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

1 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.

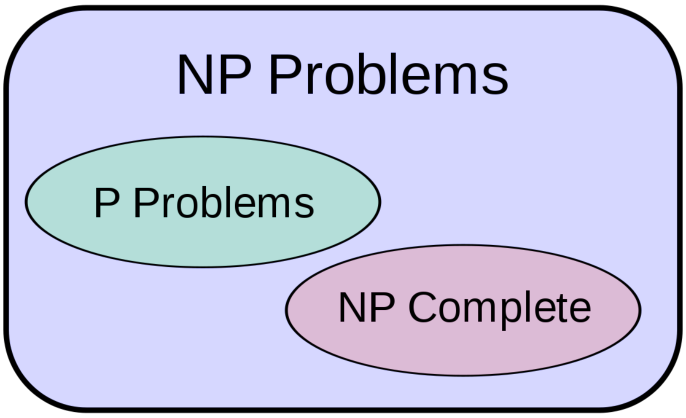

Figura 1: Complessità T(n).

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 l'algoritmo di sorting che viene fatto 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: $\boxed{8}$

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

2	5	8	4	7

Key: $\boxed{4}$

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: $\boxed{7}$

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`: $A[1, .., n]$, $A.length$.

$$\text{È noto che: } A[i] \leq key < A[i+1]$$

Pseudocodice Segue lo pseudocodice dell'`Insertion Sort`.

```
INSERTION-SORT(A)
n = A.length
for j = 2 to n // il primo elemento è già ordinato
    key = A[j] // A[1 .. j − 1] ordinato
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    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.