

Complessità degli algoritmi

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

21 luglio 2017

Complessità di un algoritmo

Quanto efficientemente risolviamo un problema?

- Dato un problema, un buon flusso di lavoro è:
 - 1 Concepiamo un algoritmo che lo risolve
 - 2 Ne valutiamo la complessità
 - 3 Se la complessità è soddisfacente, lo implementiamo
- Per la correttezza, non c'è una soluzione in generale
 - ... ma questo non nega a priori la possibilità di dimostrarla per dati casi particolari
- Per valutare la complessità ci serve rappresentare l'algoritmo in una qualche forma "comoda"

Scelta del linguaggio

Pseudocodice

- Semplice linguaggio di programmazione imperativo
- Tralascia gli aspetti non fondamentali per le nostre analisi
- Facilmente traducibile in C/Java/Python/C++
- Sintassi piuttosto asciutta (simile a Python)
- Chiaramente, è possibile effettuare analisi di complessità anche su codice scritto in un qualunque linguaggio di programmazione
 - Utile per trovare porzioni particolarmente costose

Pseudocodice - Sintassi

Procedure, assegnamenti, costrutti di controllo

- Ogni algoritmo è rappresentato con una procedura (= modifica i dati in input, non ritorna nulla)
- Operatori: Aritmetica come in C, assegnamento (\leftarrow), e confronti ($<$, \leq , $=$, \geq , $>$, \neq)
- Commenti mono-riga con \triangleright , blocchi dati dall'indentazione
- Costrutti di controllo disponibili : while, for, if-else
- Tutte le variabili sono locali alla procedura descritta
- Il tipo delle variabili non è esplicito, va inferito dal loro uso

Tipi di dato aggregato

- Ci sono gli array, notazione identica al C
- In aggiunta al C sono disponibili anche i sotto-array (slices) come in Fortran/Matlab/Python
 - $A[i..j]$ è la porzione di array che inizia dall' i -esimo elemento e termina al j -esimo
- Sono presenti aggregati eterogenei (= strutture C)
 - L'accesso a un campo è effettuato tramite l'operatore $.$
 $A.campo1$ è il campo di nome $campo1$ della struttura A
 - Diversamente dal C, una variabile di tipo aggregato è un *puntatore* all'oggetto
 - Un puntatore non riferito ad alcun oggetto ha valore NIL

1 $y \leftarrow x$

2 $x.f \leftarrow 3$ // dopo questa riga anche $y.f$ vale 3

Pseudocodice - Convenzioni

Passaggio parametri

- Il passaggio di parametri ad una procedura viene effettuato :
 - Nel caso di tipi non aggregati: per copia
 - Nel caso di tipi aggregati: per riferimento
- Comportamento identico al C per tipi non aggregati ed array
- Diverso per le strutture (uguale a quello di Java)

Modello di esecuzione

- Lo pseudocodice è eseguito dalla macchina RAM
- Assunzione fondamentale: un singolo statement di assegnamento tra tipi base in pseudocodice è tradotto in k istruzioni dell'assembly RAM

Criteri per l'analisi

Criterio di costo

- Adottiamo il criterio di *costo costante* per l'esecuzione dei nostri algoritmi
 - La maggioranza degli algoritmi che vedremo non ha espansioni significative della dimensione dei singoli dati
 - Possiamo, in quei casi, considerare dati a precisione multipla come vettori di cifre
- Ogni statement semplice di pseudocodice è eseguito in $\Theta(k)$
- Focalizzeremo la nostra analisi principalmente sulla complessità *temporale* degli algoritmi
 - É quella che presenta variazioni più “interessanti”

Una prima analisi

Cancellare un elemento da ...

