ALGORITMI NOTEVOLI IN ARRAY 1-D

ORDINAMENTO

Proprietà degli algoritmi di sort

- **STABILITÀ**
- Un algoritmo di ordinamento è **stabile** quando mantiene l'ordine originale degli oggetti con chiavi uguali. Se ad esempio si vuole ordinare una lista di nominativi per *cognome* e la lista è già stata ordinata per *nome,* utilizzando un algoritmo stabile l'ordinamento dei *nomi* verrà rispettato come in un classico elenco
- **SUL POSTO (IN-PLACE)**
  - Un algoritmo di ordinamento si dice **sul posto** se utilizza un numero costante di variabili oltre all'array da ordinare e **non** utilizza quindi un array di supporto.
- **ADATTIVITÀ**
  - Un algoritmo di ordinamento è **adattivo** quando trae vantaggio dagli elementi già ordinati.

BUBBLE SORT

RAPPRESENTA IL SORTING PER SCAMBIO

- **Bubble sort** è un semplice algoritmo di ordinamento non molto efficiente. si utilizza a scopi didattici grazie alla sua.
  L'algoritmo deve il suo nome al modo in cui gli elementi vengono ordinati: quelli più piccoli "risalgono" verso un'estremità della lista, mentre quelli più grandi "affondano" verso l'estremità opposta, come le bolle

6 5 3 1 8 7 2 4

Versioni algoritmo

```
public static void BubbleSortBase(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
        for (int j = 1; j < (arr.Length - i); j++)
            if (arr[j - 1] > arr[j])
            {
                int temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
}
```

```
public static void BubbleSort(int[] arr)
{
    bool ordered = false;
    for (int i = 0; i < arr.Length && !ordered; i++)
    {
        ordered = true;
        for (int j = 1; j < (arr.Length - i); j++)
            if (arr[j - 1] > arr[j])
            {
                int temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
                ordered = false;
            }
    }
}
```

Descrizione

- Ad ogni iterazione, si considerano, una ad una, tutte le possibili coppie di elementi adiacenti, scambiandoli se risultano nell'ordine errato. Dopo ogni iterazione, l'elemento *massimo* è in fondo alla parte di array considerata. Grazie all'utilizzo della variabile **ordered**, è possibile ottimizzare l'algoritmo fermandone l'esecuzione se il ciclo *for* più interno non ha effettuato alcuno scambio.

Proprietà e Complessità Computazionale versione NON base

- Adattivo solo nella versione NON base

STABILE
IN-PLACE
ADATTIVO

TEMPI DI ESECUZIONE

Caso peggiore: l'array è ordinato in maniera inversa; la prima iterazione effettuerà *n* controlli, la successiva *n-1* e così via, quindi **$O(N^2)$**

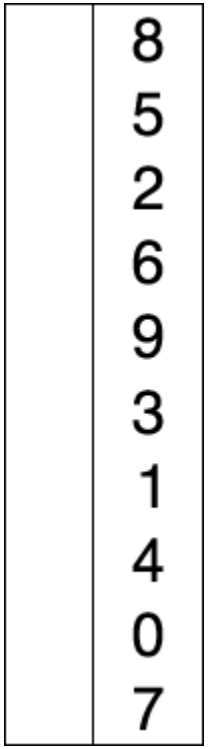
Caso migliore: l'array è già ordinato e verrà effettuata una sola iterazione della sequenza **O(N)**

Caso peggiore	$O(N^2)$
Caso medio	$O(N^2)$
Caso migliore	O(N)

SELECTION SORT

RAPPRESENTA IL SORTING PER SELEZIONE

- Il **selection sort** o **ordinamento per selezione**, opera dividendo la sequenza di input in due parti: la sottosequenza di elementi già *ordinati* (che occupa le prime posizioni dell'array) e la sottosequenza di elementi *non ordinati* (che occupa il resto dell'array).
- Inizialmente, la sottosequenza ordinata è vuota, mentre quella non ordinata rappresenta l'intero input. L'algoritmo seleziona di volta in volta il numero minore nella sottosequenza non ordinata e lo sposta in quella ordinata.



```
public static void SelectionSort(int[] arr)
{
    for (int i = 0; i < arr.Length - 1; i++)
    {
        for (int j = i + 1; j < arr.Length; j++)
            if (arr[j] < arr[i])
            {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
    }
}
```

Il ciclo esterno si utilizza per tenere conto dell'*i-esima* posizione dell'array, mentre il secondo ciclo serve per trovare l'elemento più piccolo a partire dall'*i-esima +1* posizione. Se viene trovato un elemento più piccolo di quello in posizione *i*, avviene lo scambio.

Proprietà e Complessità Computazionale

STABILE
IN-PLACE
NON ADATTIVO

TEMPI DI ESECUZIONE

Dal punto di vista del numero di operazioni svolte dall'algoritmo, non esiste un caso favorevole o uno sfavorevole: qualunque sia la disposizione iniziale degli elementi della sequenza da ordinare, il numero di operazioni effettuate per ordinare la sequenza sarà pari ad **$O(N^2)$**.

