Algoritmi e strutture dati

L04 - 02/10/2024

Notazioni asintotiche

$f, g : \mathbb{N} \to \mathbb{R}^+$

- **Limitazione superiore**
  $f(n)$ è $\mathcal{O}$-grande di $g(n) \quad \to f(n) = \mathcal{O}(g(n))$
  se $\exists\ c > 0,\ n_0 \in \mathbb{N} \mid \forall n > n_0 :\ f(n) \leq c \cdot g(n)$
- **Limitazione inferiore**
  $f(n)$ è Ω-grande di $g(n) \quad \to f(n) = \Omega(g(n))$
  se $\exists\ c > 0,\ n_0 \in \mathbb{N} \mid \forall n > n_0 : f(n) \geq c \cdot g(n)$
- **Stesso ordine di grandezza**
  $f(n)$ è Θ-grande di $g(n) \quad \to f(n) = \Theta(g(n))$
  se $\exists\ c, d > 0,\ n_0 \in \mathbb{N} \mid \forall n > n_0 : c \cdot g(n) \leq f(n) \leq d \cdot g(n)$
  $f(x) = \Theta(g(n)) \iff f(x) = O(g(n)) \land f(x) = \Omega(g(n))$

Proprietà importanti

- $f(n) = \mathcal{O}(g(n)) \Rightarrow \forall k > 0 \quad k \cdot f(n) = \mathcal{O}(g(n))$ *(le costanti non sono perciò rilevanti)*
- $\left.\begin{array}{l} f_1(n) = \mathcal{O}(g_1(n)) \\ f_2(n) = \mathcal{O}(g_2(n)) \end{array}\right\} \begin{array}{l} f_1(n) + f_2(n) = \mathcal{O}(g_1(n) + g_2(n)) \\ f_1(n)\ \cdot\ f_2(n) = \mathcal{O}(g_1(n)\ \cdot\ g_2(n)) \end{array}$

proprietà analoghe per Ω e Θ

Non è vero che $f_1(n) - f_2(n) = \mathcal{O}(g_1(n) - g_2(n))$

Algoritmi precedenti

- Prodotti iterati: $T(x, y) = 3y + 3 = \Theta(y)$ crescita lineare
- Potenza ricorsiva (due varianti):

  $$\begin{array}{ll} T(x,y) \leq 6\log_2 y + 8 & \quad T(x,y) = \mathcal{O}(\log_2 y) \\ T(x,y) \leq 7\log_2 y + 10 & \quad T(x,y) = \mathcal{O}(\log_2 y) \end{array}$$

  È dimostrabile che in entrambi i casi $T(x, y) = \Theta(\log_2 y)$, il che significa che al posto dei minori uguali nelle soluzioni delle due equazioni di ricorrenza è possibile sostituirvici gli uguali (l'algoritmo cresce con il $\log_2 y$)

Osservazione: per via della proprietà del cambio di base dei logaritmi, nei simboli di Landau non è importante specificarne la base poiché è possibile cambiarla dividendo di fatto per una costante (costante che è trascurabile).

[...] numero n in una base b > 1 quante cifre utilizza?

L05 - 04/10/2024

Macchina ad accesso diretto (RAM)

[...]

```
ALGORITMO minimo (sequenza s) -> elemento
    min <- primo elemento di s
    WHILE non hai ispezionato tutta s DO
        x <- prossimo elemento di s
        IF x è minore di min THEN min <- x
    RETURN min
```

[...]

Valutazione del costo computazionale

Criterio di costo uniforme

Tempo: ogni istruzione elementare utilizza un'unità di tempo *indipendentemente* dalla grandezza degli operandi.
Spazio: ogni variabile elementare utilizza un'unità di spazio *indipendentemente* dal valore contenuto.

- È un criterio ragionevole quando i valori trattati dall'algoritmo sono di grandezza limitata, ma diventa inadeguato quando si manipolano quantità arbitrariamente grandi: in tal caso è necessario tenere conto della loro grandezza, e in particolare della lunghezza delle loro rappresentazioni.

Algoritmo potenza riflessiva

```
ALGORITMO xx(intero x) -> intero
    p <- 1
    FOR i <- 1 TO x DO
        p <- p * x
    RETURN p
```

- Ci sono $\Theta(x)$ assegnamenti e prodotti
- Ci sono $\Theta(x)$ confronti e incrementi
- Poichè ciascuna delle operazioni principali viene eseguita x volte, e ognuna di queste operazioni costa $O(1)$ per via del criterio di costo uniforme, il tempo totale richiesto dall'algoritmo è $\Theta(x)$.

```go
packege main
import . "fmt"

var n int

func xx(x int) int { //algoritmo codificato in golang
    var p int = 1
    for i := 1; i <= x; i++ {
        p = p * x
    }
    return p
}

func main() {
    Print("Inserire un intero positivo: ")
    Scan(&n)
    Println(n, "^", n, "è uguale a", xx(n))
}
```

```
$> go run xx.go
Inserire un intero positivo: 7
7 ^ 7 è uguale a 823543

$> go run xx.go
Inserire un intero positivo: 15
15 ^ 15 è uguale a 437893890380859375

$> go run xx.go
Inserire un intero positivo: 20
20 ^ 20 è uguale a -2101438300051996672
```

- Con numero troppo grandi, che si parli di interi in 32 bit o in 64 bit, il risultato della potenza va in overflow. Non è quindi più possibile rappresentare questi dati con strutture primitive: il costo delle operazioni elementari non può più essere 1, e di conseguenza non è più possibile utilizzare il criterio di costo uniforme. Questo perché è necessario manipolare più bit per rappresentare numeri più grandi.

Criterio di costo logaritmico

Tempo: il tempo di calcolo di ciascuna operazione è *proporzionale alla lunghezza* dei valori coinvolti.
Spazio: la lunghezza della rappresentazione del dato.

- Negli interi è il numero di bit, e nelle stringhe è il numero di caratteri.

Il costo di un'operazione elementare è proporzionale alla lunghezza della rappresentazione degli operandi. Quando perciò si va incontro a valori troppo grandi non si può più usare il criterio di costo uniforme.

- La rappresentazione di un intero è composta da un numero di bit pari a $\log x$. Di conseguenza un'operazione su un numero di quantità grandi avrà un costo logaritmico, che dipende dal numero di cifre che lo compongono.

Applicato all'algoritmo della potenza riflessiva

Tempo:

- Dopo l'i-esima iterazione p contiene x^i
- All'i-esima iterazione: $p \leftarrow p * x$ e di conseguenza $x^i \leftarrow x^{i-1} * x$
  - Il costo del prodotto è: $\log x^{i-1} + \log x = (i-1)\log x + \log x = i \log x$
  - Il costo dell'assegnamento a p è: #bit da copiare, ossia $i \log x$

  L'i-esima iterazione costa quindi $\Theta(i \log x)$
- Il tempo totale utilizzando il criterio del costo logaritmico è:

$$\sum_{i=1}^{x} \Theta(i \log x) = \Theta(\sum_{i=1}^{x} i \log x) = \underset{\substack{\downarrow \\ \text{formula di Gauss per la somma dei primi } n \text{ numeri naturali}}}{\Theta(\frac{x(x+1)}{2} \log x)} = \Theta(x^2 \log x)$$

Spazio:

- $\Theta(\log x^x) = \Theta(x \log x)$ numero di bit per memorizzare x^x, ovvero l'output

Complessità di algoritmi (tempo)

Si supponga di avere un algoritmo $\mathcal{A}$ e un'istanza I.

- **Tempo in funzione dell'input**:
  - $\text{tempo}(I) = \text{tempo impiegato da } \mathcal{A} \text{ su input } I$
- **Tempo in funzione della lunghezza dell'input $T : \mathbb{N} \rightarrow \mathbb{N}$:**
  - Tempo massimo utilizzato su input di lunghezza n (stima nel caso peggiore):
    $T(n) = \max\{\text{tempo}(I) \mid |I| = n\}$
- **Tempo medio $T_{\text{avg}} : \mathbb{N} \rightarrow \mathbb{N}$:**
  - Media dei tempi utilizzati su input di lunghezza n, pesata rispetto alle probabilità:

$$T_{\text{avg}}(n) = \sum_{|I|=n} \text{Prob}(I) \cdot \text{ tempo}(I)$$

Tempo polinomiale rispetto a tempo esponenziale

- **Algoritmo polinomiale**: algoritmi che lavorano in un tempo limitato da un polinomio (rispetto alla lunghezza dell'input). Sono considerati ragionevoli o praticabili.
- **Algoritmo esponenziale**: algoritmi che utilizzano un tempo esponenziale e sono considerati impraticabili.

Esempio di un computer che esegue 1 miliardo di operazioni al secondo (ogni operazione impiega 1 nsec):

	$n = 10$	$n = 20$	$n = 50$	$n = 60$	$n = 100$
n	10 10 nsec	20 20 nsec	50 50 nsec	60 60 nsec	100 100 nsec
n^2	100 100 nsec	40 400 nsec	2500 2.5 μsec	3600 3.6 μsec	10.000 10 μsec
n^3	1000 1 μsec	8000 8 μsec	125.000 125 μsec	216.000 216 μsec	1.000.000 1 msec
2^n	$\approx$ 1000 1 μsec	$\approx$ 1.000.000 1 msec	$\approx 10^{15}$ 10^6sec $\approx$ 11.5 giorni	$\approx 10^{18}$ 32 anni	$\approx 10^{30}$ $3 \cdot 10^{10}$ millenni

- Anche utilizzando computer fino a 1000 volte più veloci, gli algoritmi che utilizzano tempo esponenziale non ne traggono alcun beneficio significativo.

Complessità di problemi

Si supponga di avere un problema $\mathcal{P}$ e si consideri la risorsa tempo. Quanto tempo si impiega per risolvere $\mathcal{P}$?

- **Limitazione superiore**: trovo un algoritmo $\mathcal{A}$ che risolve $\mathcal{P}$ in tempo $T(n)$
  $\Rightarrow$ tempo $T(n)$ è sufficiente per risolvere $\mathcal{P}$ ($\mathcal{P}$ si risolve in tempo $O(T(n))$)
- **Limitazione inferiore**: dimostro che ogni algoritmo che risolve $\mathcal{P}$ deve utilizzare almeno un tempo $T'(n)$
  $\Rightarrow$ tempo $T'(n)$ è necessario per risolvere $\mathcal{P}$ ($\mathcal{P}$ si risolve in tempo $\Omega(T'(n))$)

$\mathcal{P}$ è risolubile in tempo $O(T(n))$ significa che esiste un algoritmo che risolve $\mathcal{P}$ e utilizza tempo $O(T(n))$
$\mathcal{P}$ richiede tempo $\Omega(T(n))$ significa che ogni algoritmo che risolve $\mathcal{P}$ utilizza tempo $\Omega(T(n))$

- Se riesco a dimostrare che ogni algoritmo richiede almeno tempo $T(n)$ e trovo poi un algoritmo che si risolve in tempo $T(n)$ allora si è trovato l'algoritmo migliore possibile.

Definizioni analoghe possono essere usate per altre risorse come lo spazio.

L06 - 07/10/2024

Strutture indicizzate

Array

Un array è una collezione di elementi dello stesso tipo, ciascuno dei quali è accessibile in base alla posizione.

Caratteristiche tipiche:

- Memorizzato in una porzione contigua di memoria
- Acesso mediante indice (posizione)
- Tempo di accesso indipendente dalla posizione del dato (l'importante è sapere l'indirizzo di memoria)

Limitazione:

- È una struttura statica, ossia non è possibile aggiungere nuove posizioni

Nota: nei singoli linguaggi di programmazione (es. Go) alcune caratteristiche degli array e delle relative variabili possono essere differenti.

Variabili array

- Sono riferimenti, ossia puntatori, all'array

```
A = [5, 3, 2, 4, 7, 6]
B <- A
B[1] <- 0
stampa(A[1]) //0
```

```
PROCEDURA azzera(Array X[0...n-1])
    FOR i <- 0 TO n-1 DO
        x[i] <- 0

ALGORITMO ...
    ...
    azzera(A)
    ...
```

- X è un puntatore all'array passato nella procedura, e quindi è soltanto una variabile che contiene un indirizzo di memoria

Ricerca in un array

Input: array A, elemento x
Output: indice i tale che $A[\ i\ ] = x$; -1 se A non contiene x

Ricerca sequenziale

```
ALGORITMO ricercaSequenziale(Array A[0..n-1], elemento x) -> indice
    i <- 0
    WHILE i < n AND A[i] != x DO //lazy evaluation
        i <- i + 1
    IF i = n THEN RETURN -1
             ELSE RETURN i
```

Nota: questo algoritmo funziona per via della *lazy evalutaion* o *short-circuit behaviour*: negli operatori AND e OR, se l'espressione è già determinata dal valore dell'operando sinistro, allora l'operando destro non viene valutato.

- Di conseguenza questi due operatori non sono effettivamente commutativi

Tempo: $\Theta(n)$ dove $n = \text{len(A)}$

Ricerca in array ordinato

Ricerca binaria o dicotomica ricorsiva

```
FUNZIONE ricercaRicorsiva(Array A, indice sx, indice dx, elemento x) -> indice
    IF dx <= sx THEN RETURN -1
    ELSE
    |   m <- (sx + dx)/2
    |   IF x = A[m] THEN RETURN m
    |   ELSE IF x < A[m] THEN
    |       RETURN ricercaRicorsiva(A, sx, m, x)
    |   ELSE
    |       RETURN ricercaRicorsiva(A, m+1, dx, x)

ALGORITMO ricercaBinaria(Array A[0..n-1], elemento x) -> indice
    RETURN ricercaRicorsiva(A, 0, n, x)
```

- indice sx incluso, indice dx escluso

$$
\begin{gathered}
A \to \{1, 5, 7, 12, 16, 18, 20, 22\} x = 12 \\
\text{ricercaRic(A, 0, 8, 12)}: \\
\begin{bmatrix} sx & 0 \\ dx & 8 \\ m & 4 \end{bmatrix} \Rightarrow \text{ricercaRic(A, 0, 4, 12)} \\
\text{ricercaRic(A, 0, 4, 12)}: \\
\begin{bmatrix} sx & 0 \\ dx & 4 \\ m & 2 \end{bmatrix} \Rightarrow \text{ricercaRic(A, 3, 4, 12)} \\
\text{ricercaRic(A, 3, 4, 12)}: \\
\begin{bmatrix} sx & 3 \\ dx & 4 \\ m & 3 \end{bmatrix} \\
\text{End of search: } x = 12 \text{ found at index 3.}
\end{gathered}
$$

[...]