- ... un vettore

CANCELLAELV(v, len_v, e)

```
1   $i \leftarrow 0$ 
2  while  $v[i] \neq e$ 
3       $i \leftarrow i + 1$ 
4  while  $i < len_v - 1$ 
5       $v[i] \leftarrow v[i + 1]$ 
6   $v[len_v - 1] \leftarrow 0$ 
```

- ... una lista

CANCELLAELL(l, e)

```
1   $p \leftarrow l$ 
2  while  $p.value \neq e$ 
3       $p \leftarrow p.next$ 
4   $p \leftarrow p.next$ 
```

- Sono entrambi $\Theta(n)$ (n numero degli elementi del vettore/lista) nel caso pessimo, oppure $\mathcal{O}(n)$ in generale

Un altro esempio

Moltiplicazione di matrici: $\dim(A) = \langle n, m \rangle$ $\dim(B) = \langle m, o \rangle$

MATRIXMULTIPLY(A, B)

```
1  for  $i \leftarrow 0$  to  $n - 1$ 
2      for  $j \leftarrow 0$  to  $o - 1$ 
3           $C[i][j] \leftarrow 0$ 
4          for  $k \leftarrow 0$  to  $m - 1$ 
5               $C[i][j] \leftarrow C[i][j] + A[i][k] \cdot B[k][j]$ 
6  return  $C$ 
```

- La riga 3 viene eseguita $n \cdot o$ volte, la riga 5 viene eseguita $n \cdot m \cdot o$ volte $\rightarrow \Theta(n \cdot m \cdot o)$ (sia nel caso pessimo, che in generale) $= \Theta(n^3)$ con matrici quadrate

Ricorsione e complessità

Come calcolare la complessità di algoritmi ricorsivi?

- È possibile incontrare algoritmi la cui complessità non è immediatamente esprimibile in forma chiusa
- Il caso tipico sono algoritmi *divide et impera*:
 - Suddivido il problema in sottoproblemi con dimensione dell'input pari a una frazione $\frac{1}{b}$ dell'originale
 - Quando il sottoproblema ha ingresso di dimensioni n piccole a sufficienza, può essere risolto a tempo costante
 - Indichiamo con $D(n)$ il costo del suddividere il problema e con $C(n)$ il costo di combinare le soluzioni
- Possiamo esprimere il costo totale $T(n)$ con la seguente *equazione di ricorrenza* (o *ricorrenza*):

$$T(n) = \begin{cases} \Theta(1) & \text{se } n < c \\ D(n) + aT\left(\frac{n}{b}\right) + C(n) & \text{altrimenti} \end{cases}$$

Come risolvere le ricorrenze?

- Sono possibili 3 tecniche principali:
 - Sostituzione
 - Esame dell'albero di ricorsione
 - Teorema dell'esperto (master theorem)
- Usiamo come caso di studio la ricerca binaria:
 - Formuliamo il problema di cercare in un vettore lungo n come quello di cercare nelle sue metà superiori e inferiori
 - Costo di suddivisione (calcolo indici) costante $D(n) = \Theta(1)$
 - Costo di ricombinazione costante: sappiamo che una delle due metà non contiene per certo l'elemento cercato $C(n) = \Theta(1)$
 - Complessità espressa come $T(n) = \Theta(1) + T(\frac{n}{2}) + \Theta(1)$

Metodo di sostituzione

Ipotesi e dimostrazione

- Il metodo di sostituzione si articola in tre fasi:
 - 1 Intuire una possibile soluzione
 - 2 Sostituire la presunta soluzione nella ricorrenza
 - 3 Dimostrare per induzione che la presunta soluzione è tale per l'equazione/disequazione alle ricorrenze

- Ad esempio, con la complessità della ricerca binaria:

$$T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1)$$

- 1 Intuizione : penso sia $T(n) = \mathcal{O}(\log(n))$ ovvero $T(n) \leq c \log(n)$
- 2 Sostituisco $T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1) \leq c \cdot \log(n)$
- 3 Considero vero per ipotesi di induzione $T\left(\frac{n}{2}\right) \leq c \cdot \log\left(\frac{n}{2}\right)$ in quanto $\frac{n}{2} < n$ e sostituisco nella (2) ottenendo :
 $T(n) \leq c \cdot \log\left(\frac{n}{2}\right) + \Theta(k) = c \cdot \log(n) - c \log(2) + \Theta(k) \leq c \log(n)$

