Analisi degli algoritmi

Progettazione di Algoritmi a.a. 2020-21

Matricole congrue a 1

Docente: Annalisa De Bonis

Analisi degli algoritmi

- E' possibile progettare diversi algoritmi per risolvere uno stesso problema
  - Si pensi ad esempio agli algoritmi di ordinamento di n numeri: Merge Sort, Quick Sort, Insertion Sort, Bubble Sort, Selection Sort, Heap Sort, ...
- Un algoritmo può impiegare molto meno tempo di un altro
  - MergeSort: tempo proporzionale a nlog n (tempo pari circa a cnlogn dove c è una costante che non dipende da n)
  - QuickSort: tempo nel caso peggiore proporzionale a n^2 (tempo pari circa a $c'n^2$ dove c' è una costante che non dipende da n)

  o usare molto meno spazio di un altro
  - Alcuni algoritmi di ordinamento non utilizzano strutture dati ausiliarie in quanto ordinano "sul posto" andando a modificare la posizione degli elementi all'interno della sequenza input. Questi algoritmi richiedono solo una piccola quantità di memoria aggiuntiva che è molto inferiore rispetto alla dimensione n dell'input e in ogni momento mantengono al piu` un numero costante di elementi in memoria aggiuntiva.
    - Esempi: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort,....

Analisi degli algoritmi

- È utile avere un modo per confrontare tra loro diverse soluzioni per capire quale sia la migliore. Migliore in base ad un certo criterio di efficienza, come ad esempio uso della memoria o velocità .
- Abbiamo bisogno di tecniche di analisi che consentano di valutare un algoritmo **solo in base alle sue caratteristiche e non a quelle del codice che lo implementa o della macchina su cui è eseguito.**
- Come informatici, oltre a dover essere in grado di trovare soluzioni ai problemi, dobbiamo essere in grado di valutare la nostra soluzione e capire se c'è margine di miglioramento.
  - Limiti inferiori

Efficienza degli algoritmi

Proviamo a definire la nozione di efficienza (rispetto al tempo di esecuzione):

- ***Un algoritmo è efficiente se, quando è implementato, viene eseguito velocemente su istanze input reali.***
  - Concetto molto vago.
    - Non chiarisce **dove** viene eseguito l'algoritmo e **quanto veloce** deve essere la sua esecuzione
      - Anche un algoritmo molto cattivo può essere eseguito molto velocemente se è applicato a un input molto piccolo o se è eseguito con un processore molto veloce
      - Anche un algoritmo molto buono può richiedere molto tempo per essere eseguito se implementato male
    - ...

Efficienza degli algoritmi

- ...
- Non chiarisce cosa è un'istanza input reale
  - Noi non conosciamo a priori tutte le possibili istanze input reali
  - Alcune istanze potrebbero essere più "cattive" di altre
- Inoltre non fa capire come la velocità di esecuzione dell'algoritmo deve variare al crescere della dimensione dell'input
- Due algoritmi possono avere tempi di esecuzione simili per input piccoli ma tempi di esecuzione molto diversi per input grandi

Efficienza degli algoritmi

- Vogliamo una definizione concreta di efficienza
  - **indipendente dal processore**
  - **indipendente dal tipo di istanza**
  - che dia **una misura di come aumenta il tempo di esecuzione al crescere della dimensione dell'input**.

Efficienza

Forza bruta. Per molti problemi non triviali, esiste un naturale algoritmo di forza bruta che controlla ogni possibile soluzione.

- Tipicamente impiega tempo 2^N (o peggio) per input di dimensione N.
- Non accettabile in pratica.

- Esempio:
  - Voglio ordinare in modo crescente un array di N numeri distinti
  - Soluzione (ingenua) esponenziale: permuto i numeri ogni volta in modo diverso fino a che ottengo la permutazione ordinata (posso verificare se una permutazione è ordinata con al più N-1 confronti, confrontando ciascun elemento con il successivo)
    - Nel caso pessimo genero N! permutazioni
    - NB: $N! > 2^N$ per $N>3$

Efficienza

- **Problemi con l'approccio basato sulla ricerca esaustiva nello spazio di tutte le possibili soluzioni (forza bruta)**
  - **Ovviamente richiede molto tempo**
  - **Non fornisce alcuna informazione sulla struttura del problema che vogliamo risolvere.**

- **Proviamo a ridefinire la nozione di efficienza:** Un algoritmo è efficiente se ha una performance migliore, da un punto di vista analitico, dell'algoritmo di forza bruta.

