Appunti di Algoritmi e Strutture Dati

Algoritmi e Strutture Dati (prof. ???) - CdL Informatica Unimib - 23/24

Federico Zotti

07 Mar 2024

Indice

1 Introduzione

Un'**algoritmo** è una sequenza di istruzioni **elementari** (devono essere comprese e eseguite dall'esecutore) che permettono di risolvere un problema computazionale (ovvero per ogni possibile input produce l'output corretto).

Per definire un **problema** è necessario specificare:

- Il tipo del parametro in input
- Il tipo del risultato in output
- Il legame tra input e output

Un'**istanza** di un problema si ottiene specificando uno dei possibili valori in input specifico per il problema.

1.1 Algoritmo: Definizione dell'ordinamento di un vettore

Sort:

- Input: `Array Int` (`Dim` n) $\rightarrow A = \langle a_1, a_2, \ldots, a_n \rangle$
- Output: `Array Int` (`Dim` n) $\rightarrow A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$

A' è una permutazione di A, tale che $a'_1 \leq a'_{i+1} \quad \forall i \,.\, 1 \leq i \leq n-1$.

1.2 Scelta di un algoritmo

L'algoritmo migliore è quello che utilizza il minor numero di risorse.

Le risorse sono:

- Il tempo di esecuzione
- Lo spazio (memoria) utilizzato

1.2.1 Tempo di esecuzione

Per calcolare il tempo utilizziamo una funzione $T(n)$. n rappresenta la quantità di dati in input.

- $T_p(n)$ rappresenta il caso peggiore
- $T_n(n)$ rappresenta il caso "medio" (non è la media dei due)
- $T_m(n)$ rappresenta il caso migliore

1.2.1.1 Esempio

- Algoritmo 1: $T(n) = 100000 \cdot n$
- Algoritmo 2: $T(n) = 10 \cdot n^3$
- Algoritmo 3: $T(n) = 1 \cdot 2^n$

In questo caso il migliore dipende dal grado di n, dunque l'algoritmo 1 risulta quello più veloce. Per numeri di n molto piccoli invece è meglio calcolare caso per caso il tempo. Nel caso ci siano più n, si considera quello con il grado maggiore.

$$T(n) = 7n^3 + 2n + 10000 \approx n^3$$

1.3 Algoritmo: Ricerca sequenziale

- V: vettore di interi
- k: intero da cercare nel vettore
- p: posizione nel vettore

Ricerca sequenziale

```
function SEARCH(V[ ], k)
    p ← 1
    while (V[p] ≠ k) ∧ (p ≤ LENGTH(V) ) do
        p ← p + 1
    end while
    if p > LENGTH(V) then
        return −1
    else
        return p
    end if
end function
```


Analisi del tempo di esecuzione:

- Caso peggiore: $k \neq V[\,] \Rightarrow T(n) = 3 + 2 \cdot n + 1 \approx n$
- Caso migliore: $k = V[1] \Rightarrow T(n) = 4 \approx c$
- Caso medio: $k = V[\frac{n}{2}] \Rightarrow 3 + 2\frac{n}{2}(\pm 1) \approx n$

Se V è ordinato ci si può fermare appena trova un numero più grande di k.

Ricerca sequenziale

```
function SEARCH(V[ ], k)
    p ← 1
    while (V[p] < k) ∧ (p ≤ LENGTH(V) ) do
        p ← p + 1
    end while
    if V[p] ≠ k ∨ p > LENGTH(V) then
        return −1
    else
        return p
    end if
end function
```

Analisi del tempo di esecuzione:

- Caso migliore: $V[p] \geq k \Rightarrow 4 \approx c$
- Caso peggiore: $k > V[p] \Rightarrow 3 + 2n \approx n$
- Caso medio: $k = V\left[\frac{n}{2}\right] \Rightarrow 3 + 2\frac{n}{2} \cdot \frac{1}{2}$

Per avere un'ottimizzazione significativa si può sfruttare il fatto che il vettore è ordinato per implementare una semplice ricerca binaria (spezzare il vettore e guardare solo una metà).

Ricerca binaria

function SEARCH($V[\,], k$)
    $sx \leftarrow 1$
    $dx \leftarrow$ LENGTH(V)
    $p \leftarrow dx + sx$ div 2
    while $(V[p] \neq k) \wedge (sx < dx)$ **do**
        if $k > V[p]$ **then**
            $sx \leftarrow p + 1$
        else
            $dx \leftarrow p - 1$
        end if
        $p \leftarrow (dx + sx)$ div 2
    end while
    if $V[p] = k$ **then**
        return p
    else
        return -1
    end if
end function

Analisi del tempo di esecuzione:

- Caso migliore: $V\left[\frac{n}{2}\right] = k \Rightarrow t_m(n) = 6 \approx c$
- Caso peggiore: $k \notin V \Rightarrow T_p(n) = 5 + 4 \cdot (\log_2 n) + 1 \approx \log_2 n$
- Caso medio: $T(n) = \frac{T_p(n)}{2} \approx \log_2 n$

