

Ingegneria degli Algoritmi

Salvatore Filippone
salvatore.filippone@uniroma2.it

Algoritmo

Termine derivato dal nome di Muhammad ibn Musa al-Khwarizmi, matematico persiano vissuto tra il 780 e l'850, sulla cui vita si hanno pochissime notizie.

Dal titolo del suo libro “al-Kitab al-mukhtasar fi hisab al-jabr wa al-muqabala” deriva il termine *algebra*; a questo libro si deve una esposizione sistematica del sistema di numerazione indiano che poi venne adottato in Europa e va sotto il nome di numerazione araba (ma in realtà ha origine in India).

Khwarizmi è una regione dell'Asia centrale, attualmente divisa tra Turkmenistan, Kazakistan e Uzbekistan, cuore di un impero musulmano che comprendeva anche Iran e Afghanistan tra il 1000 e il 1200, poi caduto nel 1231 sotto l'invasione dei mongoli.

Che cosa è un algoritmo?

Che cosa è un algoritmo?

Una procedura per risolvere un problema, ossia produrre una risposta a partire dai dati, che rispetta le caratteristiche:

- 1 Definitezza; (nessuna ambiguità)
- 2 Input;
- 3 Output;
- 4 Finitezza;
- 5 Efficacia (ciascun passo può essere eseguito in tempo finito).

Per definizione un algoritmo risponde *sempre* in un tempo *finito*.
Altrimenti stiamo parlando di un *metodo computazionale*.

Il più antico algoritmo attribuito ad un singolo individuo è quello di Euclide per il calcolo del massimo comune divisore di due interi

Il più antico algoritmo attribuito ad un singolo individuo è quello di Euclide per il calcolo del massimo comune divisore di due interi

Procedura Euclide(m, n)

Input: m ed n interi positivi, $m > n$

begin

 Dividere m per n e calcolare il resto r ;

while $r \neq 0$ **do**

$m \leftarrow n, n \leftarrow r$;

 Dividere m per n e calcolare il resto r ;

Result: n

Il più antico algoritmo attribuito ad un singolo individuo è quello di Euclide per il calcolo del massimo comune divisore di due interi

Procedura Euclide(m, n)

Input: m ed n interi positivi, $m > n$

begin

 Dividere m per n e calcolare il resto r ;

while $r \neq 0$ **do**

$m \leftarrow n, n \leftarrow r$;

 Dividere m per n e calcolare il resto r ;

Result: n

Esercizio: dimostrare che:

- 1 L'algoritmo termina;
- 2 Produce il risultato atteso, ossia $MCD(m, n)$.

Per ogni problema (specificato con sufficiente precisione) c'è un algoritmo che lo risolve.

Giusto ?

Per ogni problema (specificato con sufficiente precisione) c'è un algoritmo che lo risolve.

Giusto ?

NO!

Per ogni problema (specificato con sufficiente precisione) c'è un algoritmo che lo risolve.

Giusto ?

NO!

Teorema di Turing dell'arresto (correlato al teorema di incompletezza di Gödel's):

Esistono problemi per i quali NON può esistere un algoritmo risolutivo.

Per chiarire, può ben darsi una procedura che risolve alcune istanze del problema, ma su altre istanze non può terminare in un tempo finito.

Per ogni problema (specificato con sufficiente precisione) c'è un algoritmo che lo risolve.

Giusto ?

NO!

Teorema di Turing dell'arresto (correlato al teorema di incompletezza di Gödel's):

Esistono problemi per i quali NON può esistere un algoritmo risolutivo.

Per chiarire, può ben darsi una procedura che risolve alcune istanze del problema, ma su altre istanze non può terminare in un tempo finito.

Ironicamente uno dei problemi impossibili è:

Dato un programma di uno studente, decidere se esso terminerà dato un certo input oppure se si avvierà in un ciclo infinito

Nessun algoritmo può pienamente risolvere questo problema (anche se si possono riconoscere molti casi particolari)

Per ogni problema (specificato con sufficiente precisione) c'è *un solo* algoritmo che lo risolve.

Giusto ?

Per ogni problema (specificato con sufficiente precisione) c'è *un solo* algoritmo che lo risolve.

Giusto ?

Assolutamente no!

Per ogni problema (specificato con sufficiente precisione) c'è *un solo* algoritmo che lo risolve.

Giusto ?

Assolutamente no!

Non appena si dimostra che l'insieme degli algoritmi che risolvono un certo problema è non vuoto, si apre la discussione su quale sia il “migliore”, per una qualche definizione di “migliore”

Es: problema della programmazione lineare: algoritmi del simplesso e di Karmarkar.

Tornando alla proprietà di *finitezza*:

An algorithm should be VERY finite, not just finite. (D. Knuth)

Tornando alla proprietà di *finitezza*:

An algorithm should be VERY finite, not just finite. (D. Knuth)

È del tutto naturale provare a valutare la qualità di un algoritmo descrivendone:

Complessità temporale: il tempo $T(n)$ impiegato per risolvere un problema di dimensione n ;

Complessità spaziale: la quantità di memoria impiegata $S(n)$ per risolvere un problema di dimensione n .

Si noti che non abbiamo precisato del tutto cosa intendiamo per dimensione n .

Consideriamo il gioco degli scacchi; nel 1950 Claude Shannon scrisse un articolo ipotizzando un programma per giocare a scacchi; in teoria il programma potrebbe semplicemente considerare tutte le possibili sequenze di mosse e configurazioni della scacchiera (che sono in numero finito).

Consideriamo il gioco degli scacchi; nel 1950 Claude Shannon scrisse un articolo ipotizzando un programma per giocare a scacchi; in teoria il programma potrebbe semplicemente considerare tutte le possibili sequenze di mosse e configurazioni della scacchiera (che sono in numero finito).

- ① Numero medio di (semi)mosse legali per configurazione: 30;
- ② Numero medio di combinazioni di mosse Bianco/Nero: 10^3 ;
- ③ Lunghezza media di una partita: 40 mosse;

Consideriamo il gioco degli scacchi; nel 1950 Claude Shannon scrisse un articolo ipotizzando un programma per giocare a scacchi; in teoria il programma potrebbe semplicemente considerare tutte le possibili sequenze di mosse e configurazioni della scacchiera (che sono in numero finito).

- ① Numero medio di (semi)mosse legali per configurazione: 30;
- ② Numero medio di combinazioni di mosse Bianco/Nero: 10^3 ;
- ③ Lunghezza media di una partita: 40 mosse;

Quindi dovremmo enumerare

$$10^{120}$$

configurazioni possibili della schacchiera.

