

# Complessità degli algoritmi e analisi di efficienza dei programmi

**Algoritmi e Strutture Dati + Lab**

A.A. 14/15

Informatica

Università degli Studi di Bari "Aldo Moro"

Nicola Di Mauro

# Teoria della complessità

- Lo studio della computabilità aiuta a stabilire quali problemi ammettono una soluzione algoritmica e quali no. Per i problemi computabili, è interessante conoscere la complessità degli algoritmi che li risolvono.
- *Complessità di un algoritmo*: è una misura delle risorse di calcolo consumate durante la computazione.
- *Efficienza di un algoritmo*: è inversamente proporzionale alla sua complessità.
- La teoria della complessità studia l'uso delle risorse di calcolo necessarie per la computazione di algoritmi e, più in generale, per la soluzione di problemi. Questo studio è importante al fine di:
  - poter confrontare algoritmi diversi che risolvono lo stesso problema;
  - stabilire se dei problemi ammettono algoritmi risolutivi che sono praticamente computabili.

# Valutazione dell'efficienza di programmi

- Intuitivamente un programma è più efficiente di un altro se la sua esecuzione richiede meno risorse di calcolo.
- Le risorse di calcolo che si considerano sono
  - Il tempo di elaborazione richiesto
  - La quantità di memoria necessaria
- Se si potesse valutare il tempo di calcolo in unità standard come i secondi la valutazione del costo di un programma sarebbe semplice: basterebbe eseguire il programma misurando il tempo necessario per l'esecuzione.

# Valutazione dell'efficienza di programmi /2

- Il metodo proposto non è attendibile perché bisogna considerare le condizioni in cui si effettuano le prove. Bisogna considerare:
  - L'elaboratore su cui il programma viene eseguito
  - Il compilatore usato per la traduzione
  - La modalità di ingresso dei dati
  - La significatività dei dati sui quali è eseguita la prova
- Utilizzando unità di tempo standard come i secondi non otteniamo una valutazione oggettiva del costo di esecuzione di un programma

# Complessità computazionale

- L'efficienza di un programma è legata alla complessità dell'algoritmo.
- La teoria della complessità studia in modo oggettivo l'uso delle risorse di calcolo necessarie per la computazione di algoritmi.
- Per risolvere un problema spesso sono disponibili molti algoritmi diversi .
  - Come scegliere il migliore?
- In genere si valuta la bontà di un algoritmo o si confrontano più algoritmi sulla base del comportamento che questi presentano al crescere della dimensione del problema.

# Complessità computazionale /2

- Questo studio è importante al fine di:
  - poter confrontare algoritmi diversi che risolvono lo stesso problema;
  - stabilire se dei problemi ammettono algoritmi risolutivi che sono praticamente computabili.
- Per i problemi computabili si parla di:
  - **Complessità dell'algoritmo**: è una misura delle risorse di calcolo consumate durante la computazione.
  - **Efficienza dell'algoritmo (e del programma)**: è inversamente proporzionale alla sua complessità.

# Risorse di interesse

- Le risorse di interesse sono TEMPO e SPAZIO DI MEMORIA
- Quindi, per un algoritmo, si parla di:
  - COMPLESSITA' IN TEMPO
  - COMPLESSITA' IN SPAZIO
- In ogni caso, per poter valutare la complessità occorre definire un MODELLO DI COSTO che dipende dal particolare modello di macchina astratta a cui si fa riferimento.
- Le unità di misura che si assumono in teoria della complessità non sono unità di tempo propriamente intese, per due ragioni:
  - Non ha senso parlare di tempi di esecuzione quando si considerano modelli di macchine astratte.
  - Anche se descrivessimo due algoritmi attraverso uno specifico linguaggio di programmazione disponibile per un calcolatore reale, il confronto dei tempi di esecuzione dei due programmi sarebbe lecito solo se i programmi fossero tradotti con il medesimo traduttore e fossero eseguiti con lo stesso elaboratore.

## Risorse di interesse /2

- L'analisi della efficienza di un programma è basata sull'ipotesi che il costo di ogni istruzione semplice e di ogni operazione di confronto sia pari ad un'unità di costo, indipendentemente dal linguaggio e dal sistema usato.
- Ci limiteremo a contare le operazioni eseguite o alcune operazioni chiave o preminenti ammettendo che il tempo complessivo di esecuzione sia proporzionale al numero di tali operazioni.
- Tratteremo spesso come non significative le costanti moltiplicative e studieremo le funzioni di complessità nel loro ordine di grandezza.



# La dimensione dell'input

- Il costo di esecuzione di un programma dipende anche dai particolari dati di ingresso.
- Ad esempio il costo di esecuzione di un programma di ordinamento di un insieme di numeri dipende dalle dimensioni dell'insieme che si considera.
- Per tener conto del numero di dati con cui si esegue il programma assumiamo che la dimensione dell'input rappresenta l'argomento della funzione che esprime il costo di esecuzione di un programma.

# La dimensione dell'input /2

- Si può caratterizzare tale dimensione mediante un intero  $n$  che è precisamente identificato nella Macchina di Turing come la lunghezza della porzione di nastro che contiene i dati di ingresso.
- Impiegando un elaboratore ed un suo linguaggio di programmazione, la dimensione  $n$  è lo spazio occupato, nella memoria dell'elaboratore, dai dati relativi al problema da risolvere, o più in generale un numero proporzionale a questo spazio.

## Esempi:

- se si opera su matrici  $n$  sarà il numero dei suoi elementi
- se si opera su un insieme,  $n$  sarà il numero dei suoi elementi
- se si opera su un grafo,  $n$  sarà il numero dei nodi o il numero di archi o la somma dei due numeri

# Complessità in spazio

- La complessità in spazio è il massimo spazio invaso dalla memoria durante l'esecuzione dell'algoritmo, il quale può costruire insiemi di dati intermedi o di servizio, oltre ad operare sui dati iniziali e finali.
- Anche in questo caso ci si limita in genere allo studio della complessità asintotica.
- Poiché abbiamo a disposizione memorie grandissime a basso costo, studieremo la complessità in tempo.

# Complessità in spazio /2

- Il costo è unario per i dati di tipo semplice mentre è dato da:
  - $n * \text{dimensione di un elemento}$
- per array di  $n$  elementi.
- Il costo di un record è dato dalla somma dei costi delle singole componenti.
- Esempio:
  - L'algoritmo
    - $\text{somma} = 0$
    - for  $i = 1$  to  $n$  do  $\text{somma} = \text{somma} + A[i]$
  - ha complessità in spazio pari a 13 se  $n = 10$

# Complessità in tempo e asintotica

- Fissata la dimensione  $n$ , il tempo che un algoritmo impiega a risolvere il problema è la **complessità in tempo**
- Nostro obiettivo principale sarà esprimere la complessità in tempo come funzione di  $n$  e spesso ci limiteremo a studiare il comportamento di tale funzione al crescere di  $n$  (complessità asintotica o semplicemente complessità) considerando così i soli termini prevalenti e tralasciando a volte anche le costanti moltiplicative.

# Complessità asintotica

- Lo studio della complessità asintotica è motivato dal fatto che gli algoritmi sono sempre definiti per  $n$  generico:
  - se per valori piccoli di  $n$  due algoritmi possono avere efficienza confrontabile, è sempre quello che ha il termine massimo di grado più basso a richiedere minor tempo di esecuzione per un numero illimitato di valori di  $n$  superiori ad un opportuno valore  $n_0$ .

# Il modello di costo

- Il modello di costo che si utilizza è quello di un linguaggio di programmazione lineare.
- Assumiamo che il costo di esecuzione di ogni istruzione semplice
  - Assegnazione  $\leftarrow$
  - Aritmetica  $+, -, *, /$
  - logica AND, OR, NOT
  - Confronto  $\leq, \geq, <, >, =, \neq$
  - lettura/scrittura
- è unitario.
- Il costo di esecuzione di un'istruzione composta è pari alla somma dei costi delle istruzioni che la compongono

# Il modello di costo /2

- Il costo di un'operazione di selezione: If cond then S1 else S2 è dato da:
  - costo test cond + costo S1            se cond
  - costo test cond + costo S2            se  $\neg$  cond
- Il costo di un'istruzione di ciclo: while cond do S1 è dato da:
  - costo test cond + (costo test cond + costo S1) \* n            [se il ciclo è ripetuto n volte]
- Il costo di un ciclo "repeat": repeat S1 until cond è dato da:
  - (costo test cond + costo S1) \* n            [se il ciclo è ripetuto n volte]
- Il costo di una istruzione for: for i = 1 to n do S1 è equivalente al costo del seguente ciclo

```
i = 1
while i <= n do
  S1;
  i = i+1
```

- ed è quindi dato da:  $2 + (\text{costo di S1} + 3) * n$
- Esempio:
  - Fatt = 1
  - for i = 1 to n do fatt = fatt \* i;
- Per  $n = 3$ ,      costo =  $1 + 2 + (1 + 1 + 3) * 3 = 18$



# Esempio

```
int MIN (B: vettore per riferimento; int j, k)
int m, i;
    m = B [ j ];
    for i = j + 1 to k do
        if B[i] < m then
            m = B[i]
    return m
```

C1	1
0	1
C2	1
C3	n
C4	n-1
C5	n-1
C6	1

- Si noti che l'incremento dell'indice i nel ciclo for è ripetuto n volte e non n -1. Infatti il for, come spiegato precedentemente, è un modo più compatto di scrivere un while nel quale la condizione  $i \leq k$  deve essere verificata una volta in più per poter uscire dal ciclo quando i supera k.
- Il tempo di calcolo  $T(n)$  di MIN si ottiene sommando su tutte le istruzioni il prodotto del costo di ciascuna istruzione per il numero di volte che tale istruzione è eseguita:

$$\begin{aligned} T(n) &= C1 + C2 + C3 n + C4 (n - 1) + C5 (n - 1) + C6 = \\ &= (C3 + C4 + C5) n + (C1 + C2 + C6 - C4 - C5). \end{aligned}$$

- Pertanto, il tempo di calcolo di MIN può essere espresso come
  - $T(n) = an + b$ ,
- con a e b costanti intere positive.

