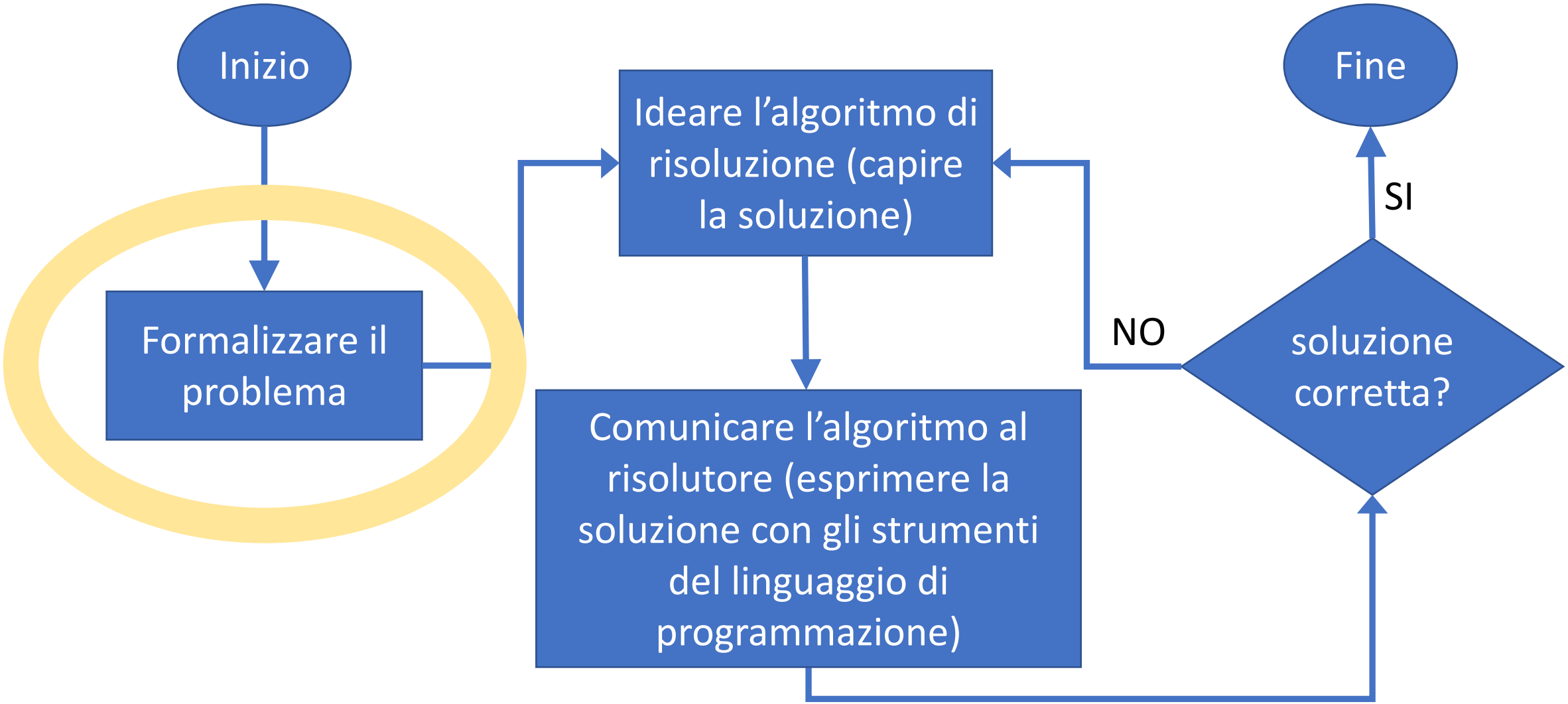
Correttezza del Codice

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Formalizzazione del Problema

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

L’ ”Algoritmo” per Creare Algoritmi



PRE and POST Condizioni

- Fino ad ora abbiamo detto che un programma è corretto se realizza la consegna.
- Cerchiamo di essere più “formali”. Anche se si può dimostrare la correttezza di frammenti di codice o interi programmi, generalmente di dimostra la correttezza di una funzione
  - La Precondizione (PRE) indica ciò che si assume vero prima della chiamata della funzione (di solito sono restrizioni sui valori degli ingressi o relazioni tra essi)
  - La Postcondizione (POST) indica una proprietà che, assumendo vera la PRE, sarà vera dopo che la funzione sarà eseguita.
    - In pratica la POST indica la consegna sottoforma di proprietà (frase che può essere vera/falsa)
- Discutere la correttezza di un programma significa mostrare che, per molti/tutti gli input, il programma calcola l’output che ci aspettiamo
  - Correttezza(Programma, POST)

PRE - POST Esempio


```
/*
        PRE: ho ingrediente 1, ingrediente 2,..., ingrediente N
        POST: nel piatto ho una porzione di pasta cacio e pepe
*/
```

Formalizzazione del Problema

Passi da eseguire per trovare un algoritmo che risolva un problema

1. Scrittura della POST
   - formalizzare il problema in una proprietà chiara aiuta a capirlo meglio
2. Ideazione dell'algoritmo / scrittura della PRE
   - Nel mentre che si pensa l'algoritmo, indicare tutte le precondizioni che si vogliono aggiungere
   - Le precondizioni non devono modificare il problema originale e renderlo troppo semplice!
   - Lo scopo di una PRE è quello di evitare di complicare molto il codice per casi rari o perché il codice non funziona per tutti gli input

```
/*
    PRE x=9
    POST stampa sqrt(x)
*/
funzione rad.quad(x) {
        scrivi 3;
}
```

PRE-POST: Ambiguità

- Le PRE/POST NON sono scritte per il calcolatore, ma per gli utenti
- Basta non essere ambigui per un utente, magari evitando di essere prolissi.
- Le POST vengono espresse con un formalismo “matematico” quando si vuole dimostrare la correttezza del codice corrispondente
  - Esempio:

```
/*
        PRE: x>=0
        POST: restituisce y: y*y=x
*/
funzione radice_quadrata (x) {
        ....
}
```

Design by Contract: Utilizzatore

- La PRE e la POST sono le informazioni che servono a chi dovrà usare il frammento di codice
  - La **POST** ci dice **cosa calcola**
  - La **PRE**, assieme al prototipo, **come (non) dobbiamo invocarla**
- Per chi usa la funzione, l'implementazione è irrelevante se la POST è dimostrata
  - Quando compriamo un'aspirapolvere robot (non studiamo come funziona internamente):
    - PRE: livello batteria>0, sacchetto non pieno, non usare su pavimento di piscina piena
    - POST: pavimento spolverato

Design by Contract: Utilizzatore

- PRE/POST realizzano un contratto tra lo sviluppatore della funzione e chi la utilizzerà
  - lo sviluppatore garantisce la POST all'utilizzatore che invoca il frammento di codice rispettando la PRE
  - se si invoca il frammento di codice ignorando la PRE, lo sviluppatore non garantisce niente
- Scrivere PRE e POST è **fondamentale** per realizzare la "programmazione per contratto" (design by contract)
  - chi usa le vostre funzioni non deve perdere tempo a capire il codice
  - fondamentale sia se si lavora in squadra, ma anche da soli!

PRE - POST Esempio


```
/*
        PRE:
        POST: restituisce il minore tra x, y, z
*/
funzione minimo(x, y, z) {
   return x;
}
```

- Contratto violato dallo sviluppatore, il codice non realizza la POST: minimo(5,4,3)==5 invece di 3

PRE Restrittive

- Siamo liberi di scegliere la PRE come vogliamo, ma dovrebbe essere più generale possibile, affinché la nostra funzione sia utilizzabile in più scenari possibili.

```
/*
        PRE: x==9;
        POST: restituisce la radice quadrata di x
*/
funzione radice_quadrata (float x) {
  return 3;
} // corretta ma non interessante, la funzione può essere applicata solo al numero 9!
```

