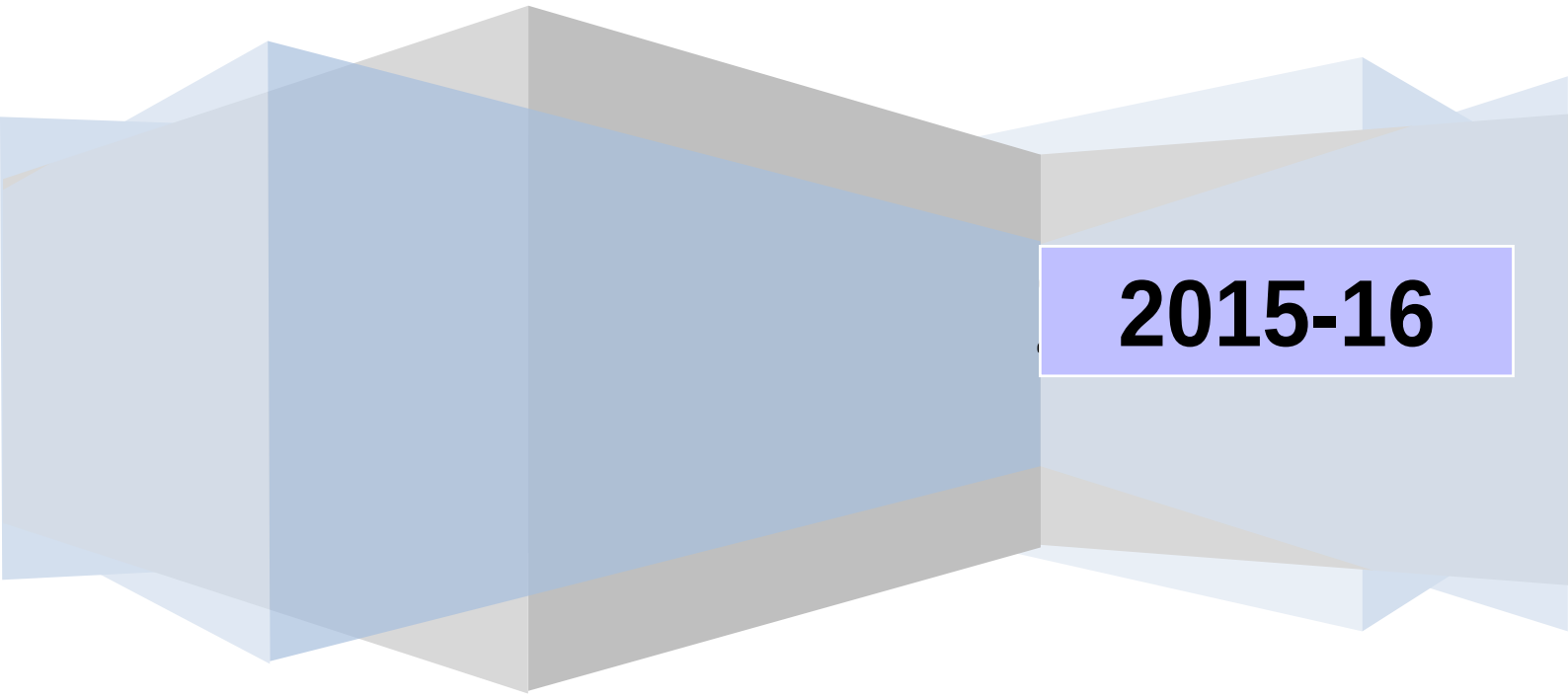
DIIES Dipartimento di
INGEGNERIA
dell'INFORMAZIONE, delle INFRASTRUTTURE e dell'ENERGIA SOSTENIBILE

Corso di Fondamenti di Informatica

Dispensa 10: Complessità Computazionale, Ordinamento e Ricerca

Prof. Domenico Rosaci



2015-16

1. Cenni sulla complessità computazionale

Uno stesso problema P può essere risolto da diversi algoritmi. Su quali basi scegliamo un certo algoritmo piuttosto che un altro? Noi vorremmo che l'algoritmo scelto usasse in maniera efficiente le risorse della macchina su cui è eseguito; in particolar modo, vorremmo che il tempo di esecuzione fosse il minore possibile.

Il tempo di esecuzione di un algoritmo dipende da diversi fattori, quali:

- L'input dell'algoritmo: ad esempio, se un algoritmo deve ordinare un vettore, ci metterà tanto più tempo quanto più grande è la dimensione del vettore, che in questo caso rappresenta l'input dell'algoritmo. Spesso, negli algoritmi che andremo a considerare, che operano su vettori, la dimensione dell'input è rappresentata dalla dimensione del vettore
- Le caratteristiche del calcolatore su cui si sta eseguendo l'algoritmo: è chiaro che lo stesso algoritmo sarà eseguito più velocemente su un calcolatore con una potente CPU che su un calcolatore con basse capacità computazionali

Conveniamo quindi di studiare il tempo di esecuzione T di un algoritmo in funzione della dimensione dell'input, che rappresentiamo genericamente con la variabile *n*. Con T(n) indichiamo quindi la *funzione di costo temporale* dell'algoritmo. Inoltre cerchiamo di esprimere T(n) in modo indipendente dalle caratteristiche del calcolatore su cui si esegue l'algoritmo: quindi, piuttosto che misurare T(n) in secondi, scegliamo di misurarlo in *numero di operazioni* richieste, grandezza chiaramente indipendente dal calcolatore utilizzato.

Inoltre, nello studio di T(n), non prendiamo in considerazione le eventuali costanti di proporzionalità esistenti nell'espressione di T(n) stesso. Ovvero, per un algoritmo avente $T(n)=3n^2+5$, diremo semplicemente che questo algoritmo ha un costo *proporzionale a* n^2. Infine, ci interessa studiare la forma funzionale di T(n) quando *n* diventa abbastanza grande, cioè, al limite per n tendente all'infinito.

Per discutere quindi della *forma funzionale* di una funzione di costo T(n) associata ad un algoritmo, argomento che va sotto il nome di *complessità computazionale*, faremo uso della cosiddetta notazione "O grande". Piuttosto cioè di calcolare esattamente la T(n) di un algoritmo, determineremo a quale *classe di complessità* O(f(n)) appartiene l'algoritmo stesso, dove O(f(n)) si legge "O di f(n)" e significa che per ogni algoritmo appartenente a questa classe esistono due costanti c e h tali che la T(n) dell'algoritmo è limitata superiormente dalla funzione c*f(n) per ogni valore di n più grande di h.

Ad esempio, dire che un algoritmo ha una funzione di costo appartenente alla classe $O(n^2)$ significa dire che, a partire da un certo valore di n (abbastanza grande), T(n) "cresce" meno velocemente della funzione n^2.

Le classi $O(n^2)$, $O(n^3)$, ecc. caratterizzano gli algoritmi cosiddetti *polinomiali*. In particolare, la classe $O(n)$ caratterizza gli algoritmi *lineari*. La classe $O(1)$ caratterizza gli algoritmi *costanti*, ovvero quelli in cui il tempo di calcolo è indipendente dalla dimensione dell'input. La classe $O(\log_2 n)$ caratterizza gli algoritmi *logaritmici*, il cui costo è inferiore a quello degli algoritmi lineari. Tutte le classi sopra citate caratterizzano algoritmi denominati *efficienti*, perchè i loro tempi di calcolo aumentano con l'aumentare della dimensione dell'input in maniera computazionalmente "accettabile". Le classi $O(2^n)$, $O(10^n)$ e così via, ovvero quelle dove la dimensione dell'input appare come esponente nella funzione di costo, caratterizzano algoritmi *inefficienti*, perchè al crescere della dimensione dell'input il costo di questi algoritmi esplode in modo esponenziale.

Un algoritmo è O(1) quando deve eseguire un numero costante di operazioni, indipendentemente dalla dimensione dell'input. Quindi, quando troviamo all'interno di un codice che abbiamo scritto un blocco di istruzioni quali somme, differenze, assegnamenti, ecc., il costo di questo blocco è O(1). Quando ripetiamo

un blocco di istruzioni avente costo O(f(n)) all'interno di un ciclo di dimensione *n*, la complessità dell'intero ciclo è O(n*f(n)). Se abbiamo due blocchi di codice consecutivi, uno di complessità O(f(n)), l'altro di complessità O(g(n)), il codice complessivo avrà per complessità la massima complessità tra O(f(n)) e O(g(n)). Con queste due semplici regole pratiche è facile determinare la complessità degli algoritmi di seguito descritti.

