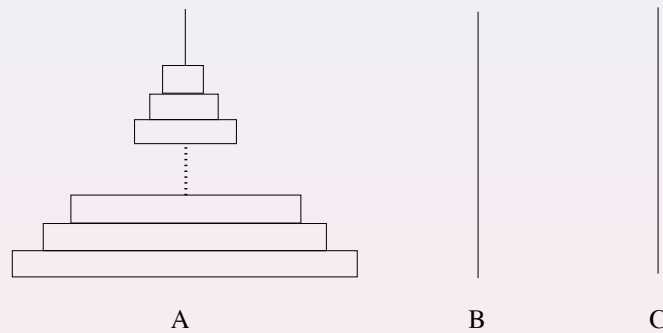


Programmazione ricorsiva: cenni

- ▶ In quasi tutti i linguaggi di programmazione evoluti è ammessa la possibilità di definire funzioni/procedure **ricorsive**: durante l'esecuzione di una funzione F è possibile chiamare la funzione F stessa.
- ▶ Ciò può avvenire
 - ▶ **direttamente**: il corpo di F contiene una chiamata a F stessa.
 - ▶ **indirettamente**: F contiene una chiamata a G che a sua volta contiene una chiamata a F .
- ▶ Questo può sembrare strano: se pensiamo che una funzione è destinata a risolvere un sottoproblema \mathcal{P} , una definizione ricorsiva sembra indicare che per risolvere \mathcal{P} dobbiamo ... saper risolvere \mathcal{P} !

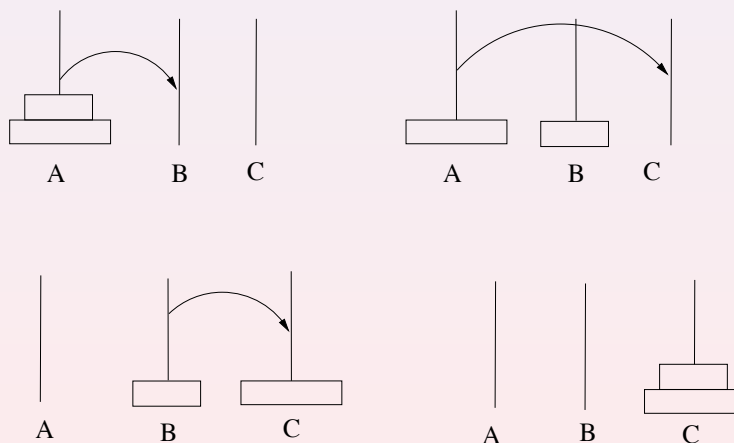
- ▶ In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi **la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.**
- ▶ La programmazione ricorsiva trova radici teoriche nel **principio di induzione ben fondata** che può essere visto come una generalizzazione del **principio di induzione** sui naturali
- ▶ La soluzione di un problema viene individuata **supponendo** di saperlo risolvere su casi più semplici.
- ▶ Bisogna poi essere in grado di risolvere **direttamente** il problema sui casi più semplici di qualunque altro.

Esempio: Torre di Hanoi (leggenda Vietnamita).



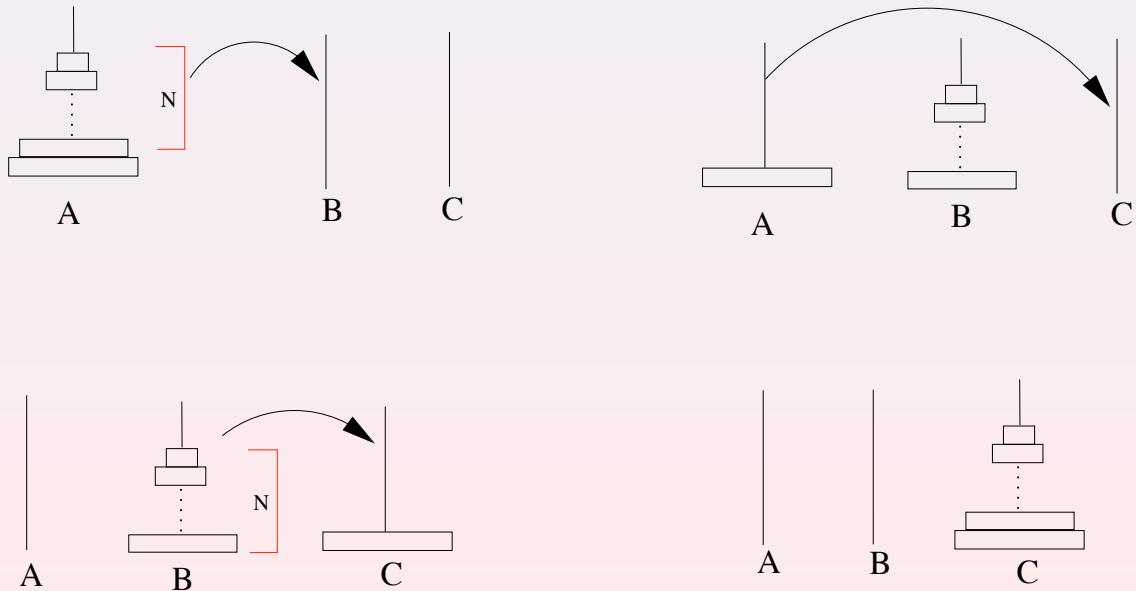
- ▶ pila di dischi di dimensione decrescente su un perno **A**
- ▶ vogliamo spostarla sul perno **C**, usando un perno di appoggio **B**
- ▶ vincoli:
 - ▶ possiamo spostare un solo disco alla volta
 - ▶ un disco più grande non può mai stare su un disco più piccolo
- ▶ secondo la leggenda: i monaci stanno spostando **64** dischi: quando avranno finito, ci sarà la fine del mondo

- ▶ Come individuare una soluzione per un numero **N** di dischi arbitrario?
 - ▶ per **N=1** la soluzione è immediata: spostiamo l'unico disco da **A** a **C**
 - ▶ se sappiamo risolvere il problema per **N=1** lo sappiamo risolvere anche per **N=2**: come?



