Fondamenti di Informatica (Elettronici)

Complessità di calcolo – Lezione 30

14 dicembre 2020

Complessità di calcolo

1. Soluzione iterativa del fattoriale
2. Efficienza dei programmi
3. Il modello di costo
4. Comportamento asintotico
5. Complessita' di un algoritmo
6. Valutazione della complessità di un programma
7. Complessita' di un problema
8. Esempio: valutazione dei polinomi
9. Collezione di risultati di complessità

Section 1

Soluzione iterativa del fattoriale

Consumo di risorse (tempo e spazio)

Programmi differenti, che implementano algoritmi differenti, possono consumare differenti quantità di risorse. Ad esempio scriviamo due versioni iterative della funzione fattoriale:

```
function fact1(n)
    fatt = 1
    for i=1:n
        fatt *= i  end
    fatt
end
```

Complessit di tempo: $O(n^2)$
Complessit di spazio: $O(1)$

```
function fact2(n)
    fatt = zeros(Int, n+1)
    fatt[1] = 1;
    for i=1:n
        fatt[i+1] = fatt[i] * i  end
    fatt
end
```

Complessit di tempo: $O(n)$
Complessit di spazio: $O(n)$

```
julia> print(collect(fact1(n) for n=0:10))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
julia> print(fact2(10))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Confronto tra esponenziali e fattoriale

Pu essere utile considerare quanto variano il polinomio n^2 e la funzione esponenziale 2^n, messe a confronto con la funzione fattoriale $n!$, al variare del parametro n:

n	n^2	2^n	$n!$
4	16	16	24
8	64	256	40320
12	144	4096	479001600
16	256	65536	20922789888000
20	400	1048576	2.43290200817664E+18
24	576	16777216	6.20448401733239E+23
48	2304	281474976710656	1.24139155925361E+61

Section 2

Efficienza dei programmi

Efficienza dei programmi 1/2

La valutazione dell'efficienza degli algoritmi e la ricerca di algoritmi sempre piu' efficienti costiruiscono quella parte dell'informatica nota come Complessita' cornputazionale.

Un programma e' tanto piu' efficiente quanto meno richiede risorse di calcolo per la sua esecuzione

Efficienza dei programmi 1/2

La valutazione dell'efficienza degli algoritmi e la ricerca di algoritmi sempre piu' efficienti costiruiscono quella parte dell'informatica nota come Complessita' cornputazionale.

Un programma e' tanto piu' efficiente quanto meno richiede risorse di calcolo per la sua esecuzione

Le risorse fondamentali richieste per l'esecuzione sono:

- la memoria

La risorsa presa in considerazione nel seguito sara' il solo tempo di calcolo. Considerazioni del tutto simili si possono fare per la memoria occupata.

Efficienza dei programmi 1/2

La valutazione dell'efficienza degli algoritmi e la ricerca di algoritmi sempre piu' efficienti costiruiscono quella parte dell'informatica nota come Complessita' cornputazionale.

Un programma e' tanto piu' efficiente quanto meno richiede risorse di calcolo per la sua esecuzione

Le risorse fondamentali richieste per l'esecuzione sono:

- la memoria
- il tempo di calcolo

La risorsa presa in considerazione nel seguito sara' il solo tempo di calcolo. Considerazioni del tutto simili si possono fare per la memoria occupata.

Efficienza dei programmi 2/2

Si potrebbe valutare il tempo di esecuzione in unita' standard di tempo (ad es. minuti), e confrontare due programmi misurando il loro tempo di esecuzione.

Efficienza dei programmi 2/2

Si potrebbe valutare il tempo di esecuzione in unita' standard di tempo (ad es. minuti), e confrontare due programmi misurando il loro tempo di esecuzione.

Questo dipende troppo da:

- caratteristiche del sistema di elaborazione;

Si cercano allora valutazioni piu' oggettive, anche se approssimate.

Efficienza dei programmi 2/2

Si potrebbe valutare il tempo di esecuzione in unita' standard di tempo (ad es. minuti), e confrontare due programmi misurando il loro tempo di esecuzione.

Questo dipende troppo da:

- caratteristiche del sistema di elaborazione;
- compilatore utilizzato per produrre il codice;

Si cercano allora valutazioni piu' oggettive, anche se approssimate.

Efficienza dei programmi 2/2

Si potrebbe valutare il tempo di esecuzione in unita' standard di tempo (ad es. minuti), e confrontare due programmi misurando il loro tempo di esecuzione.

Questo dipende troppo da:

- caratteristiche del sistema di elaborazione;
- compilatore utilizzato per produrre il codice;
- dati utilizzati per il test (che andrebbe ripetuto per numerosi valori dei dati)

Si cercano allora valutazioni piu' oggettive, anche se approssimate.

Section 3

Il modello di costo

Il modello di costo 1/2

Ipotesi fondamentale: il costo di una istruzione non dipende dal sistema di calcolo. Inoltre:

1. il costo di ogni istruzione semplice e' assunto unitario;

Il modello di costo 1/2

Ipotesi fondamentale: il costo di una istruzione non dipende dal sistema di calcolo. Inoltre:

1. il costo di ogni istruzione semplice e' assunto unitario;
2. il costo di un'istruzione composta e' assunto pari alla somma dei costi delle istruzioni componenti;

Il modello di costo 1/2

Ipotesi fondamentale: il costo di una istruzione non dipende dal sistema di calcolo. Inoltre:

1. il costo di ogni istruzione semplice e' assunto unitario;
2. il costo di un'istruzione composta e' assunto pari alla somma dei costi delle istruzioni componenti;
3. il costo di una istruzione di ciclo e' assunto pari al costo del test, sommato al costo delle istruzioni interne al ciclo, moltiplicati per il numero di volte che il ciclo viene eseguito;

Il modello di costo 1/2

Ipotesi fondamentale: il costo di una istruzione non dipende dal sistema di calcolo. Inoltre:

1. il costo di ogni istruzione semplice e' assunto unitario;
2. il costo di un'istruzione composta e' assunto pari alla somma dei costi delle istruzioni componenti;
3. il costo di una istruzione di ciclo e' assunto pari al costo del test, sommato al costo delle istruzioni interne al ciclo, moltiplicati per il numero di volte che il ciclo viene eseguito;
4. il costo di una istruzione `if` e' il costo del test sommato al maggiore tra costo della clausola `then`, oppure e' pari al costo del test sommato al costo della clausola `else`;

Il modello di costo 1/2

Ipotesi fondamentale: il costo di una istruzione non dipende dal sistema di calcolo. Inoltre:

1. il costo di ogni istruzione semplice e' assunto unitario;
2. il costo di un'istruzione composta e' assunto pari alla somma dei costi delle istruzioni componenti;
3. il costo di una istruzione di ciclo e' assunto pari al costo del test, sommato al costo delle istruzioni interne al ciclo, moltiplicati per il numero di volte che il ciclo viene eseguito;
4. il costo di una istruzione `if` e' il costo del test sommato al maggiore tra costo della clausola `then`, oppure e' pari al costo del test sommato al costo della clausola `else`;
5. il costo di attivazione di una procedura o funzione e' assunto pari alla somma dei costi di tutte le istruzioni che la compongono (trascurando ogni costo di attivazione per il passaggio di parametri, etc.)

Il modello di costo 2/2

Si noti che le ipotesi sono molto semplificative:

ad esempio le istruzioni seguenti hanno LO STESSO COSTO:

