008AA – ALGORITMICA E LABORATORIO
Appello del 22 Giugno 2010

Cognome Nome: **N. Matricola:** **Corso: A B**

Esercizio 1. *(7 punti)* Siano date k liste di dimensione totale n, ciascuna contenente interi distinti ordinati in modo crescente. Si progetti un algoritmo che esegue l'intersezione delle liste (ossia, stampa gli elementi *comuni* che appaiono in tutte le k liste) utilizzando spazio di appoggio $O(k)$ e complessità in tempo al caso pessimo $O(n \log k)$.

(Dettagliare le strutture dati utilizzate, e spiegare ANCHE a parole il funzionamento dell'algoritmo proposto.)

La soluzione mima l'approccio usato per il `merge` di due sequenze, ora esteso a k e operante su liste, mediante l'uso di una struttura dati heap. L'idea consiste nell'estrarre gli elementi in ordine crescente dall'heap, contando quelli uguali. Se sono k allora l'elemento è condiviso tra tutte le liste e quindi può essere stampato. Occorre porre attenzione nello scorrimento delle liste, al fine di garantire di non essere arrivati alla loro fine.

```
Intersezione(L[],k)
    H[0,k-1] heap di minimo contenente "elementi lista" e, aventi chiave e.key
    L[0,k-1] array di puntatori alle teste delle k liste

    for(i=0; i<k; i++) HeapInsert(H,L[i]);
    currentKey = MinHeap(H); //non estrae il minimo
    currentMatches = 0;

    while (!Empty(H)){
        e = ExtractMin(H);
        if (e.key == currentKey)
            { currentMatches++; if (currentMatches == k) print currentKey; }
        else
            { currentKey = e.key; currentMatches = 1; }

        if (e.succ != NULL) HeapInsert(e.succ);
        }
```

Cognome Nome: **N.Matr:**

Esercizio 2. *(3+3 punti)* Data la seguente funzione ricorsiva `foo`, trovare la corrispondente relazione di ricorrenza e risolverla utilizzando il teorema principale.

```
foo( n ){
if ( n < 5 ) return 1;
  else
  {
    tmp = 0;

    for ( i = 1; i <= n; i = i + 1 ) {
        for ( j = 1; j <= n; j = 2*j )
            tmp = tmp + 1;
        }

    return tmp + foo( n/3 ) + foo( n/3 );
  }
}
```

La relazione di ricorrenza è la seguente: $T(n) = O(1)$ se $n < 5$, $T(n) = 2T(n/3) + \Theta(n \log n)$ se $n \geq 5$.

Metodo I:

Si osserva che $n^{\log_b a} = n^{\log_3 2} = o(n) < \Theta(n \log n)$ per cui si ricade nel caso 3 del Teorema Master del CLR: infatti $f(n) = \Theta(n \log n) = \Omega(n^{(\log_3 2)+\epsilon})$, prendendo $0 < \epsilon \leq 1 - \log_3 2$. Per il caso III occorre anche verificare asintoticamente la condizione: $af(n/b) \leq cf(n)$ ossia $2(n/3)\log(n/3) \leq cn \log n$, per un qualche $c < 1$. E infatti $c = 2/3$ va bene. Per cui la soluzione risulta $T(n) = \Theta(n \log n)$.

Metodo II:

Ponendo $f(n) = cn \log n$, occorre vedere se esiste una costante positiva γ tale che $2f(n/3) \leq \gamma f(n)$, ossia $2(n/3)\log(n/3) \leq \gamma n \log n$. A tal fine, è sufficiente scegliere $\gamma = 2/3 < 1$, per cui la soluzione è $T(n) = O(f(n)) = O(n \log n)$.

Cognome Nome: **N.Matr:**

Esercizio 3. *(5+3 punti)*

- Si consideri la sequenza di chiavi intere $S = \{5, 1, 10, 6, 8, 7\}$. Si illustri graficamente l'evoluzione di un albero AVL inizialmente vuoto e soggetto all'inserimento delle chiavi di S, specificando a ogni inserzione: le altezze di tutti i nodi dell'albero, e il nodo critico.
- Alla fine, si stampino le chiavi dell'albero AVL secondo una visita posticipata.

Per motivi grafici indichiamo soltanto il risultato della visita posticipata: $1, 5, 7, 10, 8, 6$.

Cognome Nome: **N.Matr:**

Esercizio 4. *(7+2 punti)* Dato un grafo connesso e non orientato, un suo arco (u, v) si definisce *ponte* se la sua rimozione da G disconnette il grafo in due componenti connesse. Si progetti un algoritmo che stabilisce se G contiene un arco ponte, e se ne calcoli la sua complessità.

Esiste una soluzione di complessità in tempo $O(m(n+m))$ che consiste nell'eseguire la rimozione, a turno, di ogni arco del grafo con conseguente verifica di connettività sul grafo risultante. Se questo grafo non è connesso, allora l'arco rimosso era ponte.

Una soluzione più sofisticata consiste nell'utilizzo della seguente proprietà. Un arco di un grafo non orientato può essere o arco dell'albero DFS o arco backward (vedi proprietà dimostrata in classe, e presente sul CLR e sul CGG). Quindi si osserva che un arco (v, w) backward non può essere un ponte, in quanto i vertici v e w sono anche connessi da un cammino nell'albero DFS. Di contro un arco (v, w) dell'albero è un ponte se e solo se non appartiene a un ciclo, e quindi non esistono archi backward che connettono un discendente di w a un antenato di v nell'albero della visita DFS.

L'algoritmo ricalca dunque una semplice visita DFS con l'aggiunta di un controllo che verifica se l'arco corrente (u, v) è backward: ossia v è colorato di *grigio*. In questo caso, si ripercorre il cammino da u a v, utilizzando i puntatori π, e si marcano tutti gli archi come "non ponte". Se questo marking avviene esplicitamente, marcando il vertice destinazione dell'arco (che univocamente lo identifica trattandosi di un albero), alla fine della visita DFS si scandiscono tutti i vertici dell'albero DFS, ad eccezione della radice, e si definiscono ponte gli archi che incidono sui vertici non marcati.

L'algoritmo ha complessità $O(nm)$ visto che $O(n)$ è il costo del "marking" al caso pessimo, siccome l'albero può avere profondità $\Theta(n)$.

Esiste anche un approccio più efficiente, di complessità $O(n + m)$, che consiste nel creare gli intervalli $[i(v), i(u)]$ per ogni arco backward (u, v), ordinarli secondo la prima componente, fare fusione tra gli intervalli sovrapposti, e poi identificare i vertici x che hanno $i(x)$ non-coperto da questi intervalli-fusi.