UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Appunti di Algoritmi e Strutture Dati

a.a. 2017/2018

Autore:
Timoty Granziero

Repository:
https://github.com/Vashy/ASD-Notes

3 marzo 2018

Indice

1 Lezione del 28/02/2018

1.1 Problem Solving

1. Formalizzazione del problema;
2. Sviluppo dell'**algoritmo** (focus del corso);
3. Implementazione in un programma (codice).

Algoritmo Sequenza di passi elementari che risolve il problema.

$$\text{Input} \rightarrow \textbf{Algoritmo} \rightarrow \text{Output}$$

Dato un problema, ci sono tanti algoritmi per risolverlo.

e.g. Ordinamento dei numeri di una `Rubrica`. L'idea è quella di trovare tutte le permutazioni di ogni numero.

$$30 \text{ numeri: } \textit{complessità } 30! \cong 2 \times 10^{32}\text{ns} \Rightarrow$$
$$3^{19} \text{ anni (con } ns = \text{nanosecondi)}$$

std::vector È un esempio nel `C++` delle ragioni per cui si studia questa materia. Nella documentazione della `STL`, sono riportati i seguenti:

- `Random access`: complessità O(1);
- `Insert`: complessità O(1) ammortizzato.

Il `random access` è l'accesso a un elemento casuale del `vector`. O(1) implica che l'accesso avviene in tempo costante (pari a 1).

Per `insert` si intende l'inserimento di un nuovo elemento in coda. Avviene in tempo O(1) ammortizzato: questo perchè ogni N inserimenti, è necessario un resize del vector e una copia di tutti gli elementi nel nuovo vettore (questa procedura è nascosta al programmatore).

1.2 Cosa analizzeremo nel corso

- Tempo di esecuzione;
- Spazio (memoria);
- Correttezza;
- Manutenibilità.

1.2.1 Approfondimento sul tempo di esecuzione T(n)

- *P Problems*: complessità polinomiale. L'algoritmo è trattabile
- *NP Complete*: problemi NP completi. **e.g**: Applicazione sugli algoritmi di sicurezzza. Si basano sull'assunzione che per essere risolti debbano essere considerate tutte le soluzioni possibili.
- *NP Problems*: problemi con complessità (ad esempio) esponenziale/-fattoriale. Assolutamente non trattabili.

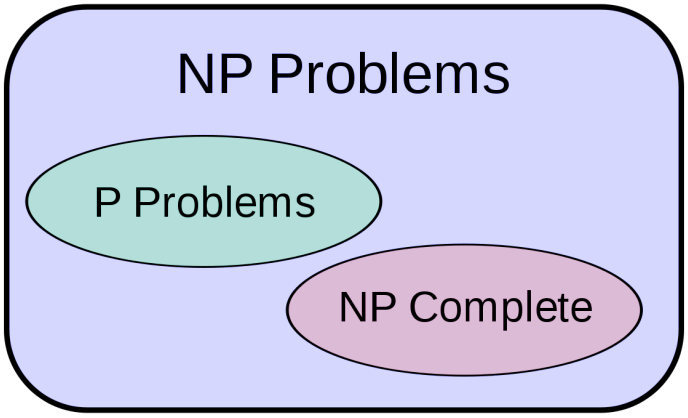

Figura 1: Complessità T(n).

1.3 Problema dell'ordinamento (sorting)

`Input`: sequenza di numeri

$$a_0 a_1 \dots a_n;$$

`Output`: permutazione

$$a'_0 a'_1 \dots a'_n$$

tale che

$$a'_0 \leq a'_1 \leq \dots \leq a'_n$$

Vedremo due algoritmi:

- `Insertion Sort`;
- `Merge Sort`.

1.4 Insertion Sort

È un algoritmo di *sorting incrementale*. Viene applicato naturalmente ad esempio quando si vogliono ordinare le carte nella propria mano in una partita a scala 40: si prende ogni carta a partire da sinistra, e la si posiziona in ordine crescente.

Astrazione Prendiamo ad esempio il seguente array:

5	2	8	4	7

Partiamo dal primo elemento: 5. È già ordinato con se stesso, quindi procediamo con il secondo elemento.
Confronto il numero 2 con l'elemento alla sua sinistra:

$2 \geq 5$? No, quindi lo inverto con l'elemento alla sua sinistra, come segue

2	5	8	4	7	Key:	8

La key analizzata è 8.

$8 \geq 5$? Sì, quindi è ordinato in modo corretto.

2	5	8	4	7	Key:	4

La key analizzata è 4.

$4 \geq 8$? No, quindi lo sposto a sinistra invertendolo con 8.
$4 \geq 5$? No, lo sposto a sinistra invertendolo con 5.
$4 \geq 2$? Sì, quindi è nella posizione corretta.

2	4	5	8	7	Key:	7

Key analizzata 7.