- ▶ Notiamo l'utilizzo del perno ausiliario **B**

- Possiamo generalizzare il ragionamento? Se sappiamo risolvere il problema per N dischi, possiamo individuare una soluzione per lo stesso problema ma con $N+1$ dischi?



- Formalizziamo il ragionamento
- Indichiamo con $\text{hanoi}(N, P1, P2, P3)$ il problema: "spostare N dischi dal perno $P1$ al perno $P2$ utilizzando $P3$ come perno d'appoggio".

```

hanoi(N, P1, P2, P3)
  if (N=1)
    sposta da P1 a P2;
  else
    {
      hanoi(N-1, P1, P3, P2);
      sposta da P1 a P2;
      hanoi(N-1, P3, P2, P1);
    }

```

Esempio: Soluzione di `hanoi(3,A,C,B)`

<code>hanoi(3,A,C,B) =</code>	<code>sposta(A, C)</code>	<code>hanoi(2,A,B,C) =</code>	<code>sposta(A,B)</code>	<code>hanoi(1,A,C,B) =</code>	<code>sposta(A,C)</code>
				<code>hanoi(1,C,B,A) =</code>	<code>sposta(C,B)</code>
		<code>hanoi(2,B,C,A) =</code>	<code>sposta(B,C)</code>	<code>hanoi(1,B,A,C) =</code>	<code>sposta(B,A)</code>
				<code>hanoi(1,A,C,B) =</code>	<code>sposta(A,C)</code>

- ▶ Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo **induttivo**.

Esempio: Definizione induttiva di somma tra due interi non negativi:

$$somma(x, y) = \begin{cases} x & \text{se } y=0 \\ 1 + (somma(x, y - 1)) & \text{se } y > 0 \end{cases}$$

- ▶ La somma di x con 0 viene definita in modo immediato;
- ▶ la somma di x con il successore di y viene definita come il successore della somma tra x e y .
- ▶ **Esempio:** somma di 3 e 2 :

$$\begin{aligned} somma(3, 2) &= 1 + (somma(3, 1)) = \\ &= 1 + (1 + (somma(3, 0))) = \\ &= 1 + (1 + (3)) = \\ &= 1 + 4 = \\ &= 5 \end{aligned}$$

Esempio: Funzione fattoriale.

- ▶ definizione iterativa: $fatt(n) = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1$
- ▶ definizione induttiva:

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 & \text{(caso base)} \\ n \cdot fatt(n - 1) & \text{se } n > 0 & \text{(caso induttivo)} \end{cases}$$

- ▶ È essenziale il fatto che, applicando ripetutamente il caso induttivo, ci riconduciamo prima o poi al caso base.

$$\begin{aligned} fatt(3) &= 3 \cdot fatt(2) = \\ &= 3 \cdot (2 \cdot fatt(1)) = \\ &= 3 \cdot (2 \cdot (1 \cdot fatt(0))) = \\ &= 3 \cdot (2 \cdot (1 \cdot 1)) = \\ &= 3 \cdot (2 \cdot 1) = \\ &= 3 \cdot 2 = \\ &= 6 \end{aligned}$$

Il codice delle due diverse versioni

- ▶ definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
        ris=ris*i;
    return ris;
}
```

- ▶ definizione ricorsiva:

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

Esempio: Programma che usa una funzione ricorsiva.

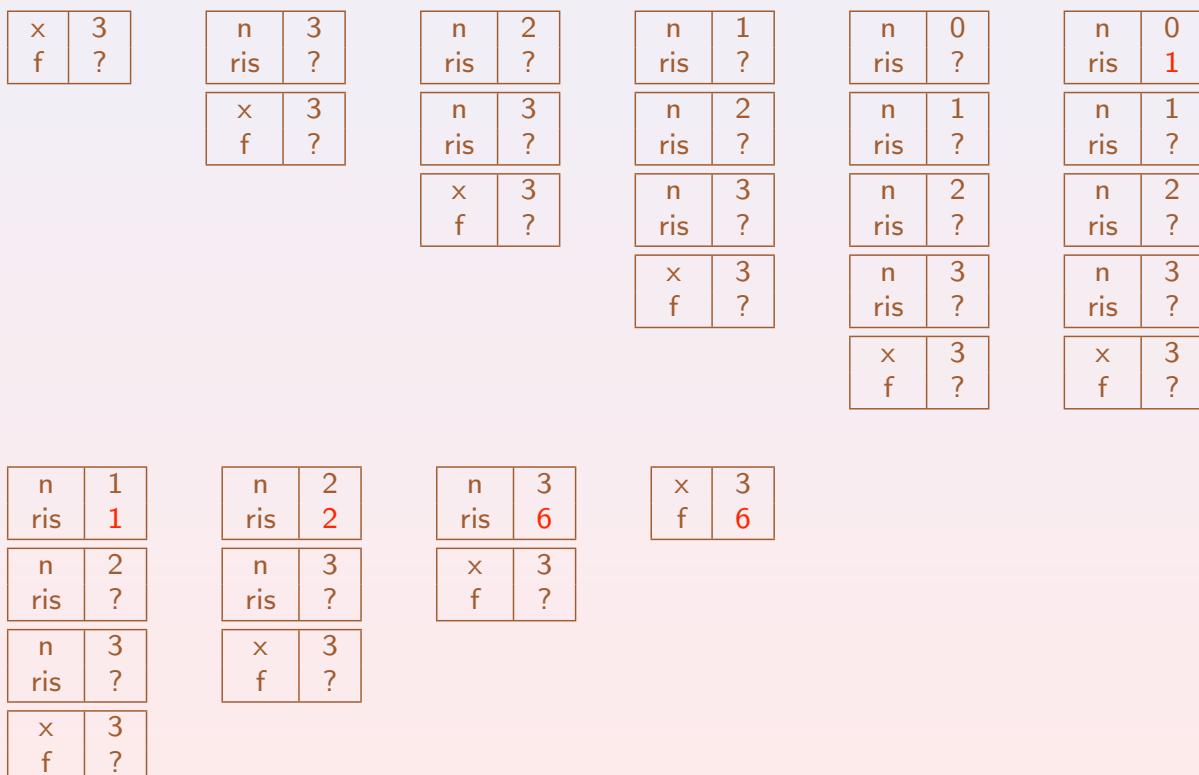
```
#include <stdio.h>