# Complessità e configurazioni

- La complessità di un algoritmo non può sempre essere caratterizzata da una sola funzione di complessità.
- A parità di dimensione di dati, il tempo di esecuzione può dipendere dalla specifica configurazione dei dati.
- Si considerano di solito tre differenti tipi di complessità: complessità nel caso medio, ottimo e pessimo.

# Complessità media

- Valore della complessità di un algoritmo, mediato su tutte le possibili occorrenze iniziali dei dati.
  - Si usa spesso la probabilità.
  - Il calcolo è spesso difficile.

# Complessità nel caso ottimo

- Si ottiene considerando, a parità di dimensione dei dati, la configurazione che dà luogo al minimo tempo di esecuzione.
- Tale complessità è per lo più di interesse secondario anche se è abbastanza facile da determinare.

# Complessità nel caso pessimo

- Si intende la complessità relativa a quella particolare occorrenza iniziale dei dati per cui l'algoritmo ha comportamento pessimo.
- Tale funzione di complessità fornisce un limite superiore alla complessità, entro cui il funzionamento dell'algoritmo è sempre garantito.

# Tre tipi di complessità

Si considerano tre tipi di complessità:

*CASO PESSIMO*: si ottiene considerando quella particolare configurazione che, a parità di dimensione dei dati, dà luogo al massimo tempo di calcolo. La corrispondente funzione di complessità è indicata con  $f_{pess}(n)$ .

*CASO OTTIMO*: si ottiene considerando la configurazione che dà luogo al minimo tempo di calcolo. La corrispondente funzione di complessità è indicata con  $f_{ott}(n)$ .

*CASO MEDIO*: si riferisce al tempo di calcolo mediato su tutte le possibili configurazioni dei dati, sempre per una data dimensione  $n$ . Funzione di complessità:  $f_{med}(n)$ .

Esistono particolari algoritmi in cui tutte le possibili configurazioni dei dati sono equivalenti per quanto riguarda la complessità in tempo. In questo caso, la complessità ottima, pessima e media coincidono.

Esempi:

$f_{pess} = f_{ott} = f_{med}$  per l'algoritmo del fattoriale.

$f_{pess} = f_{ott} = f_{med}$  per l'algoritmo del Min di  $n$  elementi.

# Ricerca lineare

ricerca\_lineare (t : array per riferimento;  
k : tipo chiave; trovato : boolean per riferimento)

/\* la procedura ricerca nella tavola l'elemento avente chiave k, se la ricerca ha successo allora la variabile booleana trovato viene posta a true, altrimenti a false \*/

```
{1}      i ← 0
{2}      repeat
{3}          i ← i + 1
{4}      until ( t [i].chiave = k) or (i = n)
{5}      if ( t [i].chiave = k) then
{6}          trovato ← true
{7}      else trovato ← false
```

# Ricerca lineare /2

Il costo di esecuzione del programma dipende dalla posizione del particolare elemento che si vuole individuare: se l'elemento cercato è il primo della tavola allora si effettua un solo confronto (questa situazione rappresenta il *caso ottimo*); se l'elemento cercato è il secondo della tavola allora si effettuano due confronti e così via. Il *caso peggiore* è costituito dalla ricerca dell'ultimo elemento o da una ricerca infruttuosa, perchè in questo caso l'algoritmo esamina tutte le componenti dell'array ed esegue il ciclo n volte.

Valutiamo ora il costo del programma nel caso di ricerca dell'ultimo elemento:

- L'istruzione 1 è eseguita 1 volta;
- L'istruzione 3 è eseguita n volte;
- Il test dell'istruzione **repeat** è eseguito n volte;
- Il test dell'istruzione **if then else** è eseguito 1 volta;
- L'istruzione in 6 è eseguita 1 volta;
- L'istruzione in 7 non viene eseguita.

Pertanto il costo di esecuzione del programma è

$$1 + n + n + 1 + 1 = 3 + 2n$$



# Ricerca lineare /3

E' facile vedere che la stessa funzione esprime anche il costo del programma nel caso di ricerca infruttuosa.

Se si vuole valutare il comportamento del programma nel *caso medio* è necessario distinguere il caso di ricerca con successo da quello di ricerca infruttuosa. Nel caso di ricerca fruttuosa, se si assume che tutti gli elementi dell'array possano essere ricercati con uguale probabilità pari a  $1/n$ , allora è facile vedere che il programma richiede mediamente  $(n + 1)/2$  confronti. Infatti se ricerchiamo l' $i$ -esimo elemento si effettuano  $i$  confronti e, quindi, il numero di confronti medio è dato da:

$$\sum_{1 \leq i \leq n} \text{Prob}(E(i)) i = \sum_{1 \leq i \leq n} i/n = (n + 1)/2$$

dove  $\text{Prob}(E(i))$  è la probabilità che l'elemento da cercare nell'array sia quello in posizione  $i$ .

# Riassumendo

- La complessità di un algoritmo è funzione della dimensione dei dati, ovvero della mole dei dati del problema da risolvere. L'individuazione della dimensione dei dati è per lo più immediata.
- Determinare la complessità in tempo (o in spazio) di un algoritmo significa determinare una funzione di complessità  $f(n)$  che fornisca la misura del tempo (o dello spazio di memoria occupato), al variare della dimensione dei dati,  $n$ .
- Le funzioni di complessità sono caratterizzate da due proprietà:
  - Assumono solo valori positivi
  - Sono crescenti rispetto alla dimensione dei dati.

# Comportamento asintotico

- Per semplificare l'analisi, non si cerca di individuare esattamente la funzione che esprime il costo del programma ma si ritiene sufficiente stabilire il suo comportamento asintotico quando le dimensioni dell'input tendono all'infinito. Si trascurano così costanti moltiplicative e termini additivi di ordine inferiore.
- Due programmi i cui costi sono espressi rispettivamente dalle funzioni  $(3 + n)$  e  $(100n + 3027)$  sono caratterizzati dalla stessa complessità asintotica.
  - $(n + 3)$  esprime un costo minore della funzione  $(100n + 3027)$  e, quindi, ignorare la costante moltiplicativa può essere in alcuni casi una semplificazione eccessiva.
- N.B.: ignorando le costanti moltiplicative e i termini additivi di ordine inferiore, l'analisi per stabilire il costo di un algoritmo viene molto semplificata e le valutazioni ottenute, pur con questa approssimazione, permettono di giungere a risultati significativi.
- Lo studio asintotico offre un criterio chiaro per confrontare due algoritmi.

# Esempi

$$A_1: f_1(n) = 3n^2 - 4n + 2$$

$$A_2: f_2(n) = 2n + 3$$

$$n = 1, 2 \quad f_1(n) < f_2(n)$$

$$n = 3, 4 \quad f_1(n) > f_2(n)$$

Qual è l'algoritmo più efficiente?

$$A_2, \text{ perché } \forall n > 2 \quad f_1(n) > f_2(n)$$

Le funzioni di complessità:

$$f_1(n) = 3n^2 + n + 1$$

$$f_2(n) = 4n^2$$

$$f_3(n) = 4n^2 + 2n$$

sono asintoticamente equivalenti, in quanto crescono tutte con il quadrato di  $n$  (Nello studio asintotico vengono trascurate le costanti moltiplicative e i termini la cui crescita asintotica è inferiore a quella di altri termini).

# Delimitazioni della complessità

- Il costo di esecuzione di un programma, o di un algoritmo, si esprime come funzione delle dimensioni dell'input in cui si ignorano le costanti moltiplicative.
- Si definiscono inoltre diversi tipi di delimitazioni che formalizzano la complessità di un programma (o di un algoritmo).

# Notazioni

- $O(f(n))$  è l'insieme di tutte le funzioni  $g(n)$  tali che esistono due costanti positive  $c$  e  $m$  per cui
  - $g(n) \leq cf(n)$  per ogni  $n \geq m$
- $g(n) \in O(f(n))$  tradizionalmente si legge “ $g(n)$  è di ordine  $f(n)$ ” e fornisce un limite superiore al comportamento asintotico della funzione  $g$  ovvero  $g$  non si comporta asintoticamente peggio di  $f$ .

# Notazioni /2

- Applicata alla funzione di complessità, la notazione  $O$  ne delimita superiormente la crescita e fornisce un indicatore di bontà dell'algoritmo.
- La funzione:  
$$f(n) = 3n^2 + 3n - 1$$
  
è un  $O(n^2)$  dal momento che esistono due costanti  $c = 4$  ed  $m = 3$  tali che
  - $\forall n \geq 3 \quad f(n) \leq 4n^2$ .
- In generale,  $f(n)$  è un  $O(n^k)$ ,  $k \geq 2$

## Notazioni /3

- $\Omega(f(n))$  è l'insieme di tutte le funzioni  $g(n)$  tali che esistono due costanti positive  $c$  e  $m$  per cui
  - $g(n) \geq cf(n)$  per ogni  $n \geq m$ .
- **$g(n) \in \Omega(f(n))$**  tradizionalmente si legge “ $g(n)$  è di ordine  $\Omega f(n)$ ” e fornisce un limite inferiore al comportamento asintotico della funzione.



# Notazioni /4

- $\Theta(f(n))$  è l'insieme di tutte le funzioni  $g(n)$  che sono sia  $\Omega f(n)$  sia  $O f(n)$ . Formalmente è l'insieme di tutte le funzioni  $g(n)$  tali che esistano tre costanti positive  $c, d$ , ed  $m$  per cui
  - $df(n) \geq g(n) \geq cf(n)$
- $g(n) \in \Theta(f(n))$  tradizionalmente si legge “ $g(n)$  è di ordine  $\Theta f(n)$ ”; la  $g$  si comporta asintoticamente esattamente come la  $f$ , per cui l'andamento di  $f$  caratterizza precisamente quello di  $g$ .