```
i = i + 1;
A = B + C * (sin(x)/2) / (B*B - C*C);
```

Comunque, esiste tra le istruzioni una relazione del tipo

$$c\, t_1 = t_2$$

t_2 differisce da t_1 al piu' per un fattore costante (ipotesi di eguale costo esatta a meno di un fattore costante)

IL COSTO DI UN PROGRAMMA VIENE VALUTATO A MENO DI UN FATTORE COSTANTE.

Alg. ricerca completa - ESEMPIO 1/2 – modello di costo

```
struct record
    chiave          ::Int
    nome            ::String
    indirizzo       ::String
    codice_fiscale::String
end

table1 = Array{record,1}()
push!(table1, record( 101, "adalberto", "via d.manin", "mnndnl123" ))
push!(table1, record( 232, "roberto", "via c.cavour", "cvrcml456" ))
push!(table1, record( 133, "alberto", "via g.garibaldi", "grbgsp789" ))
push!(table1, record( 567, "gilberto", "via p.albertelli", "lbrtpl012" ))

    function ricerca(t,k)
1       i = 1; n = length(t)
2       while (!(t[i].chiave == k) && (i < n))
3           i = i+1
4       end
5       if (t[i].chiave == k)
6           cliente = t[i]
7           return cliente
8       end
    end
```

Alg. ricerca completa - ESEMPIO 1/2 – modello di costo

Test del codice

```
julia> cliente = ricerca(table1,133)     # record presente
record(133, "alberto", "via g.garibaldi", "grbgsp789")

julia> cliente = ricerca(table1,999)     # record assente

julia> cliente = ricerca(table1,101)     # primo record
record(101, "adalberto", "via d.manin", "mnndnl123")