2. Algoritmi ricorsivi

Un algoritmo ricorsivo è un algoritmo espresso in termini di sé stesso, in modo tale che quando l'algoritmo viene eseguito su un certo input, tale input viene ridotto di dimensione e l'algoritmo viene rieseguito su tale input ridotto.

Ad esempio, il calcolo del fattoriale di un numero intero n comporta il calcolo del seguente prodotto:
fatt(n)=1*2*….*n
essendo per definizione fatt(0)=fatt(1)=0.

Un algoritmo che calcoli il fattoriale di n ha come input il valore intero n. Possiamo osservare che tale algoritmo potrebbe operare richiamando sé stesso su un input di dimensione ridotta, ovvero n-1, tramite la semplice osservazione che:

fatt(n)=(1*2*….*n-1)*n=fatt(n-1)*n

Quindi quando questo algoritmo dovesse calcolare il fattoriale di 3, richiamerebbe sé stesso per calcolare il fattoriale di 2 ed ottenuta la risposta moltiplicherebbe il risultato per 3 per determinare la soluzione finale. A sua volta, l'algoritmo richiamato per calcolare il fattoriale di 2 richiamerebbe sé stesso per calcolare il fattoriale di 1 ed ottenuta la risposta moltiplicherebbe il risultato per 2. Osserviamo poi che l'algoritmo richiamato per calcolare il fattoriale di 1 non ha bisogno di richiamare sé stesso per eseguire il suo compito, poiché produce direttamente come risultato 1. E' questo un esempio di *passo base* della ricorsione, ovvero di quella fase in cui l'algoritmo ricorsivo sa produrre il suo risultato senza richiamare ulteriormente sé stesso.

```
public static int fattRic(int n){
                //passo base
                if((n==0)||(n==1)) return 1;
                //passo ricorsivo
                else return fattRic(n-1)*n;
}
```

La versione *iterativa* di tale algoritmo sarebbe la seguente:

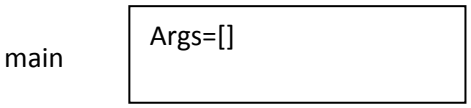
```
public static int fattIt(int n){
                if((n==0)||(n==1)) return 1;
                int fatt=1;
                for(int i=1;i<=n;i++)
                        fatt=fatt*i;
                return fatt;
        }
```

Come si vede, la versione ricorsiva è più immediata e leggibile di quella iterativa. Tuttavia un grosso svantaggio della ricorsione è rappresentato dal maggior impegno di risorse, in termini di CPU e di memoria RAM.

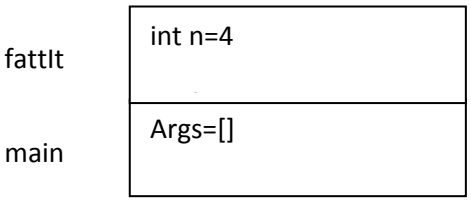
Infatti, quando un metodo viene eseguito, quello che succede in memoria RAM è che una porzione di memoria viene riservata per quel metodo, in una particolare area di memoria chiamata **stack**. Ad esempio, si consideri il seguente metodo:

```
public static void main(String []args){
        int n=4;
        System.out.println(fattIt(n));
}
```

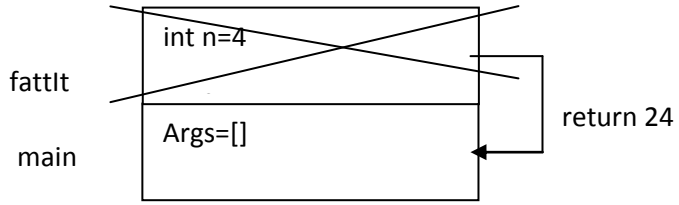
Quando il metodo viene eseguito, nello stack viene riservato un elemento, che conterrà le variabili locali del metodo.