Caso peggiore	**$O(N^2)$**
Caso medio	**$O(N^2)$**
Caso migliore	**$O(N^2)$**

INSERTION SORT

RAPPRESENTA IL SORTING PER INSERIMENTO

- È molto efficiente per l'ordinamento di array di dimensione molto piccola o per array parzialmente ordinati.
- L'idea di ordinamento è simile al modo in cui un giocatore di carte le ordina nella propria mano. Si inizia con la mano vuota e le carte capovolte sul tavolo, poi si prende una carta alla volta dal tavolo e la si inserisce nella giusta posizione. Per trovare la giusta posizione per una carta, la si confronta con le altre carte nella mano, da destra verso sinistra. Ogni carta più grande verrà spostata verso destra in modo da fare posto alla carta da inserire

6 5 3 1 8 7 2 4

```
public static void InsertionSort(int[] arr)
{
    for (int i = 1; i < arr.Length; i++)
        for (int j = i; j > 0 && arr[j] < arr[j - 1]; j--)
            if (arr[j] < arr[j - 1])
            {
                int temp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = temp;
            }

}
```

- Si utilizzano due indici: uno punta all'elemento da ordinare e l'altro all'elemento immediatamente precedente. Se l'elemento puntato dal secondo indice è maggiore di quello a cui punta il primo indice, i due elementi vengono scambiati di posto, altrimenti il primo indice avanza. L'algoritmo prende in considerazione un singolo elemento alla volta e, ad ogni iterazione, fa crescere la porzione ordinata dell'array. Il tutto si ripete finché non restano elementi da prendere in considerazione

Proprietà e Complessità Computazionale

STABILE
IN-PLACE
ADATTIVO

TEMPI DI ESECUZIONE

- **Caso peggiore:** l'array è ordinato in maniera inversa; ogni iterazione dovrà scorrere e spostare ciascun elemento della sottosequenza ordinata prima di poter inserire il primo elemento della sottosequenza non ordinata.
- **Caso migliore:** l'array è già ordinato ed in ogni iterazione il primo elemento della sottosequenza non ordinata viene confrontato solo con l'ultimo della sottosequenza ordinata;

Caso peggiore	$O(N^2)$
Caso medio	$O(N^2)$
Caso migliore	$O(N)$

SHELL SORT

Descrizione

- L’I**nsertion Sort** è efficiente per array parzialmente ordinati, ma inefficiente negli altri casi perché sposta gli elementi solo di una posizione per volta e spesso devono essere spostati più volte prima di arrivare nella posizione corretta.
- L’idea sulla quale si basa lo **Shell Sort** è che gli elementi vengono spostati di diverse posizioni alla volta man mano che risistema i valori, diminuendo gradualmente la dimensione del passo sino ad arrivare ad *1*. Alla fine lo Shell Sort eseguirà un Insertion Sort, ma i dati saranno già piuttosto ordinati e quindi il procedimento sarà efficiente.

```
public static void ShellSort(int[] arr)
{
    int h = 1;
    while (h <= arr.Length)  // Calcolo del passo iniziale.
        h = 3 * h + 1;
    h = h / 3;
    while (h >= 1)
    {
        // Questo è l'Insertion Sort con passo 'h'.
        for (int i = h; i < arr.Length; i++)
            for (int j = i; j >= h && arr[j] < arr[j - h]; j -= h)
            {
                int temp = arr[j];
                arr[j] = arr[j - h];
                arr[j - h] = temp;
            }
        h /= 3; // Passo successivo
    }
}
```

Proprietà e Complessità Computazionale

A differenza dell'**Insertion Sort** o del **Selection Sort**, lo **Shell Sort** è efficiente anche per grossi array.

