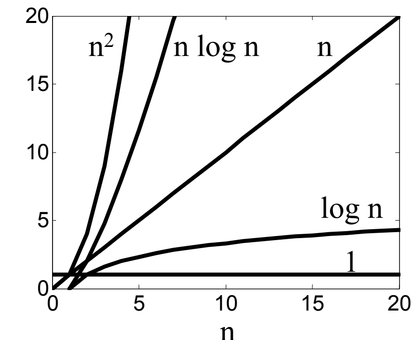
- Uno stesso problema può essere risolto da algoritmi di diversa *efficienza*: tale efficienza diventa rilevante quando la quantità di dati da manipolare diventa "grande".
- La *complessità computazionale* o efficienza può essere valutata rispetto all'uso delle seguenti risorse: *spazio* e *tempo*. In 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 $f(n)<=cg(n)$ per qualsiasi $n>=N$.
- 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,n=a.Length;
for(i=0, sum=0; i<n ;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<n ;i++){
      for(j=1,sum=a[0]; j<=i;j++)
      sum+=a[j];
Console.WriteLine ("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<n ;i++){
      for(j=i-3,sum=a[i-4]; j<=i;j++)
      sum+=a[j];
Console.WriteLine ("Somma sub "+sum);}
```

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

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

PROGETTAZIONE DI ALGORITMI

- È importante prestare attenzione alla realizzazione degli algoritmi al fine di ottenere una elevata efficienza di esecuzione.
- Per esempio, l'algoritmo per calcolare tutte le somme di tutti i sotto-array che iniziano dalla posizione 0 ha *complessità quadratica*: questo lo rende poco adatto ad applicazioni reali con dati in ingresso oltre una certa dimensione.
- Tuttavia è sufficiente prestare attenzione alla sua formulazione per poterne realizzare una versione a *complessità lineare*: calcolata la somma di un sotto-array, per calcolare la successiva somma è sufficiente aggiungere l'elemento consecutivo, evitando in tal modo di calcolare esplicitamente la somma di tutto il successivo sotto-array.

PROGETTAZIONE ALGORITMI

```
public static void SommaO2(double[] a)
{
    double somma = 0;
    int j = 0;
    for (int i = 0; i < a.Length; i++)
        for (j = 1, somma = a[0]; j <= i; j++)
            somma += a[j];
}
public static void SommaO1(double[] a)
{
    double somma = 0;
    for (int i = 0; i < a.Length; i++)
            somma += a[i];
}
```

È il valore `n` del `Main()`

PROGETTAZIONE ALGORITMI

```
Random rnd = new Random();
DateTime start, finish;
TimeSpan t;
int n = 10000;

for (int k = 2; k < 8; k++)
{
    n *=2;
    double[] v = new double[n];
    for (int i = 0; i < n; i++)
        v[i] = rnd.NextDouble();
    Console.Write("{0:E1} |\t", n);

    start = DateTime.Now;
    SommaO2(v);
    finish = DateTime.Now;
    t = finish.Subtract(start);
    Console.Write("{0:E3} |\t", t.TotalMilliseconds / 1000.0);

    start = DateTime.Now;
    SommaO1(v);
    finish = DateTime.Now;
    t = finish.Subtract(start);
    Console.WriteLine("{0:E3} |\t", t.TotalMilliseconds / 1000.0);
}
```

Una possibile uscita

n	O2	O1
2.0E+004 |	1.703E+000 |	0.000E+000 |
4.0E+004 |	6.844E+000 |	0.000E+000 |
8.0E+004 |	2.764E+001 |	0.000E+000 |
1.6E+005 |	1.108E+002 |	0.000E+000 |
3.2E+005 |	4.438E+002 |	0.000E+000 |
6.4E+005 |	1.782E+003 |	1.563E-002 |

COMPLESSITÀ ASINTOTICA

- L'analisi si complica quando il numero di esecuzioni di un ciclo dipendono dalla disposizione dei valori nell'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.