Lezione_02_DeA

1.3. Pseudocodice

Per semplicità e facilità di analisi, descriviamo un algoritmo utilizzando uno pseudocodice strutturato come segue:

> **Algoritmo** nome(parametri)
> **Input**: breve descrizione dell'istanza di input
> **Output**: breve descrizione della soluzione restituita in output
> Descrizione chiara dell'algoritmo tramite costrutti di linguaggi di programmazione e, se utile ai fini della chiarezza, anche tramite linguaggio naturale, dalla quale sia facilmente determinabile la sequenza di operazioni elementari eseguita per ogni dato input.

"Algoritmo nome (parametri)" è la firma (signature) dell'algoritmo; i parametri sono l'input.
Si può usare il linguaggio naturale per cose semplici, che sarebbero lunghe da scrivere con i costrutti del linguaggio di programmazione.

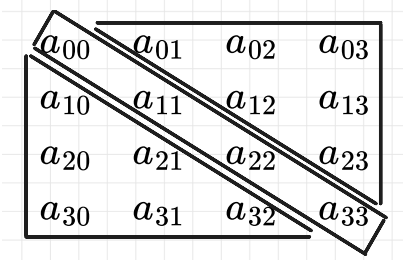
Usare *sempre* questa struttura per lo pseudocodice!

Per maggiori dettagli fare riferimento alla dispensa sulla Specifica di Algoritmi in Pseudocodice, disponibile sul Moodle del corso.

1.3.1. Esempio – Trasposta di una matrice $n \times n$ di interi

Problema computazionale: $\Pi \subseteq I \times S$
$I \equiv \{A:$ matrice $n \times n$ di interi$\}$
$S \equiv \{B:$ matrice $n \times n$ di interi$\}$
$\Pi \equiv \{(A,B) : A \in I, B \in S, B = A^t\}$

Algoritmo: idea per $n = 4$



Scambio ciascuna entry a_{ij} della matrice triangolare superiore (

$a_{ij} : i = 0, \ldots, n-2, j > i)$ con a_{ji} (l'omologa del triangolo inferiore)

Algoritmo `transpose(A)`
Input: matrice A $n \times n$ di interi a_{ij}, $0 \leq i, j \leq n$.
Output: matrice A^t.

```
for i <- 0 to n-2 do{
     for j <- i+1 to n-1 do{
          scambia a_ij con a_ji
     }
}
return A
```

1.4. Taglia di un'istanza

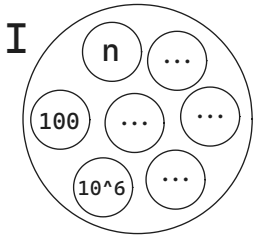
L'analisi di un algoritmo (ad esempio la determinazione della sua complessità) viene solitamente fatta *partizionando le istanze in gruppi in base alla loro taglia (size)*, in modo che le istanze di un gruppo siano tra loro *confrontabili*.

> La *taglia di un'istanza* è espressa da uno o più valori che ne caratterizzano la grandezza.

1.4.1. Esempi

- Se ho n elementi da ordinare con un `MergeSort`, la taglia è n. In questo caso fa riferimento alla dimensione fisica dell'istanza.

La taglia serve per dividere in gruppi omogenei da analizzare separatamente.



- Ordinamento di un array:
  Taglia n = numero di elementi dell'array
- Trasposta di una matrice:
  Prendendo una generica matrice $n \times m$ ho più modi di definire la taglia, in base al tipo di analisi che si vuole condurre:
  - Taglia $x = n * m$

- Taglia (n, m), con n e m rispettivamente il numero di righe e numero di colonne
- (Se $n = m$) Taglia $n =$ numero di righe/colonne

1.5. Struttura dati

Le strutture dati sono usate dagli algoritmi per organizzare e accedere in modo sistematico ai dati di input e ai dati generati durante l'esecuzione.

1.5.1. Struttura dati - definizione

> Una *struttura dati* è una collezione di oggetti corredata di *metodi* di accesso e/o modifica.

Vi sono due *livelli di astrazione*:

1. *Livello logico*: specifica l'organizzazione logica degli oggetti della collezione, e la relazione input-output di ciascun metodo (a questo livello si parla di *Abstract Data Type* o ADT)
2. *Livello fisico*: specifica il layout fisico dei dati e la realizzazione dei metodi tramite algoritmi.

1.5.1.1. Esempio

In Java a livello logico ho (gerarchia di) *interfacce*, mentre a livello fisico ho (gerarchia di) *classi*.
□

1.5.2. Esercizio

Siano A e B due array di n e m interi, rispettivamente. Scrivere un algoritmo in pseudocodice per calcolare il seguente valore:

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (A[i] \cdot B[j])$$

1.5.3. Esercizio

Sia A un array di n interi distinti, ordinato in senso crescente, e x un valore intero. Scrivere due algoritmi in pseudocodice che implementano la ricerca binaria per trovare x in A. Entrambi gli algoritmi restituiscono l'indice i tale che $A[i] = x$, se x appare in A, e -1 altrimenti. Il primo algoritmo deve essere iterativo e usare un solo ciclo, e il secondo algoritmo deve essere ricorsivo.