Metodo di sostituzione

Ipotesi e dimostrazione

- Il metodo di sostituzione si articola in tre fasi:
 - 1 Intuire una possibile soluzione
 - 2 Sostituire la presunta soluzione nella ricorrenza
 - 3 Dimostrare per induzione che la presunta soluzione è tale per l'equazione/disequazione alle ricorrenze

- Ad esempio, con la complessità della ricerca binaria:

$$T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1)$$

- 1 Intuizione : penso sia $T(n) = \mathcal{O}(\log(n))$ ovvero $T(n) \leq c \log(n)$
- 2 Sostituisco $T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1) \leq c \cdot \log(n)$
- 3 Considero vero per ipotesi di induzione $T\left(\frac{n}{2}\right) \leq c \cdot \log\left(\frac{n}{2}\right)$ in quanto $\frac{n}{2} < n$ e sostituisco nella (2) ottenendo :
 $T(n) \leq c \cdot \log\left(\frac{n}{2}\right) + \Theta(k) = c \cdot \log(n) - c \log(2) + \Theta(k) \leq c \log(n)$

Metodo di sostituzione

Esempio 2

- Determiniamo un limite superiore per $T(n) = 2T(\frac{n}{2}) + n$
- Intuiamo $\mathcal{O}(n \log(n))$, dimostriamo $T(n) \leq c(n \log(n))$
- Supponiamo vero (hp. induzione) $T(\frac{n}{2}) \leq c(\frac{n}{2} \log(\frac{n}{2}))$
- Sostituiamo ottenendo che
$$T(n) \leq 2c(\frac{n}{2} \log(\frac{n}{2})) + n \leq cn \log(\frac{n}{2}) + n =$$
$$cn \log(n) - cn \log(2) + n \leq cn \log(n) + (1 - c \log(2))n$$
 - Il comportamento asintotico è quello che vorrei
- Riesco a trovare un n_0 opportuno dal quale in poi valga la disuguaglianza, assumendo che $T(1) = 1$ per definizione?
 - Provo $n_0 = 1$, ottengo $1 \leq 0 + 1 - c \log(2)$, no.
 - Con $n_0 = 2$ funziona,
$$T(2) = 2 \cdot 1 + 2 \leq 2c \log(2) + 2 - c \log(2) = c \log(2) + 2$$

Metodo di sostituzione

Esempio 2 - Un limite più stretto

- Determiniamo un limite superiore per $T(n) = 2T(\frac{n}{2}) + 1$
- Tentiamo di provare che è $\mathcal{O}(n)$, ovvero $T(n) \leq cn$
- Supponiamo vero (hp. induzione) $T(\frac{n}{2}) \leq c\frac{n}{2}$
- Sostituiamo ottenendo che $T(n) \leq 2c\frac{n}{2} + 1 = cn + 1$
 - Non possiamo trovare un valore di c che faccia rispettare l'ipotesi che vogliamo: $cn + 1$ è sempre maggiore di cn
- In questo caso, non siamo riusciti a dimostrare il limite tramite sostituzione
- N.B.: questo *non* implica che $T(n)$ non sia $\mathcal{O}(n)$
 - Prendere come ipotesi $T(n) \leq cn - b$, con b costante, consente di dimostrare che è $\mathcal{O}(n)$