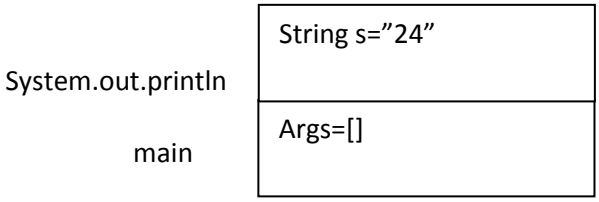
Quando il metodo main chiama il metodo *fattIt* quello che succede è che il metodo main è come se venisse sospeso, perché la CPU non opera più su di esso ma inizia ad eseguire il metodo *fattIt*. Un nuovo elemento viene inserito nello stack, sopra a quello già esistente, associata al metodo *fattIt*, che conterrà le variabili locali di tale metodo:



Quando *fattit* terminerà la sua esecuzione, calcolando il valore fatt=24 e restituendolo tramite l'istruzione return, il controllo della CPU ritornerà al metodo main e l'elemento dello stack che era associato a *fattIt* verrà eliminato.



A questo punto verrà chiamato il metodo System.out.println, e quindi verrà attivato un altro elemento sullo stack.

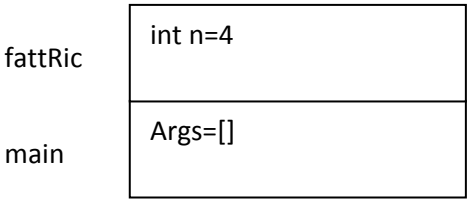


System.out.println stamperà sullo schermo la stringa "24" e poi passerà di nuovo il controllo al main, e l'elemento dello stack associato a System.out.println verrà eliminato. A questo punto resterà sullo stack solo l'elemento associato al main, che quando il main terminerà sarà anch'esso rimosso.

Ma cosa sarebbe successo se nel main ci fosse stata la chiamata ad un metodo ricorsivo, come ad esempio *fattRic*?

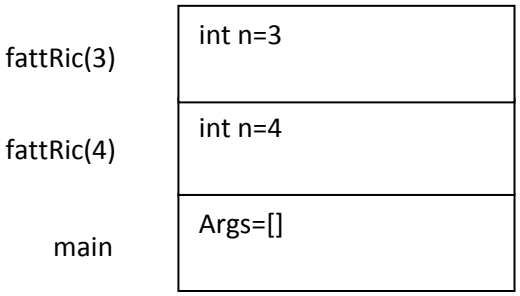
```
public static void main(String []args){
        int n=4;
        System.out.println(fattRic(n));
}
```

Al momento della chiamata a *fattRic* la situazione sullo stack sarebbe questa:

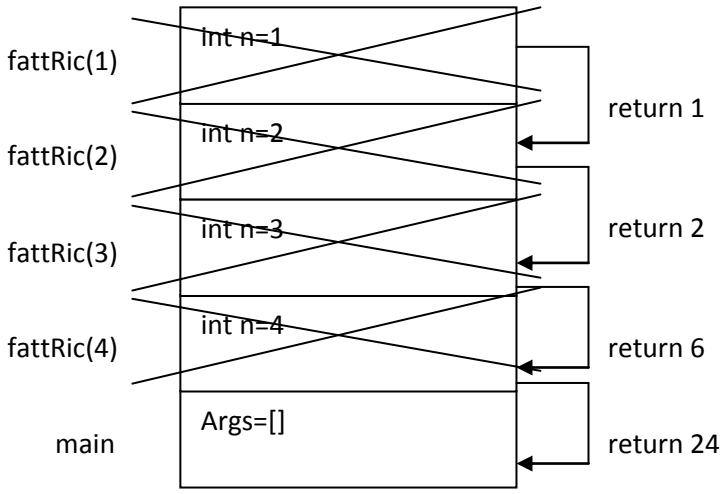


fattRic inizia la sua esecuzione, e verificando di non essere al passo base, esegue il passo ricorsivo. Ciò comporta che il metodo *fattRic* richiami sé stesso. In realtà, quello che succede è che viene invocata un'altra *istanza* di *fattRic*, diversa dall'istanza invocata dal main. Il main aveva infatti invocato *fattRic* passandogli come valore di ingresso 4, mentre adesso stiamo richiamando *fattRic* passandogli come valore di ingresso 3. Per non confondere le due istanze, chiameremo quindi *fattRic*(4) l'istanza chiamata dal main

e *fattRic*(3) l'istanza chiamata da *fattRic*(4). Al momento della chiamata di *fattRic*(3) un altro elemento viene aggiunto allo stack. Quindi il controllo della CPU è passato a *fattRic*(3), mentre *fattRic*(4) resta sospeso aspettando il risultato di *fattRic*(3).



