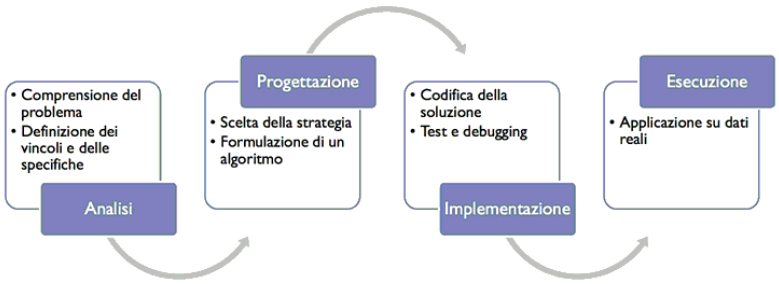
L01.1. SVILUPPO DI PROGRAMMI

Un **algoritmo** è composto da una sequenza *finita* di istruzioni *elementari*, che ha come obiettivo quello di risolvere un problema, partendo dai dati di input ed ottenendo i dati di output, risolvendo il problema dato.

FASI DELLO SVILUPPO:



- **ANALISI**:
  Specifica ***cosa*** fa il programma ed individua i dati di ***input*** e di ***output*** coi relativi ***vincoli***.
  I vincoli definiti sono: ***precondizione***, condizione definita sui dati di input che deve essere soddisfatta affinché la funzione sia applicabile, e ***postcondizione***, condizione definita su dati di output e dati di input e che deve essere soddisfatta al termine dell'esecuzione del programma (definendo cosa sono i dati di output in funzione di quelli di input).
  È buona norma utilizzare un ***dizionario dei dati*** da arricchire durante le varie fasi del ciclo di vita, ovvero una tabella il cui schema è "*identificatore*", "*tipo*" e "*descrizione*" (serve a specificare meglio l'identificatore e a descrivere il contesto in cui il dato viene usato).
  Esempio di ordinamento di una sequenza di interi:
  - Dati di ingresso: sequenza s di n interi
    - Precondizione: $n > 0$
  - Dati di uscita: sequenza s1 di n interi
    - Postcondizione: s1 è una permutazione di s dove
      $\forall i \in [0, n-2], s1_i \leq s1_{i+1}$

  Dizionario dei dati

Identificatore	Tipo	Descrizione
s	sequenza	sequenza di interi in input
s1	sequenza	sequenza di interi di output
n	intero	numero di elementi nella sequenza
i	intero	indice per individuare gli elementi nella sequenza

- **PROGETTAZIONE**:
  Definisce ***come*** il programma effettua la trasformazione specificata, progettazione dell'algoritmo per raffinamenti successivi (*stepwise refinement*), definendo i vari step e utilizzando la decomposizione funzionale.
  Esempio di ordinamento di una sequenza di interi:
  1. Input sequenza s in un array a di dimensione n
  2. Ordina array a di dimensione n
  3. Output sequenza s1 contenuta in array a di dimensione n

  Raffinamento del programma principale, ovvero definiamo delle funzioni corrispondenti agli step individuati:
  - input_array(a, n)
  - ordina_array(a, n)
  - output_array(a, n)

  Per ognuno di questi si effettua *specifica*, *progettazione*, *codifica* e *verifica*.
- **IMPLEMENTAZIONE**:
  Codifica dell'algoritmo nel linguaggio scelto, e lo si implementa su un elaboratore.
- **ESECUZIONE**:
  Verifica (testing) del programma (individuazione dei malfunzionamenti), effettuando delle scelte di casi di prova, si esegue il programma e si verificano i risultati ottenuti rispetto ai risultati attesi.