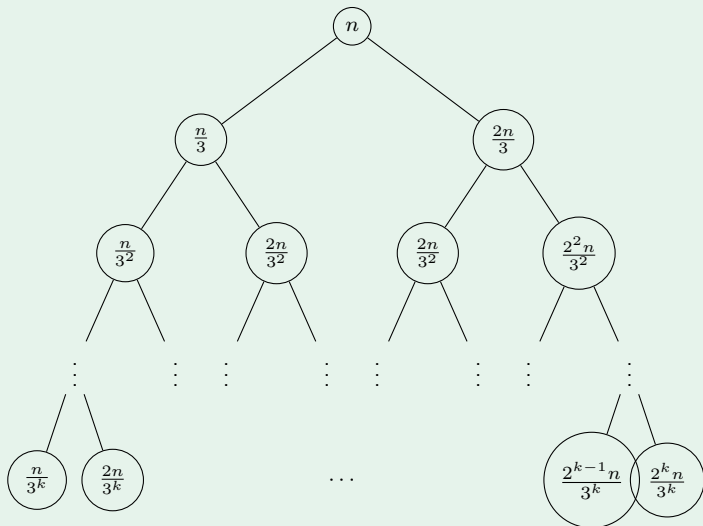
Metodo dell'albero di ricorsione

Espandere le chiamate ricorsive

- L'albero di ricorsione fornisce un aiuto per avere una congettura da verificare con il metodo di sostituzione
- É una rappresentazione delle chiamate ricorsive, indicando per ognuna la complessità
- Ogni chiamata costituisce un nodo in un albero, i chiamati appaiono come figli del chiamante
- Rappresentiamolo per $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \mathcal{O}(n)$

Metodo dell'albero di ricorsione

Espandendo completamente



Metodo dell'albero di ricorsione

Espandendo completamente

- L'albero ha la ramificazione a profondità massima posta sull'estrema destra del disegno precedente
- Sappiamo che essa ha profondità k che ricaviamo ponendo $\frac{2^k}{3^k}n = 1$ (la foglia ha un solo elemento) $\rightarrow 2^k n = 3^k \rightarrow \log_3(2^k n) = k = \log_3(2^k) + \log_3(n) = \log_3(n) + \frac{\log_2(2^k)}{\log_2(3)}$ da cui abbiamo che $k = \frac{\log_2(3) \log_3(n)}{(\log_2(3)-1)} = c \log_3(n)$
- Il costo pessimo per il contributo di un dato livello è l' $\mathcal{O}(n)$ del primo livello
- Congetturiamo che $T(n) = \Theta(n \log(n))$
 - Dimostriamolo mostrando che $T(n) = \mathcal{O}(n \log(n))$ e $T(n) = \Omega(n \log(n))$

Metodo dell'albero di ricorsione

$$T(n) = \mathcal{O}(n \log(n))$$

- Per hp. di induzione abbiamo sia che $T(\frac{n}{3}) \leq c'(\frac{n}{3} \log(\frac{n}{3}))$ e che $T(\frac{2n}{3}) \leq c'(\frac{2n}{3} \log(\frac{2n}{3}))$
- Sostituendo abbiamo
$$T(n) \leq c(\frac{n}{3} \log(\frac{n}{3})) + c(\frac{2n}{3} \log(\frac{2n}{3})) + \mathcal{O}(n) = c(\frac{n}{3}(\log(n) - \log(3))) + c(\frac{2n}{3}(\log(n) - \log(3) + \log(2))) + c'''n = c'n \log(n) - c''n + c'''n \leq c'n \log(n)$$
 per una scelta opportuna delle costanti c'', c'''

$$T(n) = \Omega(n \log(n))$$

- Hp ind. $T(\frac{n}{3}) \geq c'(\frac{n}{3} \log(\frac{n}{3}))$, $T(\frac{2n}{3}) \geq c'(\frac{2n}{3} \log(\frac{2n}{3}))$
- Sostituendo $T(n) \geq c'n \log(n) - c''n + c'''n \geq c'n \log(n)$

Teorema dell'esperto (Master theorem)

Uno strumento efficace per le ricorsioni

- Il teorema dell'esperto è uno strumento per risolvere buona parte delle equazioni alle ricorrenze.
- Affinchè sia applicabile, la ricorrenza deve avere la seguente forma: $T(n) = aT(\frac{n}{b}) + f(n)$ con $a \geq 1, b > 1$
- L'idea alla base è quella di confrontare $n^{\log_b(a)}$ (effetto delle chiamate ricorsive) con $f(n)$ (il costo di una singola chiamata)
- Le ipotesi del teorema dell'esperto sono le seguenti:
 - a deve essere costante e ≥ 1 (almeno 1 sotto-problema per chiamata ricorsiva)
 - $f(n)$ deve essere sommata, non sottratta o altro a $aT(\frac{n}{b})$
 - Il legame tra $aT(\frac{n}{b})$ e $f(n)$ deve essere polinomiale
- Se queste ipotesi sono valide, è possibile ricavare informazione sulla complessità a seconda del caso in cui ci si trova