A sua volta, *fattRic*(3), non essendo al passo base, chiamerà *fattRic*(2) e *fattRic*(2) per lo stesso motivo chiamerà *fattRic*(1). A questo punto *fattRic*(1) saprà ritornare il suo risultato senza ulteriori chiamate ricorsive, ed il risultato fornito a *fattRic*(2) permetterà anche a *fattRic*(2) di produrre il suo risultato, fornendolo a *fattRic*(3). A sua volta *fattRic*(3) fornirà il suo risultato a *fattRic*(4) che produrrà il risultato finale da ritornare al main. Inoltre ogni volta che un metodo fornisce il suo risultato, il corrispondente elemento sullo stack si svuota.



Appena ottenuto il risultato da *fattRic(4)*, il main chiamerà System.out.println per stamparlo, allo stesso modo visto sopra.

3. Algoritmi di ordinamento

In questa sezione trattiamo del problema di ordinare un vettore di interi di dimensione *n*. Stabiliamo di desiderare un ordinamento crescente per il vettore.

3.1. L'algoritmo di ordinamento *bubblesort*

L'algoritmo si basa sulla semplice idea di ordinare il vettore per "passate successive", portando l'elemento massimo in ultima posizione durante la prima "passata", il secondo massimo in penultima posizione durante la seconda "passata" e così via. Per ottenere questo scopo, ad ogni passata si eseguono (n-1) confronti tra elementi contigui, operando uno scambio tra due elementi V[i] e V[i+1] se V[i] è più grande di V[i+1].

```
void scambia(int V[], int i,int j)
{
        int temp=V[i];
        V[i]=V[j];
        V[j]=temp;
}
```

Complessità: O(1) (complessità costante, il blocco di istruzioni esegue sempre 3 istruzioni)

```
public class Ordinamento{
        public static void stampa(int v[]){
                for(int i=0;i<v.length;i++)
                        System.out.println(v[i]);
        }
        public static void scambia(int v[],int i, int j){
                int temp=v[i];
                v[i]=v[j];
                v[j]=temp;
        }
        public static void bubblesort(int v[]){
                for(int pass=1; pass<v.length-1;pass++)
                        for(int j=0;j<v.length-pass;j++)
                                if(v[j+1]<v[j]) scambia(v,j+1,j);
        }
        public static void main(String args[]){
        int v[]={7,2,4,5,1,3,8,6};
        bubblesort(v);
        stampa(v);
        }
}
```

Complessità: $O(n^2)$ (sia nel caso peggiore che nel caso migliore)

Esempio

V=[9,6,5,4,3,2]

Pass=1: V=[6,5,4,3,2,9] l'elemento più grande è stato portato in fondo al vettore
Pass=2: V=[5,4,3,2,6,9] il secondo elemento più grande è finito in penultima posizione
Pass=3: V=[4,3,2,5,6,9]

Pass=4: V=[3,2,4,5,6,9]
Pass=5 V=[2,3,4,5,6,9]

Come si vede, tutte e cinque le passate sono state necessarie. Questo è successo perché il vettore era completamente disordinato. Ma se il vettore è parzialmente ordinato, alcune passate saranno inutili.

Esempio

V=[4,9,5,6,7,8]

Dopo la prima passata il vettore è già ordinato, ma l'algoritmo non ha modo di accorgersene e continua con le altre inutili passate. L'algoritmo si accorgerebbe però che la prima passata ha già ordinato il vettore, se monitorasse ad ogni passata qual è l'indice dell'ultimo elemento scambiato. Nel nostro caso alla seconda passata l'ultimo elemento scambiato ha indice 0, quindi vuol dire che da 0 in poi tutti gli elementi sono ordinati e quindi non necessitano ulteriori passate.

