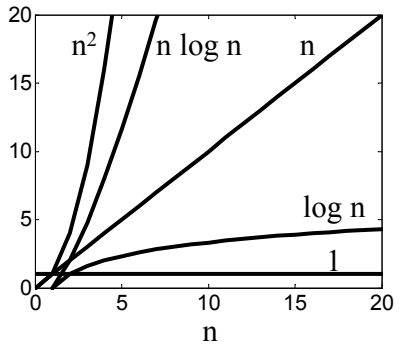
- Uno stesso problema può essere risolto da algoritmi di diversa *efficienza*: tale efficienza diventa rilevante quando la quantità di dati da manipolare diventa "grande".
- La *complessità computazionale* o efficienza può essere valutata rispetto all'uso delle seguenti risorse: *spazio* e *tempo*. In generale il fattore tempo è quello critico.
- Il *tempo di esecuzione* di un algoritmo viene espresso come una funzione della dimensione dei dati in ingresso: *T(n)*.

- Tuttavia il tempo di esecuzione di un'applicazione dipende dal sistema e dal linguaggio di implementazione.
- Allora si usa il numero di *passi base* compiuti durante l'esecuzione dell'algoritmo: per es. istruzioni di assegnamento e di confronto.
- Poiché tale misura interessa solo per grandi quantità di dati, si fa riferimento all'ordine di grandezza: *complessità asintotica.*

- Per indicare la *complessità asintotica* si usa la notazione *O-grande (Big-Oh):*
  - date due funzioni a valori positivi *f* e *g*, si dice che *f(n)* è *O(g(n))* se esistono due numeri positivi *c* ed *N* tali che $f(n) \leq cg(n)$ per qualsiasi $n \geq N$.
- Per esempio: $f(n)=2n^2 + 3n +1=O(n^2)$

Alcune classi di *complessità asintotica:*

- O(1): costante, non dipende dalla dimensione dei dati
- O(log n): logaritmica
- O(n): lineare
- O(n log n): pseudolineare
- O(n^2): quadratica



ANALISI DELLA COMPLESSITÀ DEGLI ALGORITMI

- Analizzare la complessità degli algoritmi è di estrema importanza nonostante le elevate prestazioni dei computer attuali. Algoritmi mal progettati non hanno applicazioni pratiche.

- Per esempio, su un computer che esegue un miliardo di operazioni al secondo un algoritmo pseudolineare impiega alcuni decimi di secondo a manipolare dieci milioni di dati, mentre un algoritmo quadratico impiega ventisette ore!

ESEMPI DI COMPLESSITÀ ASINTOTICA

- Un ciclo per calcolare la somma dei valori di un array:

```
int[] a = ...
int i,j,sum;
for(i=0, sum=0; i<a.length;i++)
      sum+=a[i];
```

- Vengono inizializzate due variabili, poi il ciclo itera n volte (dove n e la dimensione dell'array) eseguendo due assegnamenti, uno per aggiornare `sum` e l'altro per incrementare `i`. Quindi si ha:

  $T(n)= 2+ n\ (1+1) =2 +2\ n = O(n)$

  che è una complessità asintotica lineare.

ESEMPI DI COMPLESSITÀ ASINTOTICA

- La complessità solitamente cresce quando si usano cicli annidati. Per esempio calcolare tutte le somme di tutti i sotto-array che iniziano dalla posizione 0:

```
for(i=0; i<a.length;i++){
      for(j=1,sum=a[0]; j<=i;j++)
      sum+=a[j];
System.out.println("Somma sub "+sum);}
```

- Il ciclo più esterno esegue tre assegnamenti (`i`, `j` e `sum`) e il ciclo interno, che esegue, a sua volta, *2 i* assegnamenti (`sum` e `j`) per ogni $i \in [1,...,n-1]$. Quindi si ha:

$$T(n) = 1 + 3n + \sum_{i=1}^{n-1} 2i = 1 + 3n + n(n-1) = O(n^2)$$

ESEMPI DI COMPLESSITÀ ASINTOTICA

- Anche in presenza di cicli annidati, non è detto che la complessità cresca. Per esempio eseguire la somma dei valori che si trovano nelle ultime cinque celle dei sotto-array che partono da 0:

```
for(i=4; i<a.length;i++){
      for(j=i-3,sum=a[i-4]; j<=i;j++)
      sum+=a[j];
System.out.println("Somma sub "+sum);}
```