julia> cliente = ricerca(table1,567)     # ultimo record
record(567, "gilberto", "via p.albertelli", "lbrtpl012")
```

Il modello di costo - ESEMPIO 2/2

il costo della ricerca esaustiva (completa) nel caso peggiore (ricerca con successo nell'ultimo elemento) e':

$$2 + n + n + 2 = 2n + 4$$

La funzione di costo della ricerca esaustiva (ovvero condotta confrontando tutti gli elementi) in un array di n elementi

$$T(n) = 2n + 4$$

Il costo della ricerca esaustiva è quindi lineare con il numero degli elementi dell'array.

Section 4

Comportamento asintotico

Comportamento asintotico

Trovare una funzione esatta per il costo di un programma puo' essere molto difficile; ci si accontenta del comportamento asintotico, ovvero per valori delle dimensioni dell'input che tendono all'infinito

Si valuta il costo del programma a meno di costanti moltiplicative e di termini di ordine inferiore

Diremo che un programma ha complessita' lineare se il suo costo in funzione delle dimensioni n dell'input e' una funzione lineare:

$$T(n) = c_1 n + c_2$$

Ad esempio, si assume che le funzioni di costo del tipo

$$T_1(n) = 100n + 3045$$

$$T_2(n) = 2n + 3$$

siano asintoticamente equivalenti (nel senso che interessa solo il termine di grado massimo, privato della costante moltiplicativa)

Comportamento asintotico

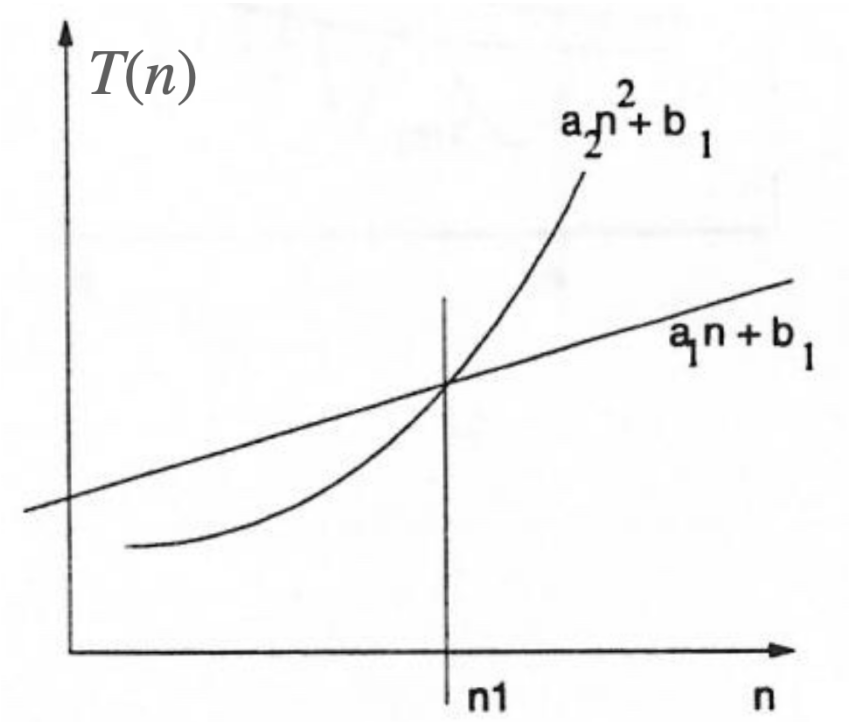
Nota bene

A partire da qualche n_1 sufficientemente grande, qualunque funzione (quadratica) di n del tipo:

$$a_2\, n^2 + b_2$$

supera qualunque funzione (lineare) del tipo:

$$a_1\, n + b_1$$



Section 5

Complessita' di un algoritmo

Notazione $O()$

Un algoritmo (programma) ha complessita $O(f(n))$ se esistono tre costanti a, b, n_0 tali che:

il numero di istruzioni $T(n)$ che vengono eseguite – nel caso peggiore – con input di misura n verifica la relazione

$$T(n) < a\,f(n) + b \quad \text{per ogni} \quad n > n_0$$

Notazione $O()$

Un algoritmo (programma) ha complessita $O(f(n))$ se esistono tre costanti a, b, n_0 tali che:

il numero di istruzioni $T(n)$ che vengono eseguite – nel caso peggiore – con input di misura n verifica la relazione

$$T(n) < a\,f(n) + b \quad \text{per ogni} \quad n > n_0$$

Nota Bene

- $O()$ fornisce una valutazione approssimata per eccesso (delimitazione superiore)

Notazione $O()$

Un algoritmo (programma) ha complessita $O(f(n))$ se esistono tre costanti a, b, n_0 tali che:

il numero di istruzioni $T(n)$ che vengono eseguite – nel caso peggiore – con input di misura n verifica la relazione

$$T(n) < a\,f(n) + b \quad \text{per ogni} \quad n > n_0$$

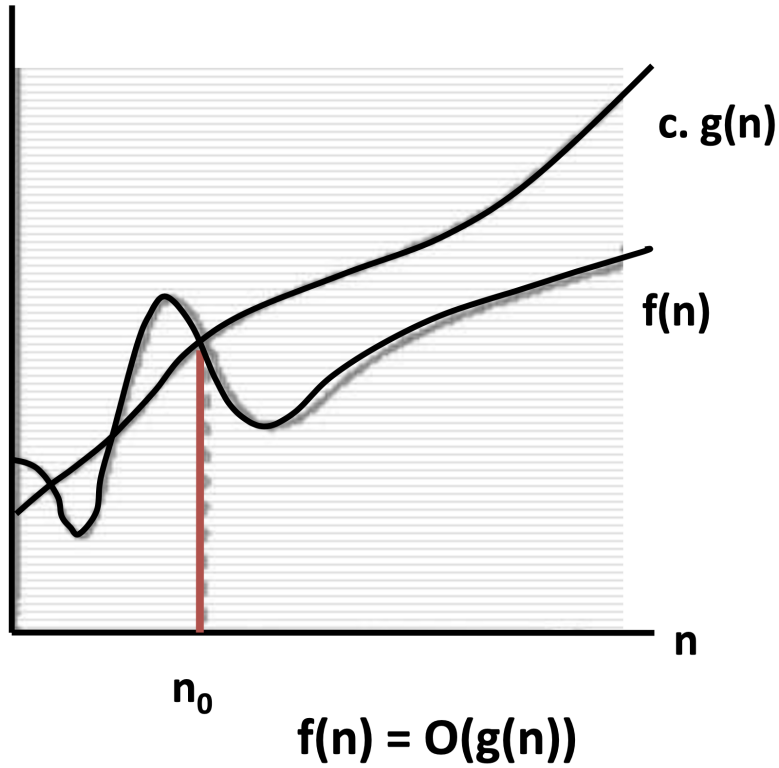
Nota Bene