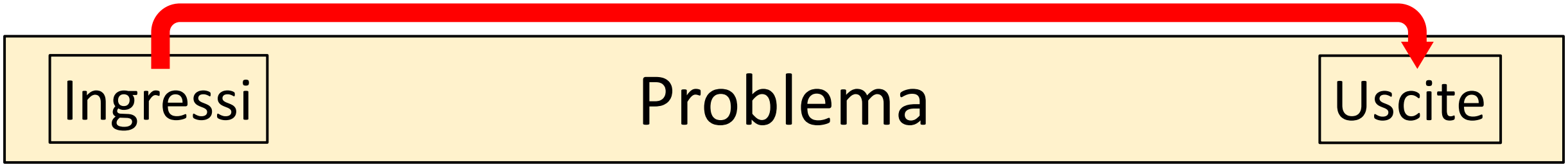
PRE


```
/*
        PRE: x>=0. //ragionevole, per i reali la radice quadrata è definita solo per i numeri positivi
        POST: restituisce la radice quadrata di x
*/
funzione radice_quadrata (x) {
        ....
}

/*      PRE:
        POST: se x<0 restituisce -1 (errore), altrimenti restituisce la radice quadrata di x
*/
funzione radice_quadrata (float x) { //se si vuole essere più generali, aggiornare la POST e il codice
        if (x<0) {return -1;}
        ....
 }
```

Rappresentazione Ingressi e Uscite

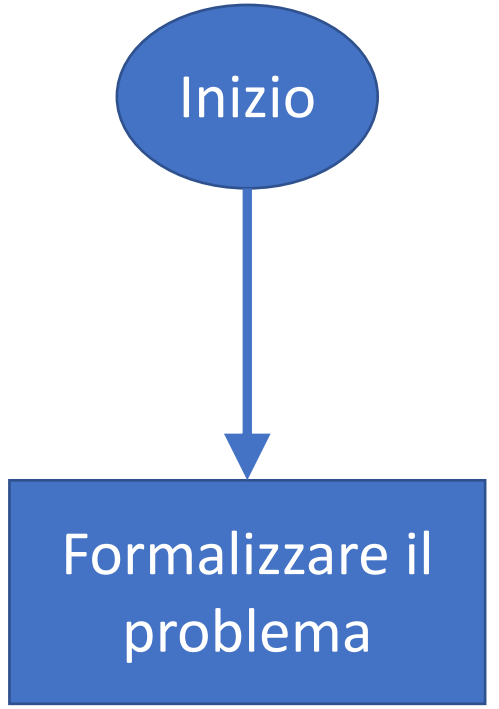
- Un programma stabilisce una relazione tra i dati in ingresso e in quelli in uscita



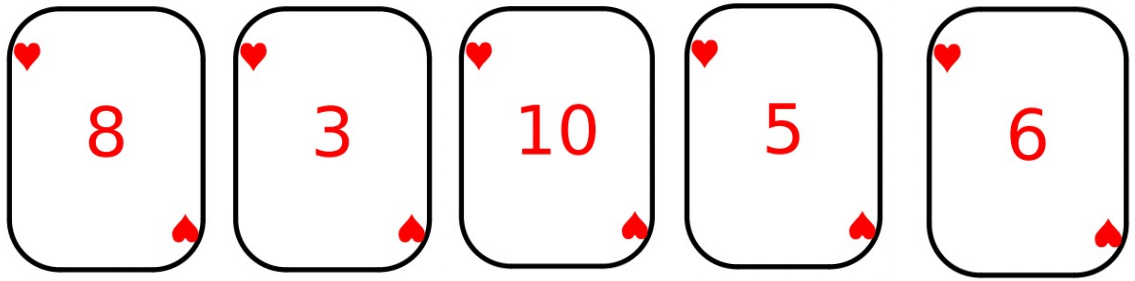
- Primo passo: Rappresentare ingressi e uscite in modo chiaro e non ambiguo

Analisi del Problema: Ordinare Carte da Gioco

- "Data una serie di carte, ordinarle"
- Abbiamo capito il problema? Possiamo fare degli esempi di input/output per
  - Accertarcene o richiedere chiarimenti al committente
  - Scovare già qualche caso particolare
  - Osservarci mentre risolviamo il problema
  - Teniamo traccia di tali input/output per la fase di test

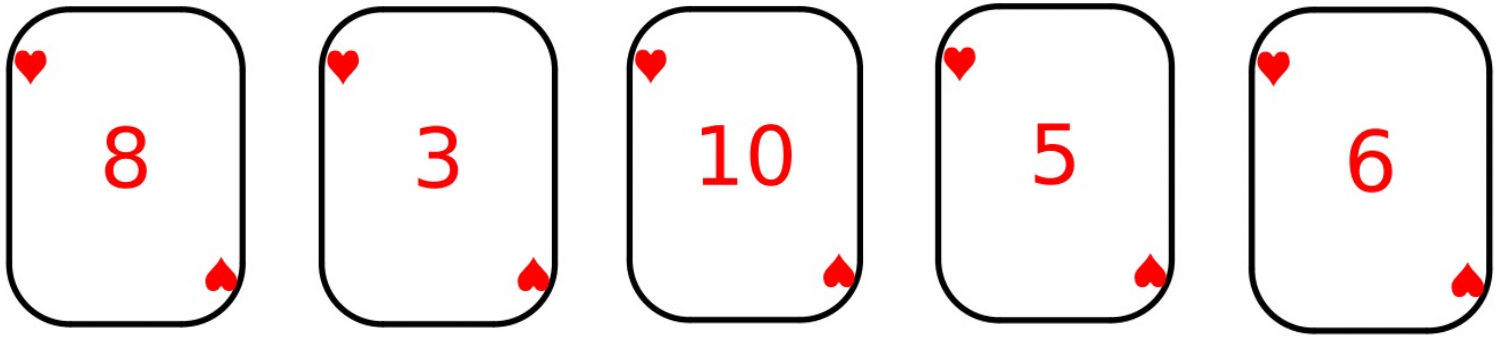


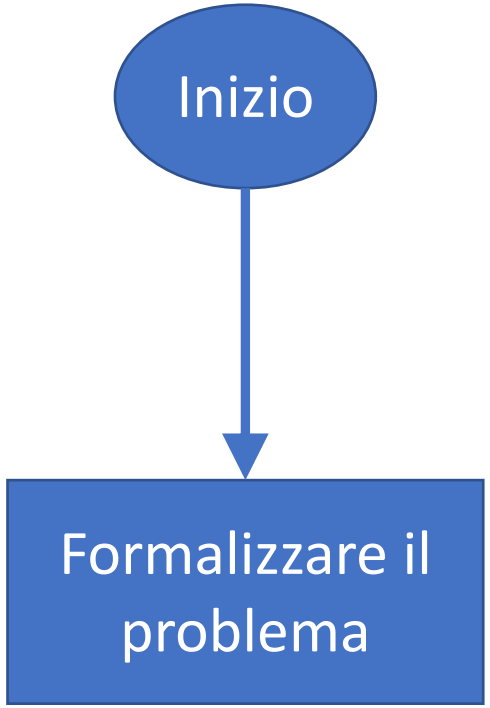
- In questa fase dobbiamo generare la PRE e la POST condizione
- e sperabilmente una bozza di idea per l'algoritmo



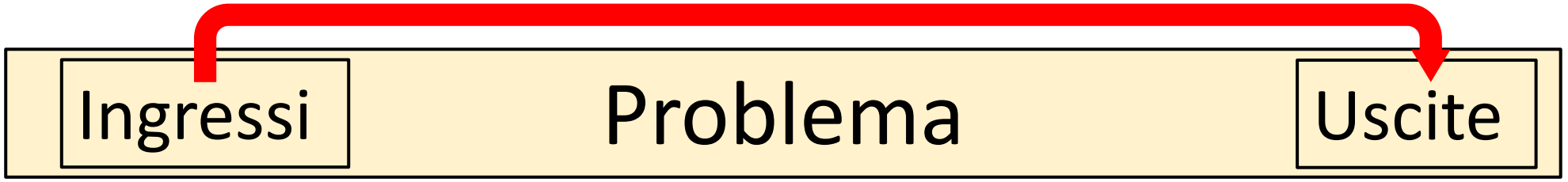
Esempio: Ordinare Carte da Gioco

- Data una serie di carte, ordinarle
- **Abbiamo capito il problema?**
  - Ordinarle in modo crescente o descrescente?
  - L'asso è la carta più bassa o quella più alta?
  - Se ci sono carte dello stesso valore ma di seme diverso?





