Cenni di Complessità computazionale

Complessità computazionale

- **Analisi della complessità**: stima del costo degli algoritmi in termini di risorse di calcolo
  - Tempo, spazio di memoria
- Esempio: dato un vettore v di n interi ordinati in maniera non decrescente verificare se un intero k è presente o meno in v

Ricerca sequenziale

```
int ricerca(int v[], int size, int k)
{   int i;
    for (i=0; i<size; i++)
      if (v[i] == k) return i;
    return -1;
}
```

… nessun vantaggio dal fatto che v è ordinato …

Valutazione del tempo di esecuzione

- Variabili che influenzano il tempo di esecuzione
  - La macchina usata
  - La dimensione dei dati
  - La configurazione dei dati
- Modello astratto per la valutazione del tempo
  - Indipendente dalla macchina usata
  - Stima in funzione della dimensione dell'input
  - Comportamento asintotico
  - Stima del caso peggiore di configurazione dei dati

Esempio di macchina astratta

- Istruzioni e condizioni atomiche hanno costo unitario
- Le strutture di controllo hanno un costo pari alla somma dei costi dell'esecuzione delle istruzioni interne, più la somma dei costi delle condizioni
- Le chiamate a funzione
  - hanno un costo pari al costo di tutte le sue istruzioni e condizioni;
  - il passaggio dei parametri ha costo nullo
- Istruzioni e condizioni con chiamate a funzioni hanno costo pari alla somma del costo delle funzioni invocate più uno

Esempio

- Calcolare il costo per l'esempio precedente nel caso v[n] = {1, 3, 9, 17, 34, 95,96, 101} e k=9

```
int ricerca(int v[], int size, int k)
{  int i;
   for (i=0; i<size; i++)
     if (v[i] == k) return i;
   return -1;
}
```

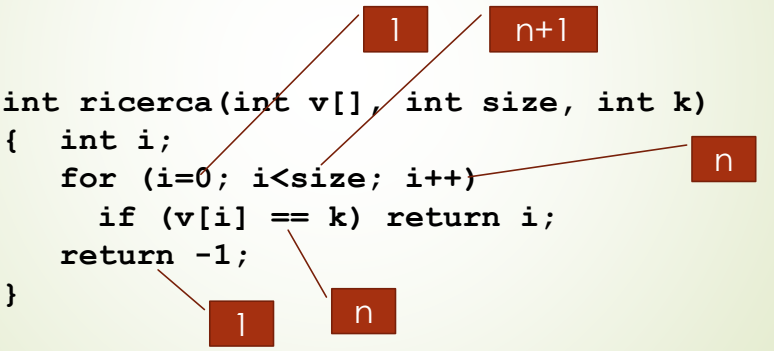
1 3 2 3 1

Totale = 10

Cosa cambia se k=10 ?

Caso peggiore

- Caso che a parità di dimensione produce il costo massimo
  - Se accettabile nel caso peggiore …
- Nel caso dell'esempio, k non presente




```
int ricerca(int v[], int size, int k)
{  int i;
   for (i=0; i<size; i++)
     if (v[i] == k) return i;
   return -1;
}
```

Totale = 3n + 3

Caso medio

- Supponiamo che il numero cercato sia presente e che ci sia equiprobabilità dell'input
  - La probabilità che k sia in posizione i (1<=i<=n) vale 1/n
  - Costo del caso in posizione i: 3i+1
  - Costo caso medio:

$$\frac{1}{n}\sum_{i=1}^{n}(3i+1)=\frac{1}{n}\left(3\frac{n^2+n}{2}+n\right)=$$

$$=\frac{3n+5}{2}$$

Costo come funzione della dimensione dell'input

- Cosa è la dimensione ?
  - Vettore … numero di elementi
  - Albero ? … numero dei nodi
  - Grafo ? … numero archi più numero nodi
- Esempio: calcolo del fattoriale, con tipo intero non limitato