Master Theorem

Caso 1

- Nel primo caso $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ per un qualche $\epsilon > 0$
- La complessità risultante è $T(n) = \Theta(n^{\log_b(a)})$
- Intuitivamente: il costo della ricorsione “domina” quello della singola chiamata
- Esempio: $T(n) = 9T(\frac{n}{3}) + n$
- Confrontiamo: $n^1 = n^{\log_3(9)-\epsilon} \Rightarrow \epsilon = 1 \quad \checkmark$
- Otteniamo che la complessità è: $\Theta(n^{\log_3(9)}) = \Theta(n^2)$

Master Theorem

Caso 2

- Nel secondo caso abbiamo che $f(n) = \Theta(n^{\log_b(a)}(\log(n))^k)$
- La complessità risultante della ricorrenza è
$$T(n) = \Theta(n^{\log_b(a)}(\log(n))^{k+1})$$
- Intuitivamente: il contributo della ricorsione e quello della singola chiamata differiscono per meno di un termine polinomiale
- Esempio: $T(n) = T(\frac{n}{3}) + \Theta(1)$
- Confrontiamo: $\Theta(1) = \Theta(n^{\lfloor \log_3(1) \rfloor}(\log(n))^k)$ è vero per un qualche k ?
 - Sì: $\Theta(1) = \Theta(n^0(\log(n))^0)$ ✓
- La complessità risultante è
$$\Theta(n^{\log_3(1)}(\log(n))^{0+1}) = \Theta(\log(n))$$

Master Theorem

Caso 3

- In questo caso abbiamo che $f(n) = \Omega(n^{\log_b(a)+\epsilon})$, $\epsilon > 0$
- Se questo è vero, deve anche valere che: $af(\frac{n}{b}) < cf(n)$ per un qualche valore di $c < 1$
- Se le ipotesi sono rispettate, abbiamo che $T(n) = \Theta(f(n))$
- Intuitivamente: il costo della singola chiamata è più rilevante della ricorsione
- Esempio: $T(n) = 8T(\frac{n}{3}) + n^3$
- Confrontiamo
 $n^3 = \Omega(n^{\log_3(8)+\epsilon}) \Rightarrow \epsilon = 3 - \log_3(8) > 0 \quad \checkmark$
- Controlliamo se $8f(\frac{n}{3}) = \frac{8}{3^3}n^3 < cn^3$ per un qualche $c < 1$?
 - Sì, basta prendere c in $(1 - (\frac{8}{3^3}); 1)$ \checkmark
- La complessità dell'esempio è: $\Theta(n^3)$

Ordinare una collezione di oggetti

Un problema ricorrente

- Tra i problemi che capita più spesso di dover risolvere, l'ordinamento di una collezione di oggetti è un classico
- Un punto chiave dell'utilità dell'ordinamento è consentire di utilizzare una ricerca binaria sulla collezione ordinata
- Analizziamo soluzioni diverse considerando la loro complessità temporale, spaziale e relative peculiarità
- Proprietà di stabilità: in breve, un algoritmo di ordinamento è stabile se non cambia di ordine elementi duplicati

Insertion Sort

Ordinamento per inserimento di interi (ordine crescente)

INSERTIONSORT(A)

```
1  for  $i \leftarrow 1$  to  $A.length - 1$ 
2
3       $tmp \leftarrow A[i]$ 
4       $j \leftarrow i - 1$ 
5      while  $j > 0$  and  $A[j] > tmp$ 
6
7           $A[j + 1] \leftarrow A[j]$ 
8           $j \leftarrow j - 1$ 
9       $A[j + 1] \leftarrow tmp$ 
```

- Complessità temporale: caso ottimo $\Theta(n)$, caso pessimo $\Theta(n^2)$, in generale $\mathcal{O}(n^2)$. Complessità spaziale $\Theta(1)$. Stabile.

