# Laboratorio di Programmazione Gr. 3 (N-Z)

## Corso di Laurea in Informatica

### Università degli Studi di Napoli Federico II

### A.A. 2022/23

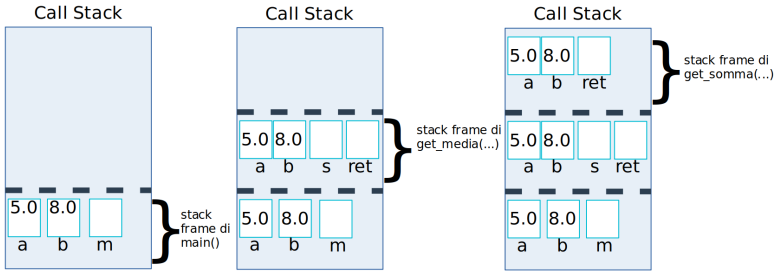
A. Apicella

# Funzioni e soluzioni ricorsive

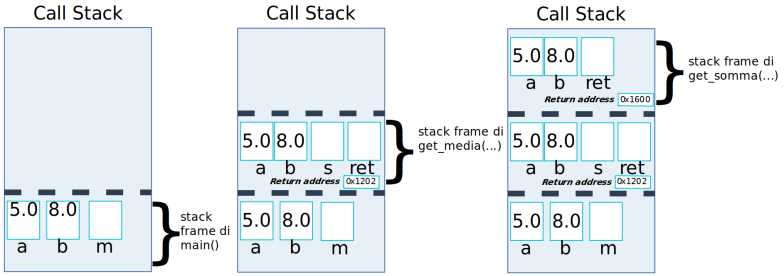
### Qualche dettaglio in più sulle invocazioni di funzioni

In [ ]:
```
# include <stdio.h>

double get_somma(double a, double b)
{
    double ret = a + b;
    return ret;
}

double get_media(double a, double b)
{
    double s;
    double ret;
    s          = get_somma(a, b);
    ret        = s/2.0;
    return ret;
}

int main()
{
    double a = 5;
    double b = 8;
    double m;
    m        = get_media(a,b);
    printf("la media tra %.2lf e %.2lf è %.2lf\n",a,b,m);

    return 0;
}
```

All'esecuzione di questo programma, il call stack al suo apice sarà rappresentato nel modo seguente:




In realtà questa rappresentazione è semplificata, in quanto ogni invocazione di funzione, oltre ai parametri attuali, memorizza nello stack anche il *return address* della funzione (assieme ad altre informazioni non direttamente accessibili).




Attraverso il return address, il programma può tornare al punto esatto in cui si era interrotto e proseguire l'esecuzione dell'invocante. In altri termini, data una invocazione di funzione, il suo return address è l'indirizzo, nella funzione invocante, dell'istruzione immediatamente successiva all' invocazione.

Nel esempio di sopra, `0x1202` sarà l'indirizzo dell'istruzione immediatamente successiva all'invocazione `get_media(a,b);` nella funzione `main(...)`, mentre `0x1600` sarà l'indirizzo dell'istruzione immediatamente successiva all'invocazione `get_somma(a, b);` nella funzione `get_media(a,b);`

Ovviamente, termine di ogni funzione l'intero stack frame verrà eliminato, compreso l'indirizzo di ritorno.

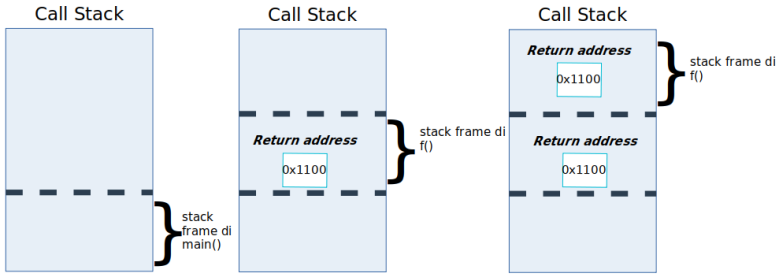
$\Rightarrow$ ogni invocazione di funzione, anche se non porta parametri, costa in termini di spazio sul call stack.