- **Definizione molto utile.** Algoritmi che hanno performance migliori rispetto agli algoritmi di forza bruta di solito **usano euristiche interessanti** e **forniscono informazioni rilevanti sulla struttura intrinseca del problema e sulla sua trattabilità computazionale.**
- **Problema con questa definizione.** Anche questa definizione è vaga. Cosa vuol dire "perfomance migliore"?

Tempo polinomiale

Proprietà desiderata. Quando la dimensione dell'input raddoppia, l'algoritmo dovrebbe risultare più lento solo di un fattore costante c

Mergesort:

per N --> tempo $c \times N \times \log N$;

per 2N --> tempo $c \times 2N \times \log(2N) = 2 \times c \times N \times (\log N + 1)$
$= 2 \times c \times N \times \log N + 2 \times c \times N$
$\leq 2 \times c \times N \times \log N + 2 \times c \times N \times \log N$
$= \mathbf{4} \times c \times N \times \log N$ per ogni $N>1$;

aumenta di al più **4** volte

Algoritmo di forza bruta: per N tempo $c \times (N-1) \times N!$
Per 2N tempo $c \times (2N-1) \times (2N)! = c \times (2N-1) \times (2N \times (2N-1) \times \cdots \times (N+1) \times N!)$
$> c \times 2 \times (N-1) \times N! \times N! = (2 \times N!) \times c \times (N-1) \times N!$
(ultima disuguaglianza perchè $(2N-1) > 2 \times (N-1)$ e $2N \times (2N-1) \times \cdots \times (N+1) > N!$)
$2 \times N!$ non è una costante

Tempo polinomiale

Def. Si dice che un algoritmo impiega tempo polinomiale (poly-time) se quando la dimensione dell'input raddoppia, l'algoritmo risulta più lento solo di un fattore costante c

Esistono due costanti $c > 0$ e $d > 0$ tali che su ciascun input di dimensione N, il numero di passi è limitato da $c N^d$.

Se si passa da un input di dimensione N ad uno di dimensione 2N allora il tempo di esecuzione passa da $c N^d$ a $c (2N)^d = c 2^d N^d$
NB: 2^d è una costante

Analisi del caso pessimo

Tempo di esecuzione nel caso pessimo. Ottenere un bound sul **più grande tempo di esecuzione possibile** per tutti gli input di una certa dimensione N.

- In genere è una buona misura di come si comportano gli algoritmi nella pratica
- Approccio "pessimistico" (in molti casi l'algoritmo potrebbe comportarsi molto meglio)
  - ma è difficile trovare un'alternativa efficace a questo approccio

Tempo di esecuzione nel caso medio. Ottenere un bound al tempo di esecuzione su un **input random** in funzione di una certa dimensione N dell'input.

- Difficile se non impossibile modellare in modo accurato istanze reali del problema mediante distribuzioni input.
- Un algoritmo disegnato per una certa distribuzione di probabilità sull'input potrebbe comportarsi molto male in presenza di altre distribuzioni.

Tempo polinomiale nel caso pessimo

Def. Un algoritmo è **efficiente** se il suo tempo di esecuzione nel caso pessimo è polinomiale.

Motivazione: Funziona veramente in pratica!

- Sebbene $6.02 \times 10^{23} \times N^{20}$ sia, da un punto di vista tecnico, polinomiale, un algoritmo che impiega questo tempo potrebbe essere inutile in pratica.
- Per fortuna, i problemi per cui esistono algoritmi che li risolvono in tempo polinomiale, quasi sempre ammettono algoritmi polinomiali il cui tempo di esecuzione è proporzionale a polinomi che crescono in modo moderato ($c \times N^d$ con c e d piccoli).
- Progettare un algoritmo polinomiale porta a scoprire importanti informazioni sulla struttura del problema

Eccezioni

- Alcuni algoritmi polinomiali hanno costanti moltiplicative e/o all'esponente grandi e sono inutili nella pratica
- Alcuni algoritmi esponenziali sono largamente usati perchè il caso pessimo si presenta molto raramente.
  - Esempio: algoritmo del simplesso per risolvere problemi di programmazione lineare

Perchè l'analisi della complessità è importante

La tabella riporta i tempi di esecuzione su input di dimensione crescente, per un processore che esegue un milione di istruzioni per secondo.
Nei casi in cui il tempo di esecuzione è maggiore di 10^{25} anni, la tabella indica che il tempo richiesto è molto lungo (very long). N.B.: la formazione del pianeta Terra risale a circa 4,54x10^9 di anni fa.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Analisi degli algoritmi

