Informatica per le Biotecnologie
Algoritmica Lezione 4

1. Il problema dell'ordinamento di dati

sorting in gergo informatico, consistente nel disporre in ordine "crescente" n elementi tra cui è definita una relazione di ordinamento totale indicata con $\leq$.

Gli elementi si memorizzano in un vettore A[0 . . . n-1]

L'algoritmo iterativo INSERTION-SORT si basa sull'ipotesi che i primi i elementi contenuti tra le posizioni **0** e $i-1$ siano ordinati: si inserisce l'elemento $A[i]$ tra questi e si ripete l'operazione a partire da $i+1$ finché tutti gli n elementi sono stati inseriti.

↓

3 8 5 2 13 10 . . .

3 8 5 2 13 10 . . .

3 8 5 2 13 10 . . .

3 5 8 2 13 10 . . .

3 8 5 2 13 10 . . .

$\downarrow$

3 5 8 2 13 10 . . .

3 8 5 2 13 10 . . .

3 5 8 2 13 10 . . .

↓

3 5 2 8 13 10 . . .

3 8 5 2 13 10 . . .

↓

3 5 8 2 13 10 . . .

↓

3 2 5 8 13 10 . . .

3 8 5 2 13 10 . . .

3 5 8 2 13 10 . . .

↓

2 3 5 8 13 10 . . .

3 8 5 2 13 10 . . .

3 5 8 2 13 10 . . .

↓

2 3 5 8 13 10 . . .

3 8 5 2 13 10 . . .

3 5 8 2 13 10 . . .

2 3 5 8 13 10 . . .

↓

2 3 5 8 10 13 . . .

INSERTION-SORT(A)

for $(i = 1;\ i \leq n-1;\ i++)$

$\{\ k = A[i]$ **;**

$j = i - 1$ **;**

while $((j \geq 0)\ \&\&\ (A[j] > k))$

$\{A[j+1] = A[j];\ j = j-1\}$**;**

$A[j+1] = k$ **;** $\}$

Prima iterazione del **for**

```
j   i                                  k = 8
3   8   5   2   13   10   . . .
```

INSERTION-SORT(A)

for $(i = 1;\ i \leq n-1;\ i++)$

$\{\ k = A[i]$;

$j = i - 1$;

while $((j \geq 0)\ \&\&\ (A[j] > k))$

$\{A[j+1] = A[j];\ j = j - 1\}$;

$A[j+1] = k;\ \}$

Primi tre assegnamenti nella seconda iterazione del **for**

```
    j   i                        k = 5
3   8   5   2   13   10   . . .
```

INSERTION-SORT(A)

for $(i = 1;\ i \leq n-1;\ i++)$

$\{\ k = A[i]$;

$j = i - 1$;

while $((j \geq 0)\ \&\&\ (A[j] > k))$

$\{A[j+1] = A[j];\ j = j - 1\}$;

$A[j+1] = k$; }

Effetto della seconda iterazione del **for**

```
j          i                                        k = 5
3    5     8     2     13     10     .   .   .
```

INSERTION-SORT(A)

for $(i = 1;\ i \leq n-1;\ i++)$

$\{\ k = A[i]$;

$j = i - 1$;

while $((j \geq 0)\ \&\&\ (A[j] > k))$

$\{A[j+1] = A[j];\ j = j - 1\}$;

$A[j+1] = k$; }

Primi tre assegnamenti nella terza iterazione del **for**

```
        j   i                         k = 2
3   5   8   2   13   10   .  .  .
```

Caso ottimo: i dati in ingresso sono ordinati in modo crescente

2 3 5 8 10 13 20 23

Il ciclo while non è mai eseguito :

il tempo $T(n)$ è di ordine $\Theta(n)$ (numero di iterazioni di **i**)

Caso pessimo: **i dati sono ordinati in modo decrescente**

23 20 13 10 8 5 3 2

Il ciclo while richiede 1 + 2 + 3 + ….. + n-1 operazioni nei successivi cicli di **i**: il tempo T(n) è di ordine $\Theta(n^2)$

Dunque INSERTION-SORT richiede tempo $O(n^2)$ considerando il caso pessimo

Un algoritmo ricorsivo di ordinamento

MERGE-SORT: ordinamento per fusione

Oltra al vettore *A* che contiene i dati che vengono riordinati man mano, è necessario un vettore di appoggio *B[0…n-1]*

MERGE-SORT(A, s, d) $\quad$ $s \leq d$

// Ordinamento di A ***tra le posizioni s,d //***

```
if (s < d) {
   m = ⌊(s + d)/2⌋;
   MERGE-SORT(A, s, m);
   MERGE-SORT(A, m + 1, d);
   FUSIONE(A, s, m, d);  }
```

Chiamata dall'esterno: MERGE-SORT(*A,0,n-1*)

Le due chiamate interne di MERGE-SORT ordinano le due metà del vettore *A*. La procedura FUSIONE le fonde in un unico vettore B e poi le trasferisce in *A*.

MERGE-SORT(A, s, d) $s \leq d$

// Ordinamento di A tra le posizioni s,d //

```
if (s < d) {
    m = ⌊(s + d)/2⌋;
    MERGE-SORT(A, s, m);
    MERGE-SORT(A, m + 1, d);
    FUSIONE(A, s, m, d);  }
```

Tutte le righe del programma, fino alle due chiamate ricorsive, hanno lo scopo di stabilire l'ordine in cui si fanno I confronti. Questi sono tutti eseguiti dalla procadura FUSIONE.

Come funziona FUSIONE ?

```
      ..  s              m                 d ..

A     ..  2   5   13   20   3   8   10   23 ..

B         2
```

	..	s			m				d	..
A	..	2	5	13	20	3	8	10	23	..
B		2	3							

	..	s			m				d	..
A	..	2	5	13	20	3	8	10	23	..
B		2	3							

	..	s			m				d	..
A	..	2	5	13	20	3	8	10	23	..
B		2	3	5						

	..	s			m				d	..
A	..	2	5	13	20	3	8	10	23	..
B		2	3	5	. . . .					
B		2	3	5	8	10	13	20		

```
      ..  s              m                 d  ..

A     ..  2    5    13   20    3    8    10   23  ..

          ↑    ↑    ↑    ↑     ↑    ↑    ↑

B         2    3    5    8     10   13   20
```

Ho ordinato *A[s:d]*.

Il sottovettore *A[s:m]* si è esaurito prima di *A[m+1:d]*: ma se avviene il contrario?

```
      .. s         m              d ..

A     .. 2   5   24   28   3   8   10   23 ..

B        2   3   5    8   10   23
```

Faccio questi trasferimenti,

ma prima devo salvare 24 28

	..	s			m				d	..
A	..	2	5	24	28	3	8	24	28	..
B		2	3	5	8	10	23			

```
FUSIONE(A, s, m, d)
  i = s; j = m + 1; k = 0;
  while ((i ≤ m) && (j ≤ d)) {
     if (A[i] ≤ A[j]) {B[k] = A[i]; i = i + 1}
     else {B[k] = A[j]; j = j + 1};
     k = k + 1 };
// se il sottovettore A[m + 1 : d] è esaurito //
  if (i ≤ m)
     { j = d − (m − i);
     for ( ; i ≤ m; i++; j++) A[j] = A[i] };
// si portano in A[s...] i k elementi di B //
  i = s; j = 0;
  for ( ; j ≤ k − 1; i++; j++) A[i] = B[j];
```

Ordine dei confronti di FUSIONE entro MERGE-SORT

```
2  20  13  5  8  23  10  3  6  9  21  12  7  16  19  4

2  20

         5  13

2   5   13  20

                 8  23

                         3  10

                 3  8  10  23

2   3   5   8  10  13  20  23

                                 6  9

                                        . . . . . . .
```

