Strutture Dati

Lezione 2
Ricorsione
Analisi degli algoritmi

Oggi parleremo di …..

- Ricorsione
- Analisi degli algoritmi
  - complessità
    - temporale
    - spaziale
    - notazioni O, Ω, θ

Ricorsione (non si sa mai!)

- Un algoritmo (o una funzione) si dice ricorsivo (o ricorsiva) quando contiene chiamate a se stesso/a
- Ricorsione diretta
  - A chiama A
- Ricorsione indiretta
  - A chiama B
  - B chiama B_1
  - B_1 chiama B_2
  - ……..
  - B_n chiama A

Ricorsione

- In una definizione ricorsiva si definisce una classe di oggetti strettamente correlati tra loro nei termini degli oggetti stessi
- La definizione ricorsiva prevede
  - una base: si definiscono uno o più oggetti semplici
  - un passo di induzione: si definiscono oggetti più grandi nei termini di quelli più piccoli della classe

Esempio 1: Fattoriale

n!=n * (n-1)! n≥1

0! =1

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


Esempio 2: Il problema dei conigli

- Al mese 1 vi è una coppia di conigli neonati
- Al mese successivo la coppia inizia il processo di riproduzione
  - ogni coppia genera una nuova coppia ogni mese
  - ogni coppia non è in grado di procreare durante il primo mese di vita
- Quante coppie vi sono al mese n , F_n?

Il problema dei conigli

- Al mese 1 vi è 1 coppia: $F_1 = 1$
- Al mese 2 la coppia non può procreare: $F_2 = 1$
- Al mese 3 la coppia può procreare: $F_3 = 2$
- Al mese 4 l'ultima coppia nata non è fertile, quindi solo la coppia originale si riproduce: $F_4 = 3$
- Al mese n vi sono le coppie presenti al mese precedente, F_{n-1} , più quelle generate nell'ultimo mese (ovvero le coppie fertili al mese n-1), F_{n-2}

$$F_n = F_{n-1} + F_{n-2} \quad \text{con } F_1 = 1 \text{ e } F_2 = 1 \text{ o } F_0 = 0 \text{ e } F_1 = 1$$

Il problema dei conigli

- Successione di Fibonacci

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8 …
0	1	1	2	3	5	8	13	21 ...

```
int fib(int n)
{
   if (n==0) return 0;
   if (n==1) return 1;
   return fib(n-1)+fib(n-2);
}
```


Ricerca binaria iterativa

```
int ricbin(int lista[], int numric, int n)
{
 int primo=0, ultimo=n-1, mezzo;

 while(primo<=ultimo) {
   mezzo = (primo + ultimo)/2;
   switch(CONFRONTA(lista[mezzo], numric)){
        case -1:  primo = mezzo + 1;
                  break;
        case 0:   return mezzo;
        case 1:   ultimo = mezzo -1;
      }
 }
 return -1;
}
```

```
int CONFRONTA(int x,int y)
{
 if(x<y) return -1;
 else  if(x==y) return 0;
       else return 1;
}
```


Ricerca binaria ricorsiva

```
int ricbin(int lista[], int numric, int primo, int ultimo)
{
 int mezzo;

 if (primo<=ultimo) {
   mezzo = (primo + ultimo)/2;
   switch(CONFRONTA(lista[mezzo], numric)){
        case -1 :  return(ricbin(lista, numric, mezzo+1, ultimo));
        case 0  :  return mezzo;
        case 1  :  return(ricbin(lista, numric, primo, mezzo -1));
        }
 }
 return -1;
}
```


Divide-et-impera

- Nella ricorsione un problema si ripropone al suo interno in sottoproblemi uguali all'originale, ma applicati a sottoinsiemi dei dati
- La soluzione globale si ottiene come combinazione delle soluzioni dei sottoproblemi (es. Fibonacci)
- Il metodo divide-et-impera consiste nei seguenti passi
  1. **DIVIDI** il problema in un certo numero di sottoproblemi
  2. **CONQUISTA** i sottoproblemi risolvendoli ricorsivamente
  3. **COMBINA** insieme le soluzioni dei sottoproblemi in una unica soluzione
- Un bilanciamento della partizione consente di guadagnare in efficienza (es. Quicksort, Mergesort)

Ricorsione

- Vantaggi
  - facilità di formulazione
  - eleganza
  - compattezza
  - correttezza
  - calcolo della complessità
- Svantaggi
  - comprensibilità delle operazioni
  - tempo di calcolo

Analisi di un algoritmo

- Un problema può essere risolto da più di un algoritmo
- Quale scegliere?
- Quello migliore!

Analisi di un algoritmo

- Criteri per giudicare la qualità di un algoritmo
  - semplicità
  - chiarezza
  - efficienza
  - quantità di risorse usate
    - tempo di esecuzione
    - quantità di memoria

Analisi di un algoritmo

- Fattori che influenzano il tempo di esecuzione
  - architettura
  - linguaggio di programmazione
  - compilatore
  - fattori esterni
  - incidenza sulla velocità di esecuzione per un fattore costante
- Le analisi che faremo saranno tutte a meno di fattori costanti

Analisi di un algoritmo

- I dati del problema (le sue dimensioni!)
- Si definisce dimensione dell'input una funzione che associa ad ogni ingresso un numero naturale che rappresenta intuitivamente la quantità di informazione contenuta nel dato
  - ordinamento: numero di oggetti da ordinare
  - gestione dati: numero di dati da gestire
  - problemi sui grafi: numero di archi e nodi del grafo
- Dipende dalla rappresentazione dei dati (struttura dati)

Analisi di un algoritmo

- Definiamo il tempo di esecuzione o complessità in tempo $T(n)$ come il numero di operazioni elementari eseguite su un input di dimensione n
- Il calcolo del tempo esatto è problematico
- Vogliamo prescindere dalla reale esecuzione dell'algoritmo (calcolo teorico della complessità in tempo)
- E' necessaria un'operazione di astrazione

Analisi di un algoritmo

- Possiamo determinare il numero totale di passi compiuti dal programma/algoritmo su un input di dimensione n
  - paragonare due programmi diversi su uno stesso input
  - capire come cresce il tempo di calcolo al variare delle caratteristiche dell'input
- Cos'è un passo di un programma?
  - linea di pseudocodice
  - istruzione

Analisi di un algoritmo

- Assegnare un costo astratto ad ogni passo dell'algoritmo comprensivo di
  - numero di operazioni aritmetiche elementari
  - costo effettivo di ogni operazione elementare
- Determinare il numero di volte che viene eseguito ciascun passo
- Ipotesi:
  - una singola istruzione richiede un tempo di esecuzione costante
  - istruzioni diverse richiedono tempi di esecuzione diversi

Esempio: un problema di conteggio

- Input
  - Un intero N dove $N \geq 1$.
- Output
  - Il numero di coppie ordinate (i , j) tali che i e j sono interi e $1 \leq i \leq j \leq N$.
- Esempio: Input N=4
  - (1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,4)
  - Output = 10

Algoritmo 1

```
int Count_1(int N)
1     sum = 0                          1
2     for i = 1 to N                   2N
3       for j = i to N                 2∑_{i=1}^{N} (N+1-i)
4         sum = sum + 1                ∑_{i=1}^{N} (N+1-i)
5     return sum                       1
```

Il tempo di esecuzione è $2+2N+3\sum_{i=1}^{N}(N+1-i)=\frac{3}{2}N^2+\frac{7}{2}N+2$

Algoritmo 2

```
int Count_2(int N)
1    sum = 0                          1
2    for i = 1 to N                   2N
3      sum = sum + (N+1-i)            4N
4    return sum                       1
```

Il tempo di esecuzione è $6N+2$

Osserviamo: $\sum_{i=1}^{N}(N+1-i)=\sum_{i=1}^{N} i = N(N+1)/2$

Algoritmo 3

$$\sum_{i=1}^{N}(N+1-i)=\sum_{i=1}^{N} i = N(N+1)/2$$

```
int Count_3(int N)
1     sum = N(N+1)/2                   4
2     return sum                       1
```

Il tempo di esecuzione è 5 unità di tempo

Riassumendo

Algoritmo	Tempo di Esecuzione
Algoritmo 1	$\frac{3}{2}N^2+\frac{7}{2}N+2$
Algoritmo 2	$6N+2$
Algoritmo 3	5

La complessità in pratica

- Per n=10
  - $T_1(n)=187$
  - $T_2(n)=62$
  - $T_3(n)=5$
- Per n=100
  - $T_1(n)=15352$
  - $T_2(n)=602$
  - $T_3(n)=5$
- Per n=500
  - $T_1(n)=376752$
  - $T_2(n)=3002$
  - $T_3(n)=5$

- Quale è l'algoritmo più conveniente?

La complessità in pratica

- Per valutare la bontà di un algoritmo occorre effettuare una analisi asintotica della complessità in tempo, ovvero per $n \to \infty$
- Occorre studiare il comportamento di $T(n)$ per grandi valori di n

Funzioni di complessità tipiche

- Sia A1 di complessità in tempo n (lineare)
- Sia A2 " " $n\log_2 n$ (loglineare)
- Sia A3 " " n^2 (quadratica)
- Sia A4 " " n^3 (cubica)
- Sia A5 " " 2^n (esponenziale)
- Sia A6 " " 3^n (esponenziale)

	Complessità	Max dim
A1	n	$6*10^7$
A2	$n\log_2 n$	$28*10^5$
A3	n^2	$77*10^2$
A4	n^3	390
A5	2^n	25

T(operazione elementare)= 1μsec (10^{-6}sec)

Dimensioni massime di input processabili in un minuto

La complessità

comp/dim	n=10	n=20	n=50	n=100	$n=10^3$	$n=10^4$	$n=10^5$	$n=10^6$
n	10 μs	20 μs	50 μs	0.1 ms	1 ms	10 ms	0.1 s	1 s
$n\log_2 n$	33.2μs	86.4μs	0.28ms	0.6 ms	9.9 ms	0.1 s	1.6 s	19.9 s
n^2	0.1 ms	0.4 ms	2.5 ms	10 ms	1 s	100 s	2.7 h	11.5 g
n^3	1 ms	8 ms	125 ms	1 s	16.6 m	11.5 g	31.7 a	≈ 300 c
2^n	1ms	1 s	35.7 a	$\approx 10^{14}$ c	••	••	••	••
3^n	59 ms	58 m	$\approx 10^8$ c	••	••	••	••	••

•• > millennio

T(operazione elementare)= 1μsec (10^{-6}sec)

La complessità

- Algoritmi di complessità $\sim n^k$ ($k \geq 2$) sono applicabili sono per n non troppo elevato
  - $2 \leq k < 3$ applicabili su input di dimensione media
  - $k \geq 3$ tempi inaccettabili
- Algoritmi di complessità lineare o quasi lineare ($n\log n$) utilizzabili anche per input di dimensioni elevate
- Algoritmi di complessità esponenziale hanno tempi di calcolo proibitivi anche per input di dimensioni limitate

La notazione O-grande

- Esprime il tempo di esecuzione in maniera "approssimata"

Definizione

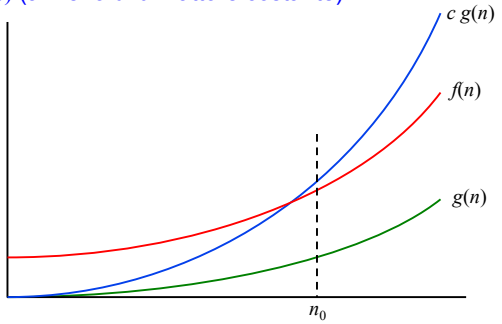
Siano $f(n)$ e $g(n)$ due funzioni definite in N e a valori in R