## *Funzione ricorsiva*

> ***Funzione ricorsiva*** (programmazione): strumento messo a disposizione da tanti (ma non tutti) i linguaggi di programmazione (e.g., FORTRAN 77 non supporta la ricorsione) che permette ad una funzione di "invocare se stessa"

Esempio:

In [ ]:
```
#include <stdio.h>

void f()
{
    printf("ciao\n");
    f();
}

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



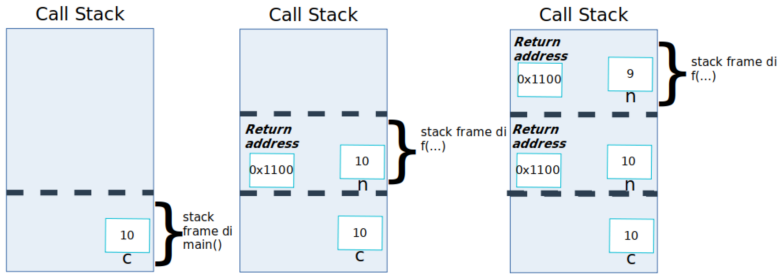

Nel caso precedente, la funzione `f()` invoca se stessa (teoricamente) all'infinito. Ad ogni invocazione, la funzione invocante:

- salva nello stack di esecuzione il punto in cui si trova (così come qualsiasi altra funzione)
- invoca sé stessa (quindi ricomincia dall'inizio)
- una volta terminata l'esecuzione, ritorna al punto in cui l'invocante si era interrotto

⇒ necessario un criterio di stop!

In [1]:
```
#include <stdio.h>

void f(int n)
{
    if(n > 0)
    {
        printf("ciao\n");
        f(n-1);
    }
}

int main()
{
    int c = 2;
    f(c);
    return 0;
}
```

```
ciao
ciao
```




In [8]:
```
#include <stdio.h>

void f(int n)
{
    while(n > 0)
    {
        printf("ciao\n");
        n--;
    }
}

int main()
{
    f(10);
    return 0;
}
```

```
ciao
ciao
ciao
ciao
ciao
ciao
ciao
ciao
ciao
ciao
```

A livello logico, il codice ricorsivo ed il codice iterativo sopra riportati sono equivalenti. La differenza sostanziale è che **il codice ricorsivo impegnerà più risorse**, in quanto, ad ogni invocazione, occuperà spazio nel call stack attraverso un nuovo stack frame.

**Piccola parentesi:**

- Da notare che anche il main è una funzione...quindi niente vieta di...

In [ ]:
```
#include <stdio.h>
int main()
{
    printf("ciao\n");
    main();
}
```

Teoricamente, ogni codice iterativo può essere convertito in codice ricorsivo e viceversa.

Esempio:

**Problema:** Data la seguente funzione per la ricerca de massimo in un vettore

```
int max_array(int v[], int n)
{
    int max = v[0];
    for(int i = 1; i < n; i++)
        if(v[i] > max)
            max = v[i];
    return max;
}
```

Definirne una versione ricorsiva

In [17]:
```
// ricerca del massimo in un vettore (funzione ricorsiva)
#include <stdio.h>
#define N 10

void max_array(int v[], int n, int i, int* max)
{
    if(i==n)
        return;
    if(i==0)
        *max = v[0];

    if(v[i] > *max)
        *max = v[i];
    max_array(v, n, i+1, max);

}


int main()
{
    int v[] = {1,5,7,2,9,4,2};
    int n   = 7;
    int max;
    max_array(v,n,0, &max);
    printf("il massimo è %d\n", max);
    return 0;
}
```

```
il massimo è 9
```

In [16]:
```
// ricerca del massimo in un vettore (funzione ricorsiva, alternativa)
#include <stdio.h>
#define N 10

int max_array(int v[], int n, int i, int max)
{
    if(i==n)
        return max;
    if(i==0)
        max = v[0];

    if( v[i] > max)
        max = v[i];
    return max_array(v, n, i+1, max);

}


int main()
{
    int v[] = {1,5,7,2,9,4,2};
    int n   = 7;
    int max;
    max = max_array(v,n,0, 0);
    printf("il massimo è %d\n", max);
    return 0;
}
```