Esempio: Ordinare Carte da Gioco



PRE: C è una lista di carte

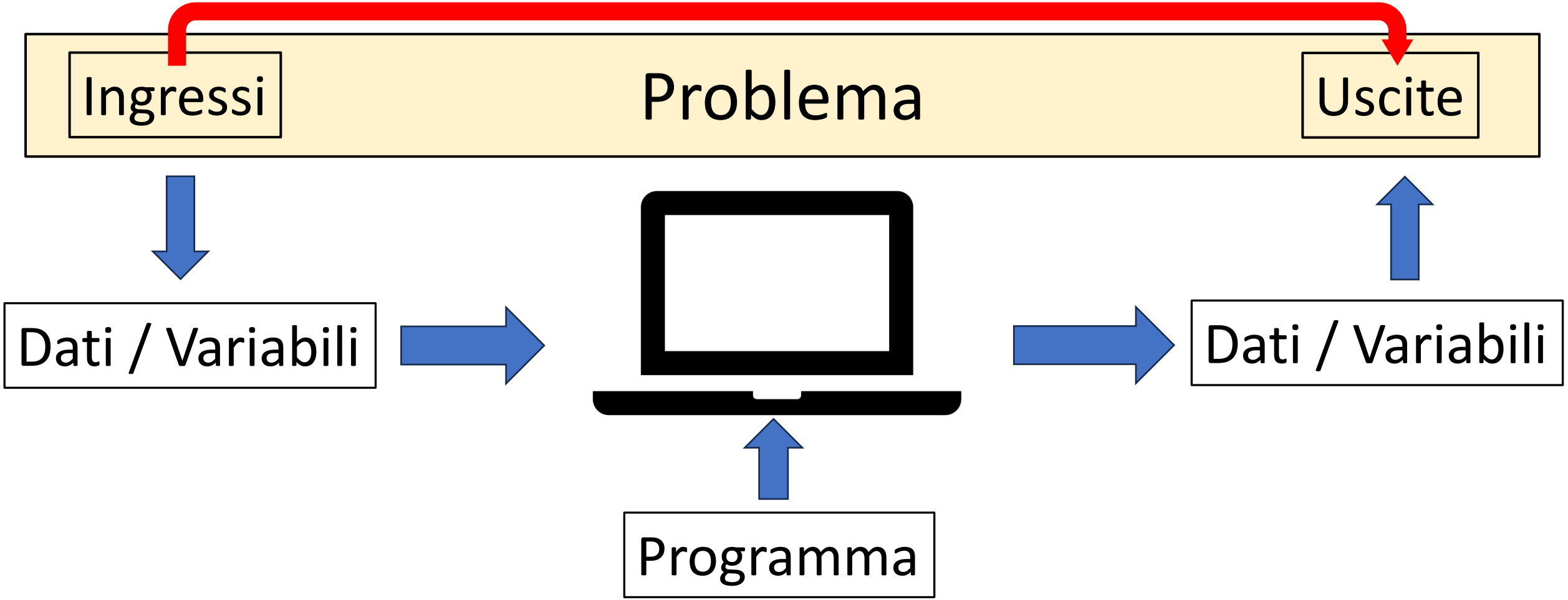
POST: restituisce C. $\forall i\ 1 \leq i < |C|.\ C[i] < C[i+1]$
dove < significa A<2<3<...<10<J<Q<K e

cuori<quadri<fiori<picche

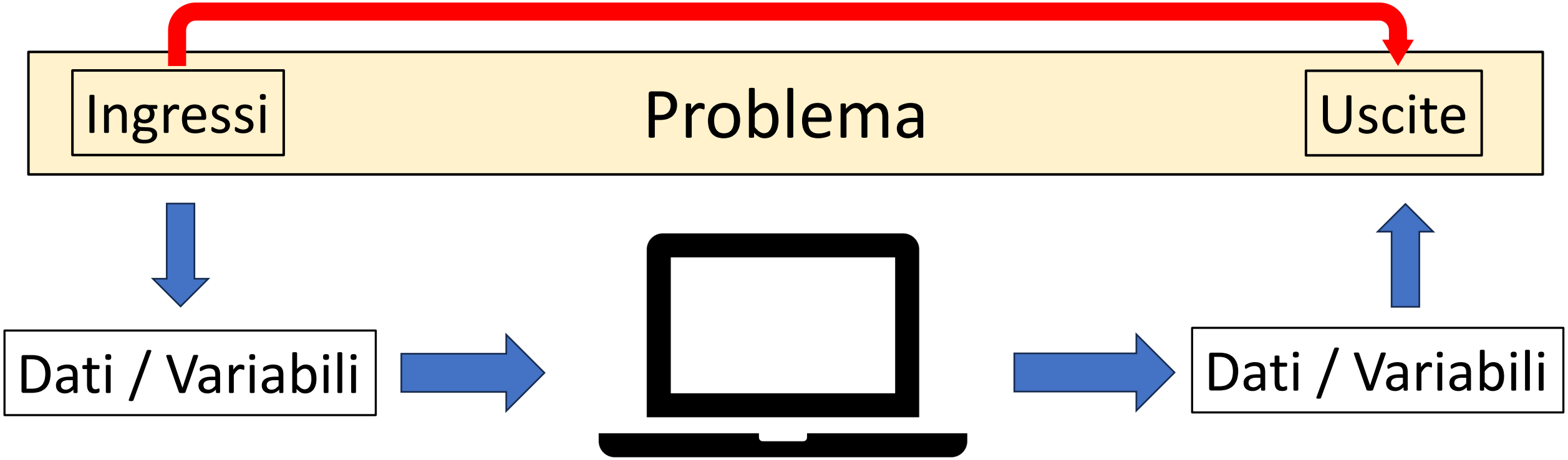
- oppure POST: restituisce C tale che ogni elemento di C è minore di quelli successivi, dove A<2<3<...<10<J<Q<K e cuori<quadri<fiori<picche
- Se vogliamo eseguire l'algoritmo al calcolatore, dobbiamo rappresentare le carte all'interno del calcolatore

Rappresentazione Ingressi e Uscite

- Un programma stabilisce una relazione tra i dati in ingresso e in quelli in uscita

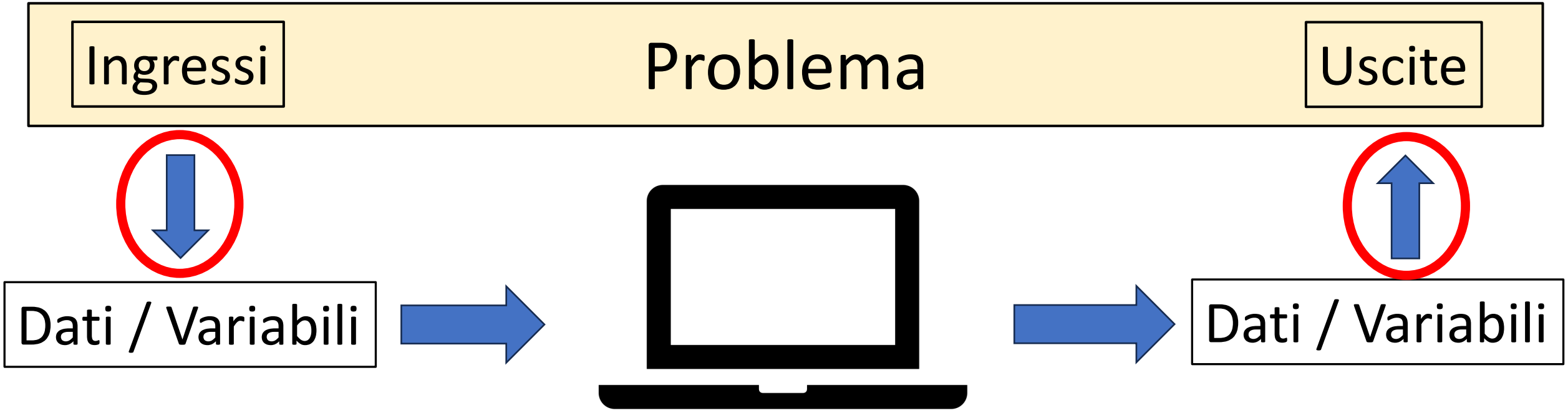


Rappresentazione Ingressi e Uscite



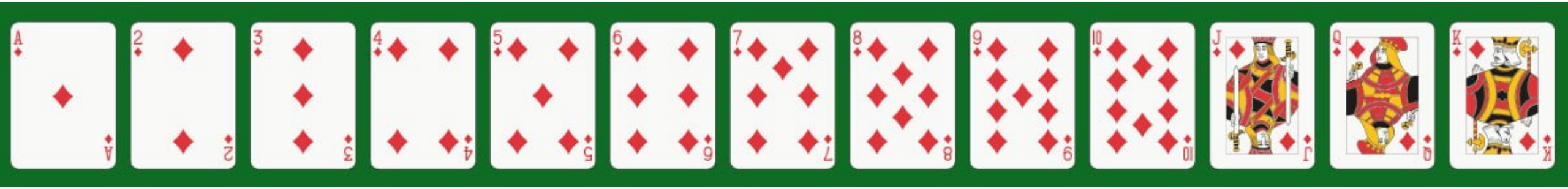
- Primo passo: Rappresentare ingressi e uscite in modo chiaro e non ambiguo tramite variabili o valori costanti

