SOMMARIO

- Algoritmi di ricerca:
  - Ricerca lineare;
  - Ricerca binaria (su elenchi già ordinati).
- Algoritmi di ordinamento:
  - Ordinamento per selezione;
  - Ordinamento veloce.

IL PROBLEMA DELLA RICERCA

- *Dato un array e un oggetto, stabilire se l'oggetto è contenuto in un elemento dell'array, riportando in caso affermativo l'indice di tale elemento.*
- Consideriamo il caso generale di un array non ordinato. Allora gli elementi dell'array vengono considerati uno dopo l'altro finché o il valore cercato è trovato o la fine dell'array è raggiunta.

RICERCA LINEARE

```
public static int SequentialSearch(int[]
                    data, int key)
{
    for (int i = 0; i < data.Length; i++)
        if (data[i] == key)
            return i;
    return -1;
}
```

- L'analisi della complessità asintotica si complica perchè il numero di esecuzioni del ciclo dipendono dalla disposizione dei valori nell'array.

RICERCA LINEARE : complessità asintotica

- Caso migliore: la ricerca richiede un solo confronto, quindi si ha *O(1)*.
- Caso peggiore: la ricerca richiede *n* confronti, quindi si ha *O(n)*.
- Caso medio: se la chiave è presente, può occupare qualsiasi posizione con la stessa probabilità, $P_i=1/n$. Quindi si devono considerare le possibili situazioni in ingresso, determinare il numero di passi eseguiti in ogni situazione e poi fare una media pesata da P_i:

$$T(n) = \sum_{i=1}^{n} P_i i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n+1)}{2} = O(n)$$

ESEMPIO: ricerca

- Consideriamo una classe `Studente` con i campi `nome` e `matricola`. Sviluppiamo un esempio completo per la ricerca di uno studente all'interno di un elenco secondo diversi criteri.

- Poiché i dati da confrontare sono oggetti, è necessario che un *criterio di confronto* venga fornito per mezzo delle interfacce `IComparable` e `IComparer`.

ESEMPIO: ricerca

```
class Studente: IComparable
{
    private string nome;
    private int matricola;

    public Studente(string nome, int matricola){
        this.nome = nome;
        this.matricola = matricola;
    }
    public int Matricola{
        get { return matricola; }
    }
    public override string ToString(){
        return nome + " " + matricola;
    }
    public int CompareTo(object obj){
        if (obj is Studente)
        {
            return nome.CompareTo(((Studente)obj).nome);
        }
        throw new ArgumentException("obj non e` uno Studente");
    }
}
```

Polimorfismo: dynamic binding.

Questo `CompareTo()` è definito in `string`.

ESEMPIO : ricerca

```
using System.Collections;
class StudenteComparatore: IComparer
{
    public int Compare(object o1, object o2)
    {
        if (o1 is Studente && o2 is Studente)
        {
            return ((Studente)o1).Matricola -
                                ((Studente)o2).Matricola;
        }
        throw new ArgumentException("o1 e o2 non
                                        compatibili");
    }
}
```


ESEMPIO : ricerca

Algoritmo polimorfico

```
class Algoritmi
{
    public static int SequentialSearch(object[] data,
                                                object key)
    {
        for (int i = 0; i < data.Length; i++)
            if (((IComparable)key).CompareTo(data[i]) == 0)
                return i;
        return -1;
    }

    public static int SequentialSearch(object[] data,
                                    object key, IComparer c)
    {
        for (int i = 0; i < data.Length; i++)
            if (c.Compare(data[i], key) == 0)
                return i;
        return -1;
    }}
```

Controllare che l'oggetto sia valido per il confronto.