```
il massimo è 9
```

I due programmi di cui sopra sono logicamente equivalenti alla versione iterativa in quanto producono le stesse soluzioni.

Da notare che, ad ogni invocazione, la funzione invocante deve inviare tutte le variabili di cui necessita l'invocata, ossia:

- dati che specificano su quale/i elemento/i lavorare (in questo caso, variabile `i`)
- dati necessari per effettuare i calcoli (in questo caso, `max`)
- dati per determinare se devono essere effettuate altre autoinvocazioni oppure no (in questo caso, `i` ed `n`)

**Problema:** Scrivere una funzione ricorsiva per il calcolo del fattoriale di un numero.

Si ricorda che il fattoriale di un numero $n$ è

- 1 se $n = 0$,
- altrimenti è dato da $n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$

Una possibile soluzione iterativa al calcolo del fattoriale in un numero $n$ è la seguente:

```
int fattoriale(int n)
{
    int i    = 1;
    int fatt = 1;
    while(i <= n)
    {
        fatt = fatt * i;
        i++;
    }
    return fatt;
}
```

una possibile conversione della funzione precedente in funzione ricorsiva è la seguente:

In [1]:
```
// Calcolo del fattoriale di un numero (funzione ricorsiva)

#include <stdio.h>
#define N 10
```

```
int fattoriale(int n, int i, int fatt)
{
    if(i > n)
        return fatt;

    return fattoriale(n, i+1, fatt*i);
}

int main()
{

    int n   = 5;
    int fatt;
    fatt= fattoriale(n,1,1);
    printf("il fattoriale è %d\n", fatt);
    return 0;
}
```

```
il fattoriale è 120
```

- ogni funzione ricorsiva deve disporre di almeno due sezioni di codice:
    - sezione dedicata al caso base: necessaria per la terminazione della funzione
    - sezione dedicata al caso ricorsivo: dove materialmente viene re-invocata la funzione
- In generale, ogni frammento di codice iterativo può essere convertito in una funzione ricorsiva, e viceversa.
- La trasformazione da iterativo a ricorsivo non sempre è intuitiva, e spesso necessita di strutture dati di appoggio. Ad esempio in certi casi è necessaria un'area di memoria con politica di accesso Pila che *simuli* l'execution stack)

# Funzione Ricorsiva Vs. Soluzione Ricorsiva

> *__Funzione__* ricorsiva (programmazione): strumento messo a disposizione da tanti (ma non tutti) i linguaggi di programmazione (e.g., FORTRAN 77 non supporta la ricorsione) che permette ad una funzione di "invocare se stessa"

E' importante distinguere tra una funzione ricorsiva ed una soluzione ricorsiva:

> *__Soluzione__* ricorsiva: soluzione ad un problema che sfrutta una possibile suddivisione di quest'ultimo in sottoproblemi più semplici da risolvere (generalmente di dimensione ridotta)

NB: **L'utilizzo di una funzione ricorsiva non implica che si stia fornendo una soluzione ricorsiva**

COROLLARIO: Se vi si chiede una soluzione ricorsiva ad un problema e voi presentate un funzione ricorsiva che non implementa una soluzione ricorsiva, non state presentando quanto richiesto.

**Esempio:** voglio una soluzione ricorsiva al calcolo del fattoriale di un numero $n$.

Per costruire una soluzione ricorsiva ad un problema una strategia può essere quella di porsi le seguenti domande:

1. esistono dei casi in cui il mio problema è facilmente risolvibile (ossia che posso considerare banale)?
2. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?
3. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?
4. posso generalizzare?

Nel caso specifico,

1. esistono dei numeri per cui il calcolo del fattoriale è particolarmente semplice da calcolare?

La risposta è sì: sappiamo infatti che per $n = 1$ (ed anche per $n = 0$) il calcolo del fattoriale è banale, ossia che $0! = 1$ ed $1! = 1$.

In realtà, sapendo che $0! = 1$, posso anche scrivere $1! = 1 \cdot 0!$.

2. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un problema un po' più complesso?

vediamo: se $n = 2$ il fattoriale sarà $2! = 2 \cdot 1 = 2 \Rightarrow 2! = 2 \cdot 1 = 2 \cdot (2 - 1)!$

Quindi ho utilizzato la soluzione $1!$ per risolvere il problema $2!$

3. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?

vediamo: se $n = 3$ il fattoriale sarà $3! = 3 \cdot 2 \cdot 1 = 3 \cdot 2! = 3 \cdot (3-1)!$

Quindi ho utilizzato la soluzione $2!$ per risolvere il problema $3!$

se $n = 4$ il fattoriale sarà $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 4 \cdot 3! = 4 \cdot (4-1)!$.

4. posso generalizzare?

In generale,

$$n! = \prod_{i}^{n} i = n \cdot \prod_{i}^{n-1} i = n \cdot (n-1)!$$

quindi

$$\forall n \geq 1,\ n! = n \cdot (n-1)!$$

Posso quindi riscrivere il fattoriale di un numero come:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

Tale soluzione è implementabile in maniera (quasi) immediata attraverso una funzione ricorsiva:

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

In [1]:
```
// Calcolo del fattoriale di un numero (soluzione ricorsiva)
#include <stdio.h>
#define N 10

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

int main()
{

    int n = 5;
    int fatt;
    fatt  = fattoriale(n);
    printf("il fattoriale di %d è %d\n", n, fatt);
    return 0;
}
```

```
il fattoriale di 5 è 120
```

Da notare che la funzione, ad ogni invocazione, *necessita soltanto dei dati su cui lavorare, che saranno di volta in volta più semplici* (in questo caso, l'unico dato su cui la funzione dovrà lavorare sarà la variabile `n` che diventa sempre più piccola).

**Problema:** Dato un insieme di numeri $S$, determinarne il massimo. Proporre una *soluzione* ricorsiva

*soluzione* iterativa: scorro attraverso un ciclo gli elementi del mio insieme,confrontando ogni elemento con il massimo locale trovato fino a quel momento.

*implementazione* iterativa:

```
int A[] = {7,6,8,3}; //NB: in generale, si ricorda che un vettore non è un insieme
int n   = 4;
int m   = A[0];
for(int i=1; i < n; i=i+1)
{
    if(A[i] > m)
        m = A[i];
}
```

1. esistono dei casi in cui il mio problema è facilmente risolvibile?

Il problema è cercare il massimo in un insieme di numeri, ossia si vuole trovare una funzione $max\_set(S)$ che restituisca il più grande da un insieme di numeri $S$ dato in input.

Intuitivamente, più l'insieme $S$ è piccolo, più il problema è semplice.

In particolare, il problema risulta particolarmente semplice se $|S| = 1$ oppure $|S| = 2$.

- se $S$ ha un solo elemento, i.e. $S = \{s\}$, allora il più grande è banalmente $s$. Posso dire che:

$$\text{se } S = \{s\},\ max\_set(S) = s$$

- se $S$ ha due elementi, i.e. $S = \{a, b\}$, allora il più grande elemento nell'insieme è il massimo tra i due elementi, che posso calcolare facilmente.

$$\text{se } S = \{a, b\}, max\_set(S) = \max(a, b)$$

con $\max(\cdot, \cdot)$ funzione di appoggio definita come $\max(a, b) = \begin{cases} a & \text{se } a \geq b \\ b & \text{altrimenti} \end{cases}$

2. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?

Il caso immediatamente più complesso è quello avente $|S| = 3$. Dato che so risolvere il problema per $|S| = 1$ ed $|S| = 2$, posso sfruttare queste soluzioni per risolvere il caso $|S| = 3$?

Il massimo tra 3 numeri contenuti in un insieme $S$ lo posso vedere come il massimo del massimo tra due numeri qualsiasi in $S$ ed il numero rimanente, ossia:

$$\text{se } S = \{a, b, c\},\ max\_set(S) = \max(c, \max(a, b))$$

Dato che sappiamo che $\max(a, b) = max\_set(\{a, b\})$,

possiamo quindi riscrivere la soluzione al problema come:

$$\text{se } S = \{a, b, c\}, max\_set(S) = \max(c, max\_set(\{a, b\}))$$

3. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?

se $|S| = 4$, posso ripetere il ragionamento di prima e sfruttare la soluzione avuta con $|S| = 3$, quindi

$$\text{se } S = \{a, b, c, d\}, max\_set(S) = \max(d, max\_set(\{a, b, c\}))$$

4. posso generalizzare?

In generale,

$$\forall |S| > 2,\ max\_set(S) = \max(s \in S, max\_set(S \setminus \{s\})$$

Posso quindi definire la seguente funzione ricorsiva:

$$max\_set(S = \{s_1, s_2, \ldots, s_n\}) = \begin{cases} s, & \text{se } S = \{s\} \\ \max(s_1, s_2), & \text{se } S = \{s_1, \\ \max(s \in S, max\_set(S \setminus \{s\})), & \text{se } |S| > 2 \end{cases}$$

Che può essere ulteriormente semplificata come:

$$max\_set(S = \{s_1, s_2, \ldots, s_n\}) = \begin{cases} s, & \text{se } S = \{s\} \\ \max(s \in S, max\_set(S \setminus \{s\})), & \text{se } |S| \geq 2 \end{cases}$$

- (Un') *Implementazione* ricorsiva:

```
int max(int a, int b)
{
    if (a >= b)
        return a;
    return b;
}

int max_set(int S[], int n)
{
    if (n == 1)
        return S[0];
   // invoco max su S con un elemento in meno (per semplicità,
tolgo quello in posizione 0).
   // invoco quindi la funzione passandogli (l'indirizzo del)
vettore a partire dal secondo elemento
    //int m = max_set(&S[1], n-1);// <=> max_set(S+1, n-1);

    //return max(S[0], m);
    return max(S[0], max_set(S+1, n-1));
}
```

Posso anche pensare di "inglobare" la funzione `max(...)` all'interno della funzione `max_set(...)`, ottenendo la seguente implementazione alternativa:

```
int max_set(int S[], int n)
{
    if (n == 1)
        return S[0];
    int m = max_set(&S[1], n-1);// <=> max(S+1, n-1);
    if (m > S[0])
        return m;
    return S[0];
}
```

In [31]:
```
#include <stdio.h>
int max(int a,int b)
{
    if (a >= b)
        return a;
    return b;
}

int max_set(int v[], int n)
{
    if ( n == 1)
        return v[0];
    return max(v[0], max_set(v+1, n-1));

}

int main()
{
    int v[] = {1,3,2,-1,5,7,4,2,3};
    int n   = 9;
    printf("il massimo è %d\n", max_set(v,n));

}
```

```
il massimo è 7
```

In [29]:
```
#include <stdio.h>
int max_set(int S[], int n)
{
    if (n == 1)
        return S[0];
    int m = max_set(&S[1], n-1);// <=> max(S+1, n-1);
   // invoco max su S con un elemento in meno (per semplicità, tolgo quel
   // Passo quindi alla funzione (l'indirizzo del) vettore a partire dal
    if (m > S[0])
        return m;
    return S[0];
}
```

```
int main()
{
  int S[] = {7,6,8,3};
  int m   = max_set(S, 4);
  printf("il massimo è %d\n",m);
  return 0;
}
```

```
il massimo è 8
```

**Ricerca del massimo in un insieme: un altro approccio ricorsivo**

Posso pensare di dividere la ricerca del massimo in un insieme come la ricerca ricorsiva dei massimi locali su due metà dell'insieme, e prendere quindi il valore più grande tra i due.

In altri termini:

- divido l'insieme in due metà
- calcolo il massimo $m_1$ nella prima metà
- calcolo il massimo $m_2$ nella seconda metà
- calcolo il massimo tra $m_1$ ed $m_2$

Per comodità, vedo l'insieme come un array $\mathbf{v}$ ed il numero di elementi che lo compone com $\#\mathbf{v}$.

Posso quindi definire la funzione massimo in questo modo:

$$max\_set(\mathbf{v}) = \begin{cases} v_1 & \text{se } \#\mathbf{v} = 1 \\ \max(max\_set(\mathbf{v}_{1,..,\lfloor \frac{\#\mathbf{v}}{2} \rfloor}),\ max\_set(\mathbf{v}_{\lfloor \frac{\#\mathbf{v}}{2} \rfloor+1,..,\#\mathbf{v}})) & \text{se } \#\mathbf{v} > 1 \end{cases}$$

In questo modo, ho *due* invocazioni ricorsive nella stessa funzione.

In [32]:
```
#include <stdio.h>
int max_set(int v[], int n)
{
    if(n == 1)
        return v[0];