Rappresentazione Ingressi e Uscite

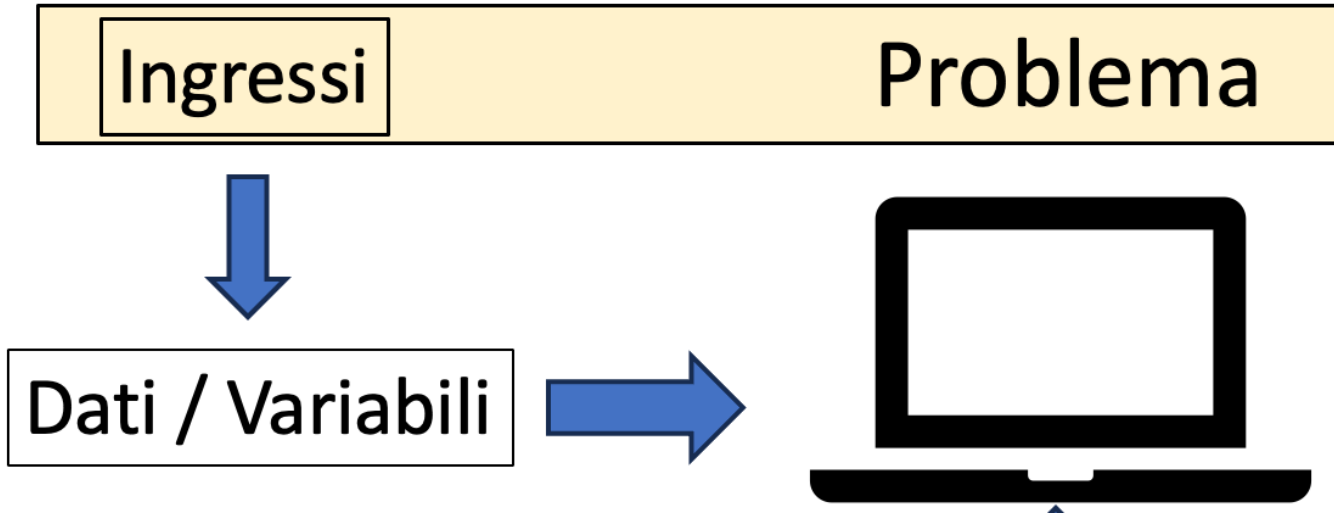


- Primo passo: Rappresentare ingressi e uscite in modo chiaro e non ambiguo tramite variabili o valori costanti

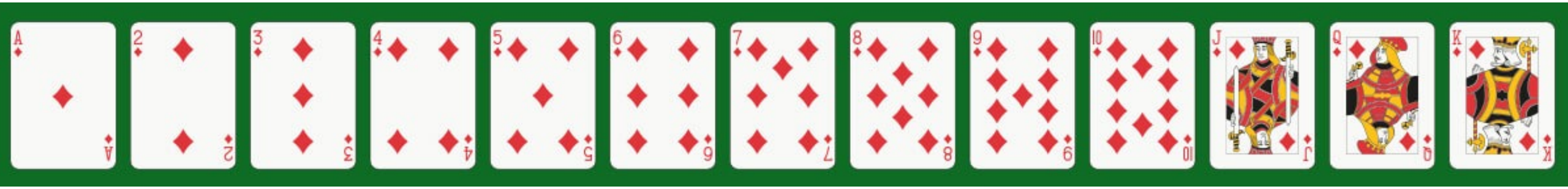
Esempio: Ordinare Carte stesso Seme



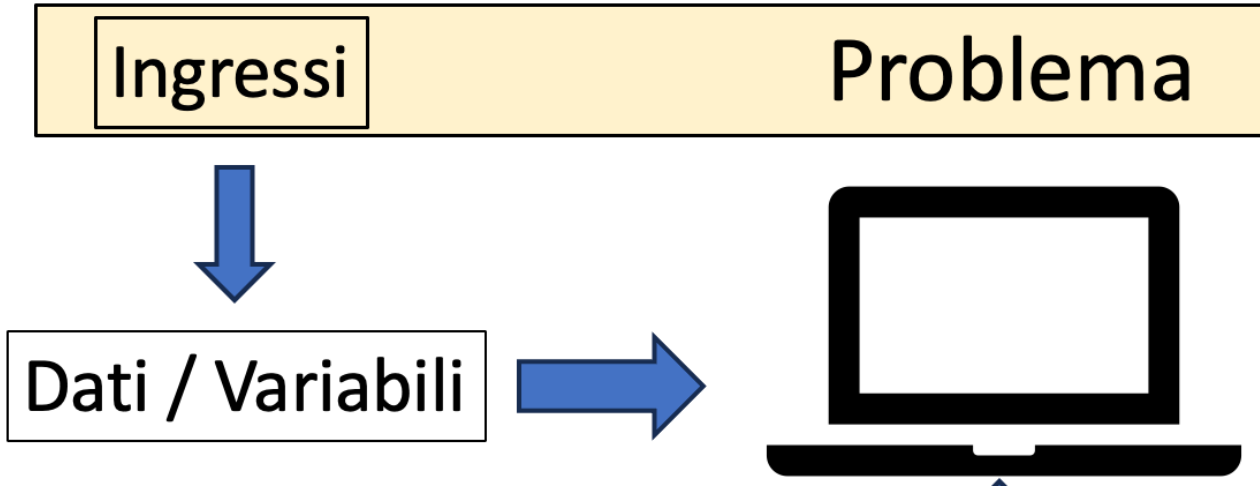
- Problema: ordinare una serie di carte dello stesso seme
- Passo 1: Come rappresentiamo le carte?



Esempio: Ordinare Carte stesso Seme

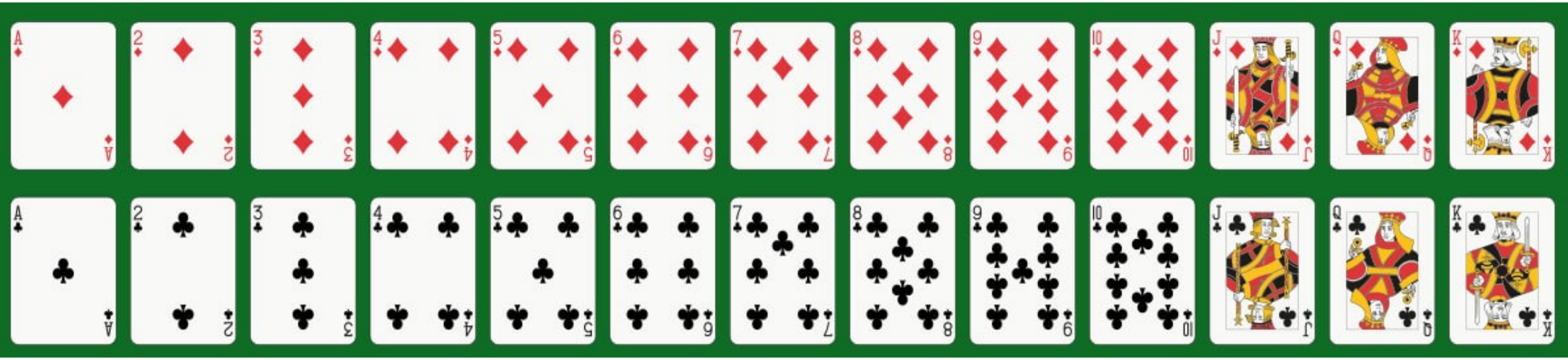


- Passo 1: Come rappresentiamo le carte?
- Usiamo una lista di numeri, uno per ogni carta da ordinare