L01.1.1 ORGANIZZAZIONE DEL CODICE

La **progettazione** è l'insieme delle attività relative al concepimento della soluzione informatica di un problema, si sviluppa a partire da una specifica (prodotta in fase di analisi), come se fosse una vera e propria architettura (software) fatta da entità autonome che interagiscono tra loro.
Ci sono 4 **principi ispiratori** di un progetto software e sono:

- **ASTRAZIONE**:
  Procedimento mentale che consente di evidenziare le caratteristiche pregnanti di un problema e offuscare o ignorare gli aspetti che si ritengono secondari rispetto ad un determinato obiettivo. Esistono due tipi:
  1. **Astrazione funzionale e procedurale**, una funzionalità di un programma è delegata ad un sottoprogramma (funzione o procedura), è definita ed usabile indipendentemente dall'algoritmo che la implementa (es. algoritmi di ordinamento di un array);
  2. **Astrazione sui dati**, un dato o un tipo di dato è totalmente definito insieme alle operazioni che sul dato possono essere fatte, pertanto, sia le operazioni che il dato (o il tipo di dato) sono usabili a prescindere dalle modalità di implementazione.
- **INFORMATION HIDING**:
  Nascondere il funzionamento interno, deciso in fase di progetto, di una parte di un programma. I ***vantaggi*** sono:
  - Parti del programma che non devono essere modificate sono inaccessibili;
  - Correzione degli errori facilitata: un errore presente in un modulo può essere corretto modificando soltanto quel modulo.

- **RIUSO DEL CODICE**:
  Pratica, estremamente comune nella programmazione, di richiamare o invocare parti di codice precedentemente già scritte ogni qualvolta risulta necessario, senza doverle riscrivere daccapo. Una soluzione è usare funzioni e librerie di funzioni.
- **MODULARITÀ**:
  È una tecnica di suddividere un progetto software (aiuta a gestire la complessità), implementando dei moduli (funzioni, classi, package).
  Un **modulo** è una unità di programma che mette a disposizione risorse e servizi computazionali (dati, funzioni, …) e devono avere due proprietà:
  1. **Elevata coesione**: le varie funzionalità messe a disposizione da un singolo modulo sono strettamente correlate;
  2. **Indipendenza**: sviluppabili separatamente dal resto del programma, con compilazione e testing separati.

  Un modulo è costituito da due parti:
  1. **Interfaccia**, definisce le risorse ed i servizi messi a disposizione dei "clienti" (programma o altri moduli), completamente visibile ai clienti;
  2. **Sezione implementativa (body)**, implementa le risorse ed i servizi esportati ed è completamente occultato.

  Altre particolarità dei moduli sono che quest'ultimo può usare altri moduli e può essere compilato indipendentemente dal modulo che lo usa.
  In C non esiste un apposito costrutto per realizzare un modulo, per esportare le risorse definite in un file (modulo), il C fornisce un particolare tipo di file, chiamato **header file** (estensione .h) che rappresenta l'interfaccia di un modulo verso gli altri moduli.
  Per accedere alle risorse messe a disposizione da un modulo bisogna includere il suo header file, esempio #include "modulo.h".
  Il modulo mette a disposizione attraverso la sua interfaccia funzioni e procedure, si presenta come una "libreria" di funzioni e per poter garantire l'information hiding non ci devono essere variabili globali e funzioni nascoste.

Esempio modulo Utils:

```
// Interfaccia del modulo: file utils.h
/* Specifica della funzione scambia */
void scambia(int * x, int * y);
// dichiarazione altre funzioni …
```

Vista del cliente, può usare tali risorse e servizi esportati dal modulo.

```
// Implementazione del Modulo: file utils.c
/* commenti relativi alla progettazione e realizzazione della funzione scambia */
void scambia(int * x, int * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}
// definizione altre funzioni …
```

Reale implementazione del modulo.

USO DEI COMMENTI:
I commenti relativi alla specifica di una funzione possono essere inseriti nell'header file prima del prototipo della funzione, serve da documentazione per chi dovrà usare la funzione (modulo client). I commenti relativi alla progettazione e realizzazione di una funzione possono essere inseriti nel file .c prima della definizione della funzione (o anche all'interno del corpo della funzione), serve da documentazione per chi dovrà modificare la funzione.