- $O()$ fornisce una valutazione approssimata per eccesso (delimitazione superiore)
- tra le tante funzioni che approssimano per eccesso $T(n)$ si cerca quella minorante.

La notazione "big Oh" — $O()$

Asymptotic *Upper* Bound

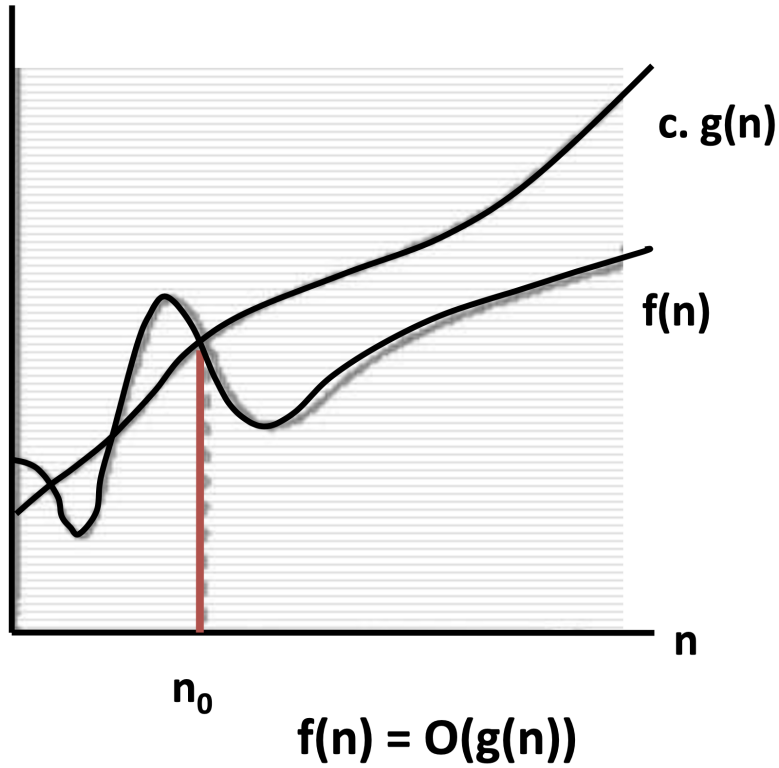
- $f(n) = O(g(n))$, se esistono costanti c e n_0 tali che $f(n) \leq c\, g(n)$ per $n \geq n_0$;



La notazione "big Oh" — $O()$

Asymptotic *Upper* Bound

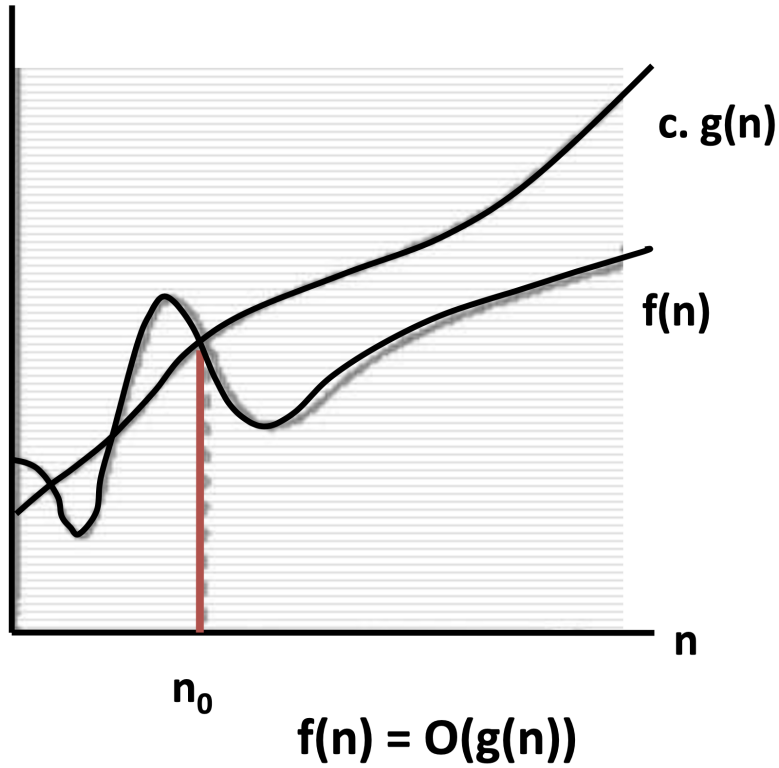
- $f(n) = O(g(n))$, se esistono costanti c e n_0 tali che $f(n) \leq c\, g(n)$ per $n \geq n_0$;



La notazione "big Oh" — $O()$

Asymptotic *Upper* Bound

- $f(n) = O(g(n))$, se esistono costanti c e n_0 tali che $f(n) \leq c\, g(n)$ per $n \geq n_0$;

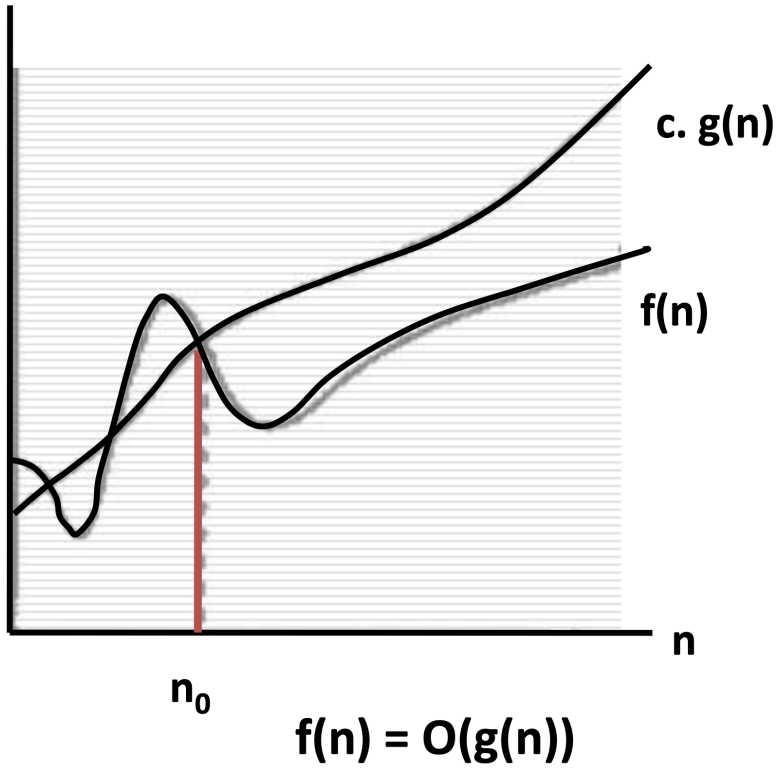


- $f(n)$ e $g(n)$ sono funzioni su numeri interi non negativi non decrescenti;

La notazione "big Oh" — $O()$

Asymptotic *Upper* Bound

- $f(n) = O(g(n))$, se esistono costanti c e n_0 tali che $f(n) \leq c\, g(n)$ per $n \geq n_0$;

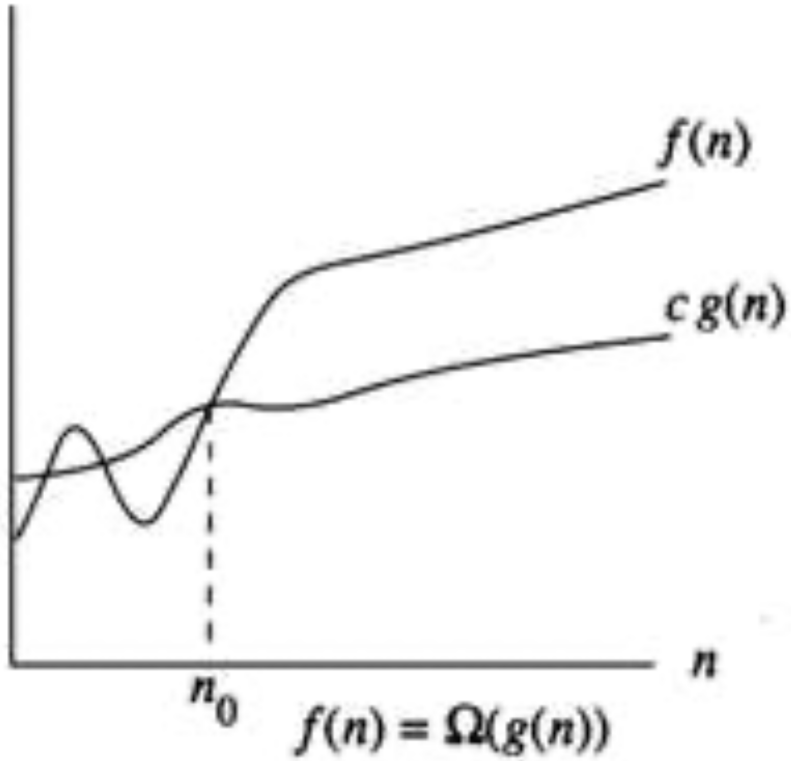


- $f(n)$ e $g(n)$ sono funzioni su numeri interi non negativi non decrescenti;
- Utilizzata per l'analisi del caso peggiore;

La notazione "big Omega" — $\Omega()$

Asymptotic *Lower* Bound

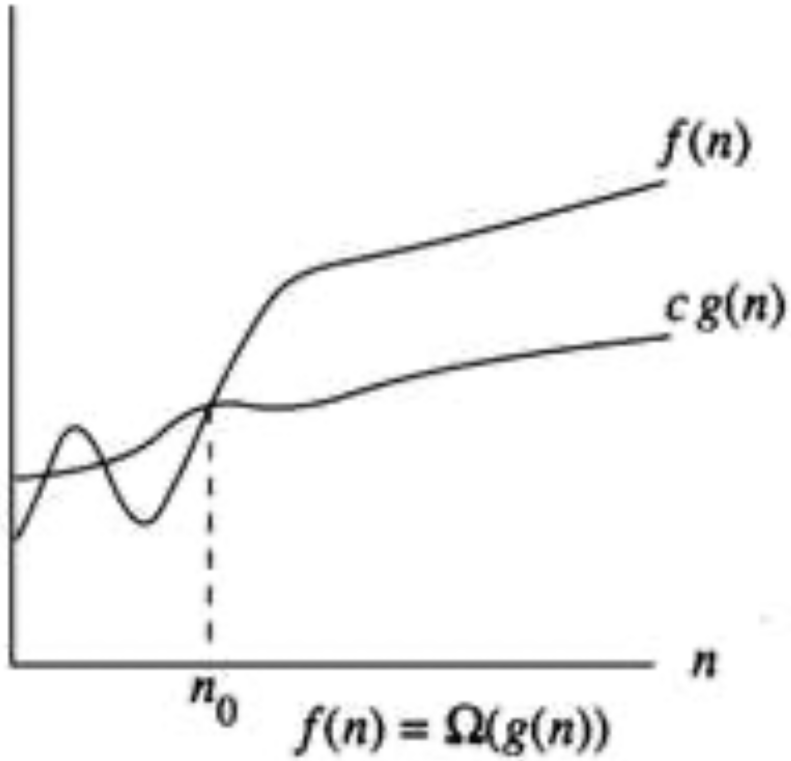
- $f(n) = \Omega(g(n))$, se esistono costanti c e n_0 tali che $c\,g(n) \leq f(n)$ per $n > n_0$;



La notazione "big Omega" — $\Omega()$

Asymptotic *Lower* Bound

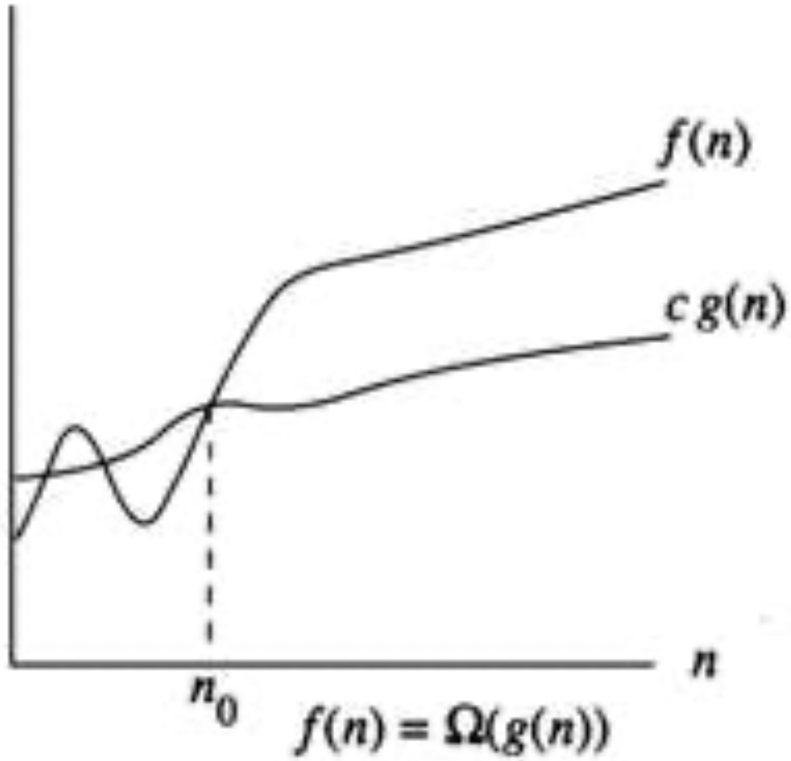
- $f(n) = \Omega(g(n))$, se esistono costanti c e n_0 tali che $c\,g(n) \leq f(n)$ per $n > n_0$;



La notazione "big Omega" — $\Omega()$

Asymptotic *Lower* Bound

- $f(n) = \Omega(g(n))$, se esistono costanti c e n_0 tali che $c\,g(n) \leq f(n)$ per $n > n_0$;

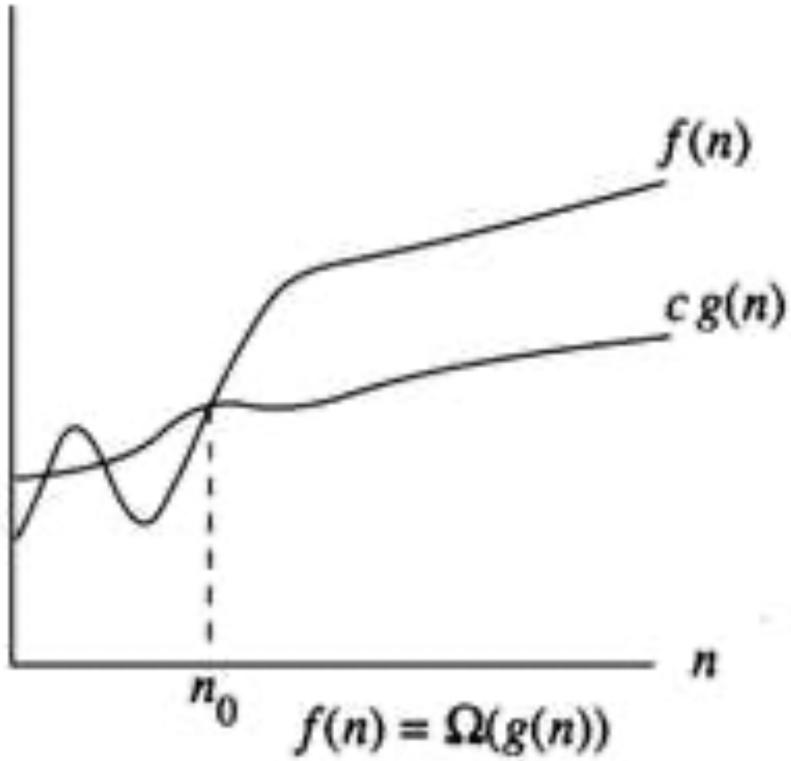


- Utilizzato per descrivere i tempi di esecuzione dei casi migliori o i limiti inferiori dei problemi asintotici;

La notazione "big Omega" — $\Omega()$

Asymptotic *Lower* Bound

- $f(n) = \Omega(g(n))$, se esistono costanti c e n_0 tali che $c\,g(n) \leq f(n)$ per $n > n_0$;

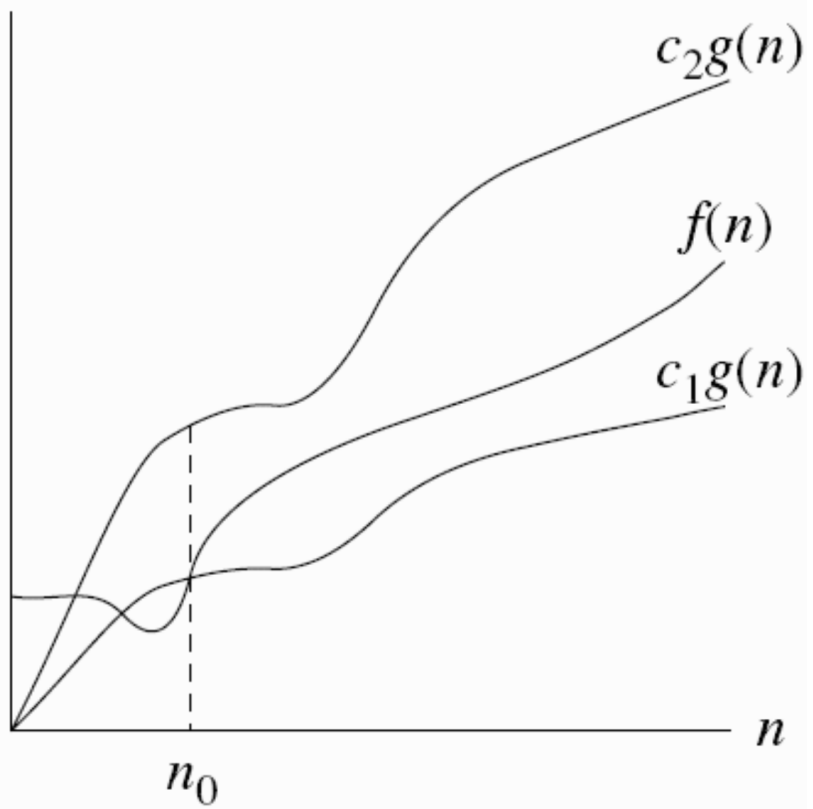


- Utilizzato per descrivere i tempi di esecuzione dei casi migliori o i limiti inferiori dei problemi asintotici;
- Ad esempio: il limite inferiore della ricerca in un array non ordinato è $\Omega(n)$.

La notazione "big Theta" — $\Theta()$

Asymptotic *Tight* Bound

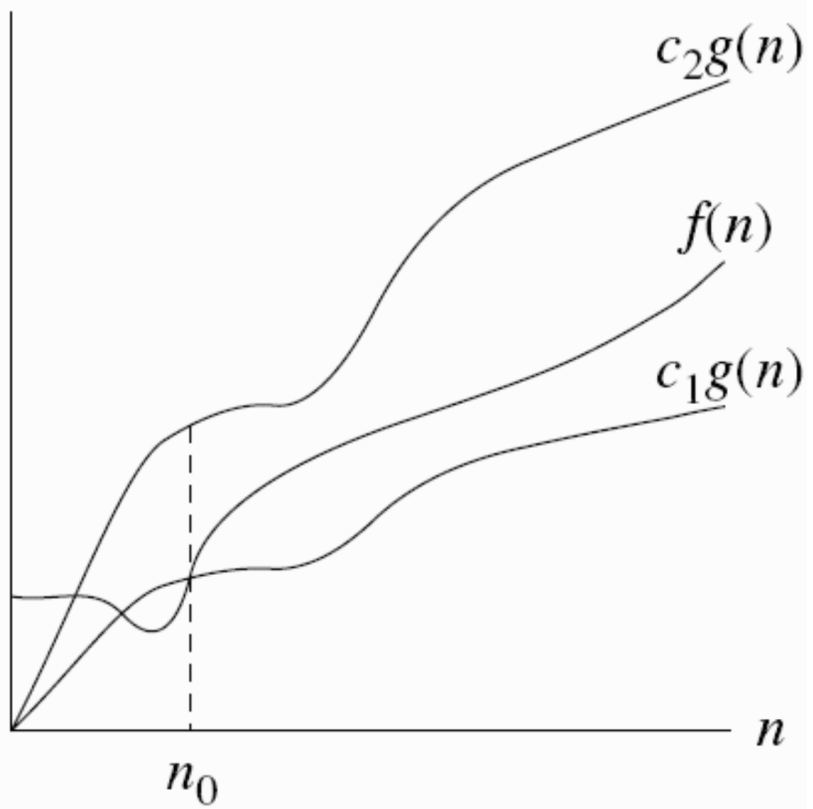
- $f(n) = \Theta(g(n))$, se esistono costanti c_1, c_2 e n_0 tali che $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per $n > n_0$;



La notazione "big Theta" — $\Theta()$

Asymptotic *Tight* Bound

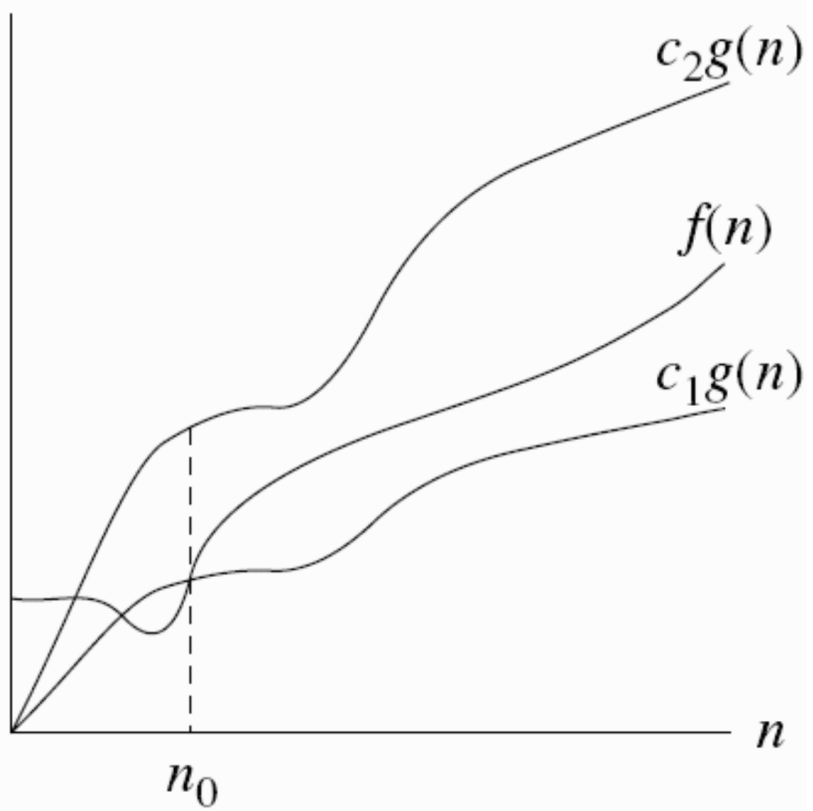