int fattric (int);

main()
{
int x, f;
scanf("%d", &x);
f = fattric(x);
printf("Fattoriale di %d:  %d\n", x, f);
}

int fattric(int n) {
int ris;
if (n == 0)
    ris = 1;
else
    ris = n * fattric(n-1);
return ris;
}
```

Evoluzione della pila (supponendo x=3).



Esempio: Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `casa` \implies `asac`

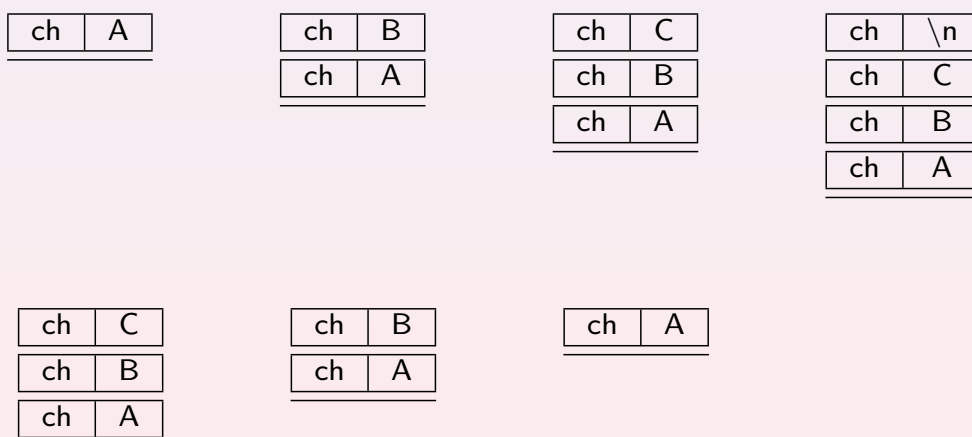
- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
 1. usando una struttura dati opportuna ma **dinamica** (liste, le vedremo più avanti)
 2. usando un procedimento ricorsivo.
 - ▶ leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;
 - ▶ il caso base è rappresentato dalla lettura del carattere di fine sequenza.

```
void invertInputRic()
{ char ch;

  ch = getchar();
  if (ch != '\n')
  {
    invertInputRic();
    putchar(ch);
  }
  else
    printf("Sequenza invertita: ");
}
```

```
main()
{
  printf("Immetti una sequenza di caratteri\n");
  invertInputRic();
  printf("\n");
}
```

Vediamo come evolve la pila per l'input `ABC\n`



L'output prodotto è il seguente

Sequenza invertita: CBA

Ricorsione multipla

- ▶ Si ha ricorsione multipla quando un'attivazione di una funzione può causare **più di una attivazione ricorsiva** della stessa funzione (es. torre di Hanoi)

Esempio: Definizione induttiva dei numeri di Fibonacci.

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \quad \text{se } n > 1 \end{aligned}$$

- ▶ $F(0), F(1), F(2), \dots$ è detta sequenza dei numeri di Fibonacci:
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
#include <stdio.h>

int fibonacci (int);

main() {
    int n;

    printf("Inserire un intero >= 0: ");
    scanf("%d", &n);
    printf("Numero %d di Fibonacci: %d\n", n, fibonacci(n));
}

int fibonacci(int i)
{
    int ris;
    if (i == 0)
        ris = 0;
    else if (i == 1)
        ris = ;
    else
        ris = fibonacci(i-1) + fibonacci(i-2);
    return ris;
}
```


Esempi di funzioni ricorsive

- ▶ Tradurre in C la definizione induttiva già vista:

$$somma(x, y) = \begin{cases} x & \text{se } y = 0 \\ 1 + (somma(x, y - 1)) & \text{se } y > 0 \end{cases}$$

```
int somma (int x, int y)
{
    int ris;
    if (y==0)
        ris = x;
    else
        ris = 1 + somma(x, y-1);
    return ris;
}
```

- ▶ Calcolo ricorsivo di x^y (si assume $y \geq 0$)

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ x \cdot x^{y-1} & \text{altrimenti} \end{cases}$$

```
int exp (int x, int y)
{
    int ris;
    if (y==0)
        ris = 1;
    else
        ris = x * exp(x, y-1);
    return ris;
}
```

- ▶ Calcolare ricorsivamente la somma degli elementi di un array v .
- ▶ Esprimiamo formalmente quanto richiesto:

$$\text{sumVet}(v, \text{dim}) = \sum_{i=0}^{\text{dim}-1} v[i]$$

- ▶ È evidente che:

$$\sum_{i=0}^{\text{dim}-1} v[i] = \begin{cases} 0 & \text{se } \text{dim} = 0 \\ v[0] + \sum_{i=1}^{\text{dim}-1} v[i] & \text{se } \text{dim} > 0 \end{cases}$$