Overloading

ESEMPIO : ricerca

```
static void Test1()
{
    Studente[] s ={new Studente("Pippo",3),
      new Studente("Anna",9),
      new Studente("Carlo",2),
      new Studente("Simona",16),
      new Studente("Anna",99),
      new Studente("Gianna",12)};

→   Studente key = new Studente("Simona", 16);

→   int i = Algoritmi.SequentialSearch(s, key);
    if (i != -1)
        Console.WriteLine("[{0}]: {1}", i, s[i]);
}
```


ESEMPIO : ricerca

- L'esempio precedente stampa: [3]: Simona 16
- Se si usa la chiave

```
Studente key = new Studente("Anna",99);
```

l'esempio stampa: [1]: Anna 9

- Se si usa il comparatore

```
int i = Algoritmi.SequentialSearch(s, key,
                new StudenteComparatore());
```

l'esempio stampa: [4]: Anna 99

RICERCA BINARIA

Se l'elenco è ordinato, si può seguire una diversa *strategia* per cercare la chiave nell'elenco.
Per prima cosa, si confronta la chiave con l'elemento centrale dell'array, se c'è corrispondenza la ricerca è finita.
Altrimenti si decide di continuare la ricerca nella metà destra o sinistra rispetto l'elemento considerato (l'array è ordinato, quindi la chiave, se presente, è maggiore o minore dell'elemento considerato) e si utilizza di nuovo la stessa strategia.
Tale algoritmo ha una formulazione elegante in termini ricorsivi.

RICERCA BINARIA

```
static int BinarySearch(object[] data, object key, int first,
                                                    int last)
{
    if (first > last)
        return -1;
    else
    {
        int mid = (first + last) / 2;

        if (((IComparable)key).CompareTo(data[mid]) == 0)
            return mid;
        else if (((IComparable)key).CompareTo(data[mid]) < 0)
            return BinarySearch(data, key, first, mid - 1);
        else
            return BinarySearch(data, key, mid + 1, last);
    }
```

RICERCA BINARIA

- Per il calcolo della complessità computazionale si consideri che lo spazio di ricerca viene ripetutamente dimezzato finché non rimane un unico elemento da confrontare (caso peggiore).
- Pertanto dopo *k* iterazioni si arriva ad avere $n/2^k = 1$ da cui $k = log_2 n$.
- L'algoritmo di ricerca binaria è molto efficiente avendo una complessità logaritmica: *O(log n)*.
- Tuttavia tale algoritmo si applica ad elenchi ordinati, pertanto diventa importante ordinare i dati prima di manipolarli.

IL PROBLEMA DELL'ORDINAMENTO

- *Dato un array i cui elementi contengono tutti una chiave di ordinamento e data una relazione d'ordine sul dominio delle chiavi, determinare una permutazione degli oggetti contenuti negli elementi dell'array tale che la nuova disposizione delle chiavi soddisfi la relazione d'ordine.*

- Vediamo per primo un algoritmo di ordinamento intuitivo ma poco efficiente, quindi utile in pratica solo per elenchi composti da poche voci.

ORDINAMENTO PER SELEZIONE

- L'idea di base è trovare l'elemento con il valore *minore* e scambiarlo con l'elemento nella prima posizione. Quindi si cerca il valore minore fra gli elementi rimasti (escludendo la prima posizione) e lo si scambia con la seconda posizione. Si continua finché tutti gli elementi sono nella posizione corretta.

- Vediamo una possibile implementazione.

ORDINAMENTO PER SELEZIONE

```
public static void SelectionSort(object[] data)
{
    int i, j, least;
    for (i = 0; i < data.Length - 1; i++)
    {
        for (j = i + 1, least = i; j < data.Length; j++)
        {
            if (((IComparable)data[j]).CompareTo(data[least])<0)
                least = j;
        }
        Swap(data, least, i);
    }
}

static void Swap(object[] d, int e1, int e2)
{
    object tmp = d[e1]; d[e1] = d[e2]; d[e2] = tmp;
}
```

ORDINAMENTO PER SELEZIONE

```
static void Test2()
{
    Studente[] s ={new Studente("Pippo",3),
        new Studente("Anna",9),
        new Studente("Carlo",2),
        new Studente("Simona",16),
        new Studente("Anna",99),
        new Studente("Gianna",12)};

    Algoritmi.SelectionSort(s);
    Print(s);
}


    ...
```


ORDINAMENTO PER SELEZIONE

```
...
static void Print(object[] s)
{
    for (int i = 0; i < s.Length; i++)
        Console.WriteLine(s[i]);
}
```

Questo esempio stampa l'array ordinato secondo il criterio implementato da `CompareTo()`: in ordine alfabetico.