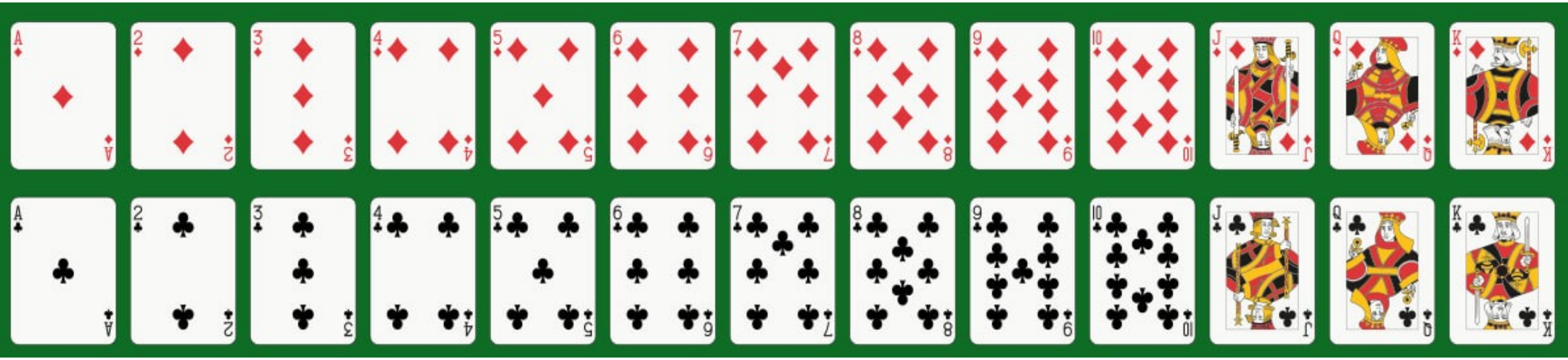
- Ogni valore della lista rappresenta una carta secondo questa trasformazione: A=1, 2=2, 3=3,..., 10=10, J=11, Q=12, K=13
- Es. L={2,1,12} →lista di tre carte: 2, asso, Q

Esempio: Ordinare Carte



- Problema: ordinare una serie di carte (che possono essere di semi diversi)
- Passo 1: Come rappresentiamo le carte?

Esempio: Ordinare Carte (Idea 1)

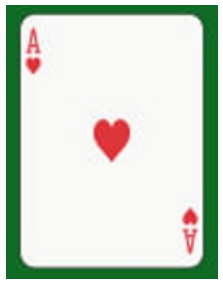
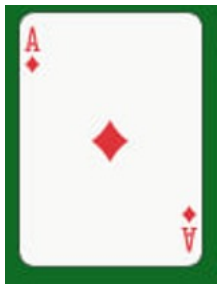
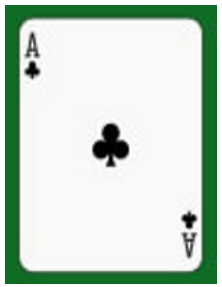


- Usiamo una lista di numeri, uno per ogni carta da ordinare
- Ogni valore della lista rappresenta una carta secondo questa trasformazione: Asso = 1, 2=2, 3=3,...,10=10, J=11, Q=12, K=13
- Usiamo una seconda lista con tant elementi quante le carte da ordinare
- L'i-esimo valore della lista rappresenta il seme della i-esima carta da ordinare

♥ = 1, ♦ = 2, ♣ = 3, ♠ = 4

Esempio: Ordinare Carte (Idea 2)

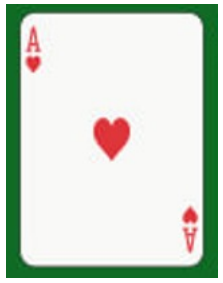
- Usiamo una lista di numeri, uno per ogni carta da ordinare
- Ogni valore della lista rappresenta una carta secondo questa trasformazione:

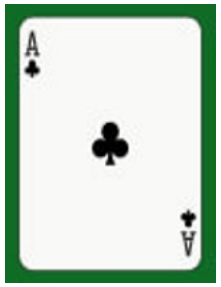
 =1  =2  =3  =4  =5  =6

 =7  =8 ...  =?

Esempio: Ordinare Carte (Idea 2)

- Usiamo una lista di numeri, uno per ogni carta da ordinare
- Ogni valore della lista rappresenta una carta secondo questa trasformazione:


=1

=2

=3

=4

=5

=6


=7

=8
...

=52

- La rappresentazione migliore dipende dal problema che vogliamo risolvere

Esempio: Ordinare Carte da Gioco

Funzione ordina(C) {

/*

PRE: $\forall i\ 0 \leq i < |C|.\ 1 \leq C[i] \leq 13$ //un solo seme

POST: restituisce

C. $\forall i\ 1 \leq i < |C|.\ C[i] < C[i+1]$

A=1,2=2,…,10=10,J=11,Q=12,K=13

*/

…. // inizio codice

}

- Si assume che le carte siano di un solo seme (per nostra semplicità, ma è facile estendere le condizioni al caso di più semi)

Esempio: Ordinare Carte da Gioco

Funzione ordina(C) {

/*

PRE: -

POST: restituisce

C. $\forall i\ 1 \leq i < |C|.\ C[i] < C[i+1]$

A=1,2=2,…,10=10,J=11,Q=12,K=13

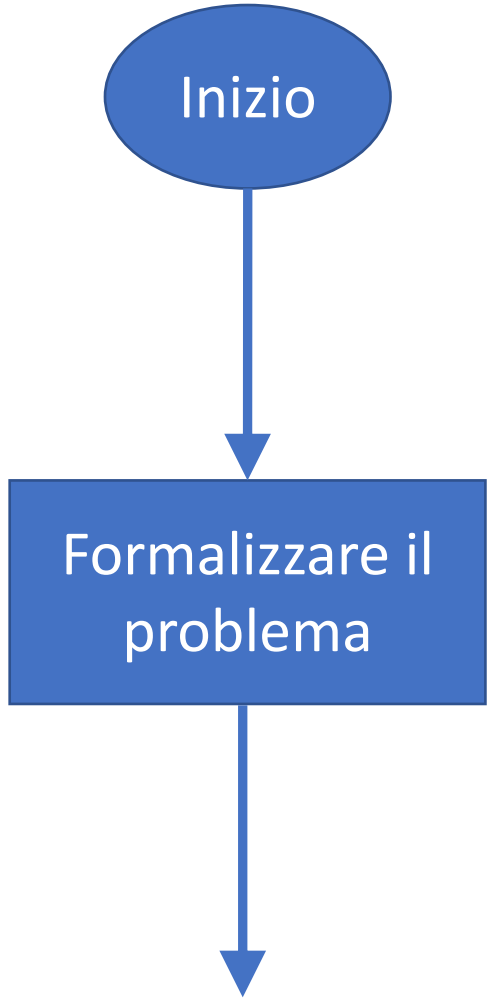
Errore se $\exists i\ C[i] < 1 \text{ OR } C[i] > 13$

*/

…. // inizio codice

}

- Si assume che le carte siano di un solo seme (per semplicità nostra, ma è facile estendere le condizioni al caso di più semi)



PRE - POST Esempio

- Se in C vogliamo assicurarci che una PRE sia rispettata all'interno del codice, possiamo usare la funzione assert, che genera un errore se la condizione al suo interno è falsa e termina il programma (altrimenti prosegue con l'esecuzione)

```
#include <assert.h>
/*
        PRE: x>=0
        POST: restituisce la radice quadrata di x
*/
int radice_quadrata (float x) {

        assert(x>=0) ;
        ....
 }
```

Utilità della PRE: esempio


```
/* PRE: A ha dim elementi; POST: restituisce min(A) se dim>0, altrimenti? */
int min_array(int A[], int dim) {

    int min=A[0];
    for(int i=0; i<dim; i+=1) {
        if(A[i]<min)
            min=A[i];
    }
    return min;
}
```

Utilità della PRE: soluzione 1


```
/* PRE: A ha dim>0 elementi; POST: restituisce min(A) */
int min_array(int A[], int dim) {

    int min=A[0];
    for(int i=0; i<dim; i+=1) {
        if(A[i]<min)
            min=A[i];
    }
    return min;
}
```

Utilità della PRE: Soluzione 2


```
/* PRE:   se dim>0, A ha almeno dim elementi;
   POST: restituisce 0 se dim=0 (min non modificato); 1 altrimenti,
                min=min(A[0]..A[dim-1]);  */
int min_array(int A[], int dim, int *min) {
   if (dim<=0) return 0;
   *min=A[0];
   for(int i=0; i<dim; i+=1)
      if(A[i] < *min)
         *min=A[i];
   return 1;
}
```