Numero massimo di confronti di FUSIONE, n = 16

2 | 20 13 | 5 8 | 23 10 | 3 6 | 9 21 | 12 7 | 16 19 | 4

n-8

2 20 | 5 13 8 23 | 3 10 6 9 | 12 21 7 16 | 4 19

n-4

2 5 13 20 | 3 8 10 23 6 9 12 21 | 4 7 16 19

n-2

2 3 5 8 10 13 20 23 | 4 6 7 9 12 16 19 21

n-1

2 3 4 5 6 7 8 9 10 12 13 16 19 20 21 23

Per $n = 16 = 2^4$, FUSIONE ha eseguito al massimo

$4n–1-2-4-8 = n \log_2 n – (n-1) =$ $\Theta(n \log n)$ confronti

Il tempo richiesto è maggiore se l'esame del sottovettore *A[m+1:d]* termina prima di quello di *A[s:m]*, ma in entrambi i casi tale tempo è di ordine $\Theta(d\text{-}s)$, quindi è di ordine $\Theta(n)$ quando lavora sull'intero vettore.

MERGE-SORT(A, s, d) $s \leq d$

// Ordinamento di A tra le posizioni s,d //

if $(s < d)$ {

$m = \lfloor (s + d)/2 \rfloor$;

MERGE-SORT(A, s, m);

MERGE-SORT$(A, m + 1, d)$;

FUSIONE(A, s, m, d); }

Tempo $T(n)$ richiesto da MERGE-SORT

- **per** $n = 1$ **(cioè per** $s = d$**) abbiamo** $T(1) = b$ costante
- **per** $n > 1$ **(cioè per** $s < d$**) la procedura richiede tempo costante** c_1 **fino al calcolo di** m**; chiama due volte se stessa su** $n/2$ **dati; esegue FUSIONE in tempo** $c_2 n$**. Dunque:**
  $T(n) = 2T(n/2) + c_1 + c_2 n$. c_1 e c_2 costanti

Abbiamo *l'equazione di ricorrenza:*

$$T(1) = b; \quad T(n) \le 2T(n/2) + cn, \quad \textbf{per } n > 1;$$

$c = c_1 + c_2$ costante

Sviluppando l'equazione si trova

$T(n)$ di ordine $\Theta(n \log n)$ (Vedi Dispensa)

ponendo $n = 2^t$ e sviluppando:

(Vedi Dispensa)

$$T(n) \leq 2T(n/2) + cn \leq 2(2T(n/4) + cn/2) + cn$$
$$= 4T(n/4) + 2cn$$
$$\leq 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn$$
$$= 2^3T(n/2^3) + 3cn$$
$$\leq .... = 2^tT(n/2^t) + tcn = nT(1) + cn\log_2 n$$
$$= cn\log_2 n + bn,$$

ovvero $\boldsymbol{T(n)}$ è di ordine O($\boldsymbol{n}$ log $\boldsymbol{n}$).

Si noti l'ordine O e non Θ perché si è maggiorato il secondo membro con $\leq$. Ma si può dimostrare che la complessità è proprio Θ(n log n).

Un limite inferiore alla complessità del sorting

Calcoliamo il numero minimo di confronti tra elementi dell'insieme da ordinare, impiegando un albero di decisione

Ogni permutazione della sequenza iniziale dei dati è una possibile soluzione del problema, quindi il numero *S(n)* di soluzioni è *n fattoriale*

Applicando la Formula di Stirling: $S(n) = n! \approx \sqrt{2\pi n}(n/e)^n$

Allocando queste soluzioni nelle foglie di un albero ternario con percorsi di computazione di lunghezza massima t **abbiamo** $3^t \geq \sqrt{2\pi n}(n/e)^n$

Applicando il logaritmo a base 3 ai due membri della diseguaglianza otteniamo:

$$t \geq (1/2)\log_3(2\pi n) + n(\log_3 n - \log_3 e) > n(\log_3 n - \log_3 e)$$

dunque $min(n) = t$ **è di ordine** $\Omega(n \log n)$.

Poiché un limite inferiore alla complessità di tempo del problema di sorting è $\Omega(n \log n)$ l'algoritmo MERGESORT che richiede tempo $\Theta(n \log n)$ ed è ottimo mentre INSERTION-SORT non lo è.

Ricordiamo tuttavia che MERGE-SORT richiede uno spazio doppio di memoria a causa dell'impiego (necessario) del vettore B mentre INSERTION-SORT è eseguito solo con scambi all'interno del vettore A.

Vi sono altri algoritmi di sorting che richiedono tempo $\Theta(n \log n)$ e non richiedono un vettore ausiliario

2. Algoritmi polinomiali e esponenziali

MERGE-SORT e INSERTION-SORT richiedono tempi $\Theta(n \log n)$ e $\Theta(n^2)$ nel caso pessimo. Indichiamoli con A_1 e A_2.

Confrontiamoli con l'algoritmo FOOLISH-SORT, indicato con A_3, che genera tutte le permutazioni dei dati d'ingresso, controllandole una a una fino a individuare quella ordinata.

FOOLISH-SORT richiede almeno tempo $n!$ per generare tutte le permutazioni (nel caso pessimo la permutazione ordinata è l'ultima generata), e tempo n per verificare l'ordinamento in ciascuna di esse. Complessivamente tempo $n x n!$

Ricordando che: $n! \approx \sqrt{2\pi n}(n/e)^n$

il tempo complessivo è $\Theta(n^{3/2}(n/e)^n)$.

Un algoritmo è detto polinomiale se la sua complessità in tempo è limitata superiormente da un polinomio in n, ovvero è di ordine $O(n^k)$ per $k>0$ costante.

Un algoritmo è detto esponenziale se la sua complessità in tempo è limitata inferiormente da una funzione che presenta n all'esponente, ovvero è di ordine $\Omega(k^f)$ per $k>0$ costante e f crescente con n.

Tempo di A1 (MERGE-SORT) Θ *(n log n)*

Tempo di A2 (INSERTION-SORT) $\Theta(n^2)$

Tempo di A3 (FOOLISH-SORT) $\Theta(n^{3/2}(n/e)^n)$

A_1 e A_2 sono algoritmi *polinomiali* (in particolare il tempo di A_1 è limitato superiormente da $n^{1+\epsilon}$ per qualsiasi $\epsilon > 0$ poiché $n^\epsilon > \log n$).

A_3 è un algoritmo *esponenziale* perché nell'espressione del tempo n figura (anche) all'esponente.

Poniamo che i tre tempi siano espressi in secondi se gli algoritmi sono codificati in un partiolare linguaggio su un particolare calcolatore

$$T_1 = \frac{1}{100} n \log_2 n, \quad T_2 = \frac{1}{100} n^2, \quad T_3 = \frac{1}{100} n^{3/2} (n/e)^n$$

n	4	8	16	32	64	128
T_1	0,08	0,24	0,64	1,60	3,84	8,96
T_2	0,16	0,64	2,56	10,24	40,96	163,84
T_3	0,38	$>10^3$	$>10^{12}$	$>10^{37}$	$>10^{91}$	$>10^{215}$

Al raddoppiare di *n* il tempo T_1 è poco più che doppio

$$T_1(2n) = \frac{1}{100} 2n \log_2(2n) = \frac{2}{100} n(\log_2 n + 1) = \frac{2}{100} n \log_2 n + \frac{2}{100} n$$

$$T_1(2n) = 2T_1(n) + \frac{2}{100} n$$

Al raddoppiare di *n* il tempo T_2 è quattro volte tanto

$$T_2(2n) = \frac{1}{100}(2n)^2 = \frac{4}{100} n^2 = 4T_2(n).$$

Con la crescita esponenziale, per semplificare i calcoli consideriamo un nuovo algoritmo A_4 di complessità "solo" 2^n anziché $(n/e)^n$

$$T_4(n) = \frac{1}{100} 2^n$$

dunque $$T_4(2n) = \frac{1}{100} 2^{2n} = \frac{1}{100} (2^n)^2$$

Per ordinare 128 elementi impiegando A_4 occorrono più di 10^{39} secondi (10^{29} millenni).

Una nuova considerazione:
Per un algoritmo arbitrario *A* eseguito su un caalcolatore *C*, impieghiamo un calcolatore *C'* di velocità k volte maggiore di quella di C e vedimo come cresce la dimensione dei dati trattabili, n e n', a pari tempo τ di elaborazione.

In prima approssimazione impiegare *C'* per τ secondi equivale a impiegare C per $k\tau$ secondi:

Se A richiede tempo lineare, $T(n) = c\,n$, abbiamo

$$c\,n = \tau\ ,\ c\,n' = k\,\tau \quad \text{quindi} \quad n' = k\,n$$

cioè la dimensionre dei dati è *k volte maggiore.*

Calcoliamo cosa accade per gli algoritmi
di ordinamento A_1, A_2, A_4

Per A_1 abbiamo: $$\frac{1}{100} n \log_2 n = \tau, \quad \frac{1}{100} n' \log_2 n' = k\tau,$$

da cui $n' \log_2 n' = k\, n \log_2 n$: poiché $\log_2 n'$ è poco maggiore di $\log_2 n$, i valori di n' sono quasi k volte i valori di n.

Poniamo $k = 10$. Per $\tau = 8{,}96$ secondi, con A_1 si possono ordinare $n = 128$ elementi su C (vedi tabella precedente) e $n' = 911$ su C', con incremento poco meno di $k = 10$

Per A_2 abbiamo:

$$\frac{1}{100}n^2 = \tau, \quad \frac{1}{100}n'^2 = k\tau,$$

da cui $n'^2 = kn^2$ ovvero $n' = \sqrt{k}\, n$

per $k = 10$ abbiamo $n' = 3{,}16\, n$

Con un algoritmo di complessità n^c, con $c > 2$, avremmo $n' = k^{1/c}\, n$, con un vantaggio tanto minore quanto più alto è il valore di c, cioè quanto meno efficiente è l'algoritmo.

Per un algoritmo esponenziale per esempio A_4 (crescita 2^n):

$$\frac{1}{100}2^n = \tau, \quad \frac{1}{100}2^{n'} = k\tau,$$

da cui $2^{n'} = k2^n$, quindi $n' = n + \log_2 k$

Per esempio moltiplicando per $k = 1024$ la velocità del calcolatore, a pari tempo τ, con A_4 si potrebbero ordinare

$$n' = n + \log_2 1024 = n + 10 \quad \text{dati.}$$

Gli algoritmi esponenziali sono inutilizzabili indipendentemente dalla velocità dei calcolatori impiegati

3. Complessità computazionale

Esistono problemi che richiedono necessariamente algoritmi esponenziali per risolverli: sono o banali o complicatissimi, e in genere non sono importanti in pratica.

Ma si incontrano molti problemi importanti che richiedono tempo esponenziale perché non siamo capaci di risolverli meglio.

Partiamo dai *problemi decisionali* che chiedono di stabilire se esiste una soluzione con una determinata proprietà: la risposta (binaria) ha forma di affermazione o negazione.

Definizione. $\mathcal{P}$ è l'insieme dei problemi decisionali risolubili in tempo polinomiale.

Problema $P_{\Sigma 2}$. **Dati un insieme S di n interi positivi e un intero k, stabilire se esistono due elementi di S la cui somma è k.**

	0	1	2	3	4	5	6	7
A	**22**	**31**	**7**	**47**	**30**	**18**	**15**	**40**

Per $k = 49$ **risposta affermativa:** $A[1] + A[5] = 49$.

Per $k = 42$ **risposta negativa.**

Possiamo provare con tutte le coppie di elementi

```
SIGMA2(A, k)
  for (i = 0; i ≤ n − 2; i++)
    for (j = i + 1; j ≤ n − 1; j++)
      if (A[i] + A[j] == k)
          return una coppia esiste;
  return nessuna coppia esiste;
```

Il tempo richiesto da SIGMA2 è di ordine *$O(n^2)$*, ma possiamo fare meglio ordinando l'insieme

	0	1	2	3	4	5	6	7
A	7	15	18	22	30	31	40	47
	i							j
		i						
							j	
						j		
			i			j		

A[2] + A[5] = 49

Operando in questo modo il tempo richiesto è θ(n log n) per ordinare il vettore, più O(n) per lo scorrimento di *i* e *j* .