- Il ciclo più interno esegue sempre lo stesso numero di assegnamenti per ogni iterazione del ciclo esterno, pertanto si ha:

  $T(n)= 1+ (n-4)(1 + 2 + 4\cdot 2) = O(n)$

COMPLESSITÀ ASINTOTICA

- L'analisi si complica quando il numero di esecuzioni di un ciclo dipendono dall'ordinamento dell'array.
- In generale si distinguono tre casi. *Caso peggiore*: l'algoritmo richiede il numero massimo di passi. *Caso migliore*: l'algoritmo richiede il numero minimo di passi. Il *caso medio* si pone tra questi estremi: si dovrebbero considerare le possibili situazioni in ingresso, determinare il numero di passi eseguiti in ogni situazione e poi fare una media. Supponendo che le probabilità siano uguali per tutti gli ingressi, ma questo non è sempre vero.

RICERCA E ORDINAMENTO: SOMMARIO

- Considerazioni sul confronto tra oggetti.
- 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;
  }
```

- La complessità di questo algoritmo è in media lineare: O(n).

CONFRONTI

- Questo algoritmo è implementato per gli interi e, in generale, per i tipi di dati di base il confronto è ovvio, ma per gli oggetti è necessario che un *criterio di confronto* venga fornito dall'utente.

- Considerando una classe `Studente` con i campi `nome` e `matricola`, in generale, non è ovvio come confrontare due oggetti `Studente`. Per questa ragione ogni classe dovrebbe implementare il metodo `compareTo()`, la cui firma si trova nell'interfaccia `Comparable`.

CONFRONTI

- Il metodo `compareTo()` si usa per un "confronto naturale", se si vuole applicare un altro criterio si definisce un comparatore per la classe, implementando l'interfaccia `Comparator` che contiene la firma del metodo `compare()`.

- Sviluppiamo un esempio completo per la ricerca di uno studente all'interno di un elenco secondo diversi criteri.

ESEMPIO

```
public class Studente implements Comparable{
   String nome;
   int matricola;

   public Studente(String nome, int matricola){
       this.nome=nome;
       this.matricola=matricola;
   }
   public String toString(){
       return nome+" "+matricola;
   }
   public int compareTo(Object o){
       Studente tmp = (Studente) o;
       return nome.compareTo(tmp.nome);
   }}
```

Polimorfismo: dynamic binding e static binding

ESEMPIO

```
import java.util.*;
public class StudenteComparator
                implements Comparator{
  public int compare(Object o1, Object o2){
      return ((Studente)o1).matricola -
                ((Studente)o2).matricola;
  }
}
```

ESEMPIO

Algoritmo polimorfico

```
import java.util.*;
public class Ricerca {
   public static int sequentialSearch(Object[] data,
                                      Object key){
        for(int i=0;i<data.length;i++)
            if (((Comparable)key).compareTo(data[i])==0)
                  return i;
        return -1;
   }
```

Overloading

```
   public static int sequentialSearch(Object[] data,
                          Object key, Comparator c){
        for(int i=0;i<data.length;i++)
            if (c.compare(data[i],key)==0)
                  return i;
        return -1;
   }
```


ESEMPIO

```
public class RicercaTest {
   public static void main(String[] args){
        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 out = Ricerca.sequentialSearch(s,key);

        if (out==-1)
              System.out.println("Elemento non trovato");
        else
              System.out.println(s[out]);
        }}
```


ESEMPIO

- L'esempio precedente stampa: Simona 16
- Se usassi la chiave

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

  l'esempio stamperebbe: Anna 9
- Se usassi il comparatore

```
int out = Ricerca.sequentialSearch(s,key,
                  new StudenteComparator());
```

l'esempio stamperebbe: 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 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 (((Comparable)key).compareTo(data[mid])==0)
            return mid;
      else if (((Comparable)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 fino a restare con un unico elemento da confrontare.
- 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

```
import java.util.*;

public class Ordinamento {
   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(((Comparable)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

```
public class OrdinamentoTest {
  public static void main(String[] args){
      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)};
      Ordinamento.selectionSort(s);
      print(s);
}
  …
```


ORDINAMENTO PER SELEZIONE

```
…
  static void print(Object[] s){
      for(int i=0; i<s.length;i++)
            System.out.println(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:

```
Ordinamento.selectionSort(s,new
            StudenteComparator());
```

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

L'algoritmo per selezione ha una complessità quadratica: $O(n^2)$.