Verifica del Codice: Testing

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Tipi di Errore

- Sintassi: insieme di regole che descrive come si possano costruire “frasi” (programmi) validi
- Errore di sintassi: il compilatore riesce a tradurre il nostro codice se e solo se rispetta la sintassi del linguaggio
- In aggiunta il compilatore analizza il codice per trovare istruzioni possibilmente problematiche anche se corrette sintatticamente, i warning
  - Es. una variabile utilizzata prima di essere assegnata
    - int x, y; y=x+2; // non si sa quanto valga x!
    - Es. int x=3; if (x=2) {printf(“x=5”);} // l’assegnamento ha valore di verità vero, quindi corretto ma forse si voleva evitare di avere un effetto collaterale?
    - Diversi compilatori decidono quali warning visualizzare (troppi warning in programmi lunghi potrebbero rendere scomodo trovare gli errori)
    - gcc **-Wall** programma.c // visualizza tutti i warning

Tipi di Errore

- Errore semantico: il programma non rispetta la POST

- Discutere la correttezza di un programma significa mostrare che, per ogni input, se la PRE è vera, la POST è anch'essa verificata, oppure fornire un controesempio

- Quindi per mostrare che un programma non è corretto rispetto alla POST, basta fornire un esempio di input/output che non sia quello aspettato

Discutere la Correttezza


```
/*  PRE:
    POST: stampa Il valore minimo tra le 3 variabili x,y,z
*/
void minimo(int x, int y, int z) {
if ( ( x <= y) && (x <= z) )
       printf("%d", x);

   if ( ( y <= x) && (y <= z) )
       printf("%d", y);

   if ( ( z <= x) && (z <= y) )
       printf("%d", z);
}
```

Discutere la Correttezza


```
/*  PRE:
    POST: stampa un numero che è Il valore
              minimo tra le 3 variabili
*/
void minimo(int x, int y, int z) {
if ( ( x <= y) && (x <= z) )
      printf("%d", x);

   if ( ( y <= x) && (y <= z) )
      printf("%d", y);

   if ( ( z <= x) && (z <= y) )
      printf("%d", z);
}
```

Input: minimo(3,3,3)

Output ottenuto: 333

Dalla POST mi aspettavo 3 come output* → il programma non è corretto

*non sbagliate a calcolare l'output aspettato!

Esercizio: Discutere la Correttezza


```
#include <stdio.h>
int main() {
        int a=1,b=2,c=3;
        if (a>=b) {
                if(a>=c) {
                        printf("%d è il massimo\n", a);
                } else {
                        printf("%d è il massimo\n", b);
                }
        } else if (b>=c) {
                printf("%d è il massimo\n", b);
        } else {
                 printf("%d è il massimo\n", c);
        }
}
```

Esercizio: Discutere la Correttezza


```
#include <stdio.h>
int main() {
        int a=1,b=2,c=3;
        if (a>=b) {
                if(a>=c) {
                        printf("%d è il massimo\n", a);
                } else {
                        printf("%d è il massimo\n", b);
                }
        } else if (b>=c) {
                printf("%d è il massimo\n", b);
        } else {
                 printf("%d è il massimo\n", c);
        }
}
```

POST: stampa "x è il massimo" dove x=max(a,b,c)

Adesso che abbiamo la POST ha senso chiedersi se la funzione è corretta

Testing

- Un'unità è un frammento di codice che può essere testato (di solito una funzione)
- unit test è una coppia (input, output) che sappiamo essere corretta (calcolata con un altro software?)
  - se l'output del nostro programma corrisponde con quello aspettato, il test è passato!
  - I test si raggruppano in collezioni: suite di test
- Più test si fanno, più si è certi della correttezza del codice
  - vasta copertura: i nostri test devono controllare più codice possibile (tutti i rami di un if-else). Pensare a casi particolari, ecc...
  - A meno che non possiate testare tutte le coppie (input, output), i test identificano i problemi, non ci dimostrano che il codice è corretto, ma ci dimostrano se il codice non è corretto
  - Attenti che il test stesso può essere sbagliato!

Esercizio

- Dato un numero intero, scrivere una funzione che calcoli la frequenza di ciascuna cifra del numero, che venga poi stampata nel main.

Es. per 1213 si stamperà nel main:

frequenza 0 = 0

frequenza 1 = 2

frequenza 2 = 1

frequenza 3 = 1

...

frequenza 9 = 0

Soluzione da Analizzare


```
void contaCifre(int n, int frequenza[]) {


    while(n>0) {
        frequenza[n % 10] += 1;
        n = n / 10;
    }
}
```

Funzioni di Test


```
void test_contaCifre(void) {

    int frequenza[10];
    int res[10] = {0};

    res[1] = 1; res[2]=2; res[3]=1;
    contaCifre(1223, frequenza);
    assert(confronta_array(frequenza, res, 10)==1);

    res[1] = 1; res[2]=2; res[3]=1;
    contaCifre(-1223, frequenza);
    assert(confronta_array(frequenza, res, 10)==1);
    printf("Test funzione contaCifre. tutti i test passati\n");
}
```

Funzioni di test

- forniscono una documentazione dei test che sono stati fatti
- permettono di replicare velocemente tutti i test se cambiamo il codice

Funzioni di Test


```
void test_contaCifre(void) {

    int frequenza[10];
    int res[10] = {0};

    res[1] = 1; res[2]=2; res[3]=1;
    contaCifre(1223, frequenza);
    assert(confronta_array(frequenza, res, 10)==1);

    res[1] = 1; res[2]=2; res[3]=1;
    contaCifre(-1223, frequenza);
    assert(confronta_array(frequenza, res, 10)==1);
    printf("Test funzione contaCifre. tutti i test passati\n");
}
```

Test Driven Development

- Si scrivono iterativamente i test e poi il codice che ha lo scopo di superare solamente quei test

Funzioni di Test

- Forniscono una documentazione dei test che sono stati fatti
- Permettono di replicare velocemente tutti i test se cambiamo il codice
- Spesso le funzioni di test vengono aggiunte in file separati, compilati in eseguibili appositi che vengono invocati per eseguire test su tutte le funzioni assieme → test suite
- Quando si testano output reali, permettere una certa tolleranza dell'errore

Dimostrazioni di Correttezza

- “A meno che non possiate testare tutte le coppie (input, output), i test identificano i problemi, non ci dimostrano che il codice è corretto, ma ci dimostrano se il codice non è corretto”
- Teoricamente Il modo migliore per assicurarsi che un programma fa sempre ciò che vogliamo (realizza sempre la POST) è dimostrarlo (come se fosse un teorema matematico)
- Si assume vera la PRE
- Si calcolano i cambiamenti allo stato (memoria + output) apportati dal frammento di codice che stiamo esaminando (che hanno effetto sulla POST)
- Si mostra che ogni possibile stato del calcolatore realizza la POST
  - Per ogni possibile punto di uscita del codice
- Le dimostrazioni di correttezza sono a volte complesse, specialmente al crescere della complessità del codice, sono sostituite da batterie di test, ma...

Discutere la Correttezza

- Come dimostriamo che la POST sia verificata?
- Esistono formalismi (triple di Hoare) che indicano come lo stato del calcolatore (memoria + video), dato un generico input, cambia dopo l'esecuzione di ciascun comando
- Ogni comando (assegnamento, if, while) ha associata una tripla di Hoare che mostra come il comando cambia lo stato del calcolatore
- assumiamo vera la PRE; applichiamo le proprietà delle triple di Hoare per ciascun comando nel nostro frammento di codice, costruendo ulteriori asserzioni (che descrivono lo stato del calcolatore)
- Per ogni punto di uscita della funzione, mostriamo che possiamo dedurre la verità della POST dalle asserzioni che abbiamo accumulato sullo stato

PRE - POST Esempio


```
void minimo(int x, int y, int z) {
printf("Il minore dei tre valori è ");
   if (x < y) { // so che x<y
      if (x < z) { // x < y e x < z
         printf("%d\n", x);
      } else { // z <= x < y
         printf("%d\n", z);
      }
   } else { // y <= x
      if (y < z) { // y < z e y <= x
         printf("%d\n", y);
      } else { // z <= y <= x
         printf("%d\n", z);
      }
   }
```

```
/*
   PRE:
   POST: stampa "Il minore dei tre
            valori è a\n" dove a è il
            valore minore tra x,y,z
*/
```

Per l'IF sappiamo che all'interno dei due rami la condizione è vera o falsa

Dimostrazioni di Correttezza

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Correttezza Funzioni Ricorsive

- La correttezza delle funzioni ricorsive si basa su principio di induzione matematica

$$\frac{P(0) \qquad P(n) \Rightarrow P(n+1))}{\forall n \,.\, P(n)}$$

- Per dimostrare che una proprietà vale per tutti i numeri naturali, dimostriamo che
- vale per 0 (o il più piccolo valore dell'insieme che stiamo considerando)
- se assumiamo che vale per n (ipotesi induttiva), dimostriamo che vale per n+1

Dimostrazioni per Induzione: Esempio

- P(n) ⇔ 1+2+...+n = n(n+1)/2, per n>0
- P(1) ⇔ 1 = 1(1+1)/2 = 1
- P(n) → P(n+1): assumendo P(n) vera riesco a dimostrare P(n+1)?

