ASD Homework 1

Francesco Iannaccone e Matteo Conti

October 2021

Le matricole saranno allegate assieme al prossimo Homework in quanto, essendoci laureati alla seduta di Settembre della settimana scorsa, dobbiamo ancora completare la procedura di immatricolazione alla magistrale.

1 Homework Esercitativi

1.1 Notazione asintotica

- $f(n) = O(f(n)^2)$

  L'affermazione è falsa nel caso di $f(n) = \frac{1}{n}$, infatti $\frac{1}{n} \geq \frac{1}{n^2}, \forall\ n \geq 1$.

  E' invece vera per $f(n) = n$, infatti $n \leq n^2, \forall\ n$.

- $f(n) + O(f(n)) = \Theta(f(n))$

  L'affermazione è vera: $f(n)+O(f(n)) = \Omega(f(n))$ è banalmente dimostrato poiché $f(n) \leq f(n) + O(f(n))$.

  Vale in oltre che: $f(n) + O(f(n)) \leq f(n) + cf(n) \leq c_2 f(n)$.

  Si ha dunque $c_1 f(n) \leq f(n) + O(f(n)) \leq c_2 f(n) \Rightarrow f(n) + O(f(n)) = \Theta(f(n))$.

- $f(n) = \Omega(g(n))$ e $f(n) = o(g(n))$

  L'affermazione è falsa: se infatti $f(n) = o(g(n)) \Rightarrow \forall\ c > 0, f(n) < c{*}g(n)$, la cosa è in evidente contraddizione con l'affermazione $f(n) = \Omega(g(n))$, ovvero che $\exists c : f(n) \geq c * g(n)$

1.2 Complessità

Si può facilmente verificare che $n + a = \Theta(n)$, risulta infatti che $\frac{1}{2}n \leq n + a \leq 2n, \forall n \geq 2a$.

Sfruttando la monotonia dell'elevamento a potenza: $c_1 n \leq n + a \leq c_2 n \Rightarrow c_1^b n^b \leq (n + a)^b \leq c_2^b n^b \Rightarrow (n + a)^b = \Theta(n^b)$.

1.3 Complessità

- $2^{n+1} = O(2^n)$ è vero, risulta infatti che $2^{n+1} = 2\ 2^n = O(2^n)$.
- $2^{2n} = O(2^n)$ è falso, risulta infatti che $2^{2n} = 4^n$ e che $2^n = o(4^n)$.

1.4 Ricorrenze

- $T(n) = 2T(n/3) + nlg(n)$

  Risulta che: $nlg(n) = \Omega(n^{log_3(2)+\epsilon}) \approx \Omega(n^{0.63+\epsilon})$, basta infatti considerare $\epsilon < 1 - log_3(2)$ affinché l'andamento polinomiale di $nlg(n)$ superi quello di $n^{0.63+\epsilon}$, inoltre $\frac{2}{3}\ n\ lg(n/3) \leq c * nlg(n)$ per una data $c = \frac{2}{3} < 1$ ed n sufficientemente grande. Applicando il terzo caso del teorema dell'esperto risulta che $T(n) = \Theta(nlg(n))$.

- $T(n) = 3T(n/5) + lg^2(n)$

  Risulta che: $lg^2(n) = O(n^{log_5(3)-\epsilon})$, basta infatti considerare $\epsilon < log_5(3)$ e l'andamento polinomiale domina su quello logaritmico. Applicando il primo caso del teorema dell'esperto risulta dunque che $T(n) = \Theta(n^{log_5(3)})$.

1.5 Ricorrenze

$T(n) = T(n/3) + T(2n/3) + cn$

I rami più corti dell'albero computazionale hanno lunghezza $log_3(n) + 1$ mentre i più lunghi hanno lunghezza $log_{\frac{3}{2}}(n) + 1$, inoltre ad ogni livello ci sono sempre cn operazioni poiché $T(n)$ è suddiviso in $\mathrm{T}(\frac{2n}{3})$ e $\mathrm{T}(\frac{n}{3})$. Andando a considerare solo i primi $log_3(n)+1$ livelli dell'albero si ottiene un limite inferiore di $T(n)$:

$T(n) \geq \sum_{i=0}^{log_3(n)} cn = cn(log_3(n) + 1) = \Theta(nlog_3(n)) \Rightarrow T(n) = \Omega(nlog_3(n))$

2 Homework di verifica

2.1 Inversioni

- L'array $(2, 3, 8, 6, 1)$ presenta 5 inversioni:

  $(2, 1), (3, 1), (8, 1), (6, 1), (8, 6)$

- L'array di n elementi con più inversioni è quello ordinato in ordine decrescente: $(n, n-1, \ldots, 1)$, che presenta $\sum_{i=0}^{n-1} n - i - 1 = \frac{n(n-1)}{2}$ inversioni.