Consideriamo il gioco degli scacchi; nel 1950 Claude Shannon scrisse un articolo ipotizzando un programma per giocare a scacchi; in teoria il programma potrebbe semplicemente considerare tutte le possibili sequenze di mosse e configurazioni della scacchiera (che sono in numero finito).

- ① Numero medio di (semi)mosse legali per configurazione: 30;
- ② Numero medio di combinazioni di mosse Bianco/Nero: 10^3 ;
- ③ Lunghezza media di una partita: 40 mosse;

Quindi dovremmo enumerare

$$10^{120}$$

configurazioni possibili della schacchiera.

Per mettere in prospettiva:

- Numero di atomi nell'universo: 10^{80}
- Diametro dell'universo misurato in diametri elettronici: 10^{39}

La Trasformata di Fourier Discreta (dimensione N): uno strumento essenziale della tecnologia delle comunicazioni:

$$F(k) = \sum_{0 \leq j < N} \omega_N^{kj} f(j), \quad \omega_N^{kj} = e^{2\pi i \frac{jk}{N}} \quad 0 \leq k < N$$

La Trasformata di Fourier Discreta (dimensione N): uno strumento essenziale della tecnologia delle comunicazioni:

$$F(k) = \sum_{0 \leq j < N} \omega_N^{kj} f(j), \quad \omega_N^{kj} = e^{2\pi i \frac{jk}{N}} \quad 0 \leq k < N$$


Questo è un prodotto matrice-vettore (complesso), per cui il costo è $8N^2$ operazioni aritmetiche elementari,

La Trasformata di Fourier Discreta (dimensione N): uno strumento essenziale della tecnologia delle comunicazioni:

$$F(k) = \sum_{0 \leq j < N} \omega_N^{kj} f(j), \quad \omega_N^{kj} = e^{2\pi i \frac{jk}{N}} \quad 0 \leq k < N$$

Questo è un prodotto matrice-vettore (complesso), per cui il costo è $8N^2$ operazioni aritmetiche elementari, ma nel 1965 Cooley e Tukey (ri)scoprirono un modo di calcolarlo (FFT) con solo $5N \log(N)$ operazioni!

| Size | DFT | FFT |
|---------|-------|-------------|
| 10 | 800 | 166 |
| 100 | 80000 | 3321.93 |
| 1000 | 8e+06 | 49828.9 |
| 5000 | 4e+08 | 307193 |
| 10000 | 8e+08 | 664386 |
| 50000 | 4e+10 | 3.90241e+06 |
| 100000 | 8e+10 | 8.30482e+06 |
| 500000 | 4e+12 | 4.73289e+07 |
| 1000000 | 8e+12 | 9.96578e+07 |

Senza la FFT non avremmo: comunicazioni via satellite, telefoni cellulari, TAC, PET, VOIP, 

Dalla discussione precedente, possiamo ora dedurre cosa fa un informatico (la maggior parte del tempo) di fronte ad un problema:

- 1 Cerca di trovare almeno un algoritmo che lo risolva;

Dalla discussione precedente, possiamo ora dedurre cosa fa un informatico (la maggior parte del tempo) di fronte ad un problema:

- ❶ Cerca di trovare almeno un algoritmo che lo risolva;
- ❷ Studia l'esistenza di eventuali altri algoritmi di soluzione, e ne confronta l'efficienza;

Dalla discussione precedente, possiamo ora dedurre cosa fa un informatico (la maggior parte del tempo) di fronte ad un problema:

- ❶ Cerca di trovare almeno un algoritmo che lo risolva;
- ❷ Studia l'esistenza di eventuali altri algoritmi di soluzione, e ne confronta l'efficienza;
- ❸ Studia la rappresentazione dei dati necessaria alla soluzione del problema;

Dalla discussione precedente, possiamo ora dedurre cosa fa un informatico (la maggior parte del tempo) di fronte ad un problema:

- ❶ Cerca di trovare almeno un algoritmo che lo risolva;
- ❷ Studia l'esistenza di eventuali altri algoritmi di soluzione, e ne confronta l'efficienza;
- ❸ Studia la rappresentazione dei dati necessaria alla soluzione del problema;
- ❹ **Costruisce una "buona" implementazione, per una qualche definizione di "buona".**

Per un qualunque algoritmo si devono dimostrare la *terminazione* (in un tempo finito) e la *correttezza*.

In informatica la correttezza degli algoritmi si dimostra spesso per induzione, ovvero utilizzando degli *invarianti*

Per un qualunque algoritmo si devono dimostrare la *terminazione* (in un tempo finito) e la *correttezza*.

In informatica la correttezza degli algoritmi si dimostra spesso per induzione, ovvero utilizzando degli *invarianti*

Il principio di induzione matematica

Se abbiamo che:

- Una affermazione è vera per il numero 0 (o comunque per un valore minimo Q);
- La verità di una affermazione per il numero k implica la verità per il numero $k + 1$ (con $k \geq Q$);

Allora l'affermazione è vera per TUTTI i numeri naturali (maggiori o uguali a Q).

Dimostrazione per induzione

Se riusciamo a dimostrare che:

- La tesi è vera per $n = 0$ (oppure $n = 1$);
- Se la tesi è vera per $k = n - 1$, allora è vera anche per $k = n$ (la verità per $k = n - 1$ implica la verità per $k = n$);

Allora la tesi è vera per *qualsunque valore di n* .

Procedura Minsearch(A, n)

Input: Array contenente i valori $A[i]$, $i = 1, \dots, n$;

$min \leftarrow A[1]$;

for $i \leftarrow 2$ **to** n **do**

if $A[i] < min$ **then**

$min \leftarrow A[i]$;

Result: min

Procedura Minsearch(A, n)

Input: Array contenente i valori $A[i]$, $i = 1, \dots, n$;

$min \leftarrow A[1]$;

for $i \leftarrow 2$ **to** n **do**

if $A[i] < min$ **then**

$min \leftarrow A[i]$;

Result: min

Dimostrazione di correttezza

- 1 Prima dell'inizio del ciclo stiamo considerando solo il vettore $A[1 : 1]$, ed il valore di min è inizializzato correttamente;
- 2 Alla iterazione i stiamo considerando il vettore $A[1 : i]$, ma sappiamo che min contiene il minimo del vettore $A[1 : i - 1]$; allora considerando il valore $A[i]$, si può avere che $A[i] < min$ e min viene aggiornato, oppure no, e min rimane eguale. In entrambi i casi al termine della iterazione min contiene il minimo di $A[1 : i]$, ossia la proprietà invariante che volevamo.

Procedura $\text{BinarySearch}(A, v, i, j)$

Input: Array ordinato $A[i]$, $i = 1, \dots, n$, chiave v , estremi del sottovettore corrente i, j ;

if $i > j$ **then**

Result: 0

else

$m \leftarrow \lfloor (i + j) / 2 \rfloor$;

if $A[m] = v$ **then**

Result: m

else if $A[m] < v$ **then**

Result: $\text{BinarySearch}(A, v, m + 1, j)$

else

Result: $\text{BinarySearch}(A, v, i, m - 1)$

Procedura $\text{BinarySearch}(A, v, i, j)$

Input: Array ordinato $A[i]$, $i = 1, \dots, n$, chiave v , estremi del sottovettore corrente i, j ;

if $i > j$ **then**

Result: 0

else

$m \leftarrow \lfloor (i + j) / 2 \rfloor$;

if $A[m] = v$ **then**

Result: m

else if $A[m] < v$ **then**

Result: $\text{BinarySearch}(A, v, m + 1, j)$

else

Result: $\text{BinarySearch}(A, v, i, m - 1)$

Dimostrazione di correttezza (proprietà di tricotomia)

- Se il sottovettore corrente è vuoto $i > j$, allora la procedura risponde con il valore 0, cioè chiave non trovata;
- Altrimenti, se la chiave è eguale all'elemento mediano allora viene trovata correttamente;
- Altrimenti, se la chiave è maggiore dell'elemento mediano, si potrebbe trovare nel sottovettore destro, che è più piccolo del vettore corrente, e per induzione il risultato è corretto;
- Altrimenti, se la chiave è minore dell'elemento mediano, si potrebbe trovare nel sottovettore sinistro, che è più piccolo del vettore corrente, e per induzione il risultato è corretto.



Esempio: algoritmo di Euclide

Il Massimo Comune Divisore di due numeri $MCD(m, n) \geq 1$ è il più grande numero intero che divida entrambi i numeri dati.

Il Massimo Comune Divisore di due numeri $MCD(m, n) \geq 1$ è il più grande numero intero che divida entrambi i numeri dati.

Procedura Euclide(m, n)

Input: m ed n interi positivi, $m > n$

while $n \neq 0$ **do**

$(q, r) \leftarrow m/n$ (quoziente e resto);

$m \leftarrow n$;

$n \leftarrow r$;

Result: m

Il Massimo Comune Divisore di due numeri $MCD(m, n) \geq 1$ è il più grande numero intero che divida entrambi i numeri dati.

Procedura Euclide(m, n)

Input: m ed n interi positivi, $m > n$

while $n \neq 0$ **do**

$(q, r) \leftarrow m/n$ (quoziente e resto);

$m \leftarrow n$;

$n \leftarrow r$;

Result: m

Dimostrazione di terminazione

Nella divisione intera $(q, r) \leftarrow m/n$ abbiamo sempre $r < n$, quindi la sequenza r_1, r_2, \dots, r_k è *strettamente* discendente, e non può esistere nei numeri naturali una sequenza strettamente discendente ed infinita (altra formulazione del principio di induzione).

Dimostrazione di correttezza (parte 1)

$$(q_1, r_1) \leftarrow m/n \quad \Leftrightarrow \quad m = nq_1 + r_1$$

$$(q_2, r_2) \leftarrow n/r_1 \quad \Leftrightarrow \quad n = r_1q_2 + r_2$$

$$(q_3, r_3) \leftarrow r_1/r_2 \quad \Leftrightarrow \quad r_1 = r_2q_3 + r_3$$

$$(q_4, r_4) \leftarrow r_2/r_3 \quad \Leftrightarrow \quad r_2 = r_3q_4 + r_4$$

...

$$(q_{k-1}, r_{k-1}) \leftarrow r_{k-3}/r_{k-2} \quad \Leftrightarrow \quad r_{k-3} = r_{k-2}q_{k-1} + r_{k-1}$$

$$(q_k, 0) \leftarrow r_{k-2}/r_{k-1} \quad \Leftrightarrow \quad r_{k-2} = r_{k-1}q_k + 0$$

Dalle equazioni precedenti segue che:

- r_{k-1} divide esattamente r_{k-2} (in quanto r_k è zero);
- Ma siccome $r_{k-3} = r_{k-2}q_{k-1} + r_{k-1}$, allora r_{k-1} divide esattamente anche r_{k-3} ;
- E quindi anche r_{k-4} ; ...; E quindi r_{k-1} divide esattamente n ;
- E quindi r_{k-1} divide esattamente m ; da cui segue che r_{k-1} è *un divisore comune di m e n* ,

e quindi necessariamente

$$r_{k-1} \leq \text{MCD}(m, n).$$

Dimostrazione di correttezza (parte 2)

- $MCD(m, n)$ divide m ed n , quindi anche $m - nq_1 = r_1$;
- $MCD(m, n)$ divide n ed r_1 , quindi anche $n - r_1q_2 = r_2$;
- $MCD(m, n)$ divide r_1 ed r_2 , quindi anche $r_1 - r_2q_3 = r_3$;
- ...
- $MCD(m, n)$ divide r_{k-3} ed r_{k-2} , quindi anche $r_{k-3} - r_{k-2}q_{k-1} = r_{k-1}$;

Da cui segue che necessariamente abbiamo

$$MCD(m, n) \leq r_{k-1}.$$

Dimostrazione di correttezza (parte 2)

- $MCD(m, n)$ divide m ed n , quindi anche $m - nq_1 = r_1$;
- $MCD(m, n)$ divide n ed r_1 , quindi anche $n - r_1q_2 = r_2$;
- $MCD(m, n)$ divide r_1 ed r_2 , quindi anche $r_1 - r_2q_3 = r_3$;
- ...
- $MCD(m, n)$ divide r_{k-3} ed r_{k-2} , quindi anche $r_{k-3} - r_{k-2}q_{k-1} = r_{k-1}$;

Da cui segue che necessariamente abbiamo

$$MCD(m, n) \leq r_{k-1}.$$

Conclusione

Abbiamo dimostrato che r_{k-1} è un divisore comune di m ed n , e che valgono

$$r_{k-1} \leq MCD(m, n) \quad \text{e} \quad MCD(m, n) \leq r_{k-1},$$

per cui si *deve* avere

$$MCD(m, n) = r_{k-1},$$

ossia l'algoritmo calcola correttamente il MCD .

Complessità computazionale

Torniamo quindi alla domanda fondamentale:

Quanto costa risolvere un problema dato?

Siccome il tempo è denaro, ma anche viceversa, calcoliamo quanto tempo impiegherà un programma per risolvere il problema:

Identifichiamo tutte le operazioni eseguite dal programma su una certa istanza di un problema, e valutiamo quanto tempo costa ciascuna operazione.

In pratica spesso:

- Si seleziona e conta un sottoinsieme delle operazioni;
- Si assume che tutte le operazioni richiedano lo stesso tempo;

Queste assunzioni ovviamente producono solo una approssimazione del primo ordine (a volte parecchio grossolana).

