

# Algoritmi e Principi dell'Informatica

Tema d'esame del 14 Giugno 2021

## 1 Informatica teorica

### Esercizio 1 (8 punti + bonus)

Si consideri il linguaggio  $L_1 = \neg L_2$ , dove  $L_2 = (ab)^+$ .

- a) Si definisca  $L_1$  nella logica MFO, se possibile, altrimenti se ne motivi l'impossibilità.
- b) **Bonus.** Con riferimento alle proprietà dei linguaggi *star-free*, si chiarisca se è possibile fornire un'espressione insiemistica per  $L_1$  facendo uso del simbolo dell'insieme vuoto ( $\emptyset$ ), degli insiemi di una singola lettera ( $\{a\}$  e  $\{b\}$ ), degli operatori di unione ( $\cup$ ), intersezione ( $\cap$ ), complemento ( $\neg$ ) e concatenamento ( $\cdot$ ), senza usare gli operatori di Kleene ( $*$  e  $+$ ).  
Suggerimento: si noti che  $\neg\emptyset = \{a, b\}^*$ .

### Soluzione

- a) Si può definire  $L_1$  in MFO mediante la formula seguente:

$$\begin{aligned} & \neg( \\ & \exists x(x = 0 \wedge a(x)) && \wedge \\ & \forall x(a(x) \rightarrow \exists y(\text{succ}(x, y) \wedge b(y))) && \wedge \\ & \forall x((b(x) \wedge \neg \text{last}(x)) \rightarrow \exists y(\text{succ}(x, y) \wedge a(y))) && \wedge \\ & \exists x(\text{last}(x) \wedge b(x)) \\ & ) \end{aligned}$$

- b) Un'espressione per  $L_1$  che non usi gli operatori di Kleene è la seguente:

$$\begin{aligned} & (\neg\emptyset \cdot \{a\} \cdot \{a\} \cdot \neg\emptyset) \quad (\text{parole che hanno due } a \text{ consecutive}) \\ & \cup \\ & (\neg\emptyset \cdot \{b\} \cdot \{b\} \cdot \neg\emptyset) \quad (\text{che hanno due } b \text{ consecutive}) \\ & \cup \\ & \neg(\{a\} \cdot \neg\emptyset) \quad (\text{parole che non iniziano con } a) \\ & \cup \\ & \neg(\neg\emptyset \cdot \{b\}) \quad (\text{parole che non finiscono con } b) \end{aligned}$$

### Esercizio 2 (8 punti, solo a) e b) per studenti con riduzione della prova)

Nel corso di *Prova finale di API* si richiede di sviluppare in C una funzione  $f$  da  $\mathbb{N}$  a  $\mathbb{N}$ . Un generatore di test, fornito agli studenti, enumera gli input per  $f$  e i corrispondenti output attesi.

- È decidibile il problema di stabilire se la codifica di  $f$  fatta da un generico studente è corretta rispetto ad ogni possibile caso di test fornito dal generatore?
- Ada e Pasquale si sono consultati prima di scrivere le proprie implementazioni. È decidibile il problema di stabilire se le loro codifiche di  $f$  sono equivalenti?
- È semidecidibile il problema del punto a)?<sup>†</sup>

### Soluzione

- No per Rice (la correttezza è una proprietà non banale della funzione calcolata: esiste almeno una funzione corretta, ed almeno una funzione non corretta).
- Sì (domanda chiusa).
- No: come noto, la correttezza non è nemmeno semidecidibile. In particolare, testando uno dopo l'altro i vari casi di test (con tecnica diagonale, mediante dovetailing) posso eventualmente trovare, se c'è, una discrepanza tra l'output predetto dal generatore e quello ottenuto con la codifica di  $f$ ; è quindi semidecidibile la presenza di una discrepanza, ed essendo indecidibile il problema della correttezza (cioè assenza di discrepanze), allora la correttezza non può essere nemmeno semidecidibile.

## 2 Algoritmi e strutture dati

### Esercizio 3 (8 punti + bonus)

Descrivere in modo rigoroso (o in pseudocodice) un algoritmo che riceve in ingresso una sequenza  $S$  di numeri interi e la riordina nel modo seguente: il primo numero è il più grande di  $S$ , il secondo è il più piccolo di  $S$ , il terzo è il secondo più grande di  $S$ , il quarto è il secondo più piccolo di  $S$ , e così via. Per esempio, data in ingresso la sequenza  $S = [4, 8, 23, 10, 17, 0, 3]$ , l'algoritmo la deve riordinare nella seguente maniera:  $[23, 0, 17, 3, 10, 4, 8]$ . Si fornisca la complessità asintotica dell'algoritmo scritto. L'algoritmo descritto deve avere complessità temporale minima; verrà attribuito un punteggio **bonus** a fronte di una dimostrazione della minimalità della complessità temporale dell'algoritmo proposto.

