Settimana 1

|------------------------------------------- Lezione 1 - 3/10/2023

Orario Martedi 14:30 – 16:00

Mercoledi 12:30 – 14:00

Problema: trovare il minimo di un array, limite minimo di n-1 confronti.

|------------------------------------------- Lezione 2 – 4/10/2023

Problema: trovare il minimo di un array (continua)

Esistono algoritmi che trovano il minimo in più di n-1 confronti.

**Concetto di complessità**

Obbiettivo: Confrontare algoritmi tra di loro per decidere quale sia il migliore caso per caso.

Gli algoritmi possono essere confrontati su vari criteri:

* Costo di esecuzione (memoria, cpu, …)
* Tempo di esecuzione
* Economico

Analizzeremo la **complessità legata al tempo di esecuzione**, ma la stessa teoria è applicabile alla memoria o altri criteri di analisi degli algoritmi. Si vuole scegliere l’algoritmo più veloce nel tempo di esecuzione.

Il tempo di esecuzione può cambiare al variare della macchina su cui l’algoritmo viene lanciato; quindi, è inutile misurare il tempo fisico di un algoritmo. Per determinare se un algoritmo è più veloce basta individuare il numero di istruzioni, soprattutto quelle che hanno un impatto significativo sul tempo di esecuzione.

La **complessità** è una funzione che prende in input la *dimensione di un problema* e fornisce in output il tempo di esecuzione di un algoritmo.

Ad esempio, per le matrici, la dimensione del problema è *righe* \* *colonne.*

Per sapere se stiamo misurando la dimensione di un problema nella maniera corretta lo si evince dai risultati a seconda dell’input.

Consideriamo un algoritmo un insieme di blocchi di istruzioni messi in sequenza.

C1(n)

C2(n)

C3(n)

Il nostro interesse è il tempo pessimo entro il quale il programma termina, quindi dobbiamo trovare l’input pessimo. Se abbiamo

IF COND

B1

ELSE

B2

La complessità di un blocco con condizione diventa:

In alcuni casi, la complessità dipende dalla condizione del problema (ad esempio la ricorsione).

Altro caso:

WHILE COND

B

La complessità diventa

Non sappiamo il numero esatto di iterazioni, ma sappiamo il limite superiore di volte che viene eseguito. Il numero che si ottiene è un limite superiore al numero di iterazioni fatte.

Esistono tanti metodi che ci portano a sovrastimare il numero di istruzioni che viene eseguito.

Il modo per capire se la previsione è accurata, si deve trovare un limite inferiore, quanto più vicini sono il limite inferiore e quello superiore, tanto più è precisa la previsione sulla complessità dell’algoritmo.

Qualunque stima facciamo, siamo obbligati a fare approssimazioni, per eccesso o per difetto. In questi casi è necessario avere un limite superiore al caso pessimo ed un limite inferiore al caso pessimo.

Non serve calcolare il tempo medio, perché servirebbe una conoscenza probabilistica dell’input.

Esercizio: ricerca del minimo elemento di un array

n <- LENGTH[A] // 1 istruzione

m <- A[1] // 2 istruzioni

FOR i <- 1 TO n: // 4 \* (n – 1) + 1 + 2\*(n-1) istruzioni

IF A[i] < m // 4 istruzioni (tutto il blocco if)

m <- a[i] // 2 istruzioni, accesso ed assegnamento

RET m // 1 istruzione

La formula della complessità dell’algoritmo diventa quindi:

Alla fine, la complessità è una formula del tipo

Avendo due algoritmi con complessità

Alg. 1:

Alg. 2:

Conviene sempre scegliere l’algoritmo 1, questo perché se , nell’Alg. 1 sto moltiplicando la complessità per 1,000, mentre nell’Alg. 2 la complessità è moltiplicata per 1,000,000.

A noi, quindi, interessa sapere qual è il **grado della funzione risultante**.

Esempio: prodotto di due matrici

MULT(A, B):

n <- ROWS[A]

m <- COLS[A]

l <- COLS[B]

FOR i <- 1 TO n: // costo cicl. n\*m\*l\*a

FOR j <- 1 TO l: // costo cicl. m\*l\*a

C[i,j] <- 0

FOR k <- 1 TO m // costo cicl. Int. m\*a~~+b~~

C[i,j] <- C[i,j] + A[i,j]\*B[k,j]

RET C

Quindi si ottiene che la complessità è lineare in *n, l* ed *m*, con una costante *a*, che non ci interessa (ma potrebbe essere stimata).

Settimana 2

|------------------------------------------ Lezione 3 – 10/10/2023

Delle funzioni di complessità ci interessa principalmente il loro ordine di grandezza, ad esempio è importante sapere che al raddoppio della dimensione del problema se il tempo di completamento raddoppia oppure quadruplica.

In alcuni casi il problema potrebbe avere complessità diverse sulla base di eventuali condizioni che si verificano (ad esempio rami if o rami else). Quello che si deve considerare in questi casi è il caso peggiore.

La ricerca di un elemento costa ‘n’, ma esiste un caso in cui costa ‘1’, ovvero quando l’elemento che cerco è il primo. (Il caso pessimo è quando l’elemento cercato non c’è).

**Definizione del concetto di ordine di grandezza di una funzione O GRANDE**

(Notazione: **definitivamente** := ‘da un certo punto in po’)

Esempio:

Vedi su appunti cartacei

Teorema

Dimostrazione

**Definizione di o piccolo**

Teorema

Definizione di theta di g

Notazione

Sia A un algoritmo.

Affermare che signfica che tale che A termina sempre entro tempo g.

Affermare che significa che esiste uno schema di input t.c. su quello schema A richiede almeno g operazioni con .

Se combaciano allora diciamo che .

Sia P un problema

Dire che significa che che risolve P, con .

Dire che significa che che risolve P, con .

Se combaciano diciamo che .

EQUAZIONE DI RICORRENZA

Calcolo del fattoriale di un numero

FATT(n)

IF n = 0

RET 1

ELSE

RET n \* FATT(n-1)

Eq. di ricorrenza:

T(n) =