P(n) = 1+2+..+n= n(n+1)/2

$$(1+2+\cdots+n)+n+1=\frac{n(n+1)}{2}+n+1=\frac{n(n+1)}{2}+\frac{2(n+1)}{2}$$
$$=\frac{(n+1)(n+2)}{2}=P(n+1)$$

Dimostrazioni per Induzione: Esempio

- P(n) ⇔ 1+2+...+n = n(n+1)/2, per n>0
- P(1) ⇔ 1 = 1(1+1)/2 = 1
- P(n) → P(n+1):

- Ho dimostrato che P(1) vale,
- ho mostrato che se vale P(n), per un n generico, allora vale anche P(n+1), quindi
- da P(1) mostriamo che vale P(2),
- da P(2) mostriamo che vale P(3)... possiamo andare avanti fino a
- raggiungere ogni n, quindi P(n) vale per ogni n!

Correttezza Funzioni Ricorsive

- P(1): la POST vale per i casi base della ricorsione?
- P(n)→P(n+1): assumendo che la POST sia vera per n (Ipotesi Induttiva), riusciamo a dimostrare che P(n+1) è vera, ovvero la POST per n+1 è vera?
- Se la risposta è si per entrambe le domande, abbiamo dimostrato la correttezza di una funzione ricorsiva!

Esempio: somma 1..N


```
/*
    PRE: n>=0;
    POST: restituisce 1+2+...+n
*/
int somma1n(int n) {

   if(n==0)
     return 0;
   else
     return n+somma1n(n-1);
}
```

- complessità espressa in termini di n (ad ogni iterazione si risolve un problema più semplice: n-1)
- P(0) = 0 → ok!
- P(n-1)→P(n)?
- Ipotesi induttiva: P(n-1) vera, ovvero P(n-1)=somma1n(n-1)=1+2+...+n-1
- P(n) = n+P(n-1)=n+1+2+...+n-1=1+2+...+n-1+n = P(n) →ok!

Esempio: potenza


```
/*
  PRE: esp>=0, base!=0;
  POST: restituisce base^esp
*/

int potenza(int base, int esp) {
  if(esp==0)
    return 1;
  return base*potenza(base, esp-1);
}
```

La complessità è espressa in termini di esp

Caso base: ?

Caso ricorsivo: ?

Esempio di Dimostrazione per Induzione


```
/*
  PRE: esp>=0, base!=0;
  POST: restituisce base^esp
*/

int potenza(int base, int esp) {
  if(esp==0)
    return 1;
  return base*potenza(base, esp-1);
}
```

Caso base: P(esp=0) → 1=base^0
→ok

Esempio di Dimostrazione per Induzione


```
/*
  PRE: esp>=0, base!=0;
  POST: restituisce base^esp
*/

int potenza(int base, int esp) {
  if(esp==0)
    return 1;
  return base*potenza(base, esp-1);
}
```

Caso base: P(esp=0) → 1=base^0 →ok

Caso ricorsivo: P(esp-1)→P(esp)?

ipotesi induttiva: P(esp-1) vera, ovvero potenza(base, esp-1)=base^(esp-1)

base*potenza(base, esp-1) =

= base*base^(esp-1) = base^esp = P(esp) ok!

Esempio: Trova Carattere in Stringa


```
/*
  PRE: s è un puntatore a stringa
  POST: restituisce 1 se x è presente in s
                    0 altrimenti
*/
int trova_char(int *s, char x) {
  if (*s=='\0')
    return 0;
  else
    return (*s==x) || trova_char(s+1, x);
}
```

- Complessità del problema in termini della lunghezza della stringa s
- P(0): l'unica stringa di lunghezza 0 è '\0', per cui x non compare in '\0'
- P(n-1)→P(n)

se trova_char(s[1..n])==1 allora x è presente in s[1..n] → viene restituito 1 →P(n) corretta

se trova_char(s[1..n])==0 viene restituito il risultato *s==x, per cui P(n) è ancora corretta

Debugging

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Motivazione

- Quando ci aspettiamo un determinato risultato da un frammento di programma, ma il risultato che otteniamo è diverso, e non capiamo perché
- vorremmo poter stampare lo stato del programma ad ogni passo finché non si individua il problema

```
5       int i, num, j;
6       printf ("Inserire un numero: ");
7       scanf ("%d", &num );
8
9       for (i=1; i<num; i++)
10          j=j*i;
11
12      printf("Il fattoriale di %d è %d\n",num,j);
13  }
```

Input: 3

Output atteso: 6

Output ottenuto: ... (non 6)

Debugger

- Un debugger è un'applicazione che esegue un nostro programma e ci permette di
  - metterlo in pausa e riprendere l'esecuzione (in punti particolari o anche istruzione per istruzione)
  - visualizzare il contenuto di variabili e informazioni sullo stato di esecuzione
  - utilissimo quando il programma non fa ciò che ci aspettiamo
- Esempi
  - gdb (gnu debugger)
  - VSCode extension (da installare sulle macchine del laboratorio)
- per poter usare il debugger su un nostro programma, dobbiamo compilarlo con l'opzione –g (preserva i nomi delle variabili ed altre info per il debugger)
- gcc –o hello –g hello.c

Debugger

- Supponiamo di avere il seguente programma di esempio

```
1  #include <stdio.h>
2  int stampa(int n) {
3          for(;n>0;n-=1) printf("ciao mondo\n");
4          return 11*n;
5  }
6  int main() {
7          int x=2; int a[3]={4,5,6};
8          stampa(x);
9  }
10
```

- compilato con gcc –o hello –g hello.c

Debugger - Bignami

Comando	abbr.	esempio	cosa fa	note
gdb		gdb hello	esegue il debugger	gdb --args hello arg1 arg2 (se con argomenti da linea di comando)
list	l	list 1,100	mostra da riga 1 a 100 del programma	100 può essere > di numero righe di hello.c
info source	i source	i source	mostra info sui sorgenti	dice anche quante righe ha il nostro codice
break	b	b 9; b stampa	interrompe l'esecuzione a quella riga	b filename.c:10 (se progetto con più file)
run	r	run	esegue il programma	se non ci sono breakpoint, esegue fino alla fine
next	n	n	esegue la prossima istruzione	
continue	c	c	riprende l'esecuzione	se non ci sono breakpoint, esegue fino alla fine
display		display x	mostra automaticamente il valore di x	esecuzione già arrivata alla dichiarazione di x
info locals	i lo	info locals	stampa il contenuto di variabili locali	(dichiarate fino a quel momento)
print	p	print x	stampa valore di espressione	print *a@2->{4,5}; print sizeof(a)->12 (si possono usare vari comandi come argomento)
whatis		whatis x;whatis a	informazioni sulla variabile (tipo)	vedere anche ptype
quit	q	q	esce dal debugger	può chiedere conferma

vedi https://www.youtube.com/watch?v=bWH-nL7v5F4 per un esempio

Debugger - Bignami

Comando	abbr.	esempio	cosa fa	note
watch		watch x	interrompe l'esecuzione ad ogni cambiamento del valore della var. x	(x deve essere già dichiarata)
start		start	esegue il programma fermandosi alla	riga "int main() {"
layout next			mostra il codice e la riga corrente	b 1; layout next; run
refresh	ctrl-l		ridisegna la schermata	utile per layout next
info b	info b	info breakpoints	mostra la lista dei breakpoint	info ha altri usi (digitate info solamente)
undisplay		undisplay 2	smette di monitorare la i-esima variabile aggiunta con display	display a; display x; undisplay 2 (non stampa più x); usare info display per conoscere indici
delete		delete 1	rimuove l'i-esimo breakpoint aggiunto	i non è la riga del breakpoint, ma il valore Num in info breakpoint
target record-full			si segna tutte le info per tornare indietro nell'esecuzione del codice	
reverse-next	rn		torna indietro di un'istruzione	
set var x		set var x=15	cambia il valore della variabile x	