- ▶ La traduzione in **C** è immediata.

```
int sumVet(int *v, int dim)
{
    if (dim==0)
        return 0;
    else
        return v[0] + sumvet(v+1,dim-1);
}
```

```
int sumVet(int *v, int dim)
{
    int somma;
    if (dim==0)
        somma = 0;
    else
        somma = v[0] + sumvet(v+1,dim-1);
    return somma;
}
```

- Calcolare ricorsivamente il numero di occorrenze dell'elemento x nell'array v .

$$f(v, x, dim) = \#\{i \in [0, dim - 1] \mid v[i] = x\}$$

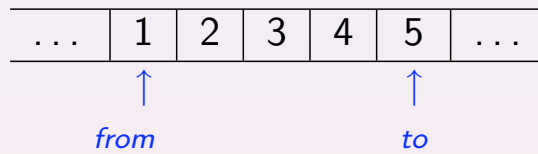
- Anche in questo caso ragioniamo induttivamente:

$$f(v, x, dim) = \begin{cases} 0 & \text{se } dim == 0 \\ f(v + 1, x, dim - 1) & \text{se } dim > 0 \wedge v[0] \neq x \\ 1 + f(v + 1, x, dim - 1) & \text{se } dim > 0 \wedge v[0] = x \end{cases}$$

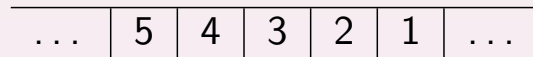
```
int occorrenze (int *v, int x, int dim)
{
    int occ;

    if (dim==0)
        occ= 0;
    else
        if (v[0]!=x)
            occ = occorrenze(v+1,x,dim-1);
        else
            occ = 1+occorrenze(v+1,x,dim-1);
}
```

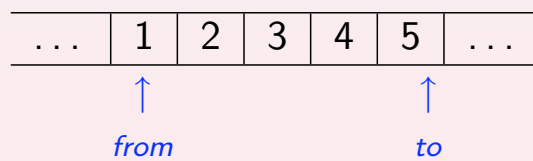
- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici *from* e *to*.



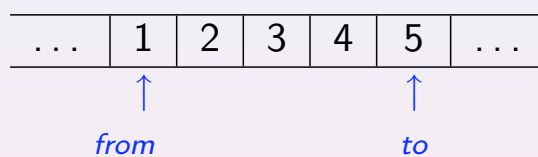
- Vogliamo ottenere:



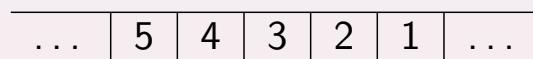
- Induttivamente:



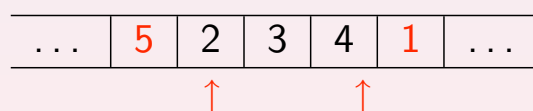
- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici *from* e *to*.



- Vogliamo ottenere:



- Induttivamente:



- Questa situazione corrisponde alla chiamata ricorsiva su una porzione più piccola del vettore

```

void swap(int *v, int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

void invertiric (int *v, int from, int to)
{
    if (from < to)
    {
        swap(v, from, to);
        invertiric(v, from+1, to-1);
    }
}

```

Si noti che la procedura non fa niente se la porzione individuata dal secondo e terzo parametro è vuota ($from > to$) o contiene un solo elemento ($from = to$)

Una soluzione alternativa per invertire tutto il vettore,

```

void invertiric (int *v, int dim)
{
    if (dim > 1)
    {
        swap(v, 0, dim-1);
        invertiric(v+1, dim-2);
    }
}

```

Schemi di programma: ricerca e verifica

- ▶ Molti problemi riguardano la ricerca di elementi in intervalli o la verifica di proprietà.
- ▶ Sviluppiamo **schemi** di programma dimostrabilmente corretti che realizzano la ricerca e la verifica.
- ▶ La soluzione di problemi concreti consiste poi nella sostituzione di alcuni **parametri** degli schemi con valori specifici dei problemi in esame.
- ▶ Distinguiamo due tipi di ricerca: ricerca **certa** e ricerca **incerta**.
 - ▶ **ricerca certa**: si vuole determinare il minimo elemento di un intervallo $[a,b)$ per il quale vale una certa proprietà \mathcal{P} , sapendo che almeno un elemento dell'intervallo soddisfa \mathcal{P} .
 - ▶ **ricerca incerta**: si vuole determinare, se esiste, il minimo elemento di un intervallo $[a,b)$ per il quale vale una certa proprietà \mathcal{P} .

Ricerca certa

- ▶ Intervallo di ricerca: $[a,b)$
- ▶ Proprietà: $\mathcal{P}(\cdot)$
- ▶ Ipotesi di certezza: $\exists i \in [a,b) . \mathcal{P}(i)$
- ▶ Stato finale:
 $x = \min \{ i \in [a,b) \mid \mathcal{P}(i) \}.$
- ▶ Lo schema generale per risolvere il problema è il seguente:

```

int x;
x=a;
while (! $\mathcal{P}(x)$ )
    x=x+1;
  
```

- ▶ Nota: l'estremo destro dell'intervallo non serve.
- ▶ Si assume che la proprietà \mathcal{P} sia esprimibile nel linguaggio.

- ▶ Proprietà invariante del ciclo:

$$x \in [a, b) \wedge (\forall j \in [a, x]. \neg \mathcal{P}(j))$$

- ▶ In altre parole, tutti gli elementi che precedono il valore corrente di x non soddisfano la proprietà \mathcal{P} .

