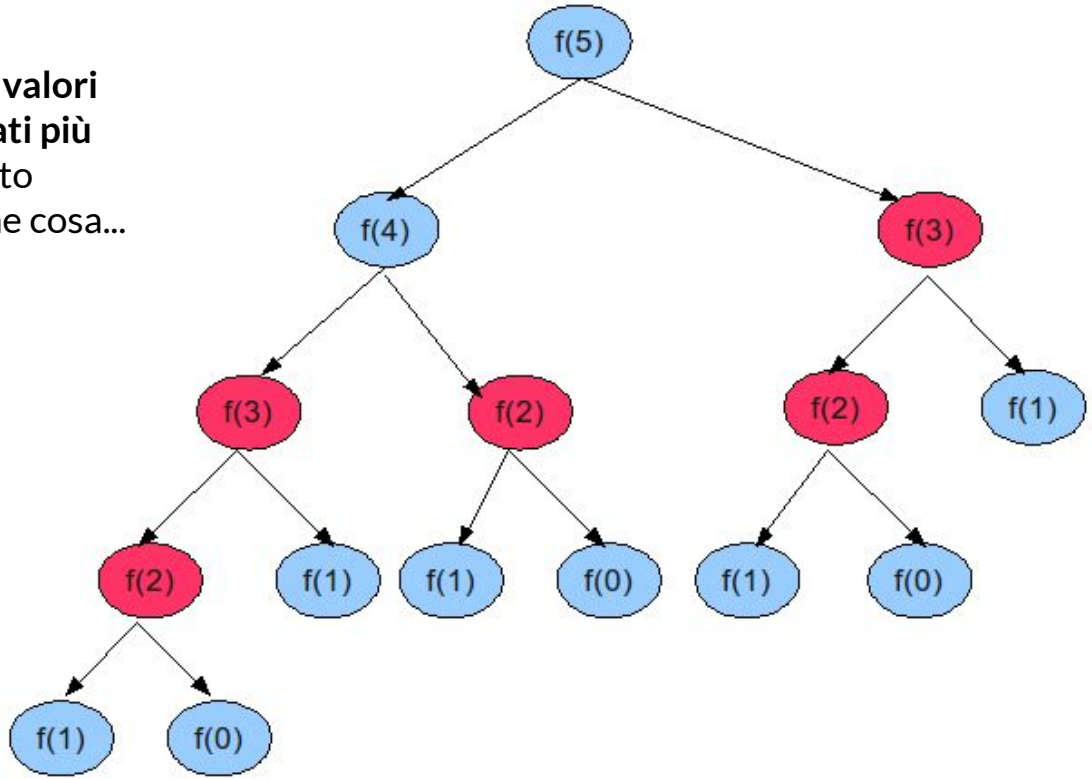
**(0), 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89**

Nell'esempio a destra, chiamiamo **fibonacci(5)**



# Esempio: fibonacci

Nota che **alcuni valori vengono calcolati più volte**, non è molto intelligente come cosa...



# Backtrack - il problema della scelta

Il ***Backtracking*** è una tecnica di programmazione che prevede di provare tutte le possibili soluzioni e di selezionare quelle ammissibili (problemi di decisione: esiste una soluzione?) oppure quelle migliori (problemi di ottimizzazione).

## QUANDO LO USO?

Quando devo esplorare tutto lo spazio delle soluzioni di un problema (es. stampare *tutte* le permutazioni di un array) oppure quando il problema da dei vincoli sull'input sufficientemente piccoli da permettere di provare tutte le possibilità e selezionare quella migliore.

Faccio una scelta. Proseguo. Se non va bene, torno indietro e faccio un'altra scelta. Questa è l'idea dietro il backtracking: ritenta, sarai più fortunato

# Easy 3

## Trova la massima somma pari di coppie di numeri 2.0

https://training.olinfo.it/#/task/easy3/statement

rileggi il testo dell'esercizio prima di continuare!