Esempio:

```
InsertionSort(a):        //n è la lunghezza di a
 For(i=1;i<n;i=i+1){
   elemDaIns=a[i];
   j=i-1;
   While(j ≥ 0 && a[j] > elemDaIns){ //cerca il posto per a[i]
        a[j+1]=a[j];      //shifto a destra gli elementi più grandi
        j=j-1;
    }
   a[j+1]=elemDaIns;
 }
```


Analisi degli algoritmi

t_i= numero di iterazioni del while all'i-esima iterazione del for

InsertionSort(a):	costo	numero di volte
For(i=1;i<n;i=i+1){	c_1	n
elemDaIns=a[i];	c_2	n-1
j=i-1;	c_3	n-1
While(j ≥ 0 && a[j] > elemDaIns){	c_4	$\sum_{i=1}^{n-1} t_i$
a[j+1]=a[j];	c_5	$\sum_{i=1}^{n-1} (t_i-1)$
j=j-1;	c_6	$\sum_{i=1}^{n-1} (t_i-1)$
}		
a[j+1]=elemDaIns;	c_7	n-1
}		

Analisi di InsertionSort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n} (t_i - 1) + c_7(n-1)$$

Nel caso pessimo t_i =i+1 per ogni i (elementi in ordine decrescente)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} (i+1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n} i + c_7(n-1)$$

Analisi di InsertionSort

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} (i+1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n-1} i + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4((\sum_{i=1}^{n-1} i) + n - 1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n-1} i + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{(n-1)n}{2} + n - 1) + c_5(\frac{(n-1)n}{2}) + c_6(\frac{(n-1)n}{2}) + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n^2}{2} + \frac{n}{2} - 1) + c_5(\frac{n^2}{2} - \frac{n}{2}) + c_6(\frac{n^2}{2} - \frac{n}{2}) + c_7(n-1) \\
&= (c_4 + c_5 + c_6)\frac{n^2}{2} + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 - c_3 - c_4 - c_7) \\
&= an^2 + bn + c
\end{aligned}$$

Ordine di grandezza

- Nell'analizzare la complessità di InsertionSort abbiamo operato delle astrazioni
- Abbiamo ignorato il valore esatto prima delle costanti c_i e poi delle costanti a, b e c.
- Il calcolo di queste costanti per alcuni algoritmi può essere molto stancante ed è inutile rispetto alla classificazione degli algoritmi che vogliamo ottenere.
- Queste costanti inoltre dipendono
  - dalla macchina su cui si esegue il programma
  - dal tipo di operazioni che contiamo
    - Operazioni del linguaggio ad alto livello
    - Istruzioni di basso livello in linguaggio macchina

Ordine di grandezza

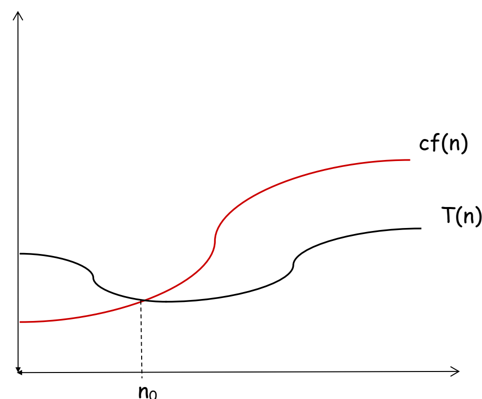
- Possiamo aumentare il livello di astrazione considerando solo l'ordine di grandezza
- Consideriamo solo il termine "dominante"
  - Per InsertionSort: an^2
  - Giustificazione: più **grande** è n, minore è il contributo dato dagli altri termini alla stima della complessità
- Ignoriamo del tutto le costanti
  - Diremo che il tempo di esecuzione di InsertionSort ha ordine di grandezza n^2
  - Giustificazione: più grande è n, minore è il contributo dato dalle costanti alla stima della complessità

Efficienza asintotica degli algoritmi

- Per input piccoli può non essere corretto considerare solo l'ordine di grandezza ma per input "abbastanza" grandi è corretto farlo
- Esempio: $10n^2+100n+10$
  per $n<10$, il secondo termine è maggiore del primo
  man mano che n cresce il contributo dato dai termini meno significativi diminuisce