    // cerco il max sulla prima metà di v
    int m1 = max_set(v, n/2);
    // cerco il max sulla seconda metà di v (più eventuale elemento riman
    int m2 = max_set(v+n/2, n/2+n%2);
    if (m1 >= m2)
        return m1;
    return m2;
}

int main()
{
  int v[] = {7,6,8,3,9,11,2,4,13,21,24,1,3};
  int m   = max_set(v, 13);
  printf("il massimo è %d\n",m);
  return 0;
}
```

```
il massimo è 24
```

## Ricapitolando

In generale, dato un problema $P$, un insieme di dati $D_t$ con $t$ "indice di complessità" del problema sull'insieme di dati (i.e., $t+1 > t \Rightarrow$ problema su $D_{t+1}$ più complesso che su $D_t$), una soluzione $S$ del problema $P$ è ricorsiva se può essere definita come:

$$S(D_t) = \begin{cases} S_0 & se\ D_t = D_0, \\ f(S(D_j), D_t) & se\ D_t \neq D_0 \wedge j < t \end{cases}$$

Dove $D_0$ è un insieme di dati su cui il problema ha una soluzione $S_0$ banale/nota ed $f$ una funzione in grado di sfruttare:

- una soluzione ottenuta su un insieme di dati $D_j$ che dà luogo ad un problema di più semplice soluzione (i.e. $j < t$)
- i dati con complessità $t$ (se necessario).

## Repetita iuvant

**L'utilizzo di una funzione ricorsiva non implica che si stia fornendo una soluzione ricorsiva**

In [1]:
```
// Dato un insieme di numeri S, determinarne il massimo.
// Una soluzione che sembra ricorsiva, ma che ricorsiva  NON è!

#include <stdio.h>
void max(int S[], int n, int idx, int* m)
{
    if(idx < n-1)
        max(S,n,idx+1,m); /* nonostante la funzione sia ricorsiva,
                             non lavora su un sottoproblema più semplice,
                             ma sullo stesso problema.
                            Cambia solo il punto in esame attraverso idx
                            */
    if(S[idx] > *m)
        *m = S[idx];

}

int main()
{
  int S[] = {7,6,8,3};

  int m = S[0];
  max(S, 4, 0, &m);
  printf("il massimo è %d\n",m);

}

// Mera conversione di una soluzione iterativa utilizzando lo strumento r
```

```
il massimo è 8
```

## Problema

ricerca di un elemento $k$ in un insieme (e.g. $S = \{7, 6, 2, 8, 4\}, k = 8$). Si desidera una funzione che restituisca un valore logico di

- *true* (e.g., $1$), se $k \in S$,
- *false* (e.g., $0$) altrimenti.

*Soluzione iterativa:* scorro l'insieme finchè non trovo l'elemento selezionato. Se non lo trovo, l'elemento non è presente. Restituisco $1$ se è presente, $0$ altrimenti.

(Un') implementazione iterativa:

```
int S[]     = { 7,6,2,8,4 };
int n       = 5;
int k       = 8;
int trovato = 0;
for(int i = 0; i < n; i=i+1)
{
    if (S[i] == k)
    {
        trovato = 1;
        break;
    }
}
```

*Soluzione ricorsiva:*

Voglio definire una funzione $find\_in\_set(S, k)$, con $S$ insieme e $k$ valore da cercare.

1. esistono dei casi in cui il mio problema è facilmente risolvibile?

- Se il mio insieme è vuoto, ovviamente l'elemento non è presente
  $\Rightarrow \forall k,\ find\_in\_set(\emptyset, k) = false$
- Se l'insieme è composto da un solo elemento, allora:

$$find\_in\_set(S = \{s\}, k) = \begin{cases} false & \text{se } s \neq k \\ true & \text{altrimenti.} \end{cases}$$

2. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?

Se $S = \{s_1, s_2\}$, posso risolvere il problema sui due sottoinsiemi $\{s_1\}$ ed $\{s_2\}$ che, avendo cardinalità $1$, so risolvere. Se almeno uno dei due mi ha dato esito $true$, allora l'elemento $k$ è presente in $S$, altrimenti no.

$$find\_in\_set(S = \{s_1, s_2\}, k) = find\_in\_set(\{s_1\}, k) \vee find\_in\_set(\{s_2\}, k)$$

3. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?

Posso sfruttare una soluzione di $|S| \leq 2$ per $|S| = 3$? Ripetendo il ragionamento precedente, posso dividere $S$ in due sottoinsiemi:

$$\{s_1\} \text{ e } \{s_2, s_3\}$$

per ognuno dei quali so trovare la soluzione. Anche in questo caso, la soluzione del problema con $|S| = 3$ sarà data dall'$or$ logico tra le soluzioni dei due sottoinsiemi.

$$find\_in\_set(S = \{s_1, s_2, s_3\}, k) = find\_in\_set(\{s_1\}, k) \vee find\_in\_set(\{s_2, s_3\},$$

4. posso generalizzare?

$$\forall |S| > 1, find\_in\_set(S, k) = find\_in\_set(\{s \in S\}, k) \vee find\_in\_set(S \setminus \{s\}, k$$

Che da vita alla funzione ricorsiva:

$$find\_in\_set(S, k) = \begin{cases} false, & \text{se } \\ true, & \text{se } \\ find\_in\_set(\{s \in S\}, k) \vee find\_in\_set(S \setminus \{s\}, k), & \text{altr} \end{cases}$$

(Un') implementazione ricorsiva:

```
int find_in_set(int S[], int k, int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
    {
        if (k == S[0])
            return 1;
        return 0;
    }

    return find_in_set(&S[0], k, 1) || find_in_set(S+1, k, n-1);

}
```

oppure più semplicemente:

```
int find_in_set(int S[], int k, int n)
{
    if ( n == 0 )
        return 0;

    if ( k==S[0] )
        return 1;

    return find_in_set(S+1, k, n-1);
}
```

## Problema

dato un insieme, si vuole una funzione che calcoli la somma di tutti gli elementi al suo interno. Proporre una soluzione ricorsiva

1. esistono dei casi in cui il mio problema è facilmente risolvibile?

Sì, la somma di un insieme composto da un solo elemento è l'elemento stesso. La somma di un array composto da due elementi è la somma dei due elementi.

2. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?

Supponiamo di avere un insieme composto da 3 elementi. La somma sarà data dalla somma di due elementi appartenenti all'array (che sappiamo risolvere) con l'elemento rimanente 3. posso generalizzare?

$$\forall Vs.t. n = |V| > 2, sum(V) = a \in S + sum(V \setminus \{a\})$$

Considerando anche il caso base, ciò da vita alla funzione ricorsiva:

$$sum(V) = \begin{cases} a & \text{se } |V = \{a\}| = 1 \\ a + b & \text{se } |V = \{a, b\}| = 2 \\ a + sum(V \setminus \{a\}) & \text{altrimenti} \end{cases}$$

oppure più semplicemente

$$sum(V) = \begin{cases} a \in V & \text{se } |V = \{a\}| = 1 \\ a + sum(V \setminus \{a\}) & \text{altrimenti} \end{cases}$$

In [6]:
```
#include <stdio.h>
int sum(int v[], int n)
{
    if(n == 1)
        return v[0];

    return v[0] + sum(v+1, n-1);

}

int main()
{
  int v[] = {1,2,3,4};
  int m   = sum(v, 4);
  printf("la somma è %d\n",m);
  return 0;
}
```

```
la somma è 10
```

### Problema

dato un insieme, si vuole una funzione che calcoli la media aritmetica di tutti gli elementi al suo interno. Proporre una soluzione ricorsiva

Si potrebbe pensare di usare la seguente soluzione sfruttando la funzione ricorsiva `sum(...)` definita prima:

In [11]:
```
#include <stdio.h>
int sum(int v[], int n)
{
    if(n == 1)
        return v[0];

    return v[0] + sum(v+1, n-1);