Più veloce di $\mathcal{O}(n^2)$

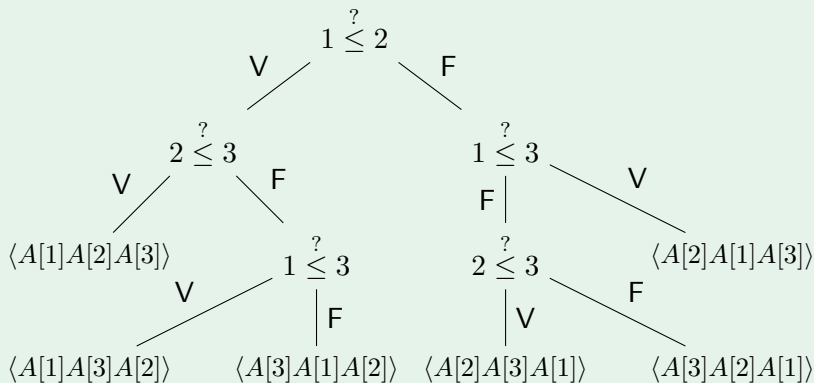
Limiti inferiori della complessità dell'ordinamento

- Abbiamo visto che nel caso pessimo l'Insertion sort è $\Theta(n^2)$
- E' possibile concepire un algoritmo più veloce? Sì
- Qual è il limite di complessità dell'ordinamento per confronto
 - É facile notare che qualunque procedura di ordinamento per n elementi è $\Omega(n)$
 - Sicuramente l'ordinamento è $\mathcal{O}(n^2)$: abbiamo l'insertion sort
- Astraiamo dalla specifica strategia di ordinamento: contiamo le azioni di confronto e scambio

Più veloce di $\mathcal{O}(n^2)$

Limiti inferiori della complessità dell'ordinamento

- Esaminiamo le decisioni per ordinare un vettore di 3 elementi



Limiti inferiori della complessità dell'ordinamento

Stima del numero di confronti

- L'albero costruito in precedenza ha tante foglie quante permutazioni
 - Per un vettore lungo n esso ha $n!$ foglie
- Assumiamo che la struttura sia la più compatta possibile
 - ogni nodo ha lo stesso numero di discendenti (figli, nipoti, pronipoti...)
- La lunghezza del più lungo dei percorsi radice-foglia è il numero max di confronti che devo fare per ordinare un vettore
- L'altezza dell'albero in questo caso è \log_2 del numero delle sue foglie $\rightarrow \log_2(n!) \approx n \log_2(n) - \log_2(e)n + \mathcal{O}(\log_2(n))$
- La complessità migliore ottenibile è $\mathcal{O}(n \log(n))$

Merge Sort

Un algoritmo $\Theta(n \log(n))$

- Per avere un algoritmo di ordinamento con complessità di caso pessimo ottima, applichiamo una strategia *divide et impera*
- Suddividiamo il vettore di elementi da ordinare in porzioni più piccole, fin quando non sono ordinabili in $\Theta(1)$, dopodichè ri-assembliamo i risultati ottenuti
 - É importante che ri-assemblare i risultati ottenuti non abbia complessità eccessiva
- Analizziamo quindi la complessità di fondere due array ordinati in un unico array, anch'esso ordinato
 - Consideriamo i due array come slices di un unico array A: $A[p..q]$, $A[q+1..r]$

Fusione di $A[p..q]$, $A[q+1..r]$ in $A[p..r]$

MERGE(A, p, q, r)

```
1   $len_1 \leftarrow q - p + 1$ ;  $len_2 \leftarrow r - q$ 
2  ALLOCA( $L[1..len_1 + 1]$ ); ALLOCA( $R[2..len_2 + 1]$ )
3  for  $i \leftarrow 1$  to  $len_1$  // Copia della prima metà
4
5       $L[i] \leftarrow A[p + i - 1]$ 
6  for  $i \leftarrow 1$  to  $len_2$  // Copia della seconda metà
7
8       $R[i] \leftarrow A[q + i]$ 
9   $L[len_1 + 1] \leftarrow \infty$ ;  $R[len_2 + 1] \leftarrow \infty$  // sentinelle
10  $i \leftarrow 1$ ;  $j \leftarrow 1$ ;
11 for  $k \leftarrow p$  to  $r$ 
12
13     if  $L[i] \leq R[j]$ 
14          $A[k] \leftarrow L[i]$ ;  $i \leftarrow i + 1$ 
15     else
16          $A[k] \leftarrow R[j]$ ;  $j \leftarrow j + 1$ 
```

