Ordinamento in tempo lineare

Il limite inferiore $\Omega(n \log n)$ vale per tutti gli algoritmi di ordinamento *generali*, ossia per algoritmi che non fanno alcuna ipotesi sul tipo degli elementi della sequenza da ordinare.

Se facciamo opportune ipotesi restrittive sul tipo degli elementi possiamo trovare algoritmi più efficienti.

Naturalmente il limite inferiore banale $\Omega(n)$ vale comunque per tutti gli algoritmi di ordinamento.

Algoritmo Counting-Sort

Assume che gli elementi dell'array siano interi compresi tra 0 e k con k costante.

Per ordinare un array $\boldsymbol{A}$ Counting-Sort richiede un secondo array $\boldsymbol{B}$ in cui mette la sequenza ordinata e un array ausiliario $\boldsymbol{C}[0..k]$.

	1	2	3	4	5	6	7	8
A	1	4	2	0	1	2	0	2

	0	1	2	3	4
C	2	2	3	0	1

	0	1	2	3	4
C	0	2	4	7	7

	1	2	3	4	5	6	7	8
B	0	0	1	1	2	2	2	4

```
Counting-Sort(A,B,k)  // A contiene a_1,...,a_n
  for i = 0 to k
    C[i] = 0
  for j = 1 to A.length
    x = A[j], C[x] = C[x] + 1
  // C[x] è il numero di elementi a_j = x
  for i = 1 to k
    C[i] = C[i] + C[i-1]
  // C[x] è il numero di elementi a_j ≤ x
  for j = A.length downto 1
    // i = C[x] è la posizione in B dove
    // mettere il prossimo a_j = x
    x = A[j], i = C[x], B[i] = x
    C[x] = C[x] - 1
```

Counting-Sort(A,B,k) // **Complessità**

```
for i = 0 to k                                  //
    C[i] = 0                                    //
for j = 1 to A.length                           //
    x = A[j], C[x] = C[x] + 1                   //
for i = 1 to k                                  //
    C[i] = C[i] + C[i-1]                        //
for j = A.length downto 1                       //
    x = A[j], i = C[x], B[i] = A[j]             //
    C[x] = C[x] - 1                             //
```

$\Theta(k)$

$\Theta(n)$

$\Theta(k)$

$\Theta(n)$

Complessità: $T^{CS}(n,k) = \Theta(n+k)$

Se $k = O(n)$ **allora** $T^{CS}(n,k) = \Theta(n)$

Osservazione:

Nell'ultimo ciclo for dell'algoritmo gli elementi dell'array ***A*** vengono copiati nell'array ***B*** partendo dall'ultimo

Cosa succede se partiamo dal primo?

	1	2	3	4	5	6	7	8
A	1	4	2	0	1'	2'	0'	2''

	0	1	2	3	4
C	1	3	6	7	7

	1	2	3	4	5	6	7	8
B		0	1'	1			2	4

Succede che l'algoritmo è ancora corretto ma gli elementi uguali vengono ricopiati in ordine inverso.

Quando un algoritmo di ordinamento mantiene l'ordine iniziale tra due elementi uguali si dice che esso è *stabile*.

L'algoritmo Counting-Sort (con l'ultimo ciclo for decrescente) è stabile.

Algoritmo Radix-Sort

Assume che i valori degli elementi dell'array siano interi rappresentabili con al più d cifre in una certa base b.

Ad esempio interi di $d = 5$ cifre decimali ($b = 10$), interi di $d = 4$ byte (cifre in base $b = 256$) o stringhe di d caratteri ($b = 256$).

Per ordinare l'array si usa d volte un algoritmo di ordinamento stabile (ad esempio Counting-Sort) per ordinare l'array rispetto a ciascuna delle d cifre partendo dalla meno significativa.

	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1
$A[1]$	1	4	2	7	9	8	9	0	8	2	2	3	0	0	3	9	0	0	3	9
$A[2]$	0	2	4	1	0	2	4	1	7	5	2	5	3	1	6	2	0	2	4	1
$A[3]$	7	5	2	5	3	1	6	2	1	4	2	7	8	2	2	3	1	2	3	9
$A[4]$	3	1	6	2	8	2	2	3	1	2	3	9	1	2	3	9	1	4	2	7
$A[5]$	9	8	9	0	7	5	2	5	0	0	3	9	0	2	4	1	3	1	6	2
$A[6]$	1	2	3	9	1	4	2	7	0	2	4	1	1	4	2	7	7	5	2	5
$A[7]$	8	2	2	3	1	2	3	9	3	1	6	2	7	5	2	5	8	2	2	3
$A[8]$	0	0	3	9	0	0	3	9	9	8	9	0	9	8	9	0	9	8	9	0

Radix-Sort(A,d) // $A[i] = c_d \ldots c_2 c_1$

for $j = 1$ **to** d

// A è ordinato rispetto alle cifre $c_{j-1} \ldots c_1$

"usa un algoritmo stabile per ordinare A rispetto alla j-esima cifra"

// A è ordinato rispetto alle cifre $c_j \ldots c_1$

// A è ordinato

Radix-Sort(*A*,*d*) // **Complessità**

for $j = 1$ **to** d

"usa Counting-Sort per ordinare A **rispetto alla** j**-esima cifra"**

Complessità: $T^{RS}(n,d,b) = \Theta(d(n+b))$

dove b è la base della numerazione e d è il numero di cifre dei numeri da ordinare.

Dovendo ordinare n numeri di m bit ciascuno possiamo scegliere $r < m$ e suddividere i numeri in $d=m/r$ "cifre" di r bit ciascuna

In questo caso la base della numerazione è $b=2^r$ e Radix-Sort richiede tempo $\Theta((m/r)(n+2^r))$

La funzione $f(r) = (m/r)(n+2^r)$ ha un minimo per r tale che $r + \log_2(r \ln 2 - 1) = \log_2 n$

Quindi il valore ottimo di r dipende soltanto da n ed è approssimativamente $\log_2 n$.