- ▶ Se il ciclo termina, all'uscita dal ciclo vale la congiunzione

$$x \in [a, b) \wedge (\forall j \in [a, x]. \neg \mathcal{P}(j)) \wedge \mathcal{P}(x)$$

che implica esattamente quanto espresso dallo stato finale:

$$x = \min \{ i \in [a, b) \mid \mathcal{P}(i) \}$$

- ▶ Osserviamo che l'invariante vale banalmente alla prima iterazione, con $x=a$.

$$a \in [a, b) \wedge (\forall j \in [a, a]. \neg \mathcal{P}(j))$$

- ▶ Verifichiamo che la proprietà

$$x \in [a, b) \wedge (\forall j \in [a, x]. \neg \mathcal{P}(j))$$

è invariante per il ciclo:

$x=a;$

while ($\neg \mathcal{P}(x)$)

$x=x+1;$

- ▶ Sia S uno stato in cui valgono le seguenti proprietà (x^S indica il valore di x in S)

1. $x^S \in [a, b) \wedge (\forall j \in [a, x^S]. \neg \mathcal{P}(j))$
2. $\neg \mathcal{P}(x^S)$

ovvero uno stato prima di una nuova iterazione del ciclo.

- ▶ 1. e 2. implicano ovviamente

$$(\forall j \in [a, x^S+1]. \neg \mathcal{P}(j))$$

- ▶ Se riusciamo anche a dimostrare che

$$x^S + 1 \in [a, b)$$

abbiamo dimostrato che la proprietà è invariante, dal momento che

$x^S + 1$ è proprio il valore di x dopo la nuova iterazione.

- ▶ Sappiamo:
 $x^s \in [a, b)$
- ▶ Supponiamo per assurdo
 $x^s + 1 \notin [a, b)$ ovvero $x^s + 1 = b$ (*)
- ▶ Abbiamo appena dimostrato
 $(\forall j \in [a, x^s + 1). \neg \mathcal{P}(j))$
 che insieme con (*) implica
 $(\forall j \in [a, b). \neg \mathcal{P}(j))$
- ▶ Ciò contraddice l'ipotesi di certezza
 $\exists i \in [a, b) . \mathcal{P}(i)$
- ▶ Dunque, dopo la nuova iterazione vale ancora la proprietà invariante.

Funzione di terminazione: Tra le tante ... $b-x$

Ricerca certa: esempio 1

- ▶ Calcolare la radice intera di un numero naturale.
- ▶ Si può esprimere come problema di ricerca certa:
 $\lfloor \sqrt{N} \rfloor = \min \{ x \in [0, N+1) \mid x^2 \leq N < (x+1)^2 \}$
- ▶ Dunque l'estremo sinistro dell'intervallo di ricerca, a nello schema, in questo caso è 0 , mentre l'estremo destro, b nello schema, è N .
- ▶ Infine la proprietà $\mathcal{P}(x)$ dello schema è $N < (x+1)^2$

```

int x;
x=0;
while ((x+1)*(x+1) <= N)
    x=x+1;

```


Ricerca certa: esempio 2

- ▶ Determinare la posizione della prima occorrenza di un dato elemento in un array, sapendo che tale elemento vi occorre almeno una volta.
- ▶ Indichiamo con `vet` l'array e con `DIM` la sua dimensione
- ▶ Vogliamo determinare:

$$x = \min\{i \in [0, DIM) \mid \text{vet}[i] = e1\}$$
- ▶ Possiamo istanziare lo schema come segue:

```
int x;
x=0;
while (vet[x] !=e1)
    x=x+1;
```

Ricerca Incerta

- ▶ Si vuole determinare, **se esiste**, il minimo elemento di un intervallo $[a,b)$ per il quale vale una certa proprietà \mathcal{P} .
- ▶ Perché lo schema di ricerca certa non va bene?

```
x=a;
while (!P(x))
    x=x+1;
```
- ▶ Se l'elemento non c'è si vanno ad esaminare valori di x che sono al di fuori dell'intervallo di ricerca e per i quali la proprietà \mathcal{P} potrebbe addirittura non essere definita (errore a tempo di esecuzione).
Esempio: Nel caso della ricerca **incerta** di un elemento in un array di dimensione `DIM` si andrebbero ad esaminare elementi del tipo `vet[x]` con $x > DIM$.
- ▶ Abbiamo bisogno di modificare lo schema in modo che l'analisi degli elementi avvenga solo all'interno dell'intervallo di ricerca e che la ricerca venga interrotta una volta esaurito l'intervallo (e non individuato alcun elemento).

Ricerca incerta

- ▶ Intervallo di ricerca: $[a,b)$
- ▶ Proprietà: $\mathcal{P}(\cdot)$
- ▶ Stato finale: $x = \min\{i \in [a,b) \mid \mathcal{P}(i)\} \text{ min } b$
 \implies dobbiamo stabilire quale valore calcolare se **nessun** elemento dell'intervallo soddisfa \mathcal{P} : una buona scelta è il valore b , che sicuramente **non** fa parte dell'intervallo.

Ricerca incerta

- ▶ Utilizziamo una variabile booleana **trovato** che fa da **sentinella**
 \implies impone l'uscita dal ciclo non appena si individua un elemento che soddisfa la proprietà
- ▶ in congiunzione con la sentinella, la guardia del ciclo assicura che l'intervallo di ricerca non sia esaurito

```
int trovato = FALSE; /* inizialmente false */
int x=a;
while (!trovato && x<b)
    if (P(x))
        trovato = TRUE; /*x soddisfa P */
    else
        x=x+1;
```

- ▶ Si suppone che le costanti **TRUE** e **FALSE** siano state definite opportunamente, ad esempio mediante le direttive

```
#define FALSE 0
#define TRUE 1
```

- ▶ Anche in questo caso possiamo stabilire una proprietà invariante del ciclo, questa volta un po' più complicata:

$$x \in [a,b] \wedge (\forall j \in [a,x]. \neg \mathcal{P}(j)) \wedge \text{trovato} \Rightarrow \mathcal{P}(x)$$