Esempio modulo Vettore:

vettore.h

```
void input_array(int a[], int n);
void output_array(int a[], int n);
void ordina_array(int a[], int n);
int ricerca_array(int a[], int n, int elem);
int minimo_array(int a[], int n);
…
```

vettore.c

```
#include <stdio.h>
#include "vettore.h"
#include "utils.h"                          // contiene funzione scambia
int minimo_i(int a[], int i, int n);        // dichiarazione locale
void input_array(int a[], int n) { … }
void output_array(int a[], int n) { … }
void ordina_array(int a[], int n) { … }
int ricerca_array(int a[], int n, int elem) { … }
int minimo_array(int a[], int n) { … }
int minimo_i(int a[], int i, int n) { … }   // usata da ordina_array
```

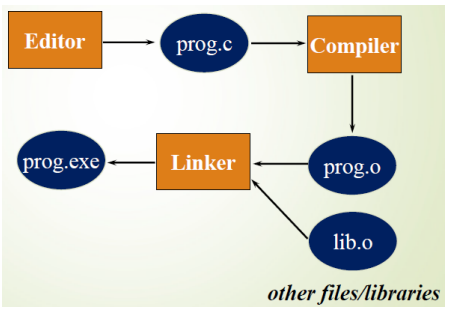
main.c

```
# include <stdio.h>
# include "vettore.h"
# define MAXELEM 100
int main(){    …   }
```

COMPILAZIONE DEI FILE:
I due moduli possono essere compilati indipendentemente:
- gcc -c utils.c
- gcc -c vettore.c
- gcc -c main.c
- gcc -c utils.c vettore.c main.c

In entrambi i casi si ottengono tre file con estensione .o e per **collegare** (link) i tre moduli e produrre l'eseguibile:
- gcc utils.o vettore.o main.o -o vettore.exe

Possibile compilazione e collegamento in un sol passo:
- gcc utils.c vettore.c main.c -o vettore.exe



MAKEFILE E MAKE:
Il **comando make** esegue compilazione e collegamento dei vari moduli che compongono il progetto.
Il **makefile** invece è costituito da specifiche del tipo: *target_file: dipendenze_da_file*.
L'esecuzione avviene tramite comando: *make target_file*
L'ordine dei comandi non è importante ma è buona norma inserire come prima specifica quella per la costruzione del programma eseguibile.

```
prog : main.o list.o item.o
        gcc main.o list.o item.o -o prog.exe
        ./prog.exe

item.o :
        gcc -c item.c

list.o :
        gcc -c list.c

main.o :
        gcc -c main.c

clean:
        rm -f *.o *.exe
```

L01.2. OPERAZIONI SU ARRAY

Un Array è una struttura contenente un certo numero di valori, tutti dello stesso tipo. Questi valori vengono chiamati ***elementi*** e possono essere selezionati individualmente tramite la loro posizione nell'array.

Per **dichiarare un vettore** dobbiamo specificare il ***tipo*** ed il ***numero di elementi***: `int a[10];`

Per **accedere ad un dato elemento del vettore** dobbiamo scrivere il ***nome del vettore*** seguito da un ***valore*** (operazione di ***indicizzazione***), gli elementi del vettore vengono sempre contati a partire da 0 ad n-1 : `a[0], a[1], …, a[9];`

INPUT DI ARRAY

```
void input_array(int *a, int n){
   int i;
   for(i=0; i<n; i++)
      scanf("%d", &a[i]);
}
```

OUTPUT DI ARRAY

```
void output_array(int *a, int n){
   int i;
   for(i=0; i<n; i++)
      printf("%d ", a[i]);
}
```

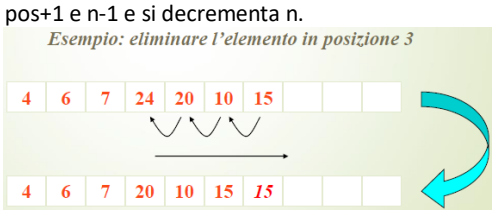
Per **eliminare un elemento dell'array** lo si elimina e si spostano, se esistono, tutti gli elementi successivi di una posizione a sinistra.