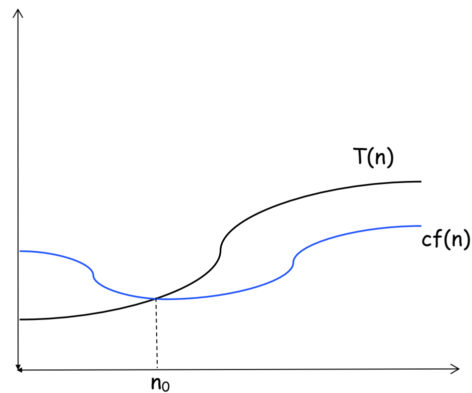
Ordine asintotico di grandezza

Limiti superiori. $T(n)$ è $O(f(n))$ se esistono delle costanti $c > 0$ ed $n_0 \geq 0$ tali che per tutti gli $n \geq n_0$ si ha $0 \leq T(n) \leq c \cdot f(n)$.



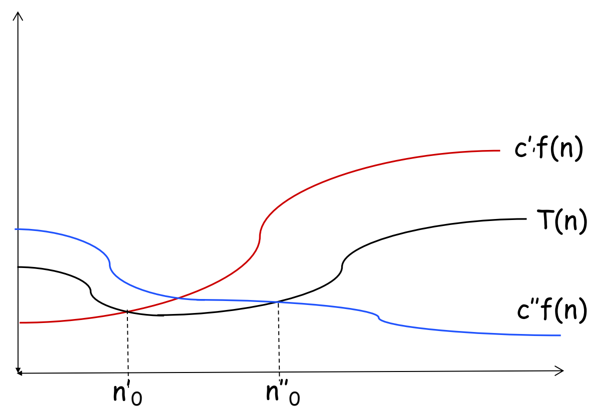
Ordine asintotico di grandezza

Limiti inferiori. T(n) è Ω(f(n)) se esistono costanti $c > 0$ ed $n_0 \geq 0$ tali che per tutti gli $n \geq n_0$ si ha $T(n) \geq c \cdot f(n) \geq 0$.



Ordine asintotico di grandezza

Limiti esatti. T(n) è Θ(f(n)) se T(n) sia O(f(n)) che Ω(f(n)).



Ordine asintotico di grandezza

- Quando analizziamo un algoritmo miriamo a trovare stime asintotiche quanto più "strette" è possibile
- Dire che InsertionSort ha tempo di esecuzione $O(n^3)$ non è errato ma $O(n^3)$ non è un limite "stretto" in quanto si può dimostrare che InsertionSort ha tempo di esecuzione $O(n^2)$
- $O(n^2)$ è un limite stretto?
  - Sì, perché il numero di passi eseguiti da InsertionSort è an^2+bn+c, con $a>0$, che non solo è $O(n^2)$ ma è anche $\Omega(n^2)$.
  - Si può dire quindi che il tempo di esecuzione di InsertionSort è $\Theta(n^2)$

Errore comune

Affermazione priva di senso. Ogni algoritmo basato sui confronti richiede almeno $O(n \log n)$ confronti.

- **Per i lower bound si usa Ω**

Affermazione corretta. Ogni algoritmo basato sui confronti richiede almeno $\Omega(n \log n)$ confronti.

Proprietà

Transitività .
- Se $f = O(g)$ e $g = O(h)$ allora $f = O(h)$.
- Se $f = \Omega(g)$ e $g = \Omega(h)$ allora $f = \Omega(h)$.
- Se $f = \Theta(g)$ e $g = \Theta(h)$ allora $f = \Theta(h)$.

Additività .
- Se $f = O(h)$ e $g = O(h)$ allora $f + g = O(h)$.
- Se $f = \Omega(h)$ e $g = \Omega(h)$ allora $f + g = \Omega(h)$.
- Se $f = \Theta(h)$ e $g = \Theta(h)$ allora $f + g = \Theta(h)$.

Bound asintotici per alcune funzioni di uso comune

Polinomi. $a_0 + a_1 n + \ldots + a_d n^d$, con $a_d > 0$, è $\Theta(n^d)$.

Dim. $O(n^d)$: dobbiamo trovare due costanti $c>0$ e $n_0 \geq 0$ tali che $a_0 + a_1 n + \ldots + a_d n^d \leq c\, n^d$ per ogni $n \geq n_0$

$a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$

$\leq |a_0| + |a_1| n + |a_2| n^2 + \ldots + |a_d| n^d$

$\leq (|a_0| + |a_1| + |a_2| + \ldots + |a_d|)\, n^d$, per ogni $n \geq 1$.