- $f(n) = \Theta(g(n))$, se esistono costanti c_1, c_2 e n_0 tali che $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per $n > n_0$;



La notazione "big Theta" — $\Theta()$

Asymptotic *Tight* Bound

- $f(n) = \Theta(g(n))$, se esistono costanti c_1, c_2 e n_0 tali che $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per $n > n_0$;

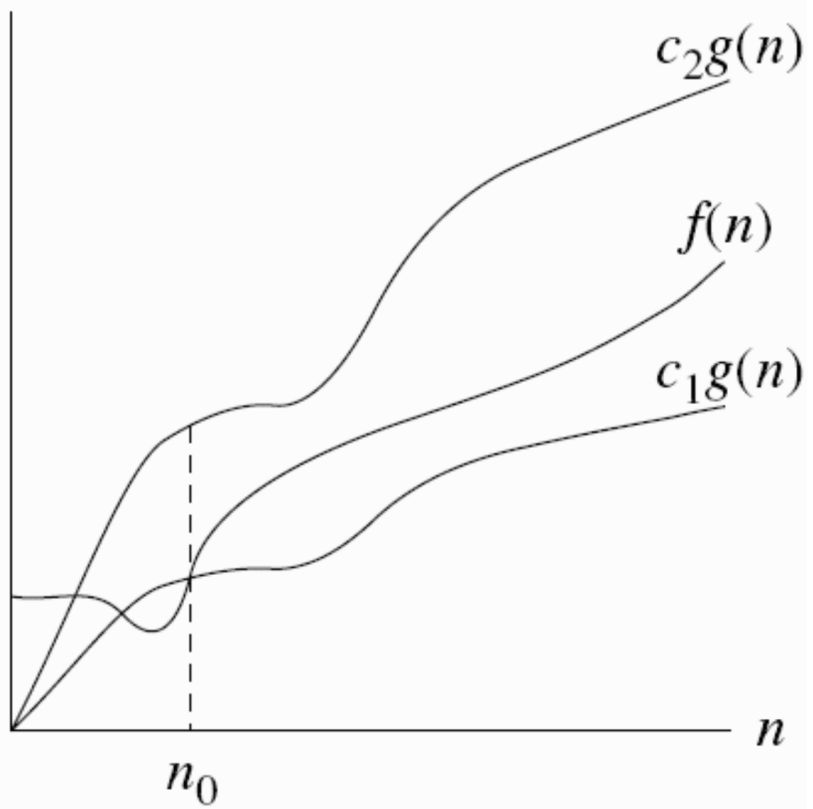


- $f(n) = \Theta(g(n))$, iff $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$;

La notazione "big Theta" — $\Theta()$

Asymptotic *Tight* Bound

- $f(n) = \Theta(g(n))$, se esistono costanti c_1, c_2 e n_0 tali che $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per $n > n_0$;



- $f(n) = \Theta(g(n))$, iff $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$;
- $O(f(n))$ è spesso usato in modo improprio invece di $\Theta(f(n))$.

Sintesi della complessità asintotica

1. Un algoritmo ha costo $O(f(n))$ se, per ogni n e per ogni input di dimensione n, impiega una quantita' di risorse proporzionale a $f(n)$ a meno di costanti additive: delimitazione superiore (upper bound)

Sintesi della complessità asintotica

1. Un algoritmo ha costo $O(f(n))$ se, per ogni n e per ogni input di dimensione n, impiega una quantita' di risorse proporzionale a $f(n)$ a meno di costanti additive: delimitazione superiore (upper bound)
2. Un algoritmo ha costo $\Omega(g(n))$ se esiste una sequenza infinita di istanze del problema di dimensioni crescenti ognuna delle quali richiede una quantità di risorse maggiore di $c\,g(n)$. Delimitazione inferiore (lower bound)

Sintesi della complessità asintotica

1. Un algoritmo ha costo $O(f(n))$ se, per ogni n e per ogni input di dimensione n, impiega una quantita' di risorse proporzionale a $f(n)$ a meno di costanti additive: delimitazione superiore (upper bound)
2. Un algoritmo ha costo $\Omega(g(n))$ se esiste una sequenza infinita di istanze del problema di dimensioni crescenti ognuna delle quali richiede una quantità di risorse maggiore di $c\,g(n)$. Delimitazione inferiore (lower bound)
3. Ci sono algoritmi che si comportano efficientemente in molti casi ma non in tutti. La definizione (2) non considera i casi facili

Sintesi della complessità asintotica

1. Un algoritmo ha costo $O(f(n))$ se, per ogni n e per ogni input di dimensione n, impiega una quantita' di risorse proporzionale a $f(n)$ a meno di costanti additive: delimitazione superiore (upper bound)
2. Un algoritmo ha costo $\Omega(g(n))$ se esiste una sequenza infinita di istanze del problema di dimensioni crescenti ognuna delle quali richiede una quantità di risorse maggiore di $c\,g(n)$. Delimitazione inferiore (lower bound)