Esempio

```
int fattoriale(int n)
{ int i = 1;
  int fatt = 1;
  while (i <=n) {
      fatt = fatt * i;
      i++; }
  return fatt;
}
```

1
1
n+1
n
n
1

Totale = 3n + 4

11

Dimensione dell'input

- Parametro n
  - Costo = 3n+4 (lineare)
- Numero d di bit necessari per rappresentare n:
  - $d \approx \log_2 n$
  - Costo = $3 \times 2^d + 4$ (esponenziale)

12

Comportamento asintotico

- Nell'analizzare la complessità di tempo di un algoritmo siamo interessati a come aumenta il tempo al crescere della taglia *n* dell'input.
- Siccome per valori "piccoli" di *n* il tempo richiesto è comunque poco, ci interessa soprattutto il comportamento per valori "grandi" di n (il comportamento asintotico)

13

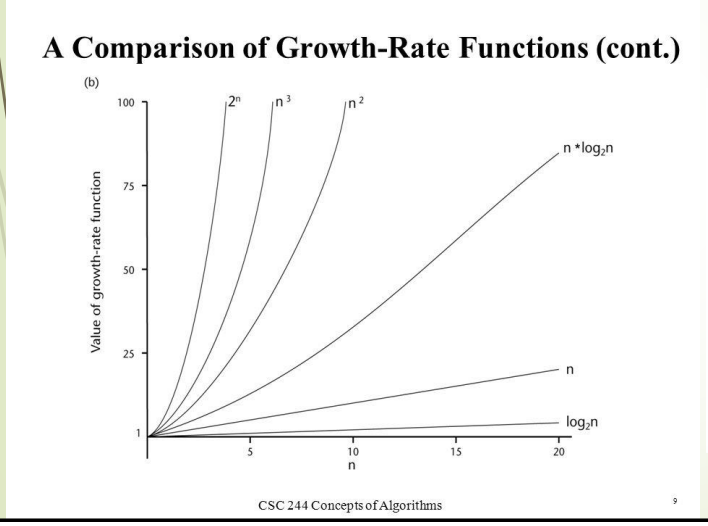
Comportamento asintotico

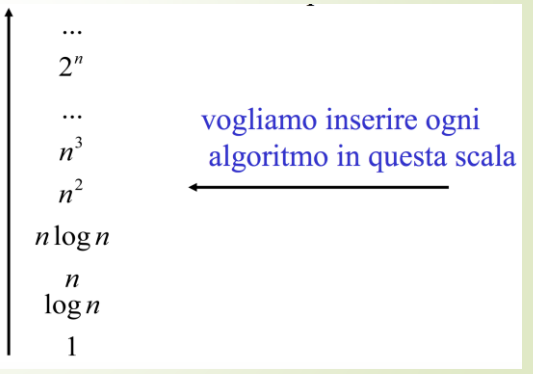
- Comportamento al crescere della dimensione n dei dati all'infinito
  - Trascurare tutte le costanti moltiplicative ed additive e tutti i termini di ordine inferiore
  - Suddivisione di algoritmi in classi di complessità
    - a — costante
    - a n+b — lineare
    - a n^2 + bn + c — quadratica
    - a $\log_g$ n + h — logaritmica
    - a^n — esponenziale
    - n^n — esponenziale

14

Comportamento asintotico

- Abbiamo una scala di complessità:





Comportamento asintotico

- Supponiamo di avere, per uno stesso problema, sette algoritmi diversi con diversa complessità.
- Supponiamo che un passo base venga eseguito in un microsecondo (10^{-6} sec).