In totale θ(n log n)

Limite inferiore al tempo di calcolo di $P_{\Sigma 2}$

Albero di decisione: Il numero di soluzioni è $S(n) = n(n-1)/2+1$ (1 rappresenta il caso in cui nessuna coppia abbia somma k). L'albero di decisione ha profondità $\geq \log_2 S(n) = \log_2(n^2/2 - n/2+1)$, quindi di ordine $\Omega(\log n)$.

Molto più forte è il limite $\Omega(n)$ che deriva dalla osservazione che tutti gli elementi di A devono essere esaminati.almeno una volta.

Rimane un gap tra i due limiti $\Omega(n)$ e $O(n \log n)$

Generalizzazione di $P_{\Sigma 2}$ a P_{Σ}:

dato un insieme S di n interi positivi e un intero k, stabilire se esiste un sottoinsieme di S di dimensioni arbitrarie la cui somma degli elementi è k.

Nell'esempio precedente la risposta affermativa per $k = 92$: $A[2] + A[4] + A[6] + A[7] = 92$.

Per P_Σ non si conosce un algoritmo polinomiale.

Si può affrontare associando ad A un *vettore di appartenenza* V contenente interi 0, 1 in cui $A[i]$ fa parte di una scelta di elementi se e solo se $V[i] = 1$.

	0	1	2	3	4	5	6	7
A	22	31	7	47	30	18	15	40
V	0	0	1	0	1	0	1	1

Vettore V per il sottoinsieme { 7, 30, 15, 40 }

Generiamo tutti i possibili vettori V di n elementi e verifichiamo se per uno di essi è verificata l'uguaglianza:

$$\sum_{i=0}^{n-1} A[i] \cdot V[i] = k$$

Esiste per esempio un algoritmo ricorsivo per generare tutti i vettori *V* .

Vedi dispense

Tale algoritmo richiede tempo esponenziale perché questi vettori sono in numero 2^n.

Per moltissimi di problemi esponenziali un *certificato* K permette di verificare, attraverso un algoritmo polinomiale $VER(K, D)$, se i dati D dell'istanza considerata hanno la proprietà desiderata.

Nel problema P_Σ il vettore V relativo a una soluzione costituisce un certificato (il calcolo della relazione su k richiede tempo $\Theta(n)$).

Definizione. $\mathcal{NP}$ **è l'insieme dei problemi decisionali verificabili in tempo polinomiale.**

P_Σ **appartiene dunque a** $\mathcal{NP}$**.**

Notare che:

L'*esistenza* di un certificato K non ha alcun effetto sull'algoritmo di soluzione perché K non è noto a priori.

Per risolvere un problema in $\mathcal{NP}$ è necessario ricorrere a un metodo enumerativo che esegua un numero esponenziale di prove.

Per ogni problema in $\mathcal{P}$ si può verificare l'esistenza di una soluzione in tempo polinomiale ignorando il certifiato e risolvendo direttamente il problema. Da ciò segue che:fondamentale:

$$\mathcal{P} \subseteq \mathcal{NP}.$$

Non è mai stato dimostrato se vale il contenimento in senso stretto tra le due classi, cioè se sia $\mathcal{P} \neq \mathcal{NP}$. Tale quesito è tra i più importanti problemi irrisolti nella matematica.

Esempi di problemi in $\mathcal{NP}$

1. Biologia molecolare: dato un insieme arbitrario di sequenze di DNA stabilire se esiste una sottosequenza comune di lunghezza assegnata.
2. Manifattura: stabilire se un insieme di pezzi di lamiera possono essere ottenuti tagliandoli da un foglio assegnato disponendoli opportunamente su di esso.
3. Scheduling: dato un insieme di procedure da eseguire su un insieme di macchine, stabilire se tali procedure possono essere distribuite tra le macchine in modo che tutte queste terminino di operare entro un tempo prefissato.

4. **Algebra: stabilire se un'equazione algebrica di secondo grado in due variabili x, y, a coefficienti interi, ammette una soluzione in cui x e y hanno valori interi (se l'equazione è di primo grado il problema è in $\mathcal{P}$).**

Questi problemi si sanno risolvere solo provando un numero esponenziale di possibili soluzioni. Si può però verificare la correttezza di una soluzione in tempo polinomiale.