Esercizi

Esercizio 0
Progettare un algoritmo per ordinare **in loco** un array *a* di *n* interi, il cui valore può essere solo 0 o 1. L'algoritmo deve richiedere tempo lineare nel caso pessimo e può solo scambiare elementi. In particolare, non può usare contatori per mantenere il numero di elementi di un certo valore.

Esercizio 1
Progettare un algoritmo efficiente di tipo divide et impera per determinare se un array non ordinato di *n* interi, contenente solo 0 e 1, contiene più 0 di 1. Analizzare la complessità dell'algoritmo trovato.

Esercizio 2
Progettare un algoritmo di tipo *divide-et-impera* per calcolare a^n co*n* $O(\log n)$ moltiplicazioni.
[*Suggerimento:* $a^n = (a^{n/2})^2$ per *n* pari, e $a^n = a\,(a^{n/2})^2$ per *n* dispari.]

Esercizio 3
Progettare un algoritmo efficiente per verificare se in un array *a* di *n* interi positivi esistono *i* e *j* tali che $a[j] = 2*a[i]$, e analizzarne la complessità.

Esercizio 4
Dato un array **ordinato** di *n* interi positivi, progettare un algoritmo efficiente per verificare se esistono due elementi nell'array la cui somma è *k*.

Esercizio 5
Dato un array **non ordinato** di *n* interi positivi, progettare un algoritmo efficiente per verificare se esistono due elementi nell'array la cui somma è *k*.

Esercizio 6
Sia *a* un array di *n* interi distinti, tale che esiste una posizione j, $0 \leq j < n$, per cui:

- gli elementi nel segmento $a[0, j]$ sono in ordine crescente;
- gli elementi in $a[j+1, n-1]$ sono in ordine decrescente;
- $a[j] > a[j+1]$, se $j < n - 1$.

1. Descrivere un algoritmo che, ricevuto in ingresso *a*, trova la posizione *j* in tempo lineare.
2. Dimostrare che, un qualunque algoritmo che risolve il problema suddetto mediante confronti, richiede tempo $\Omega(\log n)$ al caso pessimo.
3. Descrivere un algoritmo **ottimo** di tipo *divide-et-impera* per il problema precedente. Calcolare la complessità al caso pessimo dell'algoritmo indicando, e risolvendo, la corrispondente relazione di ricorrenza.

Esercizio 7
Si consideri un array *a* di *n* elementi e un valore $k < n$. In una versione modificata dell'algoritmo MERGESORT, l'algoritmo funziona normalmente fino a quando, nella riduzione ricorsiva del problema, $n > k$. Quando invece i sottoproblemi diventano sufficientemente piccoli si ordinano direttamente con una strategia diversa. In particolare gli n/k sottoinsiemi di lunghezza *k* si ordinano con INSERTIONSORT.

- Si definisca lo pseudocodice dell'algoritmo.
- Se ne studi la complessità (suggerimento: per semplificare lo studio si consideri $n=k^r$).

Esercizio 8
Dati due array *a* e *b*, di *n* e *m* interi distinti, progettare un algoritmo efficiente per costruire l'array *c* che rappresenta l'insieme intersezione di *a* e *b*.

Esercizio 9
Dati due array *a* e *b*, di *n* e *m* interi distinti, progettare un algoritmo efficiente per costruire l'array *c* che rappresenta l'insieme unione di *a* e *b*.

Esercizio 10
È dato un array *a* di *n* interi, alcuni dei quali possono essere ripetuti, ordinato in modo non decrescente. Si progetti un algoritmo che, ricevuto in ingresso *a* e un intero *k,* conta il numero *occ* di occorrenze di *k* in *a*.

- Descrivere un algoritmo che richiede tempo *O(n)*.
- Descrivere un algoritmo che richiede o tempo *O(* log *n + occ)* oppure tempo *O(* log *n)*.
- Dimostrare che l'algoritmo di complessità *O(* log *n)* è ottimo al caso pessimo.

Esercizio 11
Progettare un algoritmo di ordinamento che si comporti come MergeSort, ma che divida l'array ricorsivamente in tre parti anziché in due. Scrivere lo pseudocodice della nuova procedura *MergeSort3*. Descrivere a parole la nuova procedura *Fusione3*. Impostare e risolvere l'equazione di ricorrenza associata.

Esercizio 12
Si consideri la seguente relazione di ricorrenza

$$G(n) = \begin{cases} 1 & per\ n = 0 \\ 2 & per\ n = 1 \\ G(n-1) + 2G(n-2) & per\ n > 1 \end{cases}$$

a. *Trovare i valori di* $G(n)$, *per* $n = 2, 3, 4, 5$.

b. *Dal punto precedente, intuire la soluzione esatta di* $G(n)$ *e provarla per induzione.*

c. *Dato un intero* n, *descrivere un algoritmo efficiente per il calcolo di* $G(n)$ *analizzandone la complessità in tempo.*