Debugger - Funzioni

Comando	abbr.	esempio	cosa fa	note
break		b stampa	interrompe l'esecuzione all'inizio della funzione stampa	
up				
down				
backtrace			stampa tutto il call stack	
step		step	se l'esecuzione è ad una funzione, passa ad eseguire i comandi della funzione	next avrebbe considerato la funzione un unico comando ed eseguita tutta in un colpo
finish			va fino alla fine della funzione e	mostra il valore che restituisce la funzione

Debugger - Bignami

- gcc –o hello –g hello.c; gdb hello
  - se abbiamo argomenti da linea di comando: gdb --args hello arg1 arg2
- layout next – mostra il codice (b 1; layout next; run)
- start – esegue il programma fermandosi all'inizio della funzione main()
- break [POINT (line number, function name,...)] - dove si interrompe l'esecuzione (b)
- next– esegue la prossima istruzione (n per brevità)
- continue – continua l'esecuzione fino al prossimo breakpoint (c per brevità)
- print VAR; print *arr@len – stampa il valore di VAR o len elementi dell'array arr (si può usare print con un'espressione o un comando, es. print sizeof(arr) stampa la dimesione di arr
- display [VAR] - aggiunge VAR alle variabili di cui mostrare automaticamente il valore
- info locals – stampa il contenuto di tutte le variabili locali dichiarate fino a quella riga
- refresh – aggiorna il video, utile se si è utilizzato layout next (ctrl l)
- watch VAR – si ferma quando il valore di una variabile cambia
- whatis VAR – fornisce informazioni sulla variabile VAR (vedere anche ptype)
- quit – esce dal debugger (q)

Valutazione di Programmi

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Criteri di Valutazione

Un programma può essere valutato rispetto ai seguenti criteri (quando applicabili)

- Correttezza (le funzioni significative devono essere commentate con PRE POST): si fornisce evidenza che il codice realizzi le POST
- Efficienza: evitare codice inutile, cercare di trovare l'algoritmo più efficiente per risolvere i (sotto)problemi
- Organizzazione del codice: divisione logica del codice in funzioni. Evitare di risolvere una seconda volta problemi già risolti
- Stile: il codice deve essere leggibile dai vostri colleghi (evitare istruzioni non spiegate da me), commentare i frammenti di codice che non siano ovvi, usare nomi significativi per le variabili

Criteri di Valutazione

Correttezza

- le funzioni significative devono essere commentate con PRE POST
- il codice compila
- Se il codice non è corretto, si
- si fornisce evidenza che il codice realizzi le POST
- Efficienza: evitare codice inutile, cercare di trovare l'algoritmo più efficiente per risolvere i (sotto)problemi
- Organizzazione del codice: divisione logica del codice in funzioni. Evitare di risolvere una seconda volta problemi già risolti
- Stile: il codice deve essere leggibile dai vostri colleghi (evitare istruzioni non spiegate da me), commentare i frammenti di codice che non siano ovvi, usare nomi significativi per le variabili

Efficienza

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Efficienza

- L'efficienza di un algoritmo non viene calcolata cronometrando il tempo di esecuzione
  - perché dipendente dalla macchina
- Si usa una misura che dipende dal numero di istruzioni eseguite
- Ci interessa qual è la dipendenza rispetto all'input
  - Qual è il fattore dell'input che fa variare maggiormente il numero di istruzioni eseguite?
  - Es. la funzione che descrive il numero di istruzioni eseguite cresce in modo costante, linearmente, in modo quadratico rispetto alla dimensione dell'input?

Efficienza

- Ci interessa stimare quanto ci metterà per eseguire il programma con un input “molto grande”. Es. int X[10000000]; ordina_array(X)
  - Della funzione che conta il numero di istruzioni eseguite ci interessa il termine più grande
  - Es. se vengono eseguite 3n^2 + 50n+99999 istruzioni, il termine che domina è n^2
  - Indichiamo quindi il numero di istruzioni con O(n^2) (big-O notation), tralasciando gli altri termini (per compattezza e perché irrilevanti per grandi input)
- Un algoritmo è significativamente più efficiente di un altro se la sua complessità, utilizzando la notazione O() è minore

$$O(1) < O(n) < O(n^2)$$

- O(1) = complessità costante (numero di istruzioni costante che non dipende dall’input)

Cenni di Complessità Computazionale

- Problema: creare una funzione che, dato un array in input, verifichi se il primo elemento di un array è uguale al secondo

```
//PRE A ha dim>1 elementi
int f(int *A, int dim) {
   return A[0]==A[1];
}
```

- Il tempo di esecuzione è indipendente dalla dimensione dell'input, dim: O(1)

- Il tempo di esecuzione è costante: O(1)
  - O(1) non significa che viene eseguita esattamente una istruzione,
  - O(1) significa che il tempo di esecuzione è costante, ovvero che esiste una funzione costante che è un limite superiore al tempo di esecuzione
    - in altre parole che la complessità non cresce linearmente con la dimensione dell'input

Cenni di Complessità Computazionale

- Problema 2: creare una funzione che, dato un array in input, verifichi se il primo elemento di un array è uguale al secondo, terzo e quarto

- Tempo di esecuzione: O(1)

- Vengono eseguite più istruzioni rispetto al caso precedente, ma sono sempre un numero costante
  - Per un input enorme, eseguire 1 o 5 istruzioni non cambia molto

```
//PRE A ha dim>1 elementi
int f(int *A, int dim) {
   return (A[0]==A[1]) &&
          (A[0]==A[2]) &&
          (A[0]==A[3]);
}
```

Cenni di Complessità Computazionale

- Problema: creare una funzione che, dato un array in input e la sua dimensione, calcoli il valore massimo
- Si deve scorrere tutto l'array, confrontando ogni elemento di A con il max
- Il numero di istruzioni che vengono eseguite dipende dalla dimensione di A linearmente
- O(1) non è più un limite superiore al numero di istruzioni, ma O(n) lo è

```
int max_value(int A[], int n) {
   /*
      PRE: A ha n>0 elementi
      POST: calcola il valore massimo in A
    */
   int max = A[0];
   for(int i=1; i<n; i+=1) {
      if (A[i]>max)
         max = A[i];
   }
   return max;
}
```

Cenni di Complessità Computazionale

- Problema: funzione che verifichi se un elemento di un array è duplicato altrove nell'array.
  - Il primo elemento deve essere confrontato con ogni altro elemento dell'array;
  - il secondo elemento deve essere confrontato con ogni altro elemento tranne il primo (esso è già stato confrontato con il primo).
  - Il terzo elemento deve essere confrontato con ogni altro elemento eccetto i primi due.
- il totale dei confronti sarà: n-1 + n-2 + ... + 2 + 1
- La somma dei primi n-1 numeri è n*(n-1)/2 = n^2/2 – n/2.
- Il termine principale che influenza la complessità è quello più grande: n^2
- La complessità dell'algoritmo è perciò O(n^2)

Efficienza Computazionale

- Un algoritmo è significativamente più efficiente di un altro se la sua complessità è minore

$$O(1) < O(n) < O(n^2)$$

- nel corso di algoritmi imparerete come calcolare la complessità di un algoritmo
- Notate che nel corso abbiamo posto l'accento sul risparmio di singole istruzioni, il motivo, più che per efficienza, è per l'atteggiamento da tenere nella scrittura del codice (attenzione ai dettagli)

Efficienza Computazionale: Esempio


```
int somma1nV1(int n) {
        return n*(n+1)/2;
}
```

```
int somma1nV2(int n) {
        int somma=0;
        for(int i=0;i<n;i+=1)
                somma+=i;
        return somma;
}
```

- Complessità somma1nV1?
- Complessità somma1nV2?

Efficienza Computazionale: Esempio


```
int somma1nV1(int n) {
        return n*(n+1)/2;
}
```

```
int somma1nV2(int n) {
        int somma=0;
        for(int i=0;i<n;i+=1)
                somma+=i;
        return somma;
}
```

- Complessità somma1nV1 = O(1) // numero di operazioni costanti
- Complessità somma1nV1 = O(n) // il corpo del for viene eseguito n volte