- Il numero di swap effettuati nell'insertion sort corrisponde al numero delle inversioni nell'array.

- L'algoritmo proposto per il calcolo delle inversioni è il seguente:

```
def merge_inversioni(v, inizio , q, fine){
    n1 <- q-inizio+1
    n2 <- fine-inizio
    for i <- 1 to n1
        L[i] <- v[inizio+i-1]
    for j <-1 to n2
        R[j] <- v[q+j]
    L[n1+1] <- +∞
    R[n2+1] <- +∞
    i <- 1
    j <- 1
    N <- 0
    for k <- inizio to fine
        if L[i]>R[j]
            v[k] <- R[j]
            j <- j+1
            N <- N + (n1 - i + 1)
        else
            v[k] <- L[i]
            i <- i+1
    return N
}
```

```
def conta_inversioni(v, inizio, fine){
    N <- 0
    if inizio >= fine
        then return N
    q <- floor((inizio+fine)/2)
    N1 <- conta_inversioni(v, inizio, q)
    N2 <- conta_inversioni(v, q+1, inizio)
    N3 <- merge_inversioni(v, inizio, q, fine)
    return (N1 + N2 + N3)
}
```

L'idea fondamentale su cui si basa è quella di andare ad applicare il *merge sort* e contare ogni volta che, nella fase di merge, un elemento del vettore di sinistra è più grande di uno di quello di destra: inoltre, poichè i due vettori sono ordinati, possiamo calcolare il numero esatto di inversioni che corrisponde a $n_1 - i + 1$. Se L[i] > R[J] significa che R[j] è minore, oltre che di L[i], anche di tutti gli elementi del vettore di sinistra successivi, andiamo quindi ad aggiungere al numero di inversioni il numero di elementi successivi ad L[i] più uno.

L'algoritmo ha come effetto "collaterale" quello di ordinare il vettore, se si vuole evitare si può effettuare una copia prima di chiamare conta_inversioni.

2.2 Unimodal search

L'algoritmo proposto per la soluzione del problema è il seguente:

```
def cerca_max(v, inizio, fine){
    if inizio > fine
        return -1
    else if inizio == fine
        return v[inizio]
    else
        q <- floor((inizio+fine)/2)
        if q != inizio
            if v[q] < v[q-1]
                return cerca_max(v, inizio, q-1);
```

```
        if q != fine
            if v[q] < v[q+1]
                return cerca_max(v, q+1, fine);
        return v[q];
}
```

Ad ogni ricorsione il problema è trasformato in un sottoproblema con dimensione dell'ingresso dimezzato: il passo base ha complessità $\Theta(1)$, allo stesso tempo dopo il passo di ricorsione le operazioni condotte hanno complessità $\Theta(1)$. L'algoritmo nel complesso ha complessità $O(lg(n))$ in quanto per un nodo a livello i, la dimensione del sottoproblema è $\frac{n}{2^i}$ e quindi la dimensione 1 si raggiunge per $\frac{n}{2^i} = 1$. Il livello i in cui la dimensione è 1 è $i = lg(n)$ e, essendo il costo costante a ciascun livello dell'albero, si può dire che la complessità è $O(lg(n))$.

$$T(n) = \begin{cases} \Theta(1) & \text{se n=1} \\ T(n/2) + \Theta(1) & \text{altrimenti} \end{cases}$$

2.3 Ricorrenze

$T(n) = T(n/2) + 2^n$

Poiché $2^n = \Omega(n^{log_2(1)+\epsilon}) = \Omega(n^\epsilon)$ e inoltre la condizione $2^{n/2} \leq c * 2^n$ è verificata per almeno un $c < 1$, basta infatti considerare $c \geq 1/2^{n/2}$, risulta che $T(n) = \Theta(2^n)$ per il terzo caso del teorema dell'esperto.

Volendo fare un'ulteriore verifica del risultato facendo uso del metodo dell'albero di ricorrenza, questo risulta avere una altezza $h = lg(n) + 1$ e ogni livello ha un solo nodo ($a = 1$) con costo $2^{n/(2^i)}$. Risulta dunque che $T(n) = \sum_{i=0}^{lg(n)} 2^{n/2^i} = 2^n + 2^{n/2} + 2^{n/4} + \cdots = \Theta(2^n)$.