$7 \geq 8$? No, lo sposto a sinistra invertendolo con 8.
$7 \geq 5$? Sì, è nella posizione corretta.

Ottengo l'array ordinato:

2	4	5	7	8

Algorimo Passiamo ora all'implementazione dell'algoritmo, con uno pseudocodice similare a `Python`[1]

`Input`: $A[1, \ldots, n]$, $A.length$.

È noto che: $A[i] \leq key < A[i+1]$

Pseudocodice Segue lo pseudocodice dell'`Insertion Sort`.

INSERTION-SORT(A)
1 $n = A.length$
2 **for** $j = 2$ **to** n // il primo elemento è già ordinato
3 $key = A[j]$ // $A[1 \ldots j-1]$ ordinato
4 $i = j - 1$
5 **while** $i > 0$ and $A[i] > key$
6 $A[i+1] = A[i]$
7 $i = i - 1$
8 $A[i+1] = key$

Quando il `while` termina, ci sono due casi:

- `i = 0`: tutti gli elementi prima di `j` sono maggiori di `key`; `key` va al primo posto (1);
- `(i > 0) and (A[i]` $\leq$ `key)`: `A[i+1] = key`.

1.4.1 Invarianti e correttezza

for `A[1..j-1]` è ordinato e contiene gli elementi in `(1,j-1)` iniziali.

while `A[1..i]A[i+2..j]` ordinato e `A[i+2..j] > key`.

In uscita abbiamo:

- `j = n+1`;
- `A[1..n]` ordinato, come da invariante: vale `A[1..j-1]` ordinato, e `j` vale `n+1`.

[1] **ATTENZIONE**: verranno usati array con indici che partono da 1.

2 Lezione del 02/03/2018

2.1 Modello dei costi

Assunzione Tutte le istruzioni richiedono un tempo costante.

Rivediamo l'algoritmo:

INSERTION-SORT(A)

```
1  n = A.length
2  for j = 2 to n // il primo elemento è già ordinato
3      key = A[j] // A[1..j-1] ordinato
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

Diamo il nome c_0 alla chiamata del metodo, `InsertionSort(A)`; A ogni riga numerata, diamo il nome $c_1, c_2, .., c_8$ [1].

Vediamo il *costo* di ogni istruzione:

$\boldsymbol{c_0} \to 1$

$\boldsymbol{c_1} \to 1$

$\boldsymbol{c_2} \to n$

$\boldsymbol{c_3} \to (n-1)$

$\boldsymbol{c_4} \to (n-1)$

$\boldsymbol{c_5} \to \sum_{j=2}^{n} t_j + 1$

$\boldsymbol{c_6}, \boldsymbol{c_7} \to \sum_{j=2}^{n} t_j$

$\boldsymbol{c_8} \to (n-1)$

[1] (c_1 corrisponde alla riga 1, c_2 alla riga 2 e così via)

2.2 Complessità di IS

$$T^{IS}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n}(t_j + 1) + (c_6 + c_7)\sum_{j=2}^{n} t_j$$

$\boldsymbol{t_j}$ dipende, oltre che da n, dall'istanza dell'array che stiamo considerando. È chiaro che questo calcolo non da indicazioni chiare sull'effettiva complessità dell'algoritmo.

Andiamo ad analizzare i 3 possibili casi:

a) Caso migliore (2.2.1)

b) Caso peggiore (2.2.2)

c) Caso medio (2.2.3)

2.2.1 Caso migliore

$\rightarrow A$ ordinato $\Rightarrow t_j = 0 \ \forall j$

La **complessità** diventa:

$$T^{IS}_{min}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_5 + c_8)(n-1) = an + b \approx n$$

Ossia, si comporta come n. Il *caso migliore* **non** è interessante, visto che è improbabile si presenti.

2.2.2 Caso peggiore

$\rightarrow A$ ordinato in senso inverso $\Rightarrow \forall j \ t_j = j - 1$

La **complessità** diventa:

$$T^{IS}_{max}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} j + (c_6 + c_7)\sum_{j=2}^{n}(j-1)$$

Per valutare il costo di $\sum_{j=2}^{n} j$ e di $\sum_{j=2}^{n}(j-1)$, usiamo la **somma di Gauss**:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Otteniamo:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n}(j-1) = \sum_{i=1}^{n} n = \frac{(n-1)n}{2}$$

Per finire, ricalcoliamo $T^{IS}_{max}(n)$

$$T^{IS}_{max}(n) = a'n^2 + b'n + c' \approx n^2$$

2.2.3 Caso medio

Il caso medio è *difficile da calcolare*, e in una considerevole parte dei casi, coincide con il caso peggiore.

Comunque, l'idea è la seguente:

$$\frac{\sum_{\text{perm. di input}} T^{IS}(p)}{n!} \approx n^2 \qquad \text{posso pensare che } t_j \cong \frac{j-1}{2}$$

2.3 Divide et Impera

Un algoritmo di sorting *divide et impera* si può suddividere in 3 fasi:

divide divide il problema dato in sottoproblemi più piccoli;

impera risolve i sottoproblemi:

- ricorsivamente;
- la soluzione è nota (e.g. array con un elemento);

combina compone le soluzioni dei sottoproblemi in una soluzione del problema originale.

2.4 Merge Sort

Merge Sort[1] è un esempio di algoritmo divide et impera. Andiamo ad analizzarlo.

[1] Si consiglia uno sguardo all'algoritmo da altre fonti, poichè presentarlo graficamente in LaTeX, come è stato visto a lezione, è arduo.

Astrazione Consideriamo il seguente array A.

5	2	4	7	1	2	3	6

Lo divido a metà, ottenendo due parti separate.

5	2	4	7

1	2	3	6

Consideriamo il primo, ossia `A[1..4]` (A originale). Divido anche questo a metà.

5	2

4	7

Divido nuovamente a metà, ottenendo:

5

2

5 e 2 sono due blocchi già ordinati. Scelgo il minore tra i due e lo metto in prima posizione, mentre l'altro in seconda posizione, ottenendo un blocco composto da 2 e 5.

Riprendo con il blocco composto da 4 e 7. Lo divido in due blocchi da un elemento. Faccio lo stesso procedimento fatto per 2 e 5: metto in prima posizione 4 e in seconda posizione 7. La situazione è la seguente:

2	5

4	7

So che i blocchi ottenuti contengono elementi ordinati. Data questa assunzione, posso ragionare nel seguente modo: considero il primo elemento dei due blocchi (il 2 in questo caso) e lo metto in prima posizione. Ora considero il successivo elemento di quel blocco e lo stesso elemento del blocco che non è stato selezionato, e inserisco nell'array l'elemento minore. Continuo fino ad ottenere un blocco ordinato.

2	4	5	7

Faccio lo stesso ragionamento con la parte di array originale `A[5..8]`, ottenendo

2	4	5	7

1	2	3	6

A questo punto, i blocchi da 4 contengono elementi tra loro ordinati. Faccio lo stesso procedimento usato per comporli, per ottenere l'array originale ordinato. Considero:

- `A[1..4]`: indice $i = 1$ per scorrerlo;
- `A[5..8]`: indice $j = 1$ per scorrerlo;

Valuto `A[i]` e `A[j]`.

- Se `A[i]` $\leq$ `A[j]`, inserisco `A[i]` e incremento `i`.
- Altrimenti, inserisco `A[j]` e incremento `j`.
- Itero finchè entrambi gli indici non sono out of bounds.

Pseudocodice Segue lo pseudocodice del `Merge Sort`.

MERGE-SORT(A, p, r)
1 **if** $p < r$
2 $q = (p + r)/2$ // arrotondato per difetto
3 MERGE-SORT(A, p, q) // ordina `A[p..q]`
4 MERGE-SORT$(A, q + 1, r)$ // ordina `A[q+1..r]`
5 MERGE(A, p, q, r) // "Merge" dei due sotto-array

MERGE(A, p, q, r)
1 $n1 = q - p + 1$ // gli indici partono da 1
2 $n2 = r - q$
// **L** sotto-array sx, **R** sotto-array dx
3 **for** $i = 1$ **to** $n1$
4 $L[i] = A[p + i - 1]$
5 **for** $j = 1$ **to** $n2$
6 $R[j] = A[q + j]$
7 $L[n1 + 1] = R[n2 + 1] = \infty$
8 $i = j = 1$
9 **for** $k = p$ **to** r
10 **if** $L[i] \leq R[j]$
11 $A[k] = L[i]$
12 $i = i + 1$
13 **else** // `L[i]` $>$ `R[j]`
14 $A[k] = R[j]$
15 $j = j + 1$

2.4.1 Invarianti e correttezza

L e **R** contengono rispettivamente `A[p..q]` e `A[q+1..r]`. L'indice `k` scorre `A`. Il sotto-array `A[p..k-1]` è ordinato, e contiene `L[1..i-1]` e `R[1..j-1]`.

$$A[p\,..\,k-1] \leq L[i\,..\,n1], R[j\,..\,n2]$$
$$\Downarrow$$
$$A[p\,..\,k-1] = A[p\,..\,r+1-1] \implies A[p\,..\,r] \text{ ordinato}$$

Dimostrazione per induzione su r-p

$\Rightarrow$ Se $r - p == 0$ (oppure -1) abbiamo al più un elemento $\implies$ array già ordinato.

$\Rightarrow$ Se $r - p > 0$ per ipotesi induttiva:

- `Merge-sort(A,p,q)` ordina `A[p..q]`;
- `Merge-sort(A,q+1,r)` ordina `A[q+1..r]`;
  Per correttezza di `Merge()`, dopo la sua chiamata ottengo `A[p..r]` ordinato.