Appunti lezione del 18 Marzo 2020

Esercizio 1 - Calcolo di complessità (Appello del 18 Febbraio 2020).

Il seguende codice Java implementa un algoritmo che determina la lunghezza del più lungo sub-array non decrescente di una data porzione di un array di `int` (da indice `i` a indice `j`), ed è basato sul paradigma del divide-et-impera.

```java
private static int[] sottoseq(int a[], int i, int j) {
    // assume che i <= j e che a[i] e a[j] esistono
    int ret[] = new int[4];
    /* ret[0] il miglior risultato
     * ret[1] inizio del miglior risultato
     * ret[2] lungh. prefisso
     * ret[3] lungh. suffisso
     */
    if(i == j) {
        ret[0] = 1;
        ret[1] = i;
        ret[2] = 1;
        ret[3] = 1;
        return ret;
    }
    int m = (i+j)/2;
    int s1[] = sottoseq(a, i, m); // ****
    int s2[] = sottoseq(a, m+1, j); // ****
    boolean ord1 = s1[0] == m-i+1; // I subarray ordinato
    boolean ord2 = s2[0] == j-m; // II subarray ordinato
    boolean cont = a[m] <= a[m+1]; // suffisso1 continua con prefisso2
    if(cont && (s1[3]+s2[2] > Math.max(s1[0], s2[0]))) {
        ret[0] = s1[3]+s2[2];
        ret[1] = m+1-s1[3];
    } else if(s1[0] > s2[0]) {
        ret[0] = s1[0];
        ret[1] = s1[1];
    } else {
        ret[0] = s2[0];
        ret[1] = s2[1];
    }
    ret[2] = s1[2]; if(ord1 && cont) ret[2] += s2[2];
    ret[3] = s2[3]; if(ord2 && cont) ret[3] += s1[3];
    return ret;
}
```

Si risponda ai seguenti quesiti:

1. Determinare il costo temporale asintotico dell'algoritmo `sottoseq`, con particolare riferimento al caso in cui venga invocato con `sottoseq(a, 0, a.length-1)`.
2. Determinare il costo spaziale asintotico dell'algoritmo `sottoseq`, con particolare riferimento al caso in cui venga invocato con `sottoseq(a, 0, a.length-1)`.

Risposta al quesito 1.

Sebbene il codice possa sembrare complicato, ai fini del calcolo della complessità possiamo ragionare come segue: i) ogni invocazione di `sottoseq` contiene un numero *costante* di istruzioni e ii) nel peggiore dei casi abbiamo 2 invocazioni ricorsive di `sottoseq`, ciascuna su un array di al più $n/2$ elementi, se n era il numero di elementi presenti nell'array in input. Se $C(n)$ è il costo nel caso peggiore per array di n elementi, possiamo quindi scrivere:

$$C(n) \leq c + 2C(n/2).$$

D'altra parte, se $n = 1$ abbiamo $C(1) \leq c$. Qui, come sopra, c è una costante abbastanza grande, che essenzialmente dà il costo di caso peggiore di un'iterazione di `sottoseq` *senza considerare l'eventuale costo delle chiamate ricorsive*.

Svolgendo la ricorrenza mostrata sopra otteniamo:

$$\begin{aligned} C(n) &\leq c + 2C(n/2) \leq c + 2\left(c + 2C(n/4)\right) = c + 2c + 4C(n/4) \leq c + 2c + 4\left(c + 2C(n/8)\right) \\ &= c + 2c + 4c + 8C(n/8) \leq \ldots \leq c\sum_{j=1}^{i-1} 2^j + 2^i C(n/2^i) \end{aligned}$$

dopo i iterazioni. Chiaramente, dobbiamo fermarci quando i è tale che $n/2^i \leq 1$. Ciò ovviamente accade certamente per $i = \lceil \log_2 n \rceil$. Quindi abbiamo:

$$C(n) \leq c\sum_{j=1}^{\lceil \log_2 n \rceil - 1} 2^j + 2^{\lceil \log_2 n \rceil} C(1) \leq c\frac{2^{\lceil \log_2 n \rceil} - 1}{2 - 1} - c + 2 \cdot nC(1) < 2cn + 2cn = O(n).$$