- ▶ È facile vedere che i valori iniziali di x e **trovato** soddisfano banalmente l'invariante
- ▶ Inoltre, al termine del ciclo abbiamo due casi:
 1. $\text{trovato} = \text{TRUE} \wedge x < b$: l'invariante e questa condizione implicano $x \in [a,b) \wedge (\forall j \in [a,x]. \neg \mathcal{P}(j)) \wedge \mathcal{P}(x)$
 2. $\text{trovato} = \text{FALSE} \wedge x \geq b$: l'invariante e questa condizione implicano $x = b \wedge (\forall j \in [a,b). \neg \mathcal{P}(j))$
- ▶ Dunque possiamo controllare l'**esito** della ricerca analizzando il valore di **trovato**

- ▶ La dimostrazione formale di invarianza della proprietà vista è lasciata per esercizio
- ▶ **Funzione di terminazione**: anche in questo caso qualcosa del tipo $b - x$ sembra ragionevole.
- ▶ Il problema (formale) è che in un solo caso il valore di x non cresce (e dunque $b - x$ non decresce) strettamente.
- ▶ L'individuazione di una corretta funzione di terminazione è lasciata per esercizio.

Ricerca incerta: esempio

- ▶ Determinare la prima occorrenza di un elemento in un array.

- ▶ È un problema di ricerca incerta:

$$\min \{x \in [0, DIM) \mid \text{vet}[x] = e1\} \text{ min } DIM$$

```
int trovato = FALSE;
int x=0;
while (!trovato && x<DIM)
    if (vet[x]==e1)
        trovato = TRUE;
    else
        x=x+1;
```

- ▶ Vi sono situazioni in cui la proprietà \mathcal{P} della ricerca (certa o incerta) non è direttamente esprimibile nel linguaggio.

Esempio: Determinare (se c'è) la posizione del primo elemento di un array di interi che è uguale alla somma degli elementi che lo precedono.

- ▶ Si tratta di un problema di ricerca incerta in cui

1. l'intervallo $[a, b]$ è $[0, DIM]$
2. la proprietà $\mathcal{P}(x)$ è

$$\text{vet}[x] = \sum_{j=0}^{x-1} \text{vet}[j]$$

```
int trovato = FALSE;
int x=0;
while (!trovato && x<DIM)
    if (vet[x]== $\sum_{j=0}^{x-1}$ vet[j])
        trovato = TRUE;
    else
        x=x+1;
```

- ▶ In questi casi si utilizza la seguente tecnica:
 1. si rimpiazzano le espressioni “critiche” con variabili
 2. si impone l’uguaglianza tra le variabili così’ introdotte e le corrispondenti espressioni “critiche”, aggiungendo quanto necessario al corpo del ciclo per mantenere vere tali uguaglianze
- ▶ si noti che formalmente 2. corrisponde a rafforzare opportunamente l’invariante.
- ▶ Nell’esempio:

```

int trovato = FALSE;
int x=0;
int sommaPrecedenti = 0;
while (!trovato && x<DIM)
  if (vet[x]==sommaPrecedenti)
    trovato = TRUE;
  else
    { sommaPrecedenti = sommaPrecedenti + vet[x];
      x=x+1;      }

```

- ▶ Quale è l’invariante del ciclo così’ ottenuto?

$$\begin{aligned}
 &x \in [0, DIM] \wedge (\forall j \in [0, x). \text{vet}[j] \neq \sum_{k=0}^{j-1} \text{vet}[k]) \wedge \\
 &\text{trovato} \Rightarrow \text{vet}[x] = \sum_{k=0}^{x-1} \text{vet}[k] \wedge \\
 &\text{sommaPrecedenti} = \sum_{k=0}^{x-1} \text{vet}[k]
 \end{aligned}$$

- ▶ L’ultimo congiunto rappresenta il significato della variabile introdotta per esprimere la proprietà di ricerca \mathcal{P} .

Verifica di una proprietà

- ▶ Vogliamo verificare che tutti gli elementi di un intervallo soddisfano una certa proprietà \mathcal{P} .
 1. Facciamo una ricerca **incerta** del minimo elemento dell'intervallo per il quale **non** vale la proprietà \mathcal{P}
 2. Se non troviamo tale minimo, la verifica ha esito positivo, altrimenti ha esito negativo.
- ▶ Lo schema generale per risolvere questo problema.

```

int trovato = FALSE;
int x=a;
while (!trovato && x<b)
  if (! $\mathcal{P}$ (x))
    trovato = TRUE;
  else
    x=x+1;
if (trovato)
  /* esito negativo */
else
  /* esito positivo */

```

Tipi user-defined

- ▶ Il **C** mette a disposizione un insieme di tipi di dato predefiniti (tipi **built-in**) e dei meccanismi per definire nuovi tipi (tipi **user-defined**)
- ▶ Vediamo le regole generali che governano la definizione di nuovi tipi e quindi i costrutti linguistici (**costruttori**) che il **C** mette a disposizione.
- ▶ Tutti i tipi non predefiniti utilizzati in un programma devono essere dichiarati come ogni altro elemento del programma. Una **dichiarazione di tipo** viene fatta di solito nella parte dichiarativa del programma.
 - ▶ parte dichiarativa globale:
 - ▶ dichiarazioni di costanti
 - ▶ **dichiarazioni di tipi**
 - ▶ dichiarazioni di variabili
 - ▶ prototipi di funzioni/procedure

Dichiarazione di tipo

- ▶ Una **dichiarazione di tipo** (type declaration) consiste nella parola chiave **typedef** seguita da:
 - ▶ la **rappresentazione** o **costruzione** del nuovo tipo (ovvero la specifica di come è costruito a partire dai tipi già esistenti)
 - ▶ il nome del nuovo tipo
 - ▶ il simbolo **;** che chiude la dichiarazione

Esempio: `typedef int anno;`

- ▶ Una volta definito e nominato un nuovo tipo, è possibile utilizzarlo per dichiarare nuovi oggetti (ad es. variabili) di quel tipo.

Esempio:

```
float x;
anno a;
```

- ▶ **Nota:** In **C** si possono anche definire tipi senza usare **typedef**. Quest'ultima consente l'associazione di un nome (identificatore) a un nuovo tipo. Per uniformità e leggibilità del codice useremo spesso **typedef** per definire nuovi tipi.

Tipi semplici user-defined

Ridefinizione: Un nuovo tipo può essere definito rinominando un tipo già esistente (cioè creandone un **alias**)

```
typedef TipoEsistente NuovoTipo;
```

dove **TipoEsistente** può essere un tipo built-in o user-defined.

Esempio:

```
typedef int anno;
typedef int naturale;
typedef char carattere;
```

Enumerazione: Consente di definire un nuovo tipo **enumerando** i suoi valori, con la seguente sintassi

```
typedef enum {v1, v2, ... , vk} NuovoTipo;
```

Esempio:

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
typedef enum {gen, feb, mar, apr, mag, giu,
             lug, ago, set, ott, nov, dic} Mese;
typedef enum {m, f} sesso;
```

- ▶ I valori elencati nella definizione di un nuovo tipo enumerato, sono identificatori che rappresentano **costanti** di quel tipo (esattamente come `0`, `1`, `2`, ... sono costanti del tipo `int`, o `'a'`, `'b'`, ... sono costanti del tipo `char`).
- ▶ Dunque, se dichiariamo una variabile
`Giorno g;`
 possiamo scrivere l'assegnamento
`g = mar;`
- ▶ Le costanti dei tipi enumerati **non** vanno racchiuse tra virgolette o tra apici!

N.B. Il compilatore associa ai nomi utilizzati per denotare le costanti dei tipi enumerati valori **naturali** progressivi.

Esempio: il valore associato a `g` dopo l'assegnamento `g=mar` è il numero naturale (intero) `1`.

⇒ mancanza di astrazione: è possibile fare riferimento alla **rappresentazione** dei valori.

- ▶ La relazione tra interi e tipi enumerati consente di applicare a questi ultimi le seguenti operazioni:
 - ▶ operazioni aritmetiche: `+`, `-`, `*`, `/`, `%`
 - ▶ uguaglianza e disuguaglianza: `=`, `!=`
 - ▶ confronto: `<`, `<=`, `>`, `>=`

- ▶ Si noti che la relazione di precedenza tra i valori (che determina l'esito delle operazioni di confronto) dipende dall'**ordine** in cui vengono elencati i valori del tipo al momento della sua definizione.

Esempio: Con le dichiarazioni viste in precedenza
`lun < gio` è **vero** (un intero diverso da `0`) `apr <= feb` è **falso** (il valore intero `0`)

- ▶ Il C tratta questi tipi come ridefinizione di `int`

Tipi fai da te: i booleani

Soluzione 1

```
#define FALSE 0;
#define TRUE 1;...
typedef int Boolean;
Boolean b;
...
```

Soluzione 2

```
typedef enum {FALSE, TRUE} Boolean;...
Boolean b;
...
```

N.B. I valori vanno elencati come sopra, rispettando la convenzione adottata dal C: il valore 0 rappresenta **false**.

Esempio:

```
typedef enum {false, true} boolean;

boolean even (int n)
{
  if (n % 2 == 0)
    return true;
  else
    return false;
}

boolean implies (boolean p, boolean q)
{
  if (p)
    return q;
  else
    return true;
}
```

Esempio: Uso del costrutto `switch` con tipi enumerati

```

typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
Giorno g;
...
switch (g) {
case lun: case mar: case mer: case gio: case ven:
    printf("Giorno lavorativo");
    break;
case sab: case dom:
    printf("Week-end");
    break;
}

void stampaGiorno(Giorno g) {
switch (g) {
case lun: printf("lun");
    break;
...

case dom: printf("dom");
    break;
}

```

Tipi strutturati user-defined

- ▶ Il `C` non possiede tipi strutturati built-in, ma fornisce dei **costruttori** che permettono di definire tipi strutturati anche piuttosto complessi.
- ▶ Array e puntatori possono essere visti come **costruttori** di tipo (definiscono un tipo di dato non semplice a partire da tipi esistenti).

Uso di `typedef` con array e puntatori

- ▶ In generale, una dichiarazione di tipo mediante `typedef` ha la forma di una dichiarazione di variabile preceduta dalla parola chiave `typedef`, e con il nome di tipo al posto del nome della variabile.
- ▶ Nel caso di array e puntatori:

```

typedef TipoElemento TipoArray[Dimensione];
typedef TipoPuntato *TipoPuntatore;

```

Esempio:

```

typedef int ArrayDieciInteri[10];
typedef int MatriceTreXQuattro[3][4];
typedef int *PuntIntero;
ArrayDieciInteri vet; /* int vet[10]; */
PuntIntero p; /* int *p; */
MatriceTreXQuattro mat, mat1; /* int mat[3][4]; int mat1[3][4]; */

```

Il costruttore struct

- ▶ Una **struttura** è un'aggregazione di elementi che possono essere **eterogenei** (di tipo diverso).

Esempio:

```
struct persona {
    char nome[15];
    char cognome[20];
    int eta;
    sesso s; }
```

- ▶ la parola chiave **struct** introduce la definizione della struttura
- ▶ **persona** è l'**etichetta** della struttura, attribuisce un nome alla definizione della struttura
- ▶ **nome**, **cognome**, **eta**, **s** sono detti **campi** della struttura
- ▶ È anche possibile definire strutture con campi omogenei