- *Analisi*:
  - Dati di ingresso: Array a di n elementi, posizione pos
    - Precondizione: 0 <= pos < n
  - Dati di uscita: Array a1 di n1 elementi
    - Postcondizione: ∀i ∈[0, pos-1] a1[i] = a[i] AND ∀j ∈[pos, n1-1] a1[j] = a[j+1] AND n1 = n-1

Dizionario dei dati

Identificatore	Tipo	Descrizione
a	array	array di interi in input
n	intero	numero di elementi nell'array a
pos	intero	indice dell'elemento da eliminare
a1	array	array di interi in output
n1	intero	numero di elementi nell'array a1

- *Progettazione*:
  Si spostano indietro tutti gli elementi dell'array a compresi tra le posizioni pos+1 e n-1 e si decrementa n.



Errore tipico:
eseguire il ciclo in senso decrescente, il risultato sarà il ricopiamento a sinistra dell'elemento finale.

- *Implementazione*:

```
/*precondizione deve essere controllata dalla funzione chiamante*/
void delete(int *a, int *n, int pos){          //array a, intero n di elementi dell'array, posizione pos da eliminare
   int i;
   for(i=pos; i<(*n)-1; i++)
      a[i] = a[i+1];
   (*n)--;                                     //Decrementiamo la taglia dell'array n
}
```

Per **visitare (o analizzare) gli elementi dell'array** si scorrono tutti gli elementi, esistono due tipi di visite:

- **Visita totale**: vengono analizzati tutti gli elementi, in questo caso bisogna usare un ciclo e visitare gli elementi dell'array in un verso.
  `for(i=0; i<n; i++)` oppure `for(i=n-1; i>=0; i--)`
- **Visita finalizzata**: la visita termina quando un elemento dell'array verifica una certa condizione, in questo caso bisogna usare due condizioni di uscita, una sull'indice di scansione (visitati tutti gli elementi si esce) e l'altra che dipende dal problema specifico.

Per **ricercare un dato elemento dell'array** lo si scorre e si verifica che c'è o meno.

- *Analisi*:
  - Dati di ingresso: Array a di n interi, elemento el
    - Precondizione: n > 0
  - Dati di uscita: Intero pos
    - Postcondizione: se el è contenuto in a allora pos è la posizione della prima occorrenza di el in a altrimenti pos = -1

Dizionario dei dati

Identificatore	Tipo	Descrizione
a	array	array di interi in input
n	intero	numero di elementi nell'array
el	intero	elemento da ricercare
pos	intero	indice dell'elemento trovato o -1

- *Progettazione*:
  Si scorre l'array di input finché non si trova l'elemento o non si raggiunge la fine dell'array (***visita finalizzata***).
  Se l'elemento è stato trovato allora si restituisce la posizione corrente, in questo caso all'uscita del ciclo l'indice dell'array corrisponde a quello dell'elemento cercato.
  Altrimenti si restituisce "-1".

  Si utilizza un indice per scorrere e una variabile booleana che dice se è stato trovato o meno l'elemento.
  La condizione del ciclo sarà: `(i < n && !trovato)`

- *Implementazione*:

```
int ricerca(int a[], int n, int elem) {
   int i= 0;                          /* indice dell'array*/
   int trovato= 0;                    /* nello schema di visita indica che è stato trovato*/
   while(i<n && !trovato)             /* visita finalizzata*/
       if (a[i] == elem)
            trovato= 1;               /* permette di uscire dal ciclo*/
       else i++;                      /* se non trovato incrementa l'indice*/
   /* se trovato restituisce la posizione i dell'elemento altrimenti restituisce -1 */
   return (trovato ? i : -1);
}
```

SHORT CIRCUIT EVALUATION:

Invece dello schema di visita finalizzata precedente, lo si può ridurre nel seguente modo: `while(i < n && a[i] != elem) i++;` *(codice completo sotto)*

Se la condizione a sinistra fosse falsa la condizione a destra non viene valutata.

RICERCA BINARIA:

Per quanto riguarda la ricerca di un elemento dell'array, se quest'ultimo fosse ordinato, non sarebbe necessario arrivare alla fine dell'array per stabilire che l'elemento è stato trovato o meno, ci si può fermare appena si trova un elemento maggiore (o uguale) di quello dato.

