1. Esercizi di programmazione

Esercizio 1.1 – Cerca duplicati

Dato un vettore di n interi, scrivere un algoritmo che restituisca vero se un elemento compare due volte. Determinare una limitazione inferiore e una superiore alla complessità di questo problema.

Esercizio 1.2 – Cerca la coppia

Dato un array $A[1 \dots n]$ di interi e un intero v, scrivere un algoritmo che determini se esistono due elementi in A la cui somma sia esattamente v.

Esercizio 1.3 – Cerca la coppia porta-sfiga

Dato un array $A[1 \dots n]$ di interi e un intero v, scrivere un algoritmo che determini se esistono due elementi in A la cui somma sia esattamente 17.

2. Esercizi di comprensione notazione

Esercizio 2.1 – No limits!

Si considerino le funzioni $f(n) = n$ e $g(n) = n^{1+\sin n}$. Si dimostri che le due funzioni non sono confrontabili in ordine di grandezza, cioè che non vale né che $f(n)$ è $O(g(n))$, né che $f(n)$ è $\Omega(g(n))$.

Esercizio 2.2. – Stima di fattoriale

Dimostrare che $\log n! = \Theta(n \log n)$.

Esercizio 2.3 – Cicli FOR!

Si dimostri, per induzione, che $\sum_{i=1}^{n} i^h$ è $O(n^{h+1})$.

3. Ordinamento funzioni

Esercizio 3.1

Ordinare le seguenti funzioni in accordo alla loro complessità asintotica. Si scriva $f(n) < g(n)$ se $O(f(n)) \subset O(g(n))$. Si scriva $f(n) = g(n)$ se $O(f(n)) = O(g(n))$, ovvero se $f(n) = \Theta(g(n))$.

$$
\begin{aligned}
f_1(n) &= 2^{n+2} \\
f_2(n) &= \log^2 n \\
f_3(n) &= (\log_n(\sqrt{n})^2 n) + 1/n^2 \\
f_4(n) &= 3n^{0.5} \\
f_5(n) &= 16^{n/4} \\
f_6(n) &= 2\sqrt{n} + 4n^{1/4} + 8n^{1/8} + 16n^{1/16} \\
f_7(n) &= \sqrt{(\log n)(\log n)} \\
f_8(n) &= \frac{n^3}{(n+1)(n+3)} \\
f_9(n) &= 2^n
\end{aligned}
$$

4. Limiti superiori e inferiori

Esercizio 4.1

Trovare un limite asintotico superiore e un limite asintotico inferiore alla seguente ricorrenza, facendo uso del metodo di sostituzione:

$$T(n) = 2T(n/8) + 2T(n/4) + n$$

Esercizio 4.2

Si supponga di scrivere una variante di MergeSort, chiamata MergeSortK che, invece di suddividere l'array da ordinare in 2 parti (e ordinarle separatamente), lo suddivide in K parti, le ordina ognuna riapplicando MergeSortK, e le riunifica usando un'opportuna variante MergeK di Merge (la quale, naturalmente, fa la fusione su K sottoarray invece di 2). Come cambia, se cambia, la complessità temporale di MergeSortK rispetto a quella di MergeSort?

Esercizio 5.1 – Samarcanda

Nel gioco di Samarcanda, ogni giocatore è figlio di una nobile famiglia della Serenissima, il cui compito è di partire da Venezia con una certa dotazione di denari, arrivare nelle ricche città orientali, acquistare le merci preziose al prezzo più conveniente e tornare alla propria città per rivenderle.

Dato un vettore P di n interi in cui $P[i]$ è il prezzo di una certa merce al giorno i, trovare la coppia di giornate (x, y) con $x < y$ per cui risulta massimo il valore $P[y] - P[x]$. Calcolare la complessità e dimostrare la correttezza. È possibile risolvere il problema in $O(n)$.

Spoiler alert!

3. Ordinamento funzioni

Le funzioni da ordinare:

$$
\begin{aligned}
f_1(n) &= 2^{n+2} = 4 \cdot 2^n = \Theta(2^n) \\
f_2(n) &= \log^2 n = \Theta(\log^2 n) \\
f_3(n) &= (\log_n(\sqrt{n})^2 n) + 1/n^2 = (\log_n n^2) + 1/n^2 = 2 + 1/n^2 = \Theta(1) \\
f_4(n) &= 3n^{0.5} = \Theta(n^{1/2}) \\
f_5(n) &= 16^{n/4} = (2^4)^{n/4} = 2^{4n/4} = \Theta(2^n) \\
f_6(n) &= 2\sqrt{n} + 4n^{1/4} + 8n^{1/8} + 16n^{1/16} = \Theta(n^{1/2}) \\
f_7(n) &= \sqrt{(\log n)(\log n)} = \Theta(\log n) \\
f_8(n) &= \frac{n^3}{(n+1)(n+3)} = \Theta(n) \\
f_9(n) &= 2^n = \Theta(2^n)
\end{aligned}
$$

Una volta stabilito l'ordine Θ delle funzioni, è abbastanza semplice stabilire l'ordine corretto:

$$f_3 < f_7 < f_2 < f_4 = f_6 < f_8 < f_1 = f_5 = f_9$$

MergeSortK

La funzione di ricorrenza per MergeSortK è la seguente:

$$T(n) = \begin{cases} k(T(n/k)) + O(kn) & n > 1 \\ 1 & n = 1 \end{cases}$$

Un modo per implementare MergeK consiste nel trovare il minimo dei k valori, presenti, operazione che ha complessità $O(k)$. Ripetendo l'operazione n volte, la complessità di MergeK è pari a $O(kn)$. Calcolando $\alpha = \log_k k = 1$ e confrontandolo con n^1, è possibile vedere che il costo di MergeSortK è $O(kn \log n)$. Per valori costanti di k, questo corrisponde a $O(n \log n)$.