Nel secondo passaggio abbiamo applicato la forma chiusa della somma geometrica e ci siamo ricordati che $\lceil \log_2 n \rceil \leq \log_2 n + 1$ e che
$\sum_{j=0}^{k} a^k = 1 + \sum_{j=1}^{k} a^k = \frac{a^{k+1}-1}{a-1} \rightarrow \sum_{j=1}^{k} a^k = \frac{a^{k+1}-1}{a-1} - 1$

Si noti che allo stesso risultato si può pervenire in modo meno accurato ma pur sempre rigoroso dando un limite al numero di chiamate ricorsive (al più $2n$, vedi sopra) e poi argomentando che il costo di ciascuna invocazione (al netto di eventuali chiamate ricorsive al suo interno) è al più c dove c è una costante opportuna. Per il numero $T(n)$ di invocazioni ricorsive per array di dimensione n nel caso peggiore abbiamo:

$$T(n) \leq 1 + 2T(n/2),$$

inclusa l'invocazione corrente.

Risposta al quesito 2.

Il costo in termini di spazio dell'algoritmo è anch'esso lineare. Il motivo principale è che, in ciascuna chiamata ricorsiva, il passaggio dei parametri fa sì che non venga allocata nuova memoria per passare l'array alla funzione/metodo chiamato. In particolare, ciò che viene passato è un riferimento all'array stesso, del quale vi è una sola copia in memoria. Tutta la restante memoria allocata (variabili locali, copie locali dei parametri ecc.) è costante per ciascuna invocazione.

Esercizio 2 - Calcolo di complessità

Si considerino i seguenti metodi Java:

```
static int funct1(int x) {
    if(x <= 1) return x;
    return funct1(funct1(x/2)) + 1;
}

static long funct2(int y) {
    if(y <= 1) return y;
    return funct2(y-1)*funct1(y);
}
```

Sviluppare, argomentando adeguatamente quanto segue:

- Determinare il costo temporale asintotico dell'algoritmo descritto da `funct1(int)` in funzione della dimensione dell'input $z_1 = |x|$ (z_1 è la dimensione dell'input e x è l'input).
- Determinare il costo temporale asintotico dell'algoritmo descritto da `funct2(int)` in funzione della dimensione dell'input $z_2 = |y|$ (z_2 è la dimensione dell'input e y è l'input).

Svolgimento. Denotiamo con $f_1(x)$ ed $f_2(y)$ le funzioni calcolate da `funct1` e `funct2`. Siano poi $C_1(|x|)$ e $C_2(|y|)$ i costi computazionali di f_1 e f_2 se implementate come nel codice sopra. Prima di tutto osserviamo che f_1 è definita ricorsivamente come

$$f_1(x) = \begin{cases} x, & x \leq 1 \\ f_1(f_1(x/2)) + 1, & x > 1 \end{cases}$$

Esempio:

$$\begin{aligned} f_1(5) = f_1(f_1(5/2) + 1) + 1 = f_1(f_1(2) + 1) + 1 = f_1(f_1(f_1(2/2) + 1) + 1) + 1 \\ = f_1(f_1(f_1(1) + 1) + 1) + 1 = f_1(f_1(2) + 1) + 1 = f_1(1 + 1) + 1 = f_1(2) + 1 \\ = (f_1(1) + 1) + 1 = 1 + 1 + 1 = 3 \end{aligned}$$

Si noti incidentalmente che $x/2$ qui denota la divisione intera con troncamento. Con un po' di riflessione (e calcolando $f_1(x)$) per i primi valori (ad esempio i primi 4-5) ci si può rendere conto che il valore $f_1(x)$ non può che essere limitatato superiormente da una una qualche costante (intera).