Ovviamente, se l'elemento è maggiore di tutti quelli presenti nell'array, allora si visiterà l'intero array (caso peggiore).

Esempio:

4	6	7	10	12	15	18	20	24	30

La ricerca di 13 e di 15 terminano quando l'elemento corrente è 15, mentre quella di 40 termina quando si sono visitati tutti gli elementi.

La **ricerca lineare in un array ordinato** può essere quindi effettuata diversamente.

Prendiamo in considerazione l'implementazione precedente ma in forma ***short-cut evaluation***:

```
int ricercaord(int a[], int n, int elem) {
    int i = 0;
    while(i<n && a[i] < elem)              // visita finalizzata
        i++;
    return (a[i]==elem ? i : -1);
}
```

Una nuova implementazione consiste nel dividere l'array in due metà e confrontare l'elemento da cercare con l'elemento centrale dell'array:

- se questi due elementi sono uguali allora è stato già trovato (e ci fermiamo);
- se l'elemento da cercare è minore del centrale allora si cerca nella prima metà dell'array;
- se l'elemento da cercare è maggiore del centrale allora si cerca nella seconda metà dell'array.

Se l'elemento non è presente, l'array si ridurrà ad un solo elemento, non divisibile in due (terminazione), nel caso peggiore si visitano $\log_2 n$ elementi dell'array.

L'implementazione della **ricerca binaria** in un array ordinato sarà la seguente:

```
int ricercabin(int num, int arr[], int n){
    int begin = 0, end = n-1, center;
    while(end >= begin){
        center = (begin+end)/2;
        if(num== arr[center])
            return center;
        else if (num< arr[center])
            end = center -1;
        else if (num> arr[center])
            begin = center + 1;
    }
    return -1;
}
```

L01.3. ORDINAMENTO

Il Problema dell'Ordinamento è quello di elencare gli elementi di un insieme secondo una sequenza stabilita da una relazione d'ordine.

Gli algoritmi di ordinamento hanno determinate proprietà:

- **Stabile**, due elementi con la medesima chiave mantengono lo stesso ordine con cui si presentavano prima dell'ordinamento;
- **In loco**, in ogni istante al più è allocato un numero costante di variabili, oltre all'array da ordinare;
- **Adattivo**, il numero di operazioni effettuate dipende dall'input;
- **Interno** (dati contenuti nella memoria RAM) vs **esterno** (dati contenuti su disco o nastro o file).

Algoritmi di ordinamento semplici	Algoritmi di ordinamento avanzati
Numero di operazioni quadratico rispetto alla taglia dell'input $O(n^2)$: ▪ Selection sort ▪ Insertion sort ▪ Bubble sort	Numero di operazioni rispetto alla taglia dell'input: $O(n \log n)$: ▪ Merge sort ▪ Quick sort (caso medio $O(n \log n)$, caso peggiore $O(n^2)$)

SELECTION SORT:

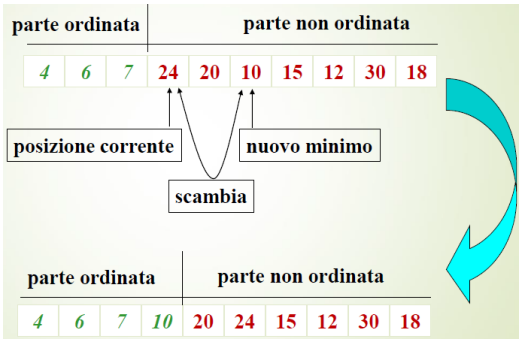
L'algoritmo consiste nella ricerca del minimo valore, presente nel vettore, e posizionarlo nella prima posizione, cercare il secondo minimo e metterlo nella seconda posizione e così via, il procedimento va eseguito n volte se n è la dimensione del vettore. Come struttura dati di ingresso necessita di un vettore. Tale algoritmo ha un approccio incrementale, pertanto il vettore concettualmente sarà suddiviso in due sotto-vettori:

- Il *sotto-vettore sinistro*, che contiene gli elementi ordinati, inizialmente è vuoto;
- Il *sotto-vettore destro*, contiene gli elementi da ordinare, inizialmente coincide con il vettore in input.