Per una valutazione accurata si dovrebbe tenere conto della sequenza di operazioni, dell'architettura di calcolo e del sistema linguaggio/compiler; in questo corso non ci occuperemo se non marginalmente di questi dettagli.

Consideriamo come esempio il seguente frammento di codice (Matlab):

```
> a=[1,2,3];  
> b=[4,5,6];  
> a+b  
[5, 7, 9]
```

- Si tratta di una specifica istanza del problema di sommare tra di loro due vettori di lunghezza 3;
- che a sua volta è un caso particolare del problema di sommare due vettori di dimensione n .

In questo caso possiamo dire che il costo sia di 3 (n) operazioni (aritmetiche).

Introduciamo ora le comuni notazioni asintotiche per il tempo di esecuzione di un programma $T(n)$ su input di dimensione n :

- $T(n)$ è $O(f(n))$ se esistono $f(n)$, C e n_0 tali che

$$T(n) \leq C \cdot f(n) \quad \text{per ogni } n > n_0$$

Introduciamo ora le comuni notazioni asintotiche per il tempo di esecuzione di un programma $T(n)$ su input di dimensione n :

- $T(n)$ è $O(f(n))$ se esistono $f(n)$, C e n_0 tali che

$$T(n) \leq C \cdot f(n) \quad \text{per ogni } n > n_0$$

- $T(n)$ è $\Omega(f(n))$ se esistono $f(n)$, C e n_0 tali che

$$T(n) \geq C \cdot f(n) \quad \text{per infiniti valori } n > n_0$$

Introduciamo ora le comuni notazioni asintotiche per il tempo di esecuzione di un programma $T(n)$ su input di dimensione n :

- $T(n)$ è $O(f(n))$ se esistono $f(n)$, C e n_0 tali che

$$T(n) \leq C \cdot f(n) \quad \text{per ogni } n > n_0$$

- $T(n)$ è $\Omega(f(n))$ se esistono $f(n)$, C e n_0 tali che

$$T(n) \geq C \cdot f(n) \quad \text{per infiniti valori } n > n_0$$

- $T(n)$ è $\Theta(f(n))$ se esistono $f(n)$, C_1 , C_2 e n_0 tali che

$$C_1 \cdot f(n) \leq T(n) \leq C_2 \cdot f(n) \quad \text{per ogni } n > n_0$$

Cosa succede se la dimensione n non è sufficiente a determinare completamente il tempo di esecuzione?

Cosa succede se la dimensione n non è sufficiente a determinare completamente il tempo di esecuzione?

Worst case: $T(n)$ è $O(f(n))$ *per qualunque* input possibile di dimensione n ;

Average case: $T(n)$ è $O(f(n))$ *in media* su tutti gli input possibili di dimensione n ;

Best case: $T(n)$ è $\Omega(f(n))$ e questo valore viene raggiunto *per alcuni* degli input di dimensione n .

Tornando all'esempio precedente, possiamo ragionevolmente assumere che l'algoritmo per la somma di due vettori di dimensione n sia $\Theta(n)$ e che non ci sia differenza fra caso migliore, caso medio e caso peggiore.

Casi comuni:

- $O(1)$: algoritmo che richiede un tempo costante;
- $O(\log(n))$: algoritmo logaritmico;
- $O(n)$: algoritmo lineare
- $O(n^k)$: algoritmo polinomiale;
- $O(a^n)$: algoritmo esponenziale.

Attenzione al passaggio dei parametri

Gli algoritmi di complessità polinomiale sono considerati *trattabili*.

Normalmente preferiamo un algoritmo con una complessità asintotica inferiore: un algoritmo $O(n^2)$ avrà prima o poi prestazioni migliori un algoritmo $O(n^3)$, anche quando i suoi coefficienti siano più grandi.

Problemi vs algoritmi

Distinguiamo attentamente:

Complessità di un algoritmo La complessità di un particolare metodo per la soluzione di un problema;

Complessità di un problema La complessità del *migliore* algoritmo che risolve quel problema.

Ad esempio, l'ordinamento per inserzione è un algoritmo $O(n^2)$ mentre il problema dell'ordinamento per confronti ammette soluzioni in tempo $O(n \log(n))$.

Supponiamo di avere a disposizione un sistema di calcolo ed un programma che implementa un certo algoritmo: il sistema è in grado di risolvere un certo problema di dimensione N in un tempo T .

Tuttavia, sappiamo che i sistemi di calcolo diventano sempre più veloci nel tempo.

Se ora acquisisco un sistema due volte più veloce, quale dimensione N_2 riesco a gestire nello stesso tempo T ?

Supponiamo di avere a disposizione un sistema di calcolo ed un programma che implementa un certo algoritmo: il sistema è in grado di risolvere un certo problema di dimensione N in un tempo T .

Tuttavia, sappiamo che i sistemi di calcolo diventano sempre più veloci nel tempo.

Se ora acquisisco un sistema due volte più veloce, quale dimensione N_2 riesco a gestire nello stesso tempo T ?

| Algoritmo | Dimensione |
|-----------|--------------|
| $O(n)$ | $2 \times N$ |

Supponiamo di avere a disposizione un sistema di calcolo ed un programma che implementa un certo algoritmo: il sistema è in grado di risolvere un certo problema di dimensione N in un tempo T .

Tuttavia, sappiamo che i sistemi di calcolo diventano sempre più veloci nel tempo.

Se ora acquisisco un sistema due volte più veloce, quale dimensione N_2 riesco a gestire nello stesso tempo T ?

| Algoritmo | Dimensione |
|-----------|------------------|
| $O(n)$ | $2 \times N$ |
| $O(n^2)$ | $1.414 \times N$ |

Supponiamo di avere a disposizione un sistema di calcolo ed un programma che implementa un certo algoritmo: il sistema è in grado di risolvere un certo problema di dimensione N in un tempo T .

Tuttavia, sappiamo che i sistemi di calcolo diventano sempre più veloci nel tempo.

Se ora acquisisco un sistema due volte più veloce, quale dimensione N_2 riesco a gestire nello stesso tempo T ?

| Algoritmo | Dimensione |
|-----------|------------------|
| $O(n)$ | $2 \times N$ |
| $O(n^2)$ | $1.414 \times N$ |
| $O(n^3)$ | $1.276 \times N$ |

Supponiamo di avere a disposizione un sistema di calcolo ed un programma che implementa un certo algoritmo: il sistema è in grado di risolvere un certo problema di dimensione N in un tempo T .

Tuttavia, sappiamo che i sistemi di calcolo diventano sempre più veloci nel tempo.