Tempi di esecuzione (in secondi) in base a n.

	n=10	n=100	n=1000	n=10^6
$\sqrt{n}$	$3*10^{-6}$	10^{-5}	$3*10^{-5}$	10^{-3}
$n+5$	$15*10^{-6}$	10^{-4}	10^{-3}	1 sec
$2*n$	$2*10^{-5}$	$2*10^{-4}$	$2*10^{-3}$	2 sec
n^2	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
n^2+n	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
n^3	10^{-3}	1 sec	10^5 (~1g)	10^{12} (~300 secoli)
2^n	10^{-3}	~$4*10^{14}$ secoli	~$3*10^{287}$ secoli	~$3*10^{301016}$ secoli

Comportamento asintotico

- Per piccole dimensioni dell'input, osserviamo che tutti gli algoritmi hanno tempi di risposta non significativamente differenti.
- L ' algoritmo di complessità esponenziale ha tempi di risposta ben diversi da quelli degli altri algoritmi (migliaia di miliardi di secoli contro secondi, ecc.)
- Per grandi dimensioni dell'input (n=10^6), i sette algoritmi si partizionano nettamente in cinque classi in base ai tempi di risposta:
  - Algoritmo rad(n) — frazioni di secondo
  - Algoritmo n+5, $2*n$ — *secondi*
  - Algoritmo n^2, n^2+n — *giorni*
  - Algoritmo n^3 — secoli
  - Algoritmo 2^n — miliardi di secoli

17

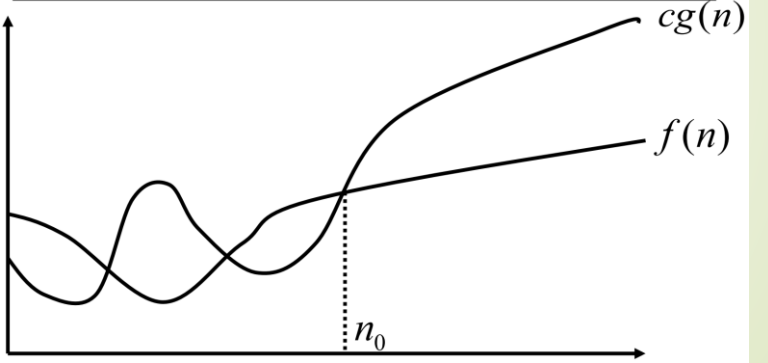
Notazione O ed Ω

- f e g funzioni dai naturali ai reali positivi
- f(n) è O di g(n), $f(n) \in O(g(n))$, se esistono due costanti positive c ed n_o tali che se $n >= n_o$, $f(n) <= c\ g(n)$
  - Applicata alla funzione di complessità f(n),la notazione O ne limita superiormente la crescita e fornisce quindi una indicazione della bontà dell'algoritmo
- f(n) è Omega di g(n), $f(n) \in \Omega(g(n))$, se esistono due costanti positive c ed n_o tali che se $n >= n_o$, $c\ g(n) <= f(n)$
  - La notazione Ω limita inferiormente la complessità, indicando così che il comportamento dell'algoritmo non è migliore di un comportamento assegnato

18

Notazione asintotica O (limite superiore asintotico)

$$O(g(n)) = \{f(n) : \text{esistono } c > 0 \text{ ed } n_0 \text{ tali che } 0 \le f(n) \le cg(n) \text{ per ogni } n \ge n_0\}$$



Esempi

$$f(n) = 2n^2 + 5n + 5 = O(n^2)$$

infatti $0 \le 2n^2 + 5n + 5 \le cn^2$
per $c = 4$ ed $n_0 = 5$

Vedremo che in generale per $a_2 > 0$

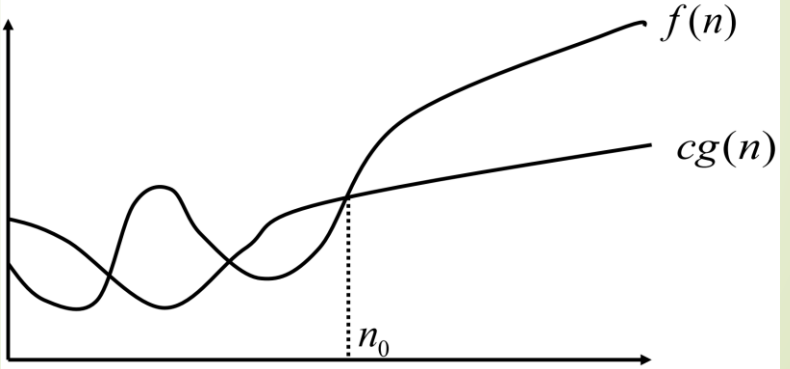
$$f(n) = a_2 n^2 + a_1 n + a_0 = O(n^2)$$