3. Ci sono algoritmi che si comportano efficientemente in molti casi ma non in tutti. La definizione (2) non considera i casi facili
4. Si ha valutazione esatta $\Theta(g(n))$ del costo di un algoritmo quando le due delimitazioni $O()$ e $\Omega()$ coincidono (a meno delle solite costanti)

Section 6

Valutazione della complessità di un programma

Regole di calcolo: Regola 1

La valutazione del costo di un algoritmo o programma richiede la determinazione di una delimitazione superiore e di una inferiore.

REGOLE PER LA DELIMITAZIONE SUPERIORE DELLA COMPLESSITA'
(senza dimostrazioni)

Regole di calcolo: Regola 1

La valutazione del costo di un algoritmo o programma richiede la determinazione di una delimitazione superiore e di una inferiore.

REGOLE PER LA DELIMITAZIONE SUPERIORE DELLA COMPLESSITA' (senza dimostrazioni)

REGOLA 1 Se un programma e' composto da due parti in sequenza, di costo $O(f(n))$ e $O(g(n))$, allora il costo dell'intero programma e'

$$O(\max\{f(n), g(n)\})$$

Regole di calcolo: Regola 1'

In particolare:

date due funzioni $f(n)$ e $g(n)$, se $f(n) = O(g(n))$, allora si avrà:

$$O(f(n) + g(n)) = O(g(n))$$

REGOLA 1' Se un programma è composto da una successione finita di parti in sequenza, allora il costo dell'intero programma è pari al costo della parte più costosa.

Regole di calcolo: Regole 2 e 2'

molto utile nella valutazione di programmi ricorsivi

REGOLA 2 Se un programma richiede per $k(n)$ volte l'esecuzione di una istruzione composta o l'attivazione di una procedura, e sia $f_i(n)$ il costo della esecuzione i-esima $(1 \geq i \geq k(n))$, allora il costo del programma e'

$$O(\sum_i f_i(n))$$

Regole di calcolo: Regole 2 e 2'

molto utile nella valutazione di programmi ricorsivi

REGOLA 2 Se un programma richiede per $k(n)$ volte l'esecuzione di una istruzione composta o l'attivazione di una procedura, e sia $f_i(n)$ il costo della esecuzione i-esima $(1 \geq i \geq k(n))$, allora il costo del programma e'

$$O(\sum_i f_i(n))$$

Regole di calcolo: Regole 2 e 2'

molto utile nella valutazione di programmi ricorsivi

REGOLA 2 Se un programma richiede per $k(n)$ volte l'esecuzione di una istruzione composta o l'attivazione di una procedura, e sia $f_i(n)$ il costo della esecuzione i-esima $(1 \geq i \geq k(n))$, allora il costo del programma e'

$$O(\sum_i f_i(n))$$

REGOLA 2' Se le $f_i(n)$ sono tutte equali e se $k(n)$ è il loro numero, allora

$$O(\sum_i f_i(n)) = O(k(n)f(n))$$

Regole di calcolo: Regola 3 (Istruzioni dominanti)

Sia dato un algoritmo il cui costo di esecuzione è $T(n)$.