### Soluzione

Un modo semplice di realizzare questo algoritmo consiste nell'ordinare l'array in ordine crescente, con un algoritmo a complessità ottima  $O(n \log(n))$ , ad esempio Merge-Sort. Ottenuto l'array ordinato se ne costruisce uno nuovo con gli elementi ordinati dall'inizio se l'indice è pari e dalla fine se è dispari (complessità  $O(n)$ ). La complessità totale è pertanto  $O(n \log(n))$ .

```
1 int *oscillante(int *array, int n) {
2     mergesort(array, n);
3     int *result = (int *) malloc(sizeof(int) * n);
4     for (int i=0; i<n; i++) {
```

---

<sup>†</sup>Gli studenti con la riduzione della prova non devono affrontare questo punto.

```

5         result[i] = (i%2 != 0) ? array[i/2] : array[n-1-i/2];
6     }
7     return result;
8 }

```

È possibile dimostrare che la soluzione fornita ha complessità ottima, osservando che, se, per assurdo, fosse disponibile un algoritmo più veloce di  $\mathcal{O}(n \log(n))$  che ordina un array dato secondo il criterio indicato, questo algoritmo consentirebbe anche di ordinare l'array stesso in un ordine crescente o decrescente. Detto infatti  $\mathcal{A}(v)$  l'algoritmo che ordina secondo l'ordine descritto nella consegna e indicata con  $T_{\mathcal{A}}(n)$  la sua complessità temporale in funzione della lunghezza di  $v$  possiamo costruire una procedura che ordina  $v$  in ordine decrescente invocando  $\mathcal{A}(v)$  su  $v$  e poi, con una scansione lineare stampando prima gli elementi pari di  $v$  dall'inizio alla fine, poi i dispari dalla fine all'inizio. Il costo di quanto fatto è quello di una scansione lineare  $\Theta(n)$  più  $T_{\mathcal{A}}(n)$ . Sapendo che il limite inferiore per la complessità dell'ordinamento per confronto è  $\mathcal{O}(n \log(n))$ , abbiamo che la complessità per la proposta di  $\mathcal{A}(n)$  è ottima.

#### Esercizio 4 (8 punti, solo punti a e b per studenti con riduzione della prova)

Risolvere le seguenti ricorrenze:

a)  $T(n) = 3T(\frac{n}{2}) + n$ ; b)  $T(n) = 16T(\frac{n}{4}) + n!$ ; c)  $T(n) = \frac{1}{2}T(\frac{n}{2}) + n^{-1}$ ;† d)  $T(n) = 4T(\frac{n}{2}) + n^2$ .†

#### Soluzione

a) Si applica il Master Theorem (caso 1):  $T(n) = \Theta(n^{\log_2 3})$ .

b) Si applica il Master Theorem (caso 3):  $T(n) = \Theta(n!)$ .

c) Non si può applicare il Master Theorem perché  $a < 1$ . Iniziamo a sviluppare qualche passo della ricorrenza:  $T(n) = \frac{1}{2}T(\frac{n}{2}) + n^{-1} = \frac{1}{4}T(\frac{n}{4}) + \frac{1}{2}(\frac{n}{2})^{-1} + n^{-1} = \frac{1}{8}T(\frac{n}{8}) + \frac{1}{4}(\frac{n}{4})^{-1} + \frac{1}{2}(\frac{n}{2})^{-1} + n^{-1} = \frac{1}{8}T(\frac{n}{8}) + \frac{1}{n} + \frac{1}{n} + \frac{1}{n} = \dots$ . Questo ci porta a ipotizzare  $T(n) = \Theta(\frac{\log n}{n})$ .

Verifichiamo quindi con il metodo di sostituzione che  $T(n) \leq c \frac{\log n}{n}$  per un'opportuna costante

$c$ . Per ipotesi induttiva abbiamo allora che  $T(\frac{n}{2}) \leq c \frac{\log \frac{n}{2}}{\frac{n}{2}}$ . Sostituendo in  $T(n)$  abbiamo

$T(n) \leq \frac{1}{2}c \frac{\log \frac{n}{2}}{\frac{n}{2}} + n^{-1} = c \frac{\log n}{n} - c \frac{\log 2}{n} + \frac{1}{n} = c \frac{\log n}{n} + \frac{1-c \log 2}{n}$ . Per completare la verifica occorre

quindi che  $c \frac{\log n}{n} + \frac{1-c \log 2}{n} \leq c \frac{\log n}{n}$ , il che avviene quando  $1 - c \log 2 < 0$ , ovvero  $c > \frac{1}{\log 2}$ .

Un'analoga verifica sussiste per  $T(n) \geq d \frac{\log n}{n}$  per un'opportuna costante  $d$ .

d) Si applica il Master Theorem (caso 2):  $T(n) = \Theta(n^2 \log n)$ .