Basta quindi prendere $n_0=1$ e $c=|a_0| + |a_1| + \ldots + |a_d|$ (è una costante)

Bound asintotici per alcune funzioni di uso comune

Dimostriamo come esercizio che $a_0 + a_1n + \ldots + a_dn^d$ è anche $\Omega(n^d)$:

- $a_0 + a_1n + \ldots + a_dn^d = a_dn^d + \ldots + a_1n + a_0 \geq a_dn^d - (|a_0| + |a_1|n + \ldots + |a_{d-1}|n^{d-1})$
- Abbiamo appena visto che un polinomio di grado d è $O(n^d)$
  - Ciò implica $a_0 + |a_1|n + \ldots + |a_{d-1}|n^{d-1} = O(n^{d-1})$
    e di conseguenza esistono due costanti $n'_0 \geq 0$ e $c' > 0$
    tali che $a_0 + |a_1|n + \ldots + |a_{d-1}|n^{d-1} \leq c'n^{d-1}$ per ogni $n \geq n'_0$
- Quindi $a_dn^d - (a_0 + |a_1|n + \ldots + |a_{d-1}|n^{d-1}) \geq a_dn^d - c'n^{d-1}$ **per ogni $n \geq n'_0$**
- da cui $a_0 + a_1n + \ldots + a_dn^d \geq a_dn^d - c'n^{d-1}$ **per ogni $n \geq n'_0$**

Bound asintotici per alcune funzioni di uso comune

✸ Per dimostrare $a_0 + a_1n + \ldots + a_dn^d = \Omega(n^d)$ dobbiamo trovare le costanti $n_0 \geq 0$ e $c > 0$ tali che $a_0 + a_1n + \ldots + a_dn^d \geq cn^d$ per ogni $n \geq n_0$

🦆 Nella slide precedente abbiamo dimostrato che esistono due costanti $n'_0 \geq 0$ e $c' > 0$ tali che $a_0 + a_1n + \ldots + a_dn^d \geq a_dn^d - c'n^{d-1}$ per ogni $n \geq n'_0$.

- Scriviamo il secondo membro della disequazione come $(a_d - c'/n)n^d$

🌵 Se per una certa costante $n''_0 > 0$ avessimo $(a_d - c'/n''_0) > 0$, allora siccome $a_d - c'/n$ è crescente si avrebbe $a_d - c'/n''_0 \leq (a_d - c'/n)$ per ogni $n \geq n''_0$. Cio\` implicherebbe che esiste una costante $c'' = a_d - c'/n''_0$ t.c. $0 < c'' \leq (a_d - c'/n)$ **per ogni $n \geq n''_0$** (e dalla 🦆 si avrebbe $a_0 + a_1n + \ldots + a_dn^d \geq c'' n^d$ per ogni $n \geq n'_0$)

- Siccome a_d è maggiore di 0 esistono sicuramente valori di n per cui $(a_d - c'/n) > 0$. Troviamoli! Imponiamo $a_d - c'/n > 0$ che è soddisfatta per $n > c'/a_d$. Quindi se prendiamo n piu\` grande di c'/a_d, si ha $(a_d - c'/n) > 0$. Prendiamo allora n''_0 in 🌵 uguale a $2c'/a_d$.
- Calcoliamo la costante c'' in 🌵. $c'' = a_d - c'/n''_0 = a_d - c'/(2c'/a_d) = a_d - a_d/2 = a_d/2$
- Abbiamo trovato le due costanti di 🌵, cioe\` $c'' > 0$ e $n''_0 \geq 0$ t.c. $c'' \leq a_d - c'/n$ per ogni $n \geq n''_0$
- Per la 🦆 si ha $a_0 + a_1n + \ldots + a_dn^d \geq (a_d - c'/n)n^d$ per ogni $n \geq n'_0$

Bound asintotici per alcune funzioni di uso comune

- Mettendo insieme i due ultimi punti della slide precedente otteniamo $a_0 + a_1 n + \ldots + a_d n^d \geq (a_d - c'/n) n^d \geq c\, n^d$ per ogni $n \geq n_0$ dove c è la costante $c = c'' = a_d/2$ ed n_0 deve essere scelto in modo che prendendo $n \geq n_0$ siano soddisfatte entrambe le diseguaglianze dei due ultimi punti della slide precedente. Prendiamo quindi $n_0 = \max\{n'_0, 2c'/a_d\}$

Ordine asintotico di grandezza

Esempio:

$T(n) = 32n^2 + 17n + 32.$

- $T(n)$ è $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$ e $\Theta(n^2)$.
- $T(n)$ non è $O(n)$, $\Omega(n^3)$, $\Theta(n)$ o $\Theta(n^3)$.