# Notazioni /5

- la funzione  $h(n)=3n^2+3n-1$  è di ordine  $n^2$  perché esistono le costanti  $c=4$  e  $n_0=3$  per cui  $h(n)<4n^2$  per  $n\geq 3$ .
- Notiamo che qualsiasi polinomio di grado  $k$  è di ordine  $n^k$  e, in accordo con la definizione, è anche di ordine  $n^j$  con  $j>k$ .
- $g(n)=3n^2+2n+1$  è  $O(n^2)$  e anche  $O(n^3)$  ma è anche di ordine  $\Omega(4n^2)$ ,  $\Omega(n^2)$ ,  $\Omega(n)$ ; inoltre è di ordine  $\theta(n^2)$ ,  $\theta(4n^2)$ , ma non di ordine  $\theta(n)$  e neppure di ordine  $\theta(n^3)$

# Proprietà

- $g \in O(f)$  implica  $(f+g) \in O(f)$
- $f_1 \in O(g_1)$ ,  $f_2 \in O(g_2)$  implica  $f_1 + f_2 \in O(g_1 + g_2)$
- $O$  e  $\Omega$  sono relazioni riflessive e transitive
- $\theta$  è una relazione di equivalenza
- $f \in O(g)$  se e solo se  $g \in \Omega(f)$

# Valutazione complessità

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

Diamo una serie di regole che aiutano a trovare la delimitazione superiore della complessità'.

**Regola 1.** Supponiamo che il programma sia composto di due parti P e Q da eseguire sequenzialmente e che i costi di P e Q siano  $S(n) = O(f(n))$  e  $T(n) = O(g(n))$ . Allora il costo del programma è  $O(\max(f(n), g(n)))$ .

*Dimostrazione.* In questo caso il costo complessivo del programma è, chiaramente, pari a

$$S(n) + T(n)$$

Inoltre si noti che

1.  $S(n) = O(f(n)) \Rightarrow$  esistono costanti  $a'$  e  $n'$  tali che  $S(n) < a' f(n)$  per  $n > n'$ ;
2.  $T(n) = O(g(n)) \Rightarrow$  esistono costanti  $a''$  e  $n''$  tali che  $T(n) < a'' g(n)$  per  $n > n''$ ;

Ne deriva che, per  $n > \max(n', n'')$

$$S(n) + T(n) < a' f(n) + a'' g(n) < (a' + a'') \max(f(n), g(n))$$

Quindi,  $O(S(n) + T(n))$  è proprio  $O(\max(f(n), g(n)))$ .

# Valutazione complessità /2

La seconda regola che presentiamo permette di valutare il costo del programma quando esso richiede più volte l'esecuzione di un insieme di istruzioni o l'attivazione di una procedura.

**Regola 2.** Supponiamo che un programma richieda per  $k$  volte l'esecuzione di una istruzione composta o l'attivazione di una procedura, e sia  $f_i(n)$  il costo relativo all'esecuzione  $i$ -esima,  $i = 1, 2, \dots, k$ . Il costo complessivo del programma è pari a

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

# Istruzione dominante

Introduciamo ora il concetto di istruzione dominante che permette, in molti casi, di semplificare in modo drastico la valutazione della complessità di un programma.

*Definizione 3.* Sia dato un programma o un algoritmo  $P$  il cui costo di esecuzione è  $t(n)$ . Una istruzione di  $P$  si dice *istruzione o operazione dominante* se, per ogni intero  $n$ , essa viene eseguita, nel caso peggiore di input avente dimensione  $n$ , un numero di volte  $d(n)$  che verifica la seguente condizione

$$t(n) < a d(n) + b$$

per opportune costanti  $a$  e  $b$ .

In altre parole, un'istruzione dominante viene eseguita un numero di volte proporzionale al costo di esecuzione di tutto l'algoritmo. E' importante osservare che in un programma, più istruzioni possono essere dominanti, ma può anche accadere che il programma non contenga affatto istruzioni dominanti.

# Istruzione dominante

**Regola 3.** Supponiamo che un programma contenga un'istruzione dominante che, nel caso peggiore di input di dimensione  $n$ , viene eseguita  $d(n)$  volte. La delimitazione superiore alla complessità del programma è  $O(d(n))$ .

Per individuare un'istruzione dominante è sufficiente, in molti casi, esaminare le operazioni che sono contenute nei cicli più interni del programma.

Ad esempio, nel caso della ricerca sequenziale considerata in un esempio precedente, è possibile valutare il costo del programma mostrato osservando che l'istruzione {3} è dominante. Infatti essa viene eseguita, nel caso peggiore,  $n$  volte; questo è sufficiente per dire che il programma ha costo lineare. Osserviamo inoltre che il test {4} del ciclo *repeat until* viene eseguito nel caso peggiore  $n$  volte e, quindi, rappresenta un'altra istruzione dominante.