Analisi di complessità

- L'algoritmo alloca due array ausiliari, grossi quanto le parti da fondere, più alcune variabili ausiliarie in numero fissato
 - Complessità spaziale $\Theta(n)$
- Tralasciando le porzioni sequenziali, l'algoritmo è composto da 3 cicli:
 - Due per copiare le parti da fondere: complessità $\Theta(n)$
 - Uno che copia in A gli elementi in ordine: complessità $\Theta(n)$
- In totale abbiamo che MERGE è $\Theta(n)$

MergeSort

Algoritmo

MERGESORT(A, p, r)

```
1  if  $p < r - 1$ 
2       $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$ 
3      MERGESORT( $A, p, q$ )
4      MERGESORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
6  else // Caso base della ricorsione: ho solo 2 elementi
7      if  $A[p] < A[r]$ 
8           $tmp \leftarrow A[r]$ 
9           $A[r] \leftarrow A[p]$ 
10          $A[p] \leftarrow tmp$ 
```

- Costo: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$: Caso 2 MT $\rightarrow \Theta(n \log(n))$

Quicksort

Un'alternativa divide-et-impera

- Quicksort ordina senza spazio ausiliario (sul posto, o *in place*)
- Quicksort applica il divide-et-impera ad una slice $A[lo..hi]$:

Dividi Scegli un elemento $A[p]$ (detto pivot) come punto di suddivisione di $A[lo..hi]$ e sposta gli elementi di $A[lo..hi]$ in modo che tutti quelli di $A[lo..p-1]$ siano minori del pivot

Impera Ordina $A[lo..p-1]$, $A[p+1..hi]$ con Quicksort

Combina Nulla! L'ordinamento è eseguito in place

QUICKSORT(A, lo, hi)

```
1  if  $lo < hi$ 
2       $p \leftarrow$  PARTITION( $A, lo, hi$ )
3      QUICKSORT( $A[lo], A[p - 1]$ )
4      QUICKSORT( $A[p + 1], A[hi]$ )
```

Quicksort

Schema di partizione di Lomuto

PARTITION(A, lo, hi)

```
1  pivot ←  $A[hi]$ 
2   $i$  ←  $lo - 1$ 
3  for  $j$  ←  $lo$  to  $hi - 1$ 
4
5      if  $A[j] \leq pivot$ 
6           $i$  ←  $i + 1$ 
7          if  $i \neq j$ 
8              SCAMBIA( $A[i], A[j]$ )
9  SCAMBIA( $A[i + 1], A[hi]$ )
10 return  $i + 1$ 
```

- Usa il cursore i per indicare l'ultimo degli elementi $< pivot$
- Complessità di PARTITIONLOMUTO: $\Theta(n)$

Quicksort

Schema di partizione di Hoare

PARTITIONHOARE(A, lo, hi)

```
1   $pivot \leftarrow A[lo]$ 
2   $i \leftarrow lo - 1; j \leftarrow hi + 1$ 
3  Loop
4       $i \leftarrow i + 1$ 
5      while  $A[i] < pivot$ 
6       $j \leftarrow j - 1$ 
7      while  $A[j] > pivot$ 
8      if  $i \geq j$ 
9      return  $j$ 
10     SCAMBIA( $A[i], A[j]$ )
```

- La complessità asintotica di PARTITIONHOARE è $\Theta(n)$
- Questo schema di partizione effettua $\frac{1}{3}$ degli scambi di quello di Lomuto, in media