Tempo lineare: O(n)

Tempo lineare. Il tempo di esecuzione è al più un fattore costante per la dimensione dell'input.

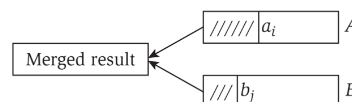
Esempio:
Computazione del massimo. Computa il massimo di n numeri $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   Se (ai > max)
      max ← ai
}
```

Il problema dell'individuazione del max di n numeri e` Ω(n)
Dim. ogni numero diverso dal massimo deve partecipare ad almeno un confronto in cui risulta < dell'altro elemento → almeno un confronto per ciascuno degli n-1 elementi diversi dal massimo

Tempo lineare: O(n)

Merge. Combinare 2 sequenze ordinate $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_m$ in una lista ordinata.




```
i = 1, j = 1
while (i≤n and j≤m) {
   if (ai ≤ bj) aggiungi ai alla fine della lista output e incrementa i
   else aggiungi bj alla fine della lista output e incrementa j
}
Ciclo per aggiungere alla lista output gli elementi non ancora
esaminati di una delle due liste input
```

Affermazione. Fondere due sequenze ordinate rispettivamente di dimensione n ed m richiede tempo O(n+m).
Dim. Dopo ogni iterazione del while o del ciclo sottostante, la lunghezza dell'output aumenta di 1.

Tempo quadratico: $O(n^2)$

Tempo quadratico. Tipicamente si ha quando un algoritmo esamina tutte le coppie di elementi input

Coppia di punti più vicina. Data una lista di n punti del piano $(x_1, y_1), \ldots, (x_n, y_n)$, vogliamo trovare la coppia più vicina.

Soluzione $O(n^2)$. Calcola la distanza tra tutte le coppie di punti.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      Se (d < min)
         min ← d
   }
}
```

← Per effettuare i confronti non c'è bisogno di estrarre la radice quadrata

A lezione, per esercizio, abbiamo svolto l'analisi dell'algoritmo nella slide precedente…

Il for esterno mi costa: tempo lineare in n + il tempo per eseguire tutte le iterazioni del for interno

Analizziamo il for interno:

Chiamiamo t_i il numero di iterazioni del for interno alla i-esima iterazione del for esterno

Quante volte viene iterato il for interno in totale? Risposta: $t_1+t_2+\ldots+t_n$.

Per ogni i si ha $t_i=(n-i)$

Quindi sommando i t_i per tutte le iterazioni del for esterno ho

$t_1+t_2+\ldots+t_n = (n-1)+(n-2)+\ldots+1+0 = n(n-1)/2$ iterazioni del for interno **IN TOTALE**

siccome la singola esecuzione del corpo del for interno richiede tempo pari ad una costante c
→ il tempo richiesto da tutte le iterazioni del for interno è $c(n(n-1)/2)=\Theta(n^2)$

Tempo totale: $\Theta(n)+\Theta(n^2)=\Theta(n^2)$

Tempo cubico: $O(n^3)$

Tempo cubico. Tipicamente si ha quando un algoritmo esamina tutte le triple di elementi.

Esempio:

Disgiunzione di insiemi. Dati n insiemi $S_1, \ldots, S_n$ ciascuno dei quali è un sottoinsieme di {1, 2, …, n}, c'è qualche coppia di insiemi che è disgiunta?

Soluzione $O(n^3)$. Per ogni coppia di insiemi, determinare se i due insiemi sono disgiunti. (Supponiamo di poter determinare in tempo costante se un elemento appartiene ad un insieme)

```
flag = true
for i = 1 to n{      //corpo iterato n volte
 for j = i+1 to n { //corpo iterato n-i volte ad ogni iterazione del for esterno
     foreach elemento p di Si { //corpo iterato al più n volte ad ogni iteraz. for su j
        if p appartiene anche a Sj //supponiamo test richiede ogni volta  O(1)
            flag = false; break;
     }
   If(flag = true)// nessun elemento di Si appartiene a Sj
         riporta che Si  e  Sj sono disgiunti
  }
}
```