# Classi di complessità

Grazie all'analisi della complessità asintotica possiamo categorizzare gli algoritmi nelle seguenti classi di complessità:

costante  $O(1)$

logaritmica  $O(\log n)$

lineare  $O(n)$

$n \log n$   $O(n \log n)$

quadratica  $O(n^2)$

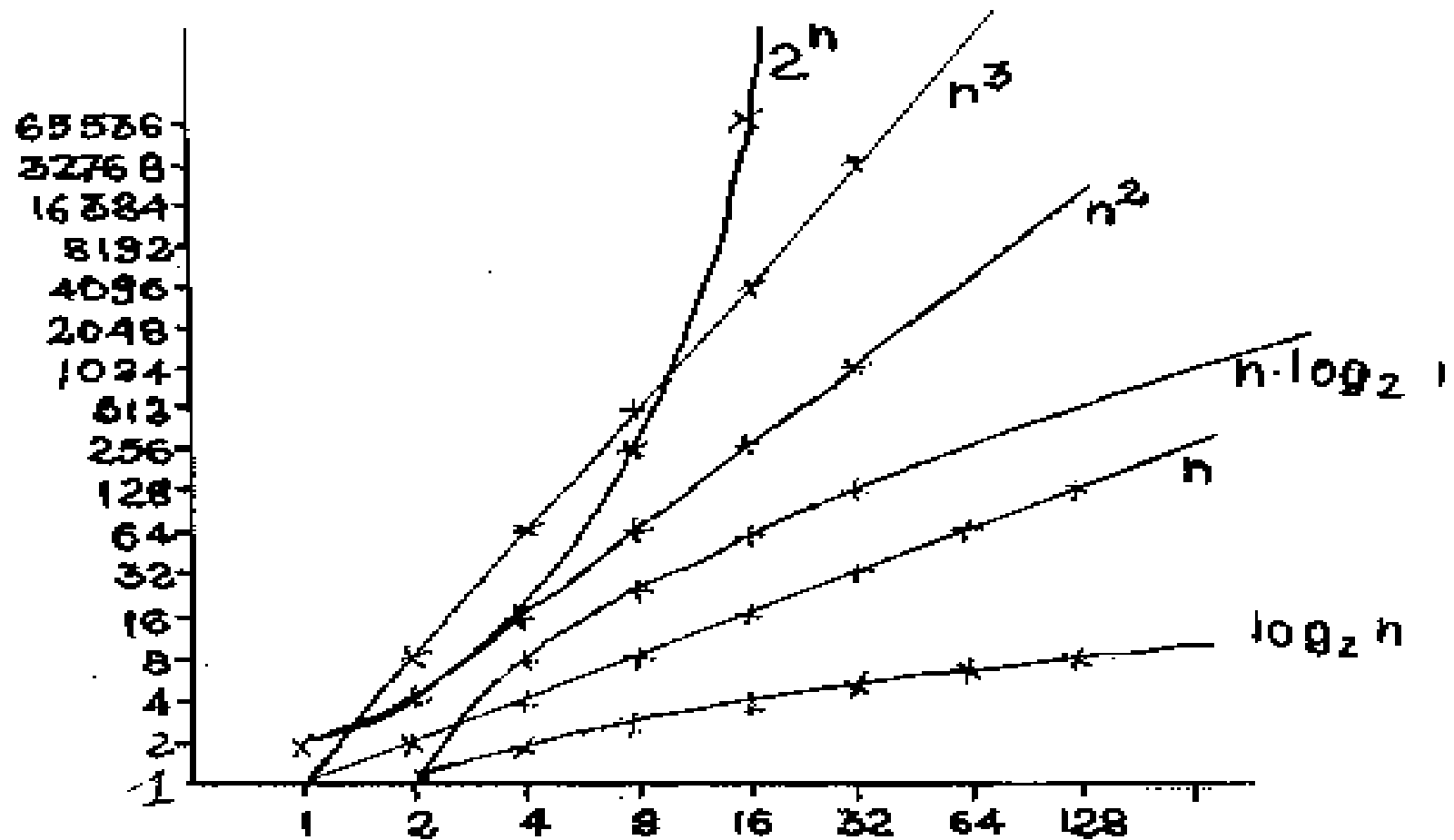
cubica  $O(n^3)$

esponenziale  $O(a^n)$   $a > 1$

Gli algoritmi con complessità costante sono più efficienti di quelli con complessità logaritmica che a loro volta sono più efficienti di quelli con complessità lineare e così via.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(a^n)$$





$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \dots < O(2^n)$$

# Algoritmi polinomiali vs. esponenziali

La distinzione di maggiore interesse in teoria della complessità è fra:

algoritmi *polinomiali*

complessità  $O(n^k)$ ,  $k > 0$

Es.:  $O(\log n)$ ,  $O(n^{10})$

algoritmi *esponenziali*

complessità  $O(a^{g(n)})$ ,  $a > 1$   $g(n)$  funzione crescente

Es.:  $O(2^n)$ ,  $O(2^{\log n})$ ,  $O(n^{\log n})$

Gli algoritmi di importanza pratica sono quelli polinomiali.