3.2. L'algoritmo di ordinamento *bubblesort con sentinella*

In questa versione dell'algoritmo bubblesort, si memorizza ad ogni passata la posizione dell'ultimo elemento scambiato in una variabile *sup* e si considera quindi che il vettore risulta ordinato da *sup* a *n-1*. La variabile *sup* viene ricalcolata ad ogni passata, quindi è stato necessario usare una variabile *ultimo_scambiato* per calcolare la posizione dell'ultimo elemento scambiato durante la singola passata ed assegnare alla fine della passata tale valore alla variabile *sup*. Questa versione dell'algoritmo risulta chiaramente più efficiente nel caso migliore, poiché è in grado di accorgersi alla prima passata che non viene effettuato nessuno scambio e che quindi non sono necessarie ulteriori passate.

```
public static void bubblesort_sentinella(int V[])
{
        int ultimo_scambiato, sup=V.length-1,i;
        while(sup!=0)
        {
        ultimo_scambiato=0;
        for(i=0;i<sup;i++)
                if(V[i]>V[i+1])
                {
                        scambia(V,i,i+1);
                        ultimo_scambiato=i;
                }
        sup=ultimo_scambiato;
        //il vettore è ordinato da sup a V.lenght-1

        }
}
```

Complessità caso peggiore: $O(n^2)$
Complessità caso migliore: $O(n)$

3.3. L'algoritmo *Selectionsort*

Questo algoritmo ordina il vettore in più passate, come nel caso del bubblesort. Alla prima passata, l'algoritmo trova l'elemento con minimo valore nel vettore, e scambia tale elemento con l'elemento di indice 0. In questo modo il vettore è ordinato fino all'elemento di indice 0. Alla seconda passata viene

trovato l'elemento di valore minimo nel sottovettore che va dall'indice 1 all'indice n-1, e tale elemento viene scambiato con l'elemento di indice 1. In questo modo il vettore sarà ordinato dall'indice 0 all'indice 1. Si va avanti così, trovando i successivi minimi nei sottovettori, fino alla passata n-1.

```
public static int minimo(int V[],int ini){
        int min=V[ini];
        int pos=-1;
        for(int i=ini;i<V.length;i++)
            if(V[i]<min){
                min=V[i];
                pos=i;
            }
        return pos;
    }
    public static void selectionsort(int V[]){
        for(int i=0;i<(V.length-1);i++)
            scambia(V,i,minimo(V,i));
    }
```

Complessità caso peggiore: $O(n^2)$
Complessità caso migliore: $O(n^2)$

3.4. L'algoritmo *Mergesort*

Questo algoritmo ordina il vettore usando una tecnica ricorsiva. Il vettore viene diviso in due metà che vengono ordinate separatamente sfruttando una chiamata ricorsiva, e quindi le due metà vengono "fuse" usando la funzione *merge*. Quest'ultima riceve in ingresso le due metà ordinate e produce un vettore V ordinato a partire da queste due metà.

```
void mergesort(int V[], int inf, int sup)
{
   int med;
   if (inf<sup)
   {
      med=(inf+sup)/2;
      mergesort(V, inf, med);
      mergesort(V, med+1, sup);
      merge(V, inf, med, sup);
   }
}

void merge(int V[], int inf, int med, int sup)
{
   int aux[]=new int[V.length];
   int i=inf, j=med+1, k=inf;
   while((i<=med) && (j<=sup))
      if (V[i]<V[j])
      {
         aux[k]=V[i];
         i++;
         k++;
      }
```

```
        else
        {
           aux[k]=V[j];
           j++;
           k++;
        }
     while(i<=med)
    {
       aux[k]=V[i];
       i++;
       k++;
    }

    while(j<=sup)
    {
        aux[k]=V[j];
       j++;
       k++;
    }
for(i=inf; i<=sup; i++)
       V[i]=aux[i];

}
```