NON STABILE
IN-PLACE
ADATTIVO

TEMPI DI ESECUZIONE

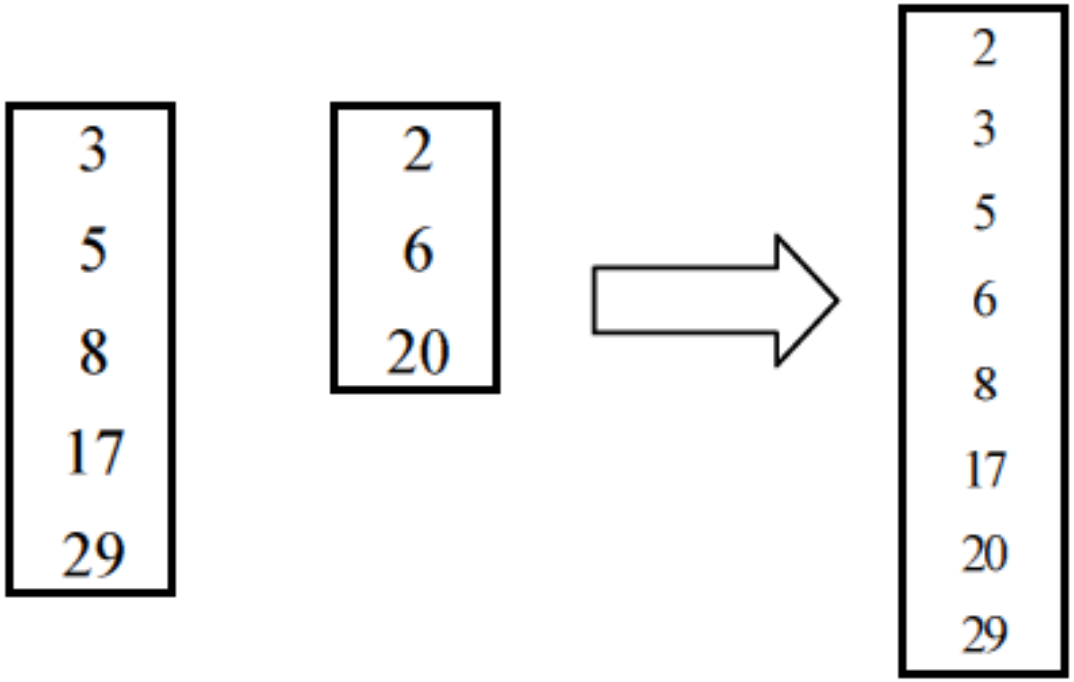
- Lo studio delle prestazioni di quest'algoritmo è complesso e richiede numerose dimostrazioni matematiche.
- Il **caso peggiore** dello Shell Sort è l'Insertion Sort base (usando un passo $h = 1$), che richiede $\mathbf{O(n^2)}$ confronti e scambi.

Caso peggiore	**$O(N^2)$**
Caso medio	**DIPENDE DAI DATI**
Caso migliore	**$O(N \log N)$**

MERGE SORT

Il problema della fusione

- Dati due array *ordinati in input, restituire in output un* terzo array *ordinato contenente gli elementi dei primi* due.



Esercizio difficile (ma risolto)

- Realizzare un programma che, dati in input due array , li ordini e poi li fonda in un unico vettore ancora ordinato.
- Suggerimento:
  1. vett[dim] , vett1[dim1], quindi vettM[dim+dim1]
  2. Utilizzare due indici diversi, i per vett e j per vett1, scandendoli in contemporanea, utilizzare per vettM k. Sia i, che j e k partono tutti da 0
  3. Ad ogni passo, se i<dim e j<dim1 confrontare vett[i] con vett1[j] e scegliere il valore minore da trasferire in vettM
  4. Se invece un array ha meno elementi dell'altro individuare quello che non è ancora finito e trasferire solo quegli elementi in vettM

Algoritmo Merge 1

```
//merge, se vett è composto da dim elementi
// e vett1 da dim1, ne segue che vettM sarà dim+dim1
int[] vettM = new int[dim+dim1];
i = 0;
j = 0;
k = 0;
while (i<dim && j<dim1)
{
    if (vett[i] <= vett1[j])
    {
        vettM[k] = vett[i];
        i++;
    }
    else
    {
        vettM[k] = vett1[j];
        j++;
    }
    k++;
}
```

Algoritmo Merge 2

```
if (i < dim)
{
    for (v = i; v < dim; v++)
    {
        vettM[k] = vett[v];
        k++;
    }
}
else
{
    for (v = j; v < dim1; v++)
    {
        vettM[k] = vett1[v];
        k++;
    }
}
Console.Write("vettM: ");
for (k = 0; k < dim+dim1; k++)
    Console.Write(vettM[k] + "\t");
```

ESERCIZIO GUIDATO

Esercizio1

- Un sito web del turismo trentino tiene un elenco aggiornato delle stazioni sciistiche e del manto nevoso (in cm, un intero). Si deve realizzare un programma che chieda in ingresso, per MAXDIM località, il nome di una località (al più 20 caratteri senza spazi), e l'altezza del manto nevoso (un intero).

Esercizio 1

1. Creare un vettore stazioni di stringhe e un vettore neve di interi (lung MAXDIM)
2. Leggere le stazioni e i cm neve da console
3. Ordinare il vettore neve (utilizzare i due array come vettori paralleli)
4. Scrivere il metodo Confronta (località1, località2) che ritorna -1 , 0 , 1 a seconda che in località1 ci sia più neve, la stessa quantità o meno neve che in località2

Esercizio 2

- Si scriva un programma che per ciascuno degli algoritmi di ordinamento visti (bubble, selection, insertion e shell) riempia un vettore di N valori interi casuali (es. 1000) e lo riordini. Il programma deve misurare i tempo impiegato per gli ordinamenti per ciascuno degli algoritmi. Impostare N in modo da eseguire un numero significativo di prove con vettori di dimensioni grandi (1000-10000), medi (100-1000) e piccoli (100). Trarne le conclusioni.

Come calcolare il tempo trascorso

1. Aggiungere using System.Diagnostics;
2. Dichiarare e istanziare un'oggetto della classe StopWatch
   sw = new StopWatch();
3. Per attivarlo (farlo partire)

```
sw.Start();
```

4. Per disattivarlo (fermarlo)

```
sw.Stop();
```

5. Per sapere quanto tempo è passato tra Start e Stop si scrive

```
Console.WriteLine("tempo: {0}", sw.ElapsedMillisecond);
```

Per reset sw.Reset();