Quicksort

Complessità

- Il calcolo di PARTITION ha complessità temporale $\Theta(n)$, con n la lunghezza del vettore di cui deve operare la partizione
- La complessità dell'intero Quicksort risulta quindi $T(n) = T(\frac{n}{a}) + T((1 - \frac{1}{a})n) + \Theta(n)$, dove il valore a dipende da quanto "bene" PARTITION ha suddiviso il vettore
- Caso pessimo: il vettore è diviso in porzioni lunghe $n - 1$ e 1
 - La ricorrenza diventa $T(n) = T(n - 1) + T(1) + \Theta(n)$
 - Si dimostra facilmente che è $\Theta(n^2)$
- Caso ottimo: il vettore è diviso in due porzioni lunghe $\frac{n}{2}$
 - La ricorrenza diventa $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
 - È la stessa del MergeSort, $\Theta(n \log(n))$
 - Una stima esatta ci dice che la costante nascosta dalla notazione Θ è 1,39: media solo il 39% più lento del caso ottimo

Un confronto tra ordinamenti per confronto

Algoritmo	Stabile?	$T(n)$ (caso pessimo)	$T(n)$ (caso ottimo)	$S(n)$
Insertion	✓	$\Theta(n^2)$	$\Theta(n)$	$O(1)$
Merge	✓	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n)$
Quick	×	$O(n^2)$	$\Omega(n \log(n))$	$O(1)$

- Sappiamo che non è possibile essere più veloci usando algoritmi di ordinamento *per confronto*
- C'è modo di fare meglio ordinando senza confrontare tra elementi?

Algoritmi non comparativi

Ordinare senza confrontare

- Il vincolo che abbiamo sulla complessità minima è legato al fatto che confrontiamo gli elementi da ordinare *tra loro*
- Nel caso in cui possiamo fare assunzioni sulla distribuzione o sul dominio degli elementi da ordinare, possiamo fare a meno dei confronti!
- Vediamo un esempio di algoritmo di ordinamento senza confronti il *counting sort*
 - Assunzione: il dominio degli elementi è *finito* e di dimensioni “ragionevoli” (dovremo rappresentarlo per esteso)
 - Intuizione: ordino calcolando l'istogramma delle frequenze e stampandone gli elementi in ordine

Counting Sort

Versione non stabile, k valore massimo degli el. di A

COUNTINGSORT(A)

```
1   $Is[0..k] \leftarrow 0$  // Nota: costo  $\Theta(k)$ 
2  for  $i \leftarrow 0$  to  $A.length$ 
3       $Is[A[i]] \leftarrow Is[A[i]] + 1$ 
4   $idxA \leftarrow 0$ 
5  for  $i \leftarrow 0$  to  $k$ 
6      while  $Is[i] > 0$ 
7           $A[idxA] \leftarrow i$ 
8           $idxA \leftarrow idxA + 1$ 
9           $Is[i] \leftarrow Is[i] - 1$ 
```

- La complessità temporale è dominata dal ciclo alle righe 5–8:
 $\mathcal{O}(n + k)$
- Se $k \gg n$ la complessità in pratica può essere molto alta

Counting Sort

Versione *stabile*, k valore massimo degli el. di A

COUNTINGSORT(A)

```
1   $B[0..A.length - 1] \leftarrow 0$ 
2   $Is[0..k] \leftarrow 0$  // Nota: costo  $\Theta(k)$ 
3  for  $i \leftarrow 0$  to  $A.length$ 
4       $Is[A[i]] \leftarrow Is[A[i]] + 1$ 
5   $s \leftarrow 0$ 
6  for  $i \leftarrow 0$  to  $k$ 
7       $s \leftarrow s + Is[i]$ 
8       $Is[i] \leftarrow s$ 
9  for  $i \leftarrow A.length - 1$  to 0
10      $idx \leftarrow Is[A[i]]$ 
11      $B[idx - 1] \leftarrow A[i]$ 
12      $Is[A[i]] \leftarrow Is[A[i]] - 1$ 
```


Counting Sort

Versione *stabile*: commento

- Il counting sort stabile parte con il calcolare il numero delle occorrenze di ogni elemento come quello classico
- A partire dall'istogramma delle frequenze I_s , lo trasforma nel vettore contenente il conteggio degli elementi con valori \leq di quello dell'indice del vettore
- Calcolato ciò, piazza un elemento calcolando la sua posizione come il valore corrente dell'informazione cumulativa contenuta in I_s
- L'informazione cumulativa è decrementata: effettivamente esiste un elemento in meno \leq all'indice del vettore