```
struct complex {
    double real;
    double imag; }
```

Campi di una struttura

- ▶ devono avere nomi univoci all'interno di una struttura
- ▶ strutture diverse possono avere campi con lo stesso nome
- ▶ i nomi dei campi possono coincidere con altri nomi già utilizzati (es. per variabili o funzioni)

Esempio:

```
int x;
struct a { char x; int y; };
struct b { int w; float x; };
```

- ▶ possono essere di tipo diverso (semplice o altre strutture)
- ▶ un campo di una struttura non può essere del tipo struttura che si sta definendo
- ▶ un campo può però essere di tipo puntatore alla struttura

Esempio:

```
struct s { int a;
           struct s *p; };
```

Dichiarazione di variabili di tipo struttura

- ▶ La definizione di una struttura non provoca allocazione di memoria, ma introduce un nuovo tipo di dato.

Esempio: `struct persona tizio, docenti[10], *p;`

- ▶ `tizio` è una variabile di tipo `struct persona`
 - ▶ `docenti` è un vettore di 10 elementi di tipo `struct persona`
 - ▶ `p` è un puntatore a una `struct persona`
 - ▶ N.B.: `persona tizio;` **Errore!**
- ▶ Una variabile di tipo struttura può essere dichiarata contestualmente alla definizione della struttura.

Esempio:

<code>struct studente {</code>		<code>struct {</code>
<code> char nome[20];</code>		<code> char nome[20];</code>
<code> long matricola;</code>		<code> long matricola;</code>
<code> struct data ddn;</code>		<code> struct data ddn;</code>
<code>} s1, s2;</code>		<code>} s1, s2;</code>

- ▶ In questo caso si può anche **omettere l'etichetta** di struttura.

Uso di typedef con strutture

- ▶ Attraverso `typedef` è possibile associare un nome ad un tipo definito mediante il costruttore `struct`.

Esempio:

```
struct data { int giorno, mese, anno; };

typedef struct data Data;
```

- ▶ `Data` è un **sinonimo** di `struct data`, che può essere utilizzato nelle dichiarazioni di variabili.

```
Data d1, d2;
Data appelli[10], *pd;
```

Operazioni sulle strutture

- ▶ Si possono assegnare variabili di tipo struttura a variabili **dello stesso tipo** struttura.

Esempio:

```
Data d1, d2;
...
d1 = d2;
```

- ▶ **Non** è possibile invece effettuare il confronto tra due variabili di tipo struttura.

Esempio:

```
struct data d1, d2;
if (d1 == d2) ...
```

Errore!

- ▶ L'equivalenza di tipo tra strutture è **per nome**.

Esempio:

```
struct s1 { int i; };
struct s2 { int i; };
struct s1 a, b;
struct s2 c;
```

a = b; **OK** a e b sono dello stesso tipo

a = c; **Errore!** a e c non sono dello stesso tipo

- ▶ Si può ottenere l'indirizzo di una variabile di tipo struttura tramite l'operatore **&**.
- ▶ Si può rilevare la dimensione di una struttura con **sizeof**.

Esempio: `sizeof(struct data)`

- ▶ Attenzione: **non** è detto che la dimensione di una struttura sia pari alla somma delle dimensioni dei singoli campi.

Accesso ai campi di una struttura

- ▶ I campi di una struttura si comportano come variabili del tipo corrispondente. L'accesso avviene tramite l'**operatore punto**

```
Data oggi;
oggi.giorno = 11; oggi.mese = 5; oggi.anno = 2009;
printf("%d %d %d", oggi.giorno, oggi.mese, oggi.anno);
```

- ▶ Accesso tramite un puntatore alla struttura.

```
Data oggi, *pd;
pd = &oggi;
(*pd).giorno = 11; (*pd).mese = 5; (*pd).anno = 2009;
```

N.B. Ci vogliono le `()` perché `"."` ha priorità più alta di `"*"`.

- ▶ **Operatore freccia**: combina il dereferenzamento e l'accesso al campo della struttura.

```
pd->giorno = 11; pd->mese = 5; pd->anno = 2009;
```

- ▶ **N.B.:** `pd->giorno` è una abbreviazione per `(*pd).giorno`.

Esempio: Accesso al campo di una struttura che è a sua volta campo di un'altra struttura.

```
struct dipendente
{
    Persona datiDip;
    Data dataAssunzione;
    int stipendio;
};
typedef struct dipendente Dipendente;

Dipendente dip, *p;
...
dip.dataAssunzione.giorno = 3;
dip.dataAssunzione.mese = 4;
dip.dataAssunzione.anno = 1997;
...
(p->dataAssunzione).giorno = 5;
(p->stipendio) = (p->stipendio) + 120;
```

Inizializzazione di strutture

- ▶ Può avvenire, come per i vettori, con un elenco di inizializzatori.

Esempio: `Data oggi = { 11, 5, 2009 }`

- ▶ Se ci sono meno inizializzatori di campi della struttura, i campi rimanenti vengono inizializzati a 0 (o al valore speciale `NULL`, se il campo è un puntatore).

Passaggio di parametri di tipo struttura

- ▶ È come per i parametri di tipo semplice:
 - ▶ il passaggio è **per valore** \implies viene fatta una **copia dell'intera struttura** dal parametro attuale a quello formale
 - ▶ è comunque possibile simulare il passaggio per indirizzo attraverso un puntatore

Nota: per passare per valore ad una funzione un vettore (il vettore, non il puntatore al suo primo elemento) è sufficiente racchiuderlo in una struttura.

Esempio:

```
struct dipendente
{
    Persona datiDip;
    Data dataAssunzione;
    int stipendio;
};
typedef struct dipendente Dipendente;

void aumento(Dipendente *p, int percentuale)
{
    int incremento;
    incremento = (p -> stipendio) * percentuale / 100;
    p -> stipendio = p -> stipendio + incremento;
}
```