Se ora acquisisco un sistema due volte più veloce, quale dimensione N_2 riesco a gestire nello stesso tempo T ?

| Algoritmo | Dimensione |
|-----------|------------------|
| $O(n)$ | $2 \times N$ |
| $O(n^2)$ | $1.414 \times N$ |
| $O(n^3)$ | $1.276 \times N$ |
| $O(2^n)$ | $N + 1$ |

Avvertenze:

- Se abbiamo a che fare con istanze piccole, può essere che l'algoritmo $O(n^3)$ sia migliore: $5n^3 < 100n^2$ per tutti gli n minori di 20. Alcuni algoritmi “ottimi” sono efficaci solo per problemi di dimensioni astronomiche, e quindi sono inutili in pratica;
- Se un programma verrà usato solo una volta o due, allora il tempo di sviluppo diventa molto più importante: un algoritmo semplice (e magari sviluppato in un linguaggio di scripting) può essere vincente;
- In alcuni casi l'algoritmo più veloce ha un costo di memoria eccessivo;
- In alcuni casi lo stesso algoritmo può essere il migliore nel caso *medio* ma anche molto scadente nel caso *peggiore*.

E per finire: mai tentare di migliorare un programma senza prima misurare e verificare le sue prestazioni

Premature optimization is the root of all evil

— D. Knuth

Problema dell'ordinamento: Abbiamo una collezione di oggetti (*records*) R_i ciascuno con una *chiave* K_i , $i = 1, \dots, n$. Le chiavi ammettono una

Relazione di ordinamento

- 1 Ogni coppia di chiavi K_i, K_j soddisfa esattamente una delle tre relazioni (tricotomia):
 $K_i < K_j$ o $K_i = K_j$ o $K_j < K_i$;
- 2 Se $K_i < K_q$ e $K_q < K_j$ allora $K_i < K_j$ (transitività).

Insieme ordinato

Vogliamo modificare l'insieme delle chiavi in modo tale che

$$i < j \Rightarrow K_i \leq K_j.$$

Il problema dell'ordinamento ammette molti metodi risolutivi. Uno dei più semplici è l'ordinamento per *inserzione*

Procedura InsertionSorting(A, n)

Input: Array contenente le chiavi $A[i]$, $i = 1, \dots, n$;

for $i \leftarrow 2$ **to** n **do**

$temp \leftarrow A[i]$;

$j \leftarrow i$;

while $j > 1$ **and** $A[j - 1] > temp$ **do**

$A[j] \leftarrow A[j - 1]$;

$j \leftarrow j - 1$;

$A[j] \leftarrow temp$;

Abbiamo ora due problemi:

- 1 Dimostrare che l'algoritmo effettivamente produce in uscita una sequenza ordinata;
- 2 Valutare la sua complessità.

Lasciamo il primo punto per ora in sospeso (da rivedere nella discussione degli algoritmi di ordinamento), ed esaminiamo invece il secondo.

Come valutare il costo di un algoritmo

- Le espressioni scalari elementari hanno costo $O(1)$;
- Il costo di una sequenza di istruzioni è dato dalla somma dei singoli costi;
- Il costo di un ciclo è la somma del costo delle singole iterazioni sull'insieme di tutte le iterazioni;
- Una istruzione condizionale ha un costo nel caso peggiore che è il massimo tra i costi della ramo `if` e del ramo `else`; per il costo *medio* occorre stimare la probabilità di ciascun ramo. In più bisogna stimare il costo della valutazione della condizione.

Queste regole possono portarci molto lontano.

Istruzioni su quantità scalari

```
a = 2.5;      % 0 or 1: Cost of assignment is often ignored;

b = a*a+1;    % Here we have 2 floating point operations;
c = b^3;      % b^3 is b*b*b, so again 2 operations;

for k=n1:n2 % This is executed (n2-n1+1) times
    c=a+b    % Cost here is 1 independent of K
end          % total cost: 1*(n2-n1+1)

if (mod(k,2) == 0) % If K is a random integer 50% prob.
    c=a*b+c;      % worst case is IF branch of cost 2
else              % average case costs 1.5
    b=b+1;        % plus 2 for evaluating (MOD()==0)
end
```


Cicli: bisogna calcolare

$$\sum_{i \in \mathcal{I}} C(i)$$

- i è una iterazione;
- \mathcal{I} è l'insieme di tutte le iterazioni;
- $C(i)$ è il costo della i -esima iterazione.

Spesso (ma non sempre) il costo per iterazione è costante; trovare \mathcal{I} è facile per i cicli `for`, ma non necessariamente per i cicli `while`.

A cicli innestati corrispondono somme multiple:

```
for  $i \leftarrow 1$  to  $m$  do  
  | for  $j \leftarrow 1$  to  $n$  do  
  |   <statement>;
```

$$Opcnt = \sum_{i=1}^m \sum_{j=1}^n C(\text{statement}_{ij}).$$

Somma (scalata) di vettori di dimensione n : $c = a + \text{alpha} * b$;

Procedura $\text{axpy}(\text{alpha}, a, b, c, n)$

for $i \leftarrow 1$ **to** n **do**

$c[i] \leftarrow a[i] + \alpha \times b[i]$;

Costo:

$$\sum_{i=1}^n 2 = 2 \sum_{i=1}^n 1 = 2n$$

Prodotto scalare di due vettori di dimensione n : $c = \mathbf{x}' * \mathbf{y}$;

Procedura $\text{dot}(x, y, n, c)$

begin

$s \leftarrow 0$;

for $i \leftarrow 1$ **to** n **do**

$s \leftarrow s + x[i] \times y[i]$;

Result: s

Costo:

$$\sum_{i=1}^n 2 = 2 \sum_{i=1}^n 1 = 2n$$

Prodotto matrice-vettore $y = y + A \cdot x$ (matrice $m \times n$):

Procedura $mv(A, x, y, m, n)$

```
for  $i \leftarrow 1$  to  $m$  do  
  | for  $j \leftarrow 1$  to  $n$  do  
    |  $y[i] \leftarrow y[i] + A[i][j] \times x[j];$ 
```

Cost:

$$\sum_{i=1}^m \sum_{j=1}^n 2 = \sum_{i=1}^m 2 \sum_{j=1}^n 1 = \sum_{i=1}^m 2n = 2n \sum_{i=1}^m 1 = 2mn$$

Prodotto matrice-matrice $C = C + A * B$ (matrici $m \times k \times n$):

Procedura mm(A, B, C, m, n, k)

```
for  $i \leftarrow 1$  to  $m$  do
  for  $j \leftarrow 1$  to  $n$  do
    for  $l \leftarrow 1$  to  $k$  do
       $C[i][j] \leftarrow C[i][j] + A[i][l] \times B[l][j];$ 
```