$$f(n) = 2 + \sin n = O(1)$$

infatti $0 \le 2 + \sin n \le c \cdot 1$
per $c = 3$ ed $n_0 = 1$

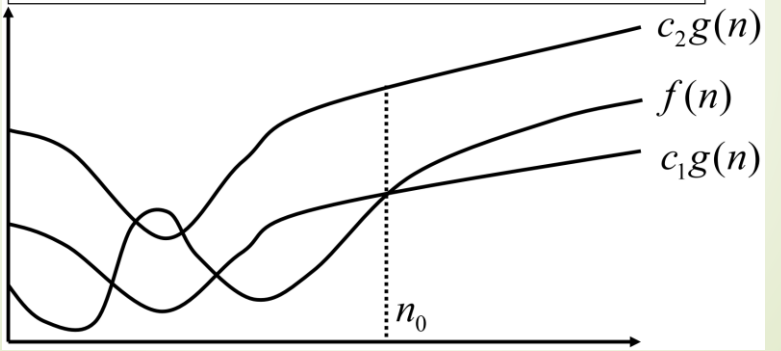
Notazione asintotica Ω (limite inferiore asintotico)

$$\Omega(g(n)) = \{f(n) : \text{esistono } c > 0 \text{ ed } n_0 \text{ tali che } f(n) \ge cg(n) \ge 0 \text{ per ogni } n \ge n_0\}$$



Notazione asintotica Θ (limite asintotico stretto)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) = \{f(n) : \text{esistono } c_1, c_2 > 0 \text{ ed } n_0 \text{ tali che per ogni } n \geq n_0 \; 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Complessità dei Problemi

- Studiare la complessità di un problema (ossia quello che un algoritmo risolve) è molto diverso dallo studiare la complessità di un algoritmo.
  - Per poter dire che un problema ha complessità O(g(n)) (ipotizziamo di parlare del caso peggiore) basta trovare un qualsiasi algoritmo che lo risolva con O(g(n)).
  - Per poter affermare che un problema è Ω(g(n)) occorre invece dimostrare matematicamente che tutti i possibili algoritmi (inventati o non) lo risolvano alla meglio come Ω(g(n)).
- Per limitare superiormente un problema basta trovare almeno un algoritmo con complessità O(g(n)),
- Per limitare il problema inferiormente bisogna studiare ogni possibile soluzione (il problema, in linea teorica, potrebbe essere risolto in tempo costante, ma si può sempre dimostrare il contrario).
  - Quando la complessità di un algoritmo è pari al limite inferiore di complessità determinato per un problema, l'algoritmo si dice ottimo (in ordine di grandezza)

Individuazione di limiti inferiori

- Dimensione n dei dati: se nel caso peggiore occorre analizzare tutti i dati allora $\Omega(n)$ è un limite inferiore alla complessità del problema
  - Esempio: ricerca di un elemento o del massimo in un array
  - E' una tecnica banale, la maggior parte dei problemi hanno limiti inferiori più alti
- Eventi contabili: la ripetizione di un evento un dato numero di volte è essenziale per la risoluzione di un problema
  - Esempio: generare tutte le permutazioni di n oggetti
  - L'evento è la generazione di una nuova permutazione che si ripete per tutte le permutazioni, ossia n! volte.