# Esempio: ordinare un array

- Ordinare un elenco di elementi è un compito molto frequente
  - ordinare numeri in modo crescente
  - ordinare numeri in modo decrescente
  - ordinare stringhe in modo alfabetico
- Vi sono molti modi per ordinare un elenco
  - Selection sort è uno dei più facili
  - non il più efficiente, ma facile da capire e da programmare

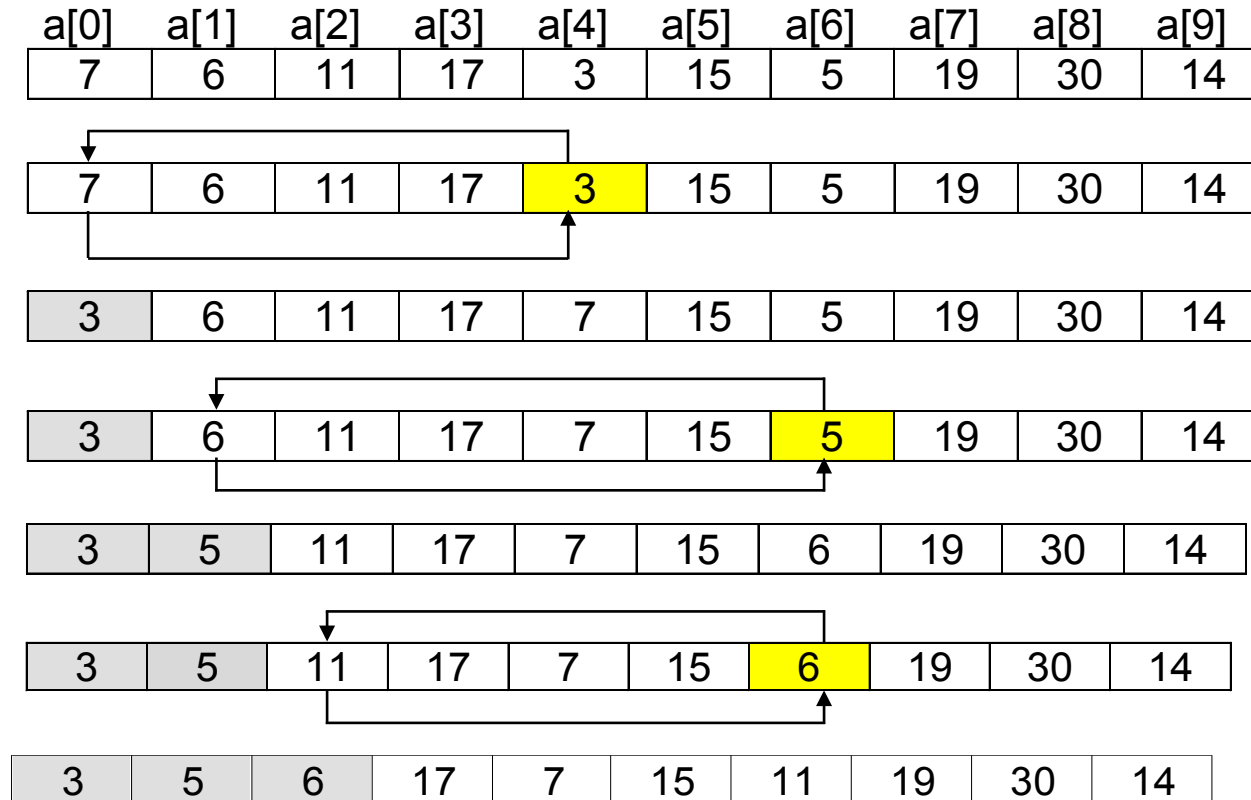
# Selection sort

- Per ordinare un array di interi in modo crescente:
  - cerca nell'array il numero più piccolo e scambialo con il primo elemento dell'array
    - la parte ordinata dell'array è ora il primo elemento, mentre quella non ancora ordinata sono i rimanenti elementi
  - cerca nella parte non ordinata il numero più piccolo e scambialo con il secondo elemento dell'array
  - ripeti la ricerca e lo scambio fino a quando tutti gli elementi sono al posto giusto
    - ogni iterazione aumenta di 1 la lunghezza della parte ordinata e diminuisce di 1 quella della parte non ordinata

# Selection sort

```
public void selectionSort(int[] a){
    int i, j, indexOfNextSmallest, min, temp;
    for (i = 0; i < a.length - 1; i++){
        min = a[i];
        indexOfNextSmallest = i;
        for (j = i+1; j < a.length; j++){
            if (a[j] < min){
                min = a[j];
                indexOfNextSmallest = j;
            }
        }
        temp = a[i];
        a[i] = a[indexOfNextSmallest];
        a[indexOfNextSmallest] = temp;
    }
}
```

# Selection sort



# Selection sort

- L'ordinamento per selezione ha un ciclo più esterno e uno più interno con proprietà simili, sebbene con scopi differenti.
- Il ciclo più esterno viene eseguito una volta per ogni valore nell'array e quello più interno confronta il valore scelto dal ciclo più esterno con molti, se non tutti, i valori rimanenti nell'array.
- Quindi esegue  $n^2$  confronti ove  $n$  è il valore di elementi dell'array. Selection sort è quindi di ordine  $n^2$ .

# Scomposizione dei costi

Per valutare il costo di un programma a partire dal costo delle sue componenti va affrontato il problema del costo di programmi con sottoprogrammi ripetuti.