Costo:

$$\sum_{i=1}^m \sum_{j=1}^n \sum_{l=1}^k 2 = 2mnk$$

Se la matrice dei coefficienti di un sistema è triangolare inferiore L , il sistema $Lx = b$ si risolve facilmente per sostituzione in avanti:

Se la matrice dei coefficienti di un sistema è triangolare inferiore L , il sistema $Lx = b$ si risolve facilmente per sostituzione in avanti:

Procedura $\text{trs}(L, x, n)$

```
for  $i \leftarrow 1$  to  $n$  do  
     $x[i] \leftarrow b[i];$   
    for  $j \leftarrow 1$  to  $i - 1$  do  
         $x[i] \leftarrow x[i] - L[i][j] \times x[j];$   
     $x[i] \leftarrow x[i] / L[i][i];$ 
```

Se la diagonale è unitaria, il passo di divisione può essere saltato. Andiamo ora a dimostrare che il numero di operazioni è $O(n^2)$.

Quale è il costo?

- Ad ogni iterazione del ciclo esterno $i = 1 \dots n$, abbiamo un ciclo interno, una assegnazione (che ora ignoriamo) e una divisione;
- Il ciclo interno $j = 1, \dots, i-1$ è un prodotto scalare, e contiene 2 operazioni per iterazione.

Ogni iterazione del ciclo esterno ha un costo diverso! Quindi:

$$\begin{aligned} \text{opcnt} &= \sum_{i=1}^n \left(1 + \left(\sum_{j=1}^{i-1} 2 \right) \right) = \left(\sum_{i=1}^n 1 \right) + 2 \left(\sum_{i=1}^n \sum_{j=1}^{i-1} 1 \right) \\ \left(\sum_{i=1}^n 1 \right) + 2 \left(\sum_{i=1}^n (i-1) \right) &= n + 2 \sum_{i=0}^{n-1} i = n + 2 \frac{(n-1)n}{2} \\ &= n + n^2 - n = n^2 = O(n^2) \end{aligned}$$

Espressioni utili:

- $$\sum_{i=n_1}^{n_2} 1 = n_2 - n_1 + 1$$

- $$\sum_{i=0}^n i = \frac{n(n+1)}{2} \approx \frac{n^2}{2} = O(n^2)$$

- $$\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3} = O(n^3)$$

- Il termine dominante può essere calcolato con un integrale

$$\sum_{i=0}^n i^3 \approx \frac{n^4}{4} = \int_0^n x^3 dx$$

Riesaminiamo l'algoritmo di insertion sorting

Procedura InsertionSorting(A, n)

Input: Array contenente le chiavi $A[i]$, $i = 1, \dots, n$;

for $i \leftarrow 2$ **to** n **do**

$temp \leftarrow A[i]$;

$j \leftarrow i$;

while $j > 1$ **and** $A[j - 1] > temp$ **do**

$A[j] \leftarrow A[j - 1]$;

$j \leftarrow j - 1$;

$A[j] \leftarrow temp$;

Abbiamo una struttura simile a quella del sistema triangolare, ma per il ciclo interno:

- Nel caso migliore, il ciclo **while** esegue un controllo ed esce immediatamente;
- Nel caso peggiore viene eseguito $i - 1$ volte;

Abbiamo quindi una complessità $\Omega(n)$ e $O(n^2)$.

Vediamo ora l'algoritmo di ricerca binaria

Procedura BinarySearch(A, v, i, j)

Input: Array ordinato $A[i]$, $i = 1, \dots, n$, chiave v , estremi del sottovettore corrente i, j ;

if $i > j$ **then**

Result: 0

else

$m \leftarrow \lfloor (i + j) / 2 \rfloor$;

if $A[m] = v$ **then**

Result: m

else if $A[m] < v$ **then**

Result: BinarySearch($A, v, m + 1, j$)

else

Result: BinarySearch($A, v, i, m - 1$)

- ① Alla prima invocazione di BinarySearch la dimensione del vettore è n ;
- ② Ad ogni invocazione ricorsiva ulteriore la dimensione si dimezza;
- ③ Nel caso peggiore è quindi necessario eseguire un numero di chiamate ricorsive k tale che

$$\frac{n}{2^k} \leq 1 \Rightarrow k = \lceil \log_2(n) \rceil;$$

- ④ Quindi BinarySearch è $O(\log(n))$.

Vediamo un approccio più strutturato per analizzare le funzioni ricorsive

Funzione ricorsiva

Ciascuna istanza di una funzione ricorsiva applicata ad un problema è una istanza base $n = 1$ (di cui quindi possiamo calcolare il costo separatamente), oppure suddivide il problema corrente di dimensione n in un certo numero di sottoproblemi di dimensione più piccola, le cui soluzioni saranno poi combinate per costruire la soluzione complessiva.

Nella analisi delle funzioni ricorsive si usano spesso delle relazioni di ricorrenza, ossia delle equazioni del tipo

$$T(n) = G(T(n - n_1), T(n - n_2), T(n - n_3), \dots, T(n - n_k))$$

con $n_j < n$. Ad esempio, una ricorrenza lineare a termini costanti sarà del tipo

$$T(n) = \left(\sum_{i=1}^k a_i T(n - i) \right) + cn^\beta, \quad n > k$$

Teorema

Ricorrenze lineari

Siano a_1, a_2, \dots, a_k costanti intere non negative, $c > 0$ e $\beta \geq 0$ costanti reali, e sia $T(n)$ definita dalla ricorrenza

$$T(n) = \begin{cases} c_n & \text{per } 1 \leq n \leq k, \\ \sum_{1 \leq i \leq k} a_i T(n-i) + cn^\beta & \text{per } n > k. \end{cases}$$

Definendo

$$a = \sum_{1 \leq i \leq k} a_i,$$

allora

$$T(n) = \begin{cases} O(n^{\beta+1}) & \text{se } a = 1, \\ O(a^n n^\beta) & \text{se } a \geq 2. \end{cases}$$

Dimostrazione

Si supponga che solo uno dei coefficienti a_i sia non nullo. Allora $a = a_j$; supponendo $n = m + p \cdot j$, con $1 \leq m \leq k$, e sostituendo

$$\begin{aligned} T(n) &= aT(n-j) + cn^\beta = a(aT(n-2j) + c(n-j)^\beta) + cn^\beta \\ &= a(a \dots a(a(T(m) + c(m+j)^\beta) + c(m+2j)^\beta) + \dots c(n-j)^\beta) + cn^\beta \\ &= a^p T(m) + \sum_{0 \leq i \leq p-1} a^i c(n-i \cdot j)^\beta \\ &\leq a^p T(m) + cn^\beta \sum_{0 \leq i \leq p-1} a^i. \end{aligned}$$

Dimostrazione.