3.5. Una classe per gestire gli ordinamenti.

```
import java.util.*;

public class Ordinamento {
        public static void scambia(int V[], int i,int j)
        {
                int temp=V[i];
                V[i]=V[j];
                V[j]=temp;
        }
        public static void inserisci(int V[],int n)
        {
                Scanner in=new Scanner(System.in);
                for(int i=0;i<n;i++){
                        System.out.print("V["+(i+1)+"]=");
                        V[i]=in.nextInt();
                }
        }
        public static void stampa(int V[])
        {
                for(int i=0;i<V.length;i++){
                        System.out.println("V["+(i+1)+"]="+V[i]);
                }
        }
        public static void bubblesort(int V[]){
                int i, pass;
                for(pass=0;pass<V.length-1;pass++)
```

```
                for(i=0;i<V.length-1;i++)
                        if(V[i]>V[i+1])
                                scambia(V,i,i+1);
}
public static void bubblesort_sentinella(int V[])
{
        int ultimo_scambiato, sup=V.length-1,i;
        while(sup!=0)
        {
                ultimo_scambiato=0;
                for(i=0;i<sup;i++)
                        if(V[i]>V[i+1]){
                        scambia(V,i,i+1);
                        ultimo_scambiato=i;
                }
                sup=ultimo_scambiato;
        //il vettore è ordinato da sup+1 a n-1
        }
}
//trova l'indice del minimo nel sottovettore di V da ini a n-1
public static int minimo(int V[],int ini){
        int min=V[ini];
        int pos=-1;
        for(int i=ini;i<V.length;i++)
                if(V[i]<min){
                        min=V[i];
                        pos=i;
                }
        return pos;
}
public static void selectionsort(int V[]){
        for(int i=0;i<(V.length-1);i++)
                scambia(V,i,minimo(V,i));
}
public static void merge(int V[], int inf, int med, int sup)
{
   int aux[]=new int[V.length];
   int i=inf, j=med+1, k=inf;
   while((i<=med) && (j<=sup))
      if (V[i]<V[j])
      {
         aux[k]=V[i];
         i++;
         k++;
      }
      else
      {
         aux[k]=V[j];
         j++;
         k++;
      }
    while(i<=med)
   {
```

```
            aux[k]=V[i];
            i++;
            k++;
        }
        while(j<=sup)
        {
            aux[k]=V[j];
            j++;
            k++;
        }
        for(i=inf; i<=sup; i++)
            V[i]=aux[i];
    }
    public static void mergesort(int V[], int inf, int sup)
    {
        int med;
        if (inf<sup)
        {
            med=(inf+sup)/2;
            mergesort(V, inf, med);
            mergesort(V, med+1, sup);
            merge(V, inf, med, sup);
        }
    }

    public static void main(String []args){
            int V[]=new int[5];
            inserisci(V,5);
            selectionsort(V);
            stampa(V);
    }
}
```

4. Gli algoritmi di ricerca

In questa sezione trattiamo del problema di ricercare un valore intero x (denominato *chiave*) in un vettore di interi V. Le funzioni descritte nel seguito risolvono questo problema, ritornando la posizione in cui x viene trovato nel caso la ricerca abbia successo, ritornando il valore false se la ricerca non ha successo.

3.1. Algoritmo di ricerca lineare.

```
int RicLin(int V[],int x)
{
    for(int i=0; i<V.length;i++)
        if(V[i]==x) return i;
    return false;
}
```

la complessità di questo algoritmo è O(n).

4.1. L'algoritmo di ricerca binaria (in forma ricorsiva)

Il seguente algoritmo può essere applicato solo al caso di vettore ordinato. E' possibile sfruttare tale informazione aggiuntiva per realizzare una soluzione più efficiente al problema della ricerca.

```
bool RicBin(int V[],int x, int ini, int fin)
{
        if(ini>fin)
                return false;
        else
        {
                int m=(ini+fin)/2;
                if(V[m]==x) return m;
                if(x<V[m])
                        return RicBin(V,x,ini,m-1);
                else
                        return RicBin(V,x,m+1,fin);

        }
}
```

Complessità: $O(\log_2 n)$

La complessità della ricerca binaria deriva dal fatto che, nel caso peggiore, l'algoritmo esegue un numero di divisioni per 2 del vettore pari a $\log_2 n$.

4.2. L'algoritmo di ricerca binaria (in forma iterativa).

```
bool ricerca_binaria (const int b[], int dim, int x, int iniziale, int finale)
{
   int centrale;
   while (iniziale<=finale)
   {
       centrale=(iniziale+finale)/2;
       if (x==b[centrale])
         return centrale;
       else if (x<b[centrale])
         finale=centrale-1;
       else
         iniziale=centrale+1;
   }

   return false;
}
```