Una istruzione si dice istruzione dominante se, per ogni intero n, essa viene eseguita nel caso peggiore di input con dimensione n, un numero di volte $d(n)$ che soddisfa, per opportune costanti a e b, la condizione

$$T(n) < a\,d(n) + b$$

REGOLA 3 La complessita' di un algoritmo e' quella delle sue istruzioni dominanti

Regole di calcolo: Regola 3 (Istruzioni dominanti)

Sia dato un algoritmo il cui costo di esecuzione è $T(n)$.

Una istruzione si dice istruzione dominante se, per ogni intero n, essa viene eseguita nel caso peggiore di input con dimensione n, un numero di volte $d(n)$ che soddisfa, per opportune costanti a e b, la condizione

$$T(n) < a\,d(n) + b$$

REGOLA 3 La complessita' di un algoritmo e' quella delle sue istruzioni dominanti

Regole di calcolo: Regola 3 (Istruzioni dominanti)

Sia dato un algoritmo il cui costo di esecuzione è $T(n)$.

Una istruzione si dice istruzione dominante se, per ogni intero n, essa viene eseguita nel caso peggiore di input con dimensione n, un numero di volte $d(n)$ che soddisfa, per opportune costanti a e b, la condizione

$$T(n) < a\,d(n) + b$$

REGOLA 3 La complessita' di un algoritmo e' quella delle sue istruzioni dominanti

NOTA BENE

Per trovare le istruzioni dominanti si esaminano i cicli piu' interni di un algoritmo

Section 7

Complessita' di un problema

Delimitazioni superiore e inferiore

Un problema ha una delimitazione superiore $O(f(n))$ alla sua complessità se esiste almeno un algoritmo di soluzione che ha complessità $O(f(n))$.

Un problema ha una delimitazione inferiore $\Omega(g(n))$ alla sua complessità se si può dimostrare che ogni algoritmo di soluzione deve avere costo $\Omega(g(n))$.

Delimitazioni superiore e inferiore

Un problema ha una delimitazione superiore $O(f(n))$ alla sua complessità se esiste almeno un algoritmo di soluzione che ha complessità $O(f(n))$.

Un problema ha una delimitazione inferiore $\Omega(g(n))$ alla sua complessità se si può dimostrare che ogni algoritmo di soluzione deve avere costo $\Omega(g(n))$.

Nota Bene

Ciò non si riferisce solo agli algoritmi noti, ma ad ogni possibile algoritmo già progettato o ancora da progettare.

Section 8

Esempio: valutazione dei polinomi

Definizioni

Un polinomio univariato di grado n a coefficienti interi è una funzione

$$p : \mathbb{R} \to \mathbb{R} \qquad \text{tale che}$$

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Gli $(a_i), 0 < i < n$, si dicono coefficienti del polinomio, e lo rappresentano completamente.

Valutare un polinomio $p \equiv (a_i)$ in corrispondenza del valore x della variabile significa calcolare $p(x)$.

Valutazione dei polinomi 1/3

Primo metodo