2. Analisi degli algoritmi

L'analisi di un algoritmo A mira a studiarne l'*efficienza* e l'*efficacia*. In particolare, essa può valutare la ***complessità*** di *tempo* e *spazio*, e la ***correttezza*** di *terminazione* (se termina o rimane in un loop) e della *soluzione del problema computazionale*.

Noi ci concentreremo sulla complessità di tempo e sulla correttezza della soluzione del problema computazionale.

2.1. Complessità in tempo

L'*obiettivo* è *stimare il tempo di esecuzione ("running time") di un algoritmo* al fine di valutarne l'efficienza e poterlo confrontare con altri algoritmi per lo stesso problema.

2.1.1. Requisiti per la complessità in tempo

La complessità in tempo deve:

1. Riguardare *tutti gli input*;
2. Permettere di *confrontare algoritmi* (senza necessariamente determinare il tempo di esecuzione esatto);
3. Essere *derivabile dallo pseudocodice*.

2.1.1.1. Esempio

Stimare sperimentalmente il tempo di esecuzione di un algoritmo (ad esempio in Java con `System.currentTimeMillis()`) è utile, ma non soddisfa i requisiti. **Perché?**

- Non si possono considerare tutti gli input (se infiniti);
- Richiede di implementare l'algoritmo con un programma e l'impatto dell'implementazione può influenzare il confronto tra algoritmi;
- La stima dipende dall'ambiente hardware/software.

2.1.1.2. Approccio che soddisfa i requisiti

- Analisi al *caso pessimo* (*worst-case*) in funzione della taglia dell'istanza (requisiti 1 e 2)
- Conteggio delle operazioni elementari nel modello RAM (requisiti 2 e 3)
- Analisi asintotica (per semplificare il conteggio)

> ⓘ **Osservazione**

Esiste anche l'analisi al caso medio (*average case*) e l'analisi probabilistica.

2.1.2. Complessità (in tempo) al caso pessimo - definizione

Sia A un algoritmo che risolve $\Pi \subseteq I \times S$.

> La complessità (in tempo) al caso pessimo di A è una funzione $t_A(n)$ definita come *il massimo numero di operazioni elementari che A esegue per risolvere un'istanza di taglia n*.

In altre parole, se chiamiamo $t_{A,i}$ il numero di operazioni eseguite da A per l'istanza i, abbiamo che

$$t_A(n) = max\{t_{A,i} : \text{ istanza } i \in I \text{ di taglia } n\}$$

La definizione rispetta i tre requisiti definiti sopra.

2.1.3. Difficoltà nel calcolo di $t_A(n)$

Determinare $t_A(n)$ per ogni n è arduo, se non impossibile, perché è *difficile identificare l'istanza peggiore di taglia n* e perché è *difficile contare il massimo numero di operazioni richieste per risolvere tale istanza* peggiore (il conteggio richiederebbe anche una specifica dettagliata del set di operazioni elementari del modello RAM).
Ma *non è necessario determinare esattamente $t_A(n)$*, infatti il tempo di esecuzione, che la complessità vuole stimare, dipende da tanti fattori che è impossibile quantificare in modo preciso, in più le diverse operazioni elementari del modello RAM possono avere impatto diverso sui tempi di esecuzione a seconda delle architetture.
Ci accontentiamo dei *limiti superiori* e i *limiti inferiori* a $t_A(n)$.

2.1.4. Esempio (`arrayMax`)

Algoritmo `arrayMax(A)`
Input: array $A[0 \div n-1]$ di $n \geq 1$ interi.
Output: max intero in A.

```
currMax<-A[0]
for i<-1 to n-1 do{
    if(currMax < A[i]) then currMax<-A[i];
```

```
}
return currMax
```

Taglia dell'istanza: n (ragionevole)

2.1.4.1. Stima di $t_{\text{arrayMax}}(n)$

È facile vedere che per una qualsiasi istanza di taglia n:

- al di fuori del ciclo for `arrayMax` esegue un numero costante (rispetto a n) di operazioni;
- in ciascuna delle $n-1$ iterazioni del ciclo for si esegue un numero costante di operazioni.
  Esistono allora quattro costanti $c_1, c_2, c_3, c_4 > 0$ tali che per ogni n valga $t_{\text{arrayMax}}(n) \leq c_1 n + c_2$ (cioè il *limite superiore*) e $t_{\text{arrayMax}}(n) \geq c_3 n + c_4$ (cioè il *limite inferiore*). Non è necessario stimare le quattro costanti per le stesse ragioni per cui non è necessario stimare esattamente la complessità.

2.2. Analisi asintotica

Si ricorre quindi all'analisi asintotica, ignorando i *fattori moltiplicativi costanti* (rispetto alla taglia dell'istanza) e i *termini additivi non dominanti*.

2.2.1. Esempio (`arrayMax`)

Riprendendo l'esempio, nel caso di `arrayMax` possiamo affermare che $t_{\text{arrayMax}}(n)$ è *al più* proporzionale a n (limite superiore) ed è anche *almeno* proporzionale a n (limite inferiore), ne consegue quindi che $t_{\text{arrayMax}}(n)$ è *proporzionale* a n (limite stretto).
□

> Per esprimere affermazione come quelle fatte sopra si usano gli **ordini di grandezza**: $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$.