Diciamo che $f(n)$ è O-grande di $g(n)$ e scriviamo $f(n) \in O(g(n))$ (oppure $f(n) = O(g(n))$) se esistono una costante $c>0$ e un numero $n_0 \in N$ tale che $\forall n \geq n_0$ si ha $f(n) \leq c*g(n)$

Si dice che $f(n)$ ha ordine di grandezza minore o uguale a quello di $g(n)$

La notazione O-grande

$g(n)$ rappresenta il **limite superiore asintotico** a $f(n)$ (a meno di un fattore costante)

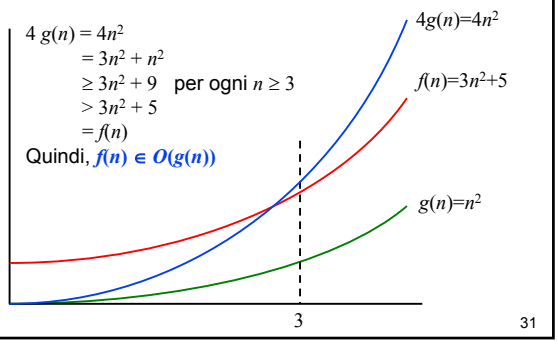


Esempio di limite superiore asintotico

$3n^2 + 5$ è $O(n^2)$

$4\,g(n) = 4n^2$
$= 3n^2 + n^2$
$\geq 3n^2 + 9$ per ogni $n \geq 3$
$> 3n^2 + 5$
$= f(n)$

Quindi, $f(n) \in O(g(n))$

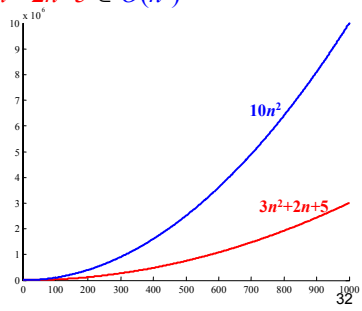


Esercizio sulla notazione O

- Mostrare che $3n^2+2n+5 \in O(n^2)$

$10n^2 = 3n^2 + 2n^2 + 5n^2$
$\geq 3n^2 + 2n + 5$

$c = 10,\ n_0 = 1$

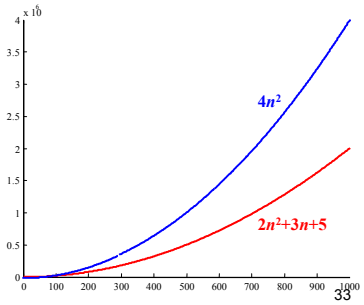


Esercizio sulla notazione O

- Mostrare che $2n^2+3n+5 \in O(n^2)$

$2n^2 + 3n + 5 \leq 4n^2$

$c = 4,\ n_0 = 3$



Utilizzo della notazione O

- In genere quando impieghiamo la notazione O, utilizziamo la formula più "*semplice*".
  - Scriviamo
    - $3n^2+2n+5 = O(n^2)$
  - Le seguenti sono tutte corrette ma in genere non le si usa:
    - $3n^2+2n+5 = O(3n^2+2n+5)$
    - $3n^2+2n+5 = O(n^2+n)$
    - $3n^2+2n+5 = O(3n^2)$

La notazione *Omega*-grande

- Esprime il tempo di esecuzione in maniera "approssimata"

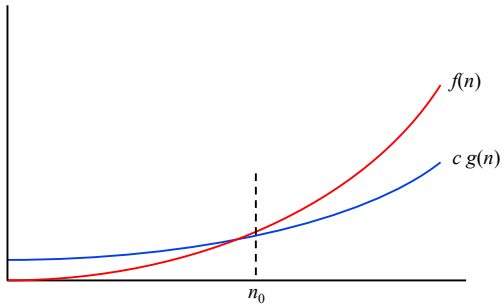
Definizione

Siano $f(n)$ e $g(n)$ due funzioni definite in N e a valori in R

Diciamo che $f(n)$ è Ω-grande di $g(n)$ e scriviamo $f(n) \in \Omega(g(n))$ se esistono una costante $c>0$ e un numero $n_0 \in N$ tale che $\forall n \geq n_0$ si ha $f(n) \geq c*g(n)$

