Algoritmica per Informatica Umanistica-Soluzione
Appello del 7/1/2009

Esercizio 1 **[punti 6]**

Modificare l'algoritmo di ordinamento in tempo lineare, Ordina (A,n), che opera su un array A di n elementi, ciascuno dei quali ha un valore massimo k < n, in modo tale che stampi tutti gli elementi che appaiono una volta sola. Determinare la complessità dell'algoritmo dato.

```
Singoli(A, n):
"definisci un array B di k elementi"
FOR (i = 0, i < k, i++) {
     B[i] = 0;}
FOR (i = 0, i < n, i++) {
     B[A[i]-1]++; }
FOR (i = 0, i < k, i++) {
     IF (B[i] == 1)
     STAMPA (i-1) };
```

L'algoritmo ha complessità O(k) + O(n) + O(k) = O(n).

Esercizio 2 **[punti 6]**

Descrivere a parole un algoritmo efficiente per risolvere lo stesso problema dell'esercizio 1, senza l'ipotesi che gli elementi abbiano un valore limitato da k, ma possano assumere valori qualsiasi.

Se gli elementi possono assumere valori qualsiasi, non si può più usare un algoritmo di ordinamento che opera in tempo lineare, bensì un algoritmo di odinamento efficiente che vada bene per il caso generale. Usiamo quindi QUICKSORT per ordinare i dati in ordine non decrescente. Poi con una scansione dell'array stamperemo un elemento in posizione i se il successivo in posizione i+1 ha valore diverso. Complessità O(nlogn) +O(n) = O(nlogn) al caso medio.

Esercizio 3 **[punti 6]**

Mostrare l'equazione di ricorrenza associata a QUICKSORT quando opera con l'array di input A contenente n elementi già ordinati.

T(n) = O(1) per n=1;
T(n) = T(n-1) + O(n) per n>1;

Perché, nel caso di elementi già ordinati il perno è l'elemento massimo e la procedura DISTRIBUZIONE (O(n)), dividerà l'array in due sottoarray di 0 elementi e di n-1 elementi, rispettivamente, da cui l'equazione. L'equazione data ha soluzione $O(n^2)$.

Esercizio 4 **[punti 6]**

Considerare il seguente algoritmo di tipo Divide et Impera che viene richiamato la prima volta come BOH! (A, 0, n) e che opera su un array A di n interi.

```
BOH!(A,sx,dx):
IF (sx > dx) RETURN TRUE;
IF (sx = = dx) {
IF ("A[sx] è pari" RETURN TRUE;
     ELSE RETURN FALSE;}
cx = (sx + dx)/2;
ps = BOH!(A, sx, cx);
pd = BOH!(A, cx+1, dx);
IF   (ps == pd) RETURN TRUE;
     ELSE RETURN FALSE;
```

Dire cosa calcola BOH!

L'algoritmo ci dice se la somma di tutti gli elementi dell'array A è pari o dispari, poiché la somma di due elementi pari è pari e e la somma di due elementi dispari è sempre pari. Per esempio simulare l'algoritmo su A = {1, 4 , 3 , 2}. Quando si arriva elementi singoli, restituisce rispettivamente FALSE, TRUE, FALSE, TRUE, combinandoli a coppie si ottiene, FALSE, FALSE e finalmente considerando tutti gli elementi si ottiene TRUE, infatti la somma degli elementi è 10 che è pari.

Esercizio 5 **[punti 6]**

Spiegare perché il seguente algoritmo ricorsivo che calcola l'ennesimo numero di Fibonacci è tremendamente inefficiente.

```
FIB(n):
IF (n = = 0) RETURN 0;
IF (n = = 1) RETURN 1;
     ELSE {
     RETURN (FIB(n-1) + FIB(n-2))
     };
```

L'algoritmo FIB è inefficiente perché si richiama ricorsivamente 2 volte, su n-1 e n-2, senza tener conto che per trovare l' (n-1)-esimo numero di Fibonacci bisogna aver già trovato l'(n-2)-esimo. Dunque tutti i calcoli relativi alla seconda chiamata ricorsiva sono inutili perché già fatti. La complessità è $O(2^n)$, inutilmente esponenziale, esiste infatti un algoritmo che risolve lo stesso problema in O(n).