Regole per la valutazione della complessità (1)

- Scomposizione
  - alg è la sequenza di alg1 ed alg2;
  - alg1 è O(g1(n)); alg2 è O(g2(n))
  - alg è O(max(g1(n),g2(n)))
- Esempio

```
i=0;
while(i<n) {                    g1(n)
  Stampastelle(i);
  i=i+1;
}
for(i=0; i<2*n; i++)            g2(n)
  scanf("%d", &numero);
```

Regole per la valutazione della complessità (1)

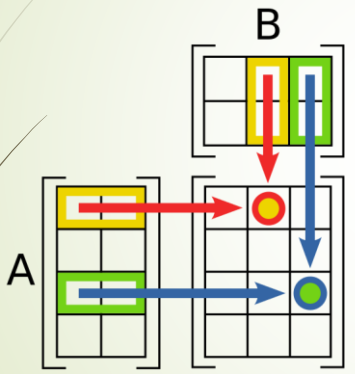
- Blocchi annidati
  - alg è composto da due blocchi annidati;
  - Blocco esterno è O(g1(n)); blocco interno è O(g2(n))
  - alg è O(g1(n)*g2(n))
- Esempio

g1(n)

g2(n)

```
for(i=0; i<n; i++) {
  scanf("%d", &j);
  printf("%d", j*j);
  do {
    scanf("%d", &numero);
    j=j+1;
  } while (j<=n);
```

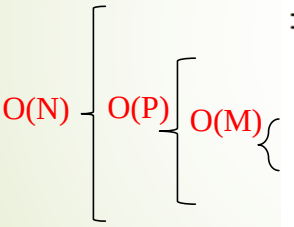

Esempio: Prodotto di matrici




```
float A[N][M], B[M][P], C[N][P];
int i, j, k;

for(i=0; i<N; i++)
  for(j=0; j<P; j++) {
    C[i][j]=0;
    for(k=0; k<M; k++)
      C[i][j]+=A[i][k] * B[k][j];
  }
```

Esempio: Prodotto di matrici




```
float A[N][M], B[M][P], C[N][P];
int i, j, k;

for(i=0; i<N; i++)
 for(j=0; j<P; j++) {
  C[i][j]=0;
  for(k=0; k<M; k++)
   C[i][j]+=A[i][k] * B[k][j];
 }
```

- Complessità asintotica del programma: O(N*P*M).

Regole per la valutazione della complessità (2)

- Sottoprogrammi ripetuti
  - alg applica ripetutamente un certo insieme di istruzioni la cui complessità all'i-esima esecuzione vale $f_i(n)$; il numero di ripetizioni è g(n)
  - alg è $O\left(\sum_{i=1}^{g(n)} f_i(n)\right)$
  - per $f_i(n)$ tutte uguali … O(g(n) f(n))

Regole per la valutazione della complessità (3)

- Operazione dominante
  - Sia f(n) il costo di esecuzione di un algoritmo alg;
  - Un'istruzione i è dominante se viene eseguita g(n) volte, con f(n) <= a g(n)
  - Se un algoritmo ha una operazione dominante allora è O(g(n))

Esempio
Ricerca Binaria

Istruzione dominante

```
int ricerca(int v [], int size, int k)
{ int inf =0, sup = size-1;
   while (sup >=inf)
   {  int med = (sup + inf ) / 2;
      if (k==v[med])
            return med;
      else if (k>v[med])
            inf = med+1;
      else sup = med-1
   }
   return -1;
}
```

Ricerca binaria
Complessità

- Osserviamo che la dimensione del problema si dimezza ad ogni ciclo.
  - Inizialmente è *n*, quindi *n*/2, poi *n*/4, e così via.
- Il ciclo si arresta quando la dimensione del problema è 1, dopo circa $\log_2 n$ iterazioni.
- Eseguiamo un numero costante di confronti per ogni iterazione. Il numero massimo di confronti sarà:

  $f(n) = O(\log n)$

Esercizio

- Scrivere la versione ricorsiva dell'algoritmo di ricerca binaria
  - Versione iterativa:

```
int ricerca(int v [], int size, int k)
{ int inf =0, sup = size-1;
   while (sup >=inf)
   {  int med = (sup + inf ) / 2;
      if (k==v[med])
                return med;
      else if (k>v[med])
                inf = med+1;
      else sup = med-1
   }
   return -1;
}
```

Ricorsione e valutazione della complessità

- Negli algoritmi ricorsivi la soluzione di un problema si ottiene applicando lo stesso algoritmo ad uno o più sottoproblemi
- La complessità viene espressa nella forma di una relazione di ricorrenza
  - La funzione di complessità f(n) è definita in termini di se stessa su una dimensione inferiore dei dati
- Per la valutazione della complessità valutiamo:
  - Il lavoro di combinazione (preparazione delle chiamate ricorsive e combinazione dei risultati ottenuti risultati ottenuti), che può essere:
    - Costante, lineare, …
  - La forma dell'equazione di ricorrenza, che può essere
    - Con o senza partizione dei dati
  - Il numero di termini ricorsivi (chiamate ricorsive nella funzione)

Ricorsione e valutazione della complessità

1. Lavoro di combinazione costante
   a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \ldots a_h T(n-h) + b$ per $n > h$
      - Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
      - Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)
   b) $T(n) = a\,T(n/p) + b$ per $n > 1$
      - $\log n$ se $a = 1$ (singola chiamata ricorsiva)
      - $n^{\log_p a}$ se $a > 1$ (più chiamate ricorsive)
2. Lavoro di combinazione lineare
   a) $T(n) = T(n-h) + b\,n + d$ per $n > h$
      Quadratico con n
   b) $T(n) = a\,T(n/p) + b\,n + d$
      - Lineare con n se $a < p$
      - $n \log n$ se $a = p$
      - $n^{\log_p a}$ se $a > p$

Relazioni di ricorrenza

Lavoro di combinazione costante (1)

```
unsigned fib(unsigned n) {
    if (n<2) return n;
   else return fib(n-1)+fib(n-2);
}
```

- Esempio: Fibonacci
  - T(0) = T(1) = c
  - T(n) = T(n-1) + T(n-2) + b
- Rientra nel caso 1.a: Lavoro di combinazione costante, senza partizione dei dati

a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots a_h T(n-h) + b$ per $n > h$

- Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
- Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)

  - Sono presenti 2 termini, quindi T(n) è esponenziale con n

Relazioni di ricorrenza

Lavoro di combinazione costante (2)

```
int fact(int n) {
   if (n==0) return 1;
   else return n*fact(n-1);
}
```

- Esempio: Fattoriale (h = 1)
  - T(0) = T(1) = c
  - T(n) = T(n-1) + b
- Rientra nel caso 1.a: Lavoro di combinazione costante, senza partizione dei dati

a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots a_h T(n-h) + b$ per $n > h$

- Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
- Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)

  - È presente un solo termine, quindi T(n) è lineare con n

Relazioni di ricorrenza

Lavoro di combinazione costante (3)

- Esempio: Ricerca binaria
  - T(1) = c
  - T(n) = T(n/2) + b
- Rientra nel caso 1.b: Lavoro di combinazione costante, con partizione dei dati

b) $T(n) = a\,T(n/p) + b$ per $n > 1$

- $\log n$ se $a = 1$ (singola chiamata ricorsiva)
- $n^{\log_p a}$ se $a > 1$ (più chiamate ricorsive)

  - È presente una sola chiamata ricorsiva (a=1), quindi T(n) è logaritmico con n

Ricerca Binaria Ricorsiva

```
int ricercaBinaria(int valore, int vettore[], int primo, int
ultimo)
{
    if (primo > ultimo) return -1;
    int mid=(primo+ultimo)/2;
    if (valore==vettore[mid])
         return mid;
    if (valore<vettore[mid])
         return ricercaBinaria(valore,vettore,primo,mid-1);
    else
         return ricercaBinaria(valore,vettore,mid+1,ultimo);
}
```