Abbiamo allora due casi:

$a = 1$ In questo caso

$$\begin{aligned} T(n) &\leq T(m) + cn^\beta(1 + 1 + \cdots + 1) \\ &= T(m) + cn^\beta p \\ &= T(m) + cn^\beta(n - m)/j = O(n^{\beta+1}); \end{aligned}$$

$a \geq 2$ In questo caso

$$\begin{aligned} T(n) &= a^p T(m) + cn^\beta((a^p - 1)/(a - 1)) \\ &\leq a^p (T(m) + cn^\beta) \\ &= a^{(n-m)/j} (T(m) + cn^\beta) = O(a^n n^\beta). \end{aligned}$$

Dimostrazione.

Nel caso generale di più di un coefficiente non nullo, necessariamente $a \geq 2$ e allora

$$\begin{aligned} T(n) &= \sum_{1 \leq i \leq k} a_i T(n-i) + cn^\beta \\ &\leq \sum_{1 \leq i \leq k} a_i T(n-1) + cn^\beta \\ &= aT(n-1) + cn^\beta \end{aligned}$$

che si riconduce al caso precedente. □

Ricorrenze lineari con partizione bilanciata:

Si divide il problema principale di dimensione n in un insieme di sottoproblemi di dimensione n/b , le cui soluzioni vengono poi ricombinate per costruire la soluzione complessiva

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ aT(\frac{n}{b}) + n^\beta & \text{se } n > 1 \end{cases}$$

Il termine n^β misura il costo di suddividere il problema in sottoproblemi e di ricombinare le loro soluzioni.

Si noti che se si avesse un costo $c \cdot n^\beta$ si può facilmente usare una funzione riscalata $U = T/c$.

Andiamo ora ad espandere la relazione di ricorrenza supponendo $n = b^k$:

$$\begin{aligned}
 T(n) &= aT\left(\frac{n}{b}\right) + n^\beta \\
 &= a\left(aT\left(\frac{n}{b^2}\right) + \left(\frac{n}{b}\right)^\beta\right) + n^\beta = a^2T\left(\frac{n}{b^2}\right) + a\left(\frac{n}{b}\right)^\beta + n^\beta \\
 &= a^3T\left(\frac{n}{b^3}\right) + a^2\left(\frac{n}{b^2}\right)^\beta + a\left(\frac{n}{b}\right)^\beta + n^\beta = \sum_{j=0}^k a^j \left(\frac{n}{b^j}\right)^\beta
 \end{aligned}$$

ovvero

$$T(n) = n^\beta \sum_{j=0}^k \left(\frac{a}{b^\beta}\right)^j$$

avendo usato:

$$T\left(\frac{n}{b^k}\right) = T(1) = 1 = 1^\beta = \left(\frac{n}{b^k}\right)^\beta$$

Notiamo ora che

$$a^k = a^{\log(n)/\log(b)} = 2^{\log(a)\log(n)/\log(b)} = n^{\log(a)/\log(b)} = n^\alpha = n^{\log_b(a)};$$

inoltre

$$a = b^\alpha, \quad \alpha = \log_b(a),$$

e ponendo

$$q = \frac{a}{b^\beta} = \frac{b^\alpha}{b^\beta} = b^{\alpha-\beta}$$

si può scrivere

$$T(n) = n^\alpha + b^{k\beta}(q^{k-1} + q^{k-2} + \dots + q + 1)$$

Caso 1:

$$\alpha > \beta$$

In questo caso $q > 1$

$$\sum_{j=0}^k \left(\frac{a}{b^\beta}\right)^j = \frac{\left(\frac{a}{b^\beta}\right)^{k+1} - 1}{\frac{a}{b^\beta} - 1};$$

che può essere approssimato asintoticamente con

$$\sum_{j=0}^k \left(\frac{a}{b^\beta}\right)^j \leq c \cdot \left(\frac{a}{b^\beta}\right)^{k+1} = \frac{ac}{b^\beta} \frac{a^k}{n^\beta}$$

quindi

$$T(n) \leq n^\beta \cdot \frac{ac}{b^\beta} \frac{a^k}{n^\beta} = \gamma \cdot a^k$$

da cui

$$T(n) = O(a^k) = O(a^{\log_b n}) = O(n^{\log_b a}) = O(n^\alpha)$$

Il passaggio precedente si dimostra facilmente osservando che

$$\log(x) = \log(y) \Rightarrow x = y,$$

Possiamo quindi dimostrare

$$a^{\log_b n} = n^{\log_b a}$$

prendendone il logaritmo

$$\log_b(a^{\log_b n}) = \log_b(n^{\log_b a})$$

che evidentemente vale

$$\log_b(n) \cdot \log_b(a) = \log_b(a) \cdot \log_b(n),$$

ossia la tesi.

Caso 2:

$$\alpha < \beta;$$

in questo caso

$$\sum_{j=0}^k \left(\frac{b^{\alpha}}{b^{\beta}} \right)^j \leq \sum_{j=0}^{\infty} \left(b^{\alpha-\beta} \right)^j = \xi < \infty,$$

da cui

$$T(n) \leq n^{\beta} \cdot \xi = O(n^{\beta}).$$

Caso 3:

$$\alpha = \beta;$$

in questo caso abbiamo

$$\sum_{j=0}^k 1 = k + 1 = O(k)$$

e quindi

$$T(n) = O(n^{\beta} k) = O(n^{\beta} \log_b n).$$

Abbiamo visto che $C \leftarrow AB$ nella versione “normale” richiede $2n^3$ operazioni

Abbiamo visto che $C \leftarrow AB$ nella versione “normale” richiede $2n^3$ operazioni
Consideriamo il problema di moltiplicare due matrici di dimensione 2×2 :

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Il modo standard di calcolare il prodotto richiede:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

per un costo di 8 moltiplicazioni e 4 addizioni.

Nell 1968 Strassen pubblicò una formula (poi migliorata da Winograd)

$$u = (A_{21} - A_{11})(B_{21} - B_{22})$$

$$v = (A_{21} + A_{22})(B_{21} - B_{11})$$

$$w = A_{11}B_{11} + (A_{21} + A_{22} - A_{11})(B_{11} + B_{22} - B_{12})$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = w + v + (A_{11} + A_{12} - A_{21} - A_{22})B_{22}$$

$$C_{21} = w + v + A_{22}(B_{21} + B_{12} - B_{11} - B_{22})$$

$$C_{22} = u + v + w$$

che richiede 7 moltiplicazioni e 15 addizioni; la commutatività non viene usata, quindi i termini possono essere sottomatrici di dimensione $\frac{n}{2} \times \frac{n}{2}$

Per cui la moltiplicazione di matrici costa

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T(\frac{n}{2}) + 15(\frac{n}{2})^2 & n > 1 \end{cases}$$

ma siccome

$$7 > 4 = 2^2$$

ne segue

$$T(n) = O(n^{\log_2(7)}) = O(n^{2.8074}).$$

Si è poi aperta la ricerca all'algoritmo asintoticamente migliore; l'attuale record è $O(n^{2.376})$, ma è efficiente solo per dimensioni assolutamente fuori portata, e quindi in pratica poco interessante (mentre quello di Strassen è valido per dimensioni ragionevoli, ma soffre di errori di arrotondamento).