**PROBLEMA: Non possiamo controllare tutte le possibili combinazioni**!!

Richiede troppo tempo

# Primo approccio

Devo far girare il mio algoritmo più velocemente…

- ➔ le operazioni che richiedono più tempo sono le somme
- ➔ vedere se un numero è pari o dispari è equivalente a dividere per 2 → fare uno shift di 1 bit a destra

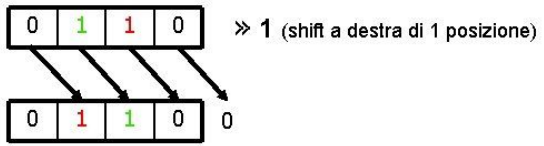
So che la somma di due numeri è pari se:

- ➔ i due numeri sono ENTRAMBI PARI
  - ◆ es. 4+6, 2+8, 14+18, 22+46, …
- ➔ i due numeri sono ENTRAMBI DISPARI
  - ◆ es. 3+7, 15+17, 19+21, …
- ➔ altrimenti, la somma è dispari

**Shift a destra**

- Si inserisce come MSB un bit a zero
- Equivale ad una divisione per due

0110 » 1 = 0011 ( 6 : 2 = 3 )
0110 » 2 = 0001 ( 6 : 4 = 1 ) troncamento!



# Primo approccio

```
for(int i=0; i<n; i++){
    for(int j=i+1; j<n; j++){
        if(sequenza[i] %2 == 0 && sequenza[j] %2 == 0 ||
                sequenza[i] %2 != 0 && sequenza[j] %2 != 0){
            if(!trovato){
                trovato = true;
                max = sequenza[i] + sequenza[j];
            }else if(sequenza[i]+sequenza[j] > max)
                max = sequenza[i]+sequenza[j];
        }
    }
}
```

Arriviamo a 70/100, un po' meglio… ma non basta!

# Secondo approccio - greedy

Devo far girare il mio algoritmo più velocemente…

- la somma pari massima è ottenuta a partire dai due numeri più grandi, che devono essere entrambi pari o entrambi dispari
- **ordino il vettore**, così trovo più facilmente i numeri più grandi (non posso assumere che il vettore sia già ordinato: se provate, vi accorgete che non è sempre ordinato)
- **parto dalla fine del vettore**, prendo due numeri fino a quando non sono entrambi pari o entrambi dispari, altrimenti scalo uno o scalo l'altro, retrocedendo verso l'inizio del vettore
- i primi due numeri entrambi pari o entrambi dispari sono quelli che danno la somma pari massima
- **trovati i numeri**, **calcolo la somma** e la stampo sul file di output

# Secondo approccio - greedy

| 7 | 6 | 5 | 10 | 8 | 4 | 3 | 9 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|

ordino il vettore

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

parto dalla fine

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

uno pari e uno dispari, non va bene

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

uno pari e uno dispari, non va bene

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

ora va bene

# Secondo approccio - greedy

Per ordinare il vettore, possiamo usare questo codice:

```
int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}
```

```
int sequenza[n];
```

```
//ordino la sequenza
qsort(sequenza, n, sizeof(int), cmpfunc);

out << trova_max(n-2, n-1, sequenza);
out.close();
```

# Secondo approccio - greedy

```
int trova_max(int i, int j, int sequenza[]){
    if(i < 0 || j < 0)
        return -1;
    if(sequenza[i] %2 == 0 && sequenza[j] %2 == 0 ||
        sequenza[i] %2 != 0 && sequenza[j] %2 != 0)
        return sequenza[i]+sequenza[j];
    else
        return max(trova_max(i-1, j, sequenza), trova_max(i-1, i, sequenza));
}
```

Come potete notare, è una **funzione ricorsiva**, ed è anche piuttosto semplice!

# Piastrelle

## Piastrellature

https://training.olinfo.it/#/task/piastrelle/statement

Esercizio che prevede l'uso di ricorsione + backtrack


SEE YOU
NEXT TIME!