Dimostriamo ora che $f_1(x) \leq 3$, per ogni x. L'affermazione è certamente vera per $x = 0, 1$. Dimostriamo per induzione che essa è vera per $x \geq 2$. Supponiamo dunque che $f_1(i) \leq 3$ per $i = 0, 1, \dots, x - 1$ e mostriamo che $f_1(x) \leq 3$. Abbiamo:

$$f_1(x) = f_1(f_1(x/2)) + 1 \leq \max\{f_1(0), f_1(1), f_1(2), f_1(3)\} + 1 \leq 3.$$

La prima disuquaglianza è vera per ipotesi induttiva ed è facile verificare direttamente che $f_1(0) = 0, f_1(1) = 1, f_1(2) = 2, f_1(3) = 2$. Ciò implica l'ultima disuguaglianza. Si osservi anche che, per $x \in \{0, 1, 2, 3\}$ la complessità del calcolo di $f(x)$ è costante, come si può verificare direttamente. Quest'ultima osservazione è usata nel paragrafo che segue.

Come ulteriore passo stimiamo il numero di invocazioni ricorsive di `funct1`. Quest'ultimo si stima dalla definizione stessa di $f_1(x)$. In particolare, $C_1(x)$ si compone di due parti che si sommano: i) il costo di `funct1(x)` una volta che il valore `funct1(x/2)` è stato calcolato e ii) il costo di `funct1(x/2)`, per definizione pari a $C_1(x/2)$. Il primo costo è limitato da una costante c come osservato sopra, quindi abbiamo:

$$C_1(x) \leq c + C_1(x/2) = \cdots \leq c \lceil \log_2 x \rceil + c = O(\log_2 x) + c,$$

dove il risultato si ottiene iterando la disuguaglianza finché l'argomento di C_1 diviene ≤ 1, cosa che accade dopo $O(\log_2 x)$ iterazioni. La presenza della costante c nell'ultima espressione è dovuta al fatto che comunque si paga un costo costante se x è 0 o 1. Infine, occorre notare che $|x| = \Theta(\log_2 x)$ se l'intero x è rappresentato in modo non artificialmente ridondante. Ciò implica $C_1(x) = O(z_1)$ + c nel nostro caso. Il costo è dunque *lineare* nella dimensione dell'input.

Per quanto riguarda `funct2`, notiamo che per $C_2(y)$ possiamo scrivere:

$$C_2(y) \leq \begin{cases} c, & y \leq 1 \\ C_2(y-1) + C_1(y) + c, & y > 1 \end{cases}$$

Per quanto riguarda la scelta di c, essa corrisponde al massimo tra il costo per il calcolo di `funct1(x)` e quello di `funct2(y)` quando $x, y \leq 1$. Sostituendo nell'equazione di ricorrenza che definisce $C_2(y)$ il limite superiore calcolato per $C_1(y)$ abbiamo:

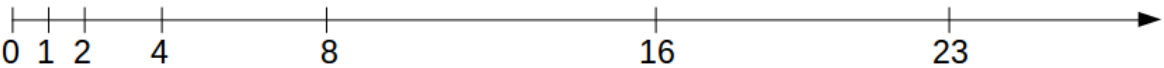
$$\begin{aligned} C_2(y) &\leq C_2(y-1) + c\lceil \log_2 y \rceil + 2c \leq C_2(y-2) + c\lceil \log(y-1) \rceil + 2c + c\lceil \log_2 y \rceil + 2c \\ &\leq \cdots \leq C_2(1) + 2cy + c\sum_{i=2}^{y} \lceil \log_2 i \rceil \leq c + 2cy + c(y-1)\lceil \log_2 y \rceil = O(y \log_2 y). \end{aligned}$$

Osservando come fatto in precedenza che $z_2 = |y| = \Theta(\log_2 y)$, abbiamo $C_2(y) = O(2^{z_2} z_2)$, quindi il costo è esponenziale rispetto alla dimensione di y.