Anna 9
Anna 99
Carlo 2
Gianna 12
Pippo 3
Simona 16

ORDINAMENTO PER SELEZIONE

Se si usa il comparatore:

```
Algoritmi.SelectionSort(s,
        new  StudenteComparatore());
```

Questo esempio stampa l'array ordinato secondo il criterio implementato da `Compare()`: in ordine di matricola.

Carlo 2
Pippo 3
Anna 9
Gianna 12
Simona 16
Anna 99

ORDINAMENTO PER SELEZIONE

- Per l'analisi delle prestazioni di tale algoritmo di ordinamento, si considerano i due cicli `for` annidati: poiché i *confronti* avvengono nel ciclo interno si ha che

$$\sum_{i=0}^{n-2}(n-1-i) = n(n-1)/2 = O(n^2)$$

- Quindi una *complessità quadratica*. Per applicazioni pratiche con un elevato numero di dati è necessario trovare una soluzione migliore.

ORDINAMENTO PER SELEZIONE

- Solitamente si tiene conto solo dei confronti tra i *dati* e non tra *indici*, perchè il confronto tra indici è trascurabile se i dati sono *oggetti* complessi.

- Risulta anche interessante valutare il numero di *scambi* effettuati da un algoritmo. L'ordinamento per selezione ha una *complessità lineare* sugli scambi, un risultato buono.

ORDINAMENTO VELOCE

- Consideriamo un algoritmo molto diffuso: il *quicksort.* La versione di base è stata sviluppata da Hoare nel 1960.

- È facilmente implementabile e ha una complessità media *O(n log n)*.

- L'algoritmo di base presenta alcuni svantaggi, tuttavia è stato migliorato al punto da diventare il metodo ideale per un gran numero di applicazioni (una versione è usata nella classe `Array` di C#).

ORDINAMENTO VELOCE

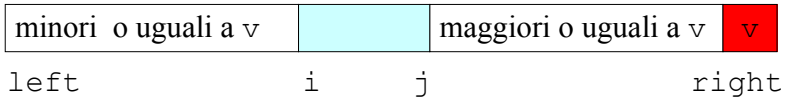
- I principali svantaggi dell'algoritmo di base sono i seguenti:
  - Nel caso peggiore ha una complessità quadratica.
  - Non è stabile. Un algoritmo di ordinamento è *stabile* se non altera l'ordine relativo di elementi dell'array aventi la stessa chiave.
- I principali vantaggi sono:
  - Una complessità media *O(n log n)*, che è ottima.
  - Opera sul *posto*: la dimensione delle "variabili ausiliarie" di cui ha bisogno è indipendente dalla dimensione dell'array da ordinare.

ORDINAMENTO VELOCE

- Il *quicksort* è un metodo che opera *partizionando* un array in *due parti* da ordinare indipendentemente.
- Lo scopo del partizionamento è quello di *riorganizzare* l'array in modo tale che: dato *l'elemento di partizionamento,* tutti gli elementi precedenti siano minori o uguali e quelli successivi siano maggiori o uguali di tale elemento di partizionamento.
- Poi si applica *ricorsivamente* ai due sotto-array lo stesso procedimento.
- Tale approccio è del tipo *divide et impera*.

ORDINAMENTO VELOCE

- La scelta dell'*elemento di partizionamento* avviene, in questa versione di base, in modo del tutto arbitrario (per es. elemento più a destra).
- Per *riorganizzare* l'array la strategia è quella di partire dai due estremi e scambiare gli elementi fuori posto.



- Dopo che gli indici di scansione `i` e `j` si sono incrociati, si deve porre l'*elemento di partizionamento* `v` nella posizione corretta, cioè scambiarlo con l'elemento più a sinistra della porzione a destra dell'array, e memorizzare tale indice `p` di partizionamento.