La notazione *Omega*-grande

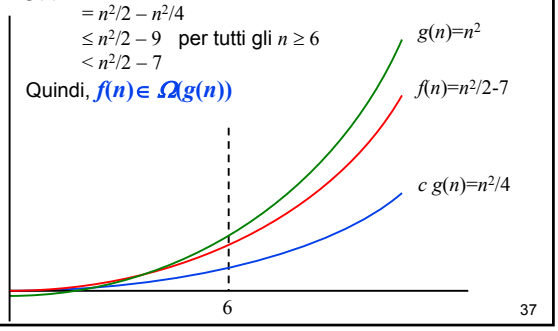
$g(n)$ è detto un **limite inferiore asintotico** di $f(n)$



Esempio di limite inferiore asintotico

$g(n)/4 = n^2/4$
$= n^2/2 - n^2/4$
$\geq n^2/2 - 9$ per tutti gli $n \geq 6$
$< n^2/2 - 7$

Quindi, $f(n) \in \Omega(g(n))$

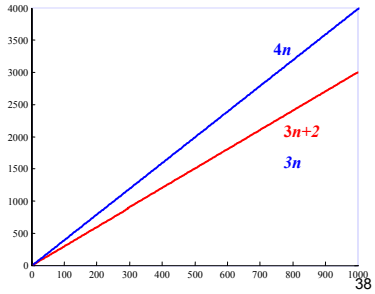


Esercizio sulla notazione O e Ω

- Mostrare che $3n+2 \in O(n)$ e $3n+2 \in \Omega(n)$

$3n + 2 \leq 4n$
$c = 4,\ n_0 = 2$

$3n + 2 \geq 3n$
$c = 3,\ n_0 = 1$



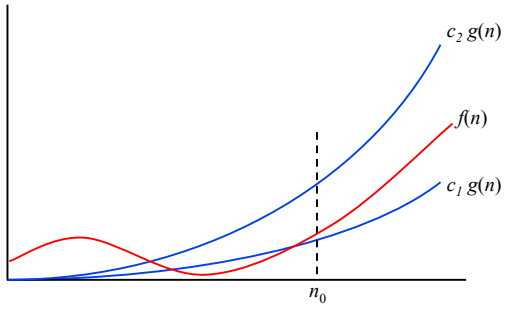
La notazione *Theta*-grande

Definizione
Siano $f(n)$ e $g(n)$ due funzioni definite in N e a valori in R

Diciamo che $f(n)$ è Θ-grande di $g(n)$ e scriviamo $f(n) \in \Theta(g(n))$ se esistono due costanti $c_1>0$, $c_2>0$ e un numero $n_0 \in N$ tale che $\forall n \geq n_0$ si ha $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

La notazione *Theta*-grande

$g(n)$ è detto un **limite asintotico stretto** di $f(n)$



Riassumendo

- O : *O-grande*: limite superiore asintotico
- Ω : *Omega-grande*: limite inferiore asintotico
- Θ : *Theta*: limite asintotico stretto
- Usiamo la *notazione asintotica* per dare un limite ad una funzione ($f(n)$), a meno di un fattore costante (c).

La complessità in spazio

- Definiamo la complessità in spazio come il massimo spazio invaso nella memoria durante l'esecuzione dell'algoritmo
- Si studia la complessità asintotica, limitandosi al suo ordine di grandezza
- Ciò che definisce la bontà di un algoritmo è la complessità in tempo

Caso medio, pessimo, ottimo

- Per complessità media si intende la complessità di un algoritmo mediato su tutte le possibili occorrenze iniziali dei dati (difficile!)
- Per complessità nel caso pessimo si intende la complessità relativa a quella particolare occorrenza iniziale dei dati per cui l'algoritmo ha comportamento pessimo
  - fornisce un limite superiore alla complessità
  - semplice da individuare
- La complessità nel caso ottimo non ci dice nulla sulla bontà di un algoritmo