Si ha la terminazione dell'algoritmo nel caso in cui la dimensione del sotto-vettore sinistro coincide con la dimensione del vettore di input.

```
void selection_sort(int *a, int n) {
    int i, m;
    for (i=0;i<n-1;i++) {
        m = min(&a[i], n-i)+i;
        swap(&a[i],&a[m]);
    }
}
```

```
int min(int *a, int n){
    int min=0, i;
    for(i=1; i<n; i++)
        if(a[i]< a[min])
            min=i;
    return min;
}
void swap(int *a, int *b){
    int temp=*a;
    *a=*b;
    *b=temp;
}
```

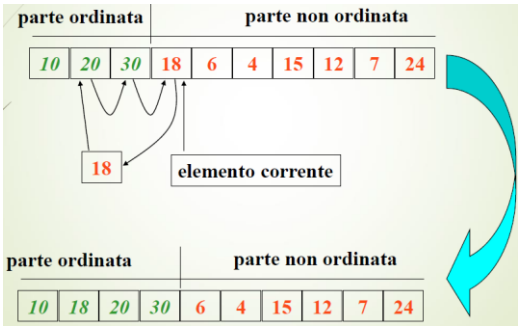


INSERTION SORT:

Effettuiamo una visita totale dell'array e ad ogni passo gli elementi che precedono l'elemento corrente sono ordinati, si inserisce l'elemento corrente nella posizione che garantisce il mantenimento dell'ordinamento e gli elementi precedenti se sono maggiori vengono spostati in avanti, il primo elemento è già "ordinato" e quindi si parte dal secondo. Si termina l'algoritmo quando il sotto-vettore ordinato ha dimensione uguale al vettore.

```
void insertion_sort(int *a, int n){
   int i;
   for(i=1; i<n;)
      insertion_sorted_array(a, a[i], &i);
}
```

```
void insertion_sorted_array(int *a, int val, int *n){
   int i;
   for(i=*n; i>0; i--)
      if(val<a[i-1])
         a[i]=a[i-1];
      else
         break;
   a[i]=val;
   (*n)++;
}
```



BUBBLE SORT:

Tale algoritmo ha un approccio incrementale, pertanto il vettore concettualmente sarà suddiviso in due sotto-vettori:

- *sotto-vettore sinistro*, che contiene gli elementi ancora da ordinare, inizialmente coincide con il vettore in input;
- *sotto-vettore destro*, contiene gli elementi già ordinati, inizialmente è vuoto.

Si ha la terminazione dell'algoritmo nel caso in cui la dimensione del sotto-vettore destro coincide con la dimensione del vettore di input.

Di questo algoritmo esistono due versioni:

1. **Iterativo**, si effettuano n-1 visite dell'array, alla visita i-esima si confrontano elementi adiacenti dal primo al (n-i)-esimo elemento e gli elementi adiacenti che non risultano ordinati vengono scambiati.

```
void bubble_sort(int *a, int n){
   int i, j;
   for(i=1; i<n; i++)
      for(j=0; j<n-i; j++)
         if(a[j]>a[j+1])
            swap(&a[j], &a[j+1]);
}
```

NB: ad ogni passo l'elemento più grande viene portato nella sua posizione finale, dopo il passo i-esimo, gli elementi tra le posizioni n-i ed n-1 risultano ordinati e nelle loro posizioni finali.

2. **Adattivo**, se in una visita dell'array non è stato effettuato alcuno scambio, allora l'array è ordinato, in tal caso si può interrompere.

```
int adaptive_bubble_sort(int *a, int n){
   int i, j, sorted=0;
   for (i=1; i<n && !sorted; i++){
      sorted=1;
      for(j=0; j<n-i; j++)
         if(a[j]>a[j+1]){
            swap(&a[j], &a[j+1]);
            sorted=0;
         }
   }
   return i-1;                //Ritorna il numero si scambi fatti
}
```