Sia  $A$  un programma la cui esecuzione consiste nella esecuzione ripetuta di chiamate di funzione il cui costo all' $i$ -esima ripetizione è  $f_i(n)$  dove  $n$  è la dimensione dell'input di  $A$ . Sia  $g(n)$  il numero di ripetizioni legate alla dimensione dell'input. Il costo del programma  $f_a(n)$  è dato da

$$O\left(\sum_{i=1}^{g(n)} f_i(n)\right)$$



# Scomposizione dei costi

## Esempio

*Si consideri la funzione in grado di determinare se tutti gli elementi di un vettore  $w$  siano presenti in un altro vettore  $v$ . Per la ricerca si può utilizzare una banale ricerca sequenziale.*

```
Bool CercaInTutti (int v[], int w[])
```

```
// v e w sono vettori di N interi. Se tutti gli elementi di w sono // in v la funzione restituisce
```

```
// true, altrimenti restituisce false
```

```
{ for (int i= 0; i<N; i++)
```

```
if (!RicercaSequenziale(v, w[i]))
```

```
return false;
```

```
return true;
```

```
}
```

*Il numero massimo di chiamate al sottoprogramma `RicercaSequenziale()` è pari ad  $N$  e si ha quando tutti gli elementi di  $w$  vengono trovati in  $v$ . Poiché il costo dell'esecuzione del sottoprogramma è delimitato superiormente da  $O(N)$  si può concludere che `CercaInTutti()` è delimitato superiormente da  $O(N^2)$*

# Costi per ADT

- Da quanto detto si deduce che, poiché gli operatori sulle strutture dati definite da noi sono, di fatto, sottoprogrammi che hanno una loro complessità e vengono applicati più volte, il costo di una realizzazione piuttosto che un'altra incide sicuramente sul costo complessivo del programma che utilizza quegli operatori.
- Ad esempio, la complessità degli operatori su lista realizzati con cursori/puntatori è costante  $O(1)$  per tutti gli operatori e l'occupazione di memoria è  $O(\text{maxlung})$  per tutte le liste. Per le realizzazioni con cursori va però considerata INIZIALIZZALISTALIBERA

# Fusione di liste

- La complessità della procedura di fusione dipende dalla complessità degli operatori su lista utilizzati:
  - CREALISTA
  - PRIMOLISTA
  - FINELISTA
  - LEGGILISTA
  - INSLISTA
  - SUCCLISTA
- Realizzazione con puntatori e doppi puntatori: la complessità di questi operatori è  $O(1)$
- Realizzazione con cursori: occorre tenere presente il costo della procedura INIZIALIZZALISTALIBERA che comunque è eseguita fuori della procedura fusione e quindi non va conteggiato a fini della complessità della procedura.

# Fusione di liste /2

- Consideriamo come operazione dominante il confronto di elem1 con elem2. Allora la complessità dipende dalla configurazione dei dati di ingresso.
- In particolare, dette m ed n le lunghezze rispettivamente di Lista1 e Lista2, possiamo distinguere i seguenti casi:
- CASO OTTIMO:
  - tutti gli elementi della lista più corta sono minori degli elementi della lista più lunga. Si effettueranno tanti confronti quanti sono gli elementi della lista più corta:
  - $f_{\text{ott}}(m, n) = \min \{m, n\}$

# Fusione di liste /3

- CASO PESSIMO:

- l'inserimento dell'ultimo elemento di una delle due liste può avvenire solo dopo l'inserimento del penultimo elemento dell'altra lista.
- Il numero di confronti sarà dato da:  $f_{\text{pess}}(m, n) = m + n - 1$

- CASO MEDIO:

- senza perdere di generalità supponiamo essere  $m \leq n$ . Consideriamo allora i seguenti eventi:
- $E_k$  : l'ultimo elemento di Lista1 è inserito prima del k-esimo elemento di Lista2 per  $k = 1, 2, 3, \dots, n$ .
- Supponiamo che i vari eventi siano tutti equiprobabili (poiché non abbiamo informazioni sufficienti per sostenere il contrario, ipotizziamo una distribuzione uniforme), allora:
  - $P(E_k) = 1/n$
- poiché i diversi eventi sono  $n$ .

# Fusione di liste /4

Il costo, espresso in termini di numero di confronti, associato ad ogni evento è dato da:

$$\text{costo}(E_k) = m + k - 1$$

N.B.:  $E_1$  corrisponde al caso ottimo,  $E_n$  al caso pessimo

Allora la complessità nel caso medio sarà data da:

$$\begin{aligned} f_{\text{med}}(m, n) &= \sum_{k=1, \dots, n} P(E_k) \text{costo}(E_k) = \\ &= 1/n \sum_{k=1, \dots, n} (m + k - 1) = \\ &= 1/n \left( \sum_{k=1, \dots, n} m + \sum_{k=1, \dots, n} k - \sum_{k=1, \dots, n} 1 \right) = \\ &= 1/n (m n + (n(n + 1)/2) - n) = \\ &= m + (n + 1)/2 - 1 \end{aligned}$$