La *Trasformata di Fourier Discreta* o DFT opera su *sequenze* (campionamenti di una funzione)

$$f_j, \quad j = 0, \dots, N - 1,$$

e si definisce come:

$$F(N)_k = \sum_{j=0}^{N-1} e^{\frac{2\pi i j k}{N}} f_j, \quad k = 0, \dots, N - 1.$$

È evidente che si tratta di un prodotto matrice-vettore (complesso) dal costo $O(N^2)$ operazioni.

La DFT si usa sistematicamente in tutte le applicazioni di elaborazione dei segnali e delle immagini, ma così come l'abbiamo appena descritta ha un costo eccessivo

Riscriviamo ora l'espressione della DFT:

$$\begin{aligned}
 F(N)_k &= \sum_{j=0}^{N-1} e^{\frac{2\pi i jk}{N}} f_j = \left(\sum_{j=0}^{N/2-1} e^{\frac{2\pi i k(2j)}{N}} f_{2j} \right) + \left(\sum_{j=0}^{N/2-1} e^{\frac{2\pi i k(2j+1)}{N}} f_{2j+1} \right), \\
 &= \left(\sum_{j=0}^{N/2-1} e^{\frac{2\pi i kj}{N/2}} f_{2j} \right) + e^{\frac{2\pi i k}{N}} \left(\sum_{j=0}^{N/2-1} e^{\frac{2\pi i kj}{N/2}} f_{2j+1} \right) = F(N/2)_k^p + e^{\frac{2\pi i k}{N}} \cdot F(N/2)_k^d.
 \end{aligned}$$

L'insieme dei valori di F_k si ottiene combinando i valori calcolati prima sui punti pari e poi su quelli dispari. Siamo nelle condizioni del teorema sulle partizioni bilanciate per cui il costo sarà

$$O(N \log(n));$$

l'algoritmo che ne risulta si chiama *FFT* ovvero Trasformata di Fourier Veloce (Fast Fourier Transform). L'articolo di Cooley e Tukey del 1965 in cui viene presentata è l'articolo di matematica applicata più citato di tutti i tempi.

Affrontiamo ora un altro aspetto e chiediamoci:

Quale è l'effettivo costo della esecuzione di una *sequenza* di m operazioni (su una certa struttura dati) ?

La domanda è pertinente in quanto potremmo avere effettuato una valutazione della complessità *nel caso peggiore* di ciascuna singola operazione, ma l'esecuzione di m operazioni consecutive ci porta naturalmente a cercare di valutare una forma di *complessità media*, che può essere abbastanza diversa.

Consideriamo un contatore implementato con un insieme di bit, e consideriamo l'operatore di aggiornamento $++$

Algoritmo 1: Add

Input: Array contenente i bit $A[i]$, $i = 0, \dots, k - 1$, la dimensione k ;

$i \leftarrow 0$;

while $i < k$ **and** $A[i] == 1$ **do**

$A[i] \leftarrow 0$;

$i \leftarrow i + 1$;

if $i < k$ **then**

$A[i] \leftarrow 1$;

L'algoritmo presentato è banalmente $O(k)$.

Se ora consideriamo m esecuzioni consecutive, vediamo che la complessità $O(mk)$ è eccessivamente pessimistica; infatti, se osserviamo la sequenza numerica

| n | A | Costo | n | A | Costo |
|-----|------|-------|-----|------|-------|
| 0 | 0000 | - | 4 | 0100 | 3 |
| 1 | 0001 | 1 | 5 | 0101 | 1 |
| 2 | 0010 | 2 | 6 | 0110 | 2 |
| 3 | 0011 | 1 | 7 | 0111 | 1 |
| | | | 8 | 1000 | 4 |

vediamo che il bit i -esimo viene alterato ogni 2^i incrementi, quindi il costo è

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

ossia il costo medio del singolo aggiornamento è $O(1)$.

Per effettuare l'analisi si può anche usare il

Metodo dell'accantonamento

Si assegna a ciascuna operazione un costo, più alto del costo minimo, e si usa il credito così accumulato per compensare le operazioni costose

Nell'esempio del contatore, si assegna un costo di 2 ad ogni operazione, 1 per cambiare un bit da 0 a 1, e 1 per riportarlo da 1 a zero. In questo modo il credito è pari al numero di bit pari ad 1 e compensa i ritorni a 0, e riotteniamo un costo $2n$.

Consideriamo l'operazione di inserimento di dati in un vettore

Algoritmo 2: Insert

Input: Array $A[i]$, $i = 0, \dots, n$, ultima posizione occupata dimensione k , elemento v ;

if $k < n - 1$ **then**

$k \leftarrow k + 1$;

$A[i] \leftarrow v$;

Occasionalmente si dovrà effettuare una *riallocazione* del vettore

- viene creata un'area temporanea più grande,
- si copiano i dati esistenti,
- si sostituisce l'area nuova a quella vecchia¹

Cosa succede se ad ogni riallocazione passiamo da n a $n + d$?

¹le operazioni di creazione dell'area temporanea e di sostituzione costano $O(1)$

Supponiamo dover gestire $N = d \cdot P$ inserimenti a partire da dimensione iniziale 0; il loro costo totale sarà

$$\sum_{k=1}^P d \cdot (k-1) = d \sum_{k=0}^{P-1} k = d \cdot \frac{P \cdot (P-1)}{2} \approx d \cdot \frac{P^2}{2} = \frac{N^2}{2d} = O(N^2),$$

ovvero

il costo *medio* del singolo inserimento sarà $O(N)$,

anche se la maggior parte degli inserimenti costa $O(1)$

Se invece si rialloca a dimensione $n + n = 2 \cdot n$, allora il numero di eventi di riallocazione P dovrà essere tale che

$$2^P \geq N \Rightarrow P \approx \log(N);$$

ogni evento di riallocazione avrà un costo lineare, ed il costo totale sarà

$$O(N \log(N)),$$

ossia

Il costo *medio* di ciascun inserimento sarà

$$O(\log(N)) \ll N.$$