ORDINAMENTO VELOCE

- Vediamo un esempio con un array di lettere:

```
                  O R D I N A L A F R A S E

[0,12]   p=4      A A D A E I L R F R O S N
[0,3]    p=1      A A D A - - - - - - - - -
[2,3]    p=2      - - A D - - - - - - - - -
[5,12]   p=8      - - - - - I L F N R O S R
[5,7]    p=5      - - - - - F L I - - - - -
[6,7]    p=6      - - - - - - I L - - - - -
[9,12]   p=10     - - - - - - - - - O R S R
[11,12]  p=11     - - - - - - - - - - - R S

                  A A A D E F I L N O R R S
```


ORDINAMENTO VELOCE

```
public static void QuickSort(object[] data,
                             int left, int right)
{
    if (left >= right) return;

    int p = Partition(data, left, right);

    QuickSort(data, left, p - 1);
    QuickSort(data, p + 1, right);
}
```

L'efficienza dell'ordinamento dipende da quanto è bilanciato il partizionamento e quindi dal valore dell'elemento di partizionamento.

Notare che la ricorsione avviene sempre su array con dimensioni strettamente inferiori a quelle di partenza.

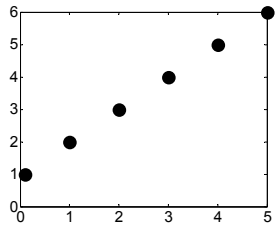
ORDINAMENTO VELOCE

```
public static int Partition(object[] data, int left,
                                                int right)
{
    int i = left - 1, j = right;
    IComparable v = (IComparable)data[right];

    for (; ; )
    {
        while (((IComparable)data[++i]).CompareTo(v) < 0) ;
        while (v.CompareTo(data[--j]) < 0)
            if (j == left) break;
        if (i >= j) break;
        Swap(data, i, j);
    }

    Swap(data, i, right);
    return i;
}
```

CARATTERISTICHE DINAMICHE

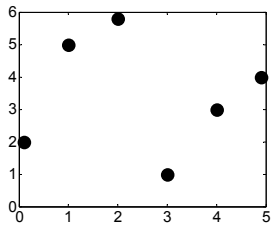
- Visualizziamo graficamente le caratteristiche dinamiche degli algoritmi di ordinamento.
- Il grafico rappresenta i valori dell'array `v` in funzione dell'indice:

caso ordinato




```
int[] v={1,2,3,4,5,6};
```

caso non ordinato

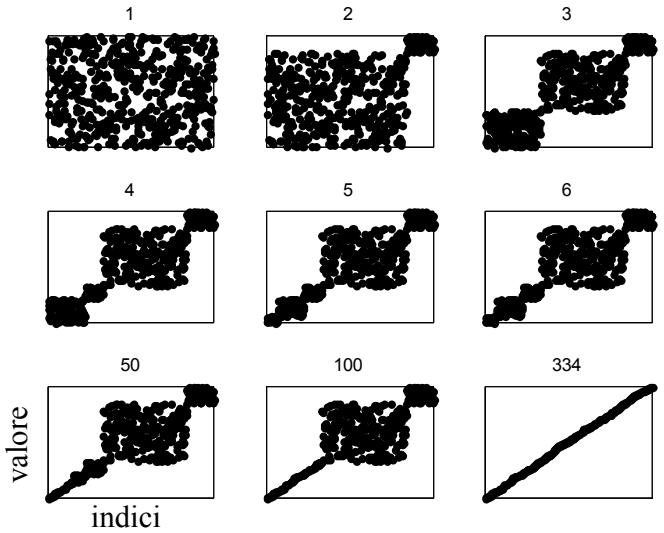



```
int[] v={2,5,6,1,3,4};
```


CARATTERISTICHE DINAMICHE

- Vediamo l'ordinamento di un array di 500 elementi, ogni elemento è un valore intero casuale scelto nell'intervallo [0,999].
- I grafici rappresentano la disposizione dei valori dell'array ad uno specifico passo del *processo di ordinamento*, il passo è indicato sopra ogni grafico.

ORDINAMENTO VELOCE



ORDINAMENTO PER SELEZIONE