CS180 Homework 6

Zhehao Wang 404380075 (Dis 1B)

May 11, 2016

1 King of the tournament

(a) Prove that the king of the tournament exists in the tournament graph.

Proof: in class we proved the following theorem (by induction, each time removing a node and all the nodes it connects to).

Theorem: given a directed graph $G = (V, E)$; there exists $I(G)$ who is an independent set in G's underlying undirected graph such that for any v in V, there exists a w in $I(G)$ such that $(w, v) \in E$ or $(w, u) \in E$ and $(u, v) \in E$ for a $u \in V$.

Since the underlying undirected graph of the tournament graph is a complete graph, all of its independent sets contain one node. Thus according to the theorem, there exists an independent set with one node n, such that n can reach any other node by a path of $length \leq 2$. Thus we have the conclusion.

(b) The algorithm's given in alg 1. Given a graph $G = (V, E)$, the algorithm returns the king of G. We start from an arbitrary node, if it's directly connected to every other node in the graph, we return this node as the king; otherwise we remove this node and all the nodes it connects to, and find the king among the remaining nodes. We do this recursively until we find a node which is directly connected to every other remaining node, and return that node as the king.

Algorithm 1 Find king of tournament

```
function KING(V, E)
    $s \leftarrow firstNode(V)$
    if $s.outDegree = |V| - 1$ then
        return $s$
    $V \leftarrow V - s$
    for each $\{v|(s, v) \in E\}$ do
        $E \leftarrow E - (s, v)$
        $V \leftarrow V - v$
    return $king(V, E)$
```

Correctness: the algorithm starts with an arbitrary node s in graph G. If s is connected to all other nodes in G, s is obviously a king. If otherwise, since we are given a complete graph, we have $\forall v \in V$, if $(s, v) \notin E$, then $(v, s) \in E$. Let G' be the remaining graph after we remove s and all the nodes s connects to in E. By (a), we have that there exists a king s' in G'. Since $(s', s) \in E$, we have that s' is also a king in G (without the removal). Thus we can recursively reduce the problem, until a king is found.

Time complexity: the recursive algorithm visits and removes each edge at most once, and all other operations in each recursion is constant time, so the overall complexity is $O(|E|)$.

2 Same real number in two sets

(a) The algorithm's given in alg 2. We first sort the set S_1 ($O(n \log n)$), then for every element in a S_2, we do a binary search in the sorted S_1 ($O(n \log n)$ for this process). Thus the algorithm is $O(n \log n)$. The correctness is obvious.

Algorithm 2 Same real number in two sets

```
function sameNumber(S1, S2)
    $S'_1 \leftarrow sort(S_1)$
    for $i \in S_2$ do
        if $binarySearch(i, S'_1) = true$ then
            return $true$
    return $false$
```

(b) The algorithm's given in alg 3. We first sort the set S_1 ($O(n \log n)$), then for every element e in a S_2, we do a binary search on $(X - e)$ in the sorted S_1 ($O(n \log n)$ for this process). Thus the algorithm is $O(n \log n)$. The correctness is obvious.

Algorithm 3 Sum of real number in two sets equals X

```
function sumEqualsX(S1, S2, X)
    $S'_1 \leftarrow sort(S_1)$
    for $i \in S_2$ do
        if $binarySearch(X - i, S'_1) = true$ then
            return $true$
    return $false$
```

(c.i) The algorithm's given in alg 4. After sorting S, we maintain two pointers that points at the left and right of the sorted set. We compare the sum of the two pointed numbers, and return if it equals with X, move left pointer right if it's smaller than X, and move right pointer left if it's larger than X, until right and left pointers meet. The initial sorting is $O(n \log n)$, and the two pointer traversal afterwards is $O(n)$, so the overall algorithm is $O(n \log n)$. The correctness is obvious from the above description.

Algorithm 4 Sum of real number in one set equals X

```
function sumEqualsX(S, X)
    $S' \leftarrow sort(S_1)$
    $left \leftarrow 0$
    $right \leftarrow size(S)$
    while $right > left$ do
        if $S'[right] + S'[left] = X$ then
            return $true$
        else if $S'[right] + S'[left] < X$ then
            $left \leftarrow left + 1$
        else
            $right \leftarrow right - 1$
    return $false$
```

(c.ii) Take the initial sorting step out of alg 4, the solution for (c.i), and we have the $O(n)$ solution for this problem. Correctness and complexity are analyzed in (c.i).

3 Find different binary number

(a) The algorithm's given in alg 5. Given an $n \times n$ matrix M, the algorithm returns the different number. For each $0 < i \leq n$, the algorithm finds the i^{th} bit of the i^{th} number in the array, and makes (1 - that number) to be the i^th bit of the resulting number.

Algorithm 5 Find different number in an array

```
function differentNumber(M)
    $n \leftarrow len(M)$
    $result \leftarrow []$
    for $0 < i \leq n$ do
        $result[i] \leftarrow 1 - M[i][i]$
    return $result$
```

Correctness: as shown in the pseudocode, given n numbers, the algorithm is guaranteed to return a number that has at least one bit that's different from each given number; thus the algorithm can find a number that's different from all given numbers.

Time complexity: one loop over n with a constant number of operations inside makes the algorithm $O(n)$.

(b) Prove that $\Omega(n)$ is the lower bound.

Proof: consider the n binary numbers we are given, we have a solution space that's a binary tree with height $n + 1$, and all nodes except the root in the binary tree represents 0 or 1 for the corresponding bit in a number.

The n given binary numbers represent n paths down the binary tree. Our result path needs to differ from each of the given paths by at least one choice. Since the given numbers could take any paths down the tree, the algorithm would need to look at at least one choice for each given number, which makes the algorithm $\Omega(n)$.

4 Radix sort different length

The algorithm's given in alg 6. The algorithm takes a list and returns a sorted list. We first do a bucket sort on the length of all numbers, so that numbers of the same length will fall into the same bucket, and then radix sort numbers in each bucket, and output the sorted results bucket by bucket.

Algorithm 6 Radix

```
function RADIXDIFFERENT(list)
    $L \leftarrow []$
    $maxlen \leftarrow 0$
    $result \leftarrow []$
    for $i \in list$ do
        if $\log_n i > maxlen$ then
            $maxlen \leftarrow \log_n i$
        $L[\log_n i].enqueue(i)$
    for $0 < i \leq maxlen$ do
        $radixSort(L[i])$
        $result.concat(L[i])$
    return $result$
```

Correctness: in the result array, for each i, all elements of length i will be sorted. And for each $j < i$, elements of length j will always come before elements of length i. So the result array is sorted.

Time complexity: let N be the total number of digits in the given list. The first loop gets the length of all elements, and put elements into corresponding buckets; this step is $O(N)$. The second loop goes through all the buckets and radix sort elements in each bucket; the complexity of radix sorting m elements of length l is $O(ml)$, thus this step is also linear to the total number of digits. So the overall algorithm is $O(N)$.