}
float average(int v[], int n)
{
    return sum(v,n)/(float)n;
}

int main()
{
  int v[] = {1,2,3,4};
  float m   = average(v, 4);
  printf("la media è %f\n",m);
  return 0;
}
```

```
la media è 2.500000
```

Per quanto funzionante, in questa implementazione la funzione average non è ricorsiva, piuttosto *sfrutta* a sua volta una funzione ricorsiva. Una possibile implementazione ricorsiva potrebbe derivare dal seguente ragionamento:

1. esistono dei casi in cui il mio problema è facilmente risolvibile?

Sì, la media di un insieme composto da un solo elemento è l'elemento stesso. La media di un array composto da due elementi è la somma dei due elementi fratto 2.

2. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?

Supponiamo di avere un insieme composto da 3 elementi $V = \{a, b, c\}$. Posso sfruttare lasoluzione del problema a complessità minore (i.e. la media di due elementi, ad esempio $average(b, c)$) per costruire la soluzione del mio problema con 3 elementi? Così come viene restituita dalla funzione $average(b, c)$ no, in quanto dobbiamo sommare i valori $a, b, c$ e dividere la somma per 3, mentre invece la funzione $average(b, c)$ ci restituisce $\frac{b+c}{2}$, effettuando quindi una divisione di troppo. Tuttavia, possiamo ricavare la somma $b + c$ semplicemente moltiplicando $\frac{b+c}{2}$ per $2$. Quindi possiamo scrivere

$$average(V = \{a, b, c\}) = \frac{a + 2 \cdot average(\{b, c\})}{3}$$

3. posso generalizzare?

$$\forall Vs.t. n = |V| > 2, average(V) = \frac{a \in V + (n-1) \cdot average(V \setminus \{a\})}{n}$$

Considerando anche il caso base, ciò da vita alla funzione ricorsiva:

$$average(V) = \begin{cases} a & \text{se } |V = \{a\}| = 1 \\ \frac{a+b}{2} & \text{se } |V = \{a, b\}| = 2 \\ \frac{a+(n-1) \cdot average(V \setminus \{a\})}{n} & \text{altrimenti} \end{cases}$$

oppure più semplicemente

$$average(V) = \begin{cases} a \in V & \text{se } |V = \{a\}| = 1 \\ \frac{a+(n-1)\cdot average(V \setminus \{a\})}{n} & \text{altrimenti} \end{cases}$$

In [16]:
```
#include <stdio.h>

float average(int v[], int n)
{
    if(n == 1)
        return v[0];
    return (v[0]+(n-1)*average(v+1,n-1))/(float)n;
}

int main()
{
  int v[] = {1,2,3,4};
  float m   = average(v, 4);
  printf("la media è %f\n",m);
  return 0;
}
```

```
la media è 2.500000
```

## Alcuni tipi di ricorsione

In base al numero di invocazioni ricorsive che si trovano all'interno di una funzione, si distingue tra:

- ricorsione *lineare*: è sufficiente una singola chiamata ricorsiva per definire il valore restituito

  `return find_in_set(S+1, k, n-1);`

- ricorsione *non lineare*: sono necessarie più chiamate ricorsive per definire il valore restituito

`return find_in_set(&S[0], k, 1) || find_in_set(S+1, k, n-1);`

In [34]:
```
#include <stdio.h>

int find_in_set(int S[], int k, int n)
{
    if (n == 0)
        return 0;

    if (k==S[0])
        return 1;

    return find_in_set(&S[1], k, n-1);
}

/*
oppure...

int find_in_set(int S[], int k, int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
    {
        if (k == S[0])
            return 1;
        return 0;
    }

    return find_in_set(&S[0], k, 1) || find_in_set(S+1, k, n-1);
}
*/


int main()
{
    int S[] = {7,6,2,8,4};
    int n   = 5;
    int k   = 9;
    int trovato = find_in_set(S,k,n);
    if(trovato == 1)
        printf("trovato\n");
    else
        printf("non trovato\n");
}
```

```
non trovato
```