```
    function pol1(a,n,x)
1       val = a[1]
2       for i=1:n
3           y = 1
4           for j=1:i
5               y = y * x
6           end
7           val += a[i+1] * y
8       end
9       return val
    end
```

Test

```
julia> a = [10,1,2,3,4];

julia> pol1(a,1,1.0)   # 11.0
julia> pol1(a,1,2.0)   # 12.0
julia> pol1(a,1,3.0)   # 13.0
julia> pol1(a,2,1.0)   # 13.0
julia> pol1(a,2,2.0)   # 20.0
julia> pol1(a,2,0.0)   # 10.0
julia> pol1(a,4,0.0)   # 10.0
```

Istruzione dominante: 5

eseguita $1 + 2 + \cdots + (n-1) + n$ volte: $T(n) = \frac{1}{2}n(n+1) + 2n + 1$

Complessità di tempo: $O(n^2)$

Valutazione dei polinomi 2/3

Secondo metodo

```
    function pol2(a,n,x)
1       val = a[1]
2       y = 1
3       for i=1:n
4           y *= x
5           val += a[i+1] * y
6       end
7       return val
    end
```

Istruzioni dominanti: 4,5
eseguite n volte: $T(n) = 2n + 3$

Complessità di tempo: $O(n)$

Test

```
julia> a = [10,1,2,3,4];

julia> pol2(a,1,1.0) # 11.0
julia> pol2(a,1,2.0) # 12.0
julia> pol2(a,1,3.0) # 13.0
julia> pol2(a,2,1.0) # 13.0
julia> pol2(a,2,2.0) # 20.0
julia> pol2(a,2,0.0) # 10.0
julia> pol2(a,4,0.0) # 10.0
```

Valutazione dei polinomi 3/3

Terzo metodo: Horner's method

$$p(x) = a_0 + x(a_1 + \cdots x(a_{n-1} + x(a_n)))$$

```
    function pol3(a,n,x)
1       val = a[n+1]
2       for i=n:-1:1
3           val = val * x + a[i]
4       end
5       return val
    end
```

```
julia> a = [10,1,2,3,4];
julia> pol3(a,1,1.0)   # 11.0
julia> pol3(a,1,2.0)   # 12.0
julia> pol3(a,1,3.0)   # 13.0
julia> pol3(a,2,1.0)   # 13.0
julia> pol3(a,2,2.0)   # 20.0
julia> pol3(a,2,0.0)   # 10.0
julia> pol3(a,4,0.0)   # 10.0
```

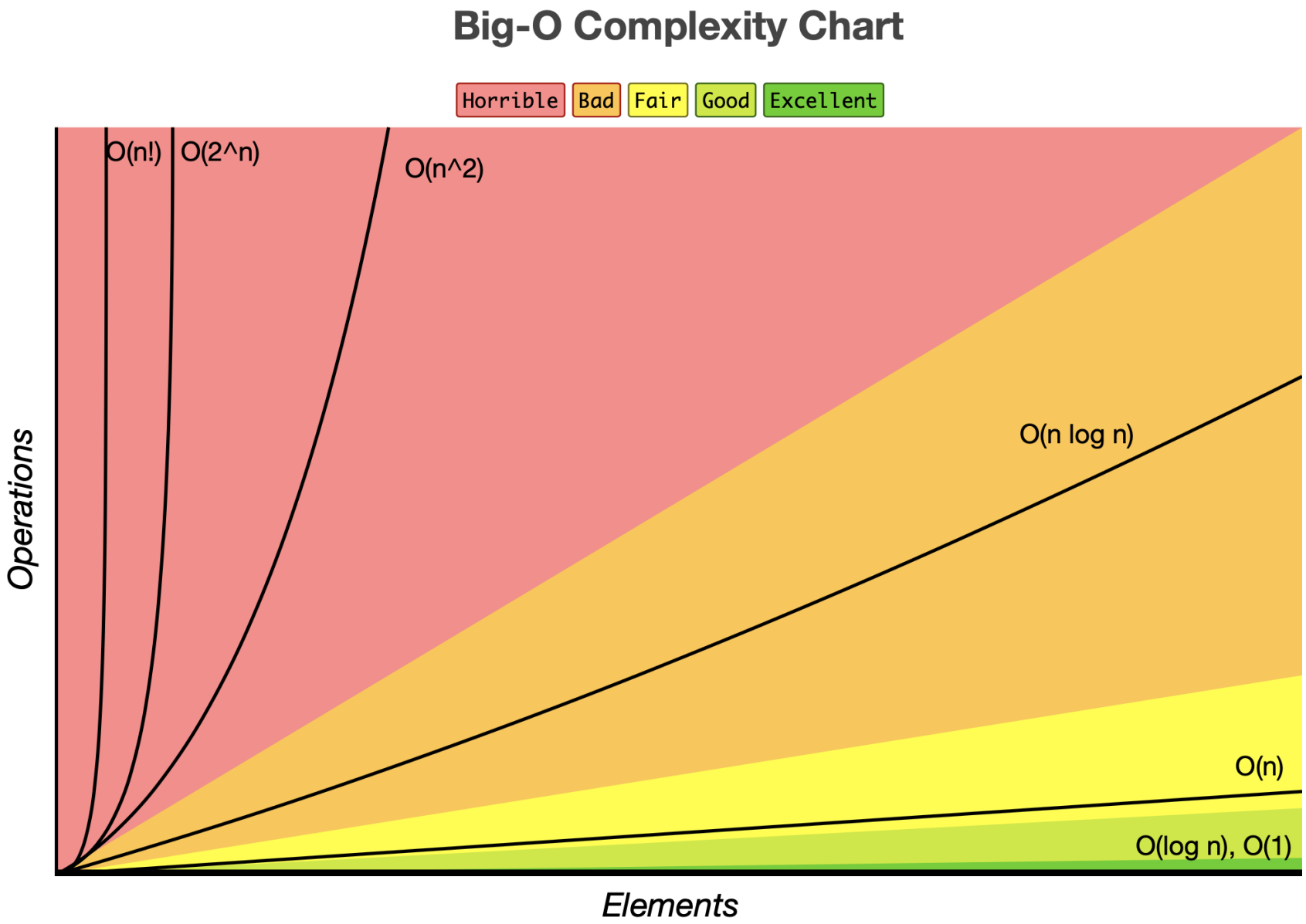
Istruzione dominante: 3
eseguite n volte: $T(n) = n + 2$

Complessità di tempo: $O(n)$

Section 9

Collezione di risultati di complessità

Confronto delle complessità asintotiche degli algoritmi



Complessità delle operazioni su strutture dati comuni

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Algoritmi di ordinamento: complessità di tempo e di spazio

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)