Esercizio 3 - Ricorsione

Si descriva un algoritmo per trovare la parte intera di $\log_2 n$ (con n un numero naturale) usando *soltanto addizioni/sottrazioni e divisioni intere*. Calcolare la complessità dell'algoritmo proposto.

Risposta. Chiediamoci innanzi tutto quale sia la parte intera del logaritmo. Consideriamo l'esempio del numero 23. La parte intera di $\log_2 23$ è 4. Come possiamo visualizzare tale valore? Si osservi la figura sottostante:



Consideriamo gli intervalli che rappresentano le potenze di 2 più piccole di 23, non includendo l'intervallo $[0, 1]$. Il numero di tali intervalli corrisponde al logaritmo in base 2 di 23. Per caratterizzare tale numero in modo algoritmico possiamo fare la seguente, ulteriore osservazione: se dividiamo iterativamente 23 per il valore 2, eseguendo ogni volta la divisione con troncamento, il numero di divisioni eseguite *prima* di ottenere un numero minore o uguale di 1 è esattamente pari al numero di intervalli che ci interessano. Il motivo è che se considero un numero x che cade nell'intervallo $[2^i, 2^{i+1})$, per $i > 0$ intero, allora $\lfloor x/2 \rfloor$ cade nell'intervallo $[2^{i-1}, 2^i)$ (la prova è molto semplice ed è lasciata allo studente). **Suggerimento:** se $x \in [2^i, 2^{i+1})$ allora $x = 2^i + a$, per $0 \leq a < 2^i$.

Osservando che il logaritmo di 1 è 0, le osservazioni precedenti si traducono immediatamente nell'algoritmo seguente (nel quale si assume $n > 0$ e intero):

```
int_log(int n) {
    if (n == 1)
        return 0;
    return 1 + int_log(n/2) // L'operatore '/' è la divisione intera con
troncamento
}
```

Non sorprende che la complessità dell'algoritmo sia $O(\log_2 n)$. Sia infatti $C(n)$ il suo costo. Se $C(1) \leq c$ possiamo immediatamente scrivere:

$$C(n) \leq c + C(n/2)$$

Come al solito la costante c è abbastanza grande da tenere sia conto del costo $C(1)$ che dei costi delle operazioni diverse dalla chiamata ricorsiva nella generica invocazione della funzione.

Va infine notato che il costo dell'algoritmo, come ormai dovrebbe essere chiaro, è logaritmico nel *valore* dell'argomento, ma non nella dimensione (numero di bit) dell'input. Per rappresentare il valore n abbiamo infatti bisogno di al più $\log_2 n$ bit. Quindi se z è la dimensione dell'input, abbiamo $z = O(\log_2 n)$, per cui il costo dell'algoritmo è *lineare* rispetto a z (in particolare, $\Theta(z)$).

Esercizio 4 - Heap

4.1. Si consideri un heap *minimale* inizialmente vuoto e si supponga di avere una sequenza di n inserimenti consecutivi con chiavi *crescenti*. Dare il costo complessivo della sequenza degli n inserimenti nel caso peggiore.

4.2. Rispondere alla stessa domanda nel caso di una sequenza di n inserimenti consecutivi con chiavi *decrescenti*.

Risposta. Per fare un esempio, nel primo caso inseriamo la sequenza di chiavi (indichiamo soltanto le chiavi) 4, 6, 23, 50, 201, ... Ogni inserimento ha un costo $O(1)$. Per vederlo è sufficiente riguardarsi la procedura upheap e capire cosa accadrebbe in un caso del genere.

Nel secondo, con considerazioni analoghe ma simmetriche possiam concludere che il costo complessivo sarà $\Omega(n \log n)$.