Solutions to Homework 3

Debasish Das
EECS Department, Northwestern University
ddas@northwestern.edu

1 Problem 2.4

Recurrence for Algorithm A

$$T(n) = 5T(\frac{n}{2}) + O(n) \tag{1}$$

Using Master's theorem we get a bound of O($n^{\log 5}$).
Recurrence for Algorithm B

$$T(n) = 2T(n-1) + O(1) \tag{2}$$

The idea to solve such recurrence is to use a recursion tree and combine the constant time operation at each level of recursion tree. Alternatively you can use substitution.

$$\begin{aligned} T(n) &= 2T(n-1) + O(1) \\ T(n-1) &= 2T(n-2) + O(1) \\ &\quad ... \\ T(2) &= 2T(1) + O(1) \end{aligned}$$

Substituting the values of T(i-1) into the equation of T(i), we get the following sum

$$T(n) = \sum_{i=0}^{n-1} 2^i \cdot O(1) \tag{3}$$

Thus we obtain T(n) as O(2^n)
Recurrence for Algorithm C

$$T(n) = 9T(\frac{n}{3}) + O(n^2) \tag{4}$$

Using Master's theorem we get O($n^2 \log n$)

If we do an order analysis, it turns out that Algorithm C is most efficient, since $\log n$ grows slower than $n^{\log 5 - 2}$.

2 Problem 2.12

In this problem we have to give an recurrence for the number of lines printed by the algorithm. The recurrence is given as follows

$$L(n) = \begin{cases} 1 + 2L(\frac{n}{2}) & \text{if n > 1} \\ 0 & \text{if n = 1} \end{cases} \tag{5}$$

Theorem 1 *L(n)* $= \Theta(n)$

Proof: Base Case: L(1) = 0 which is $\Theta(1)$
Hypothesis: c_1·k $\leq$ L(k) $\leq$ c_2·(k-1), k < n
Induction: L(n) = 1 + 2L($\frac{n}{2}$) $\geq$ 1 + 2($c_1 \cdot \frac{n}{2}$)=1+$c_1 n$=$k_1 n$ where k_1 is a constant equal to $c_1 + \frac{1}{n}$
Similarly for the other bound, L(n) = 1 + 2L($\frac{n}{2}$) $\leq$ 1 + 2($c_2 \cdot (\frac{n}{2} - 1)$) = 1+$c_2$n - 2$c_2$ = k_2(n-1) where $k_2 = c_2 - \frac{(c_2-1)}{(n-1)}$
□
Using above result we can say that L(n) is $\Theta(n)$. We can do a more accurate analysis using recursion tree and establish that the line will be printed n-1 times, which is still $\Theta(n)$.

3 Problem 2.14

Given an array of n elements, we need to remove the duplicate elements from the array in O($n \log n$) time. Idea is to maintain the order of elements in the array after the duplicates are removed. The following example explains the idea. Let the array A has following numbers: 2 3 1 3 1 4.
Once the duplicate numbers are removed the output array should be 2 3 1 4.
Note that the order of elements in the final output array is maintained. In other words the final array is not sorted.

```
function remove-duplicate(a[1...n])
Input:  An array of numbers a[1...n]
Output:  Array A with duplicates removed
Construct an array temp[1..n]:
  temp[i] has two fields key and value
for i = 1 to n
  temp[i].value = a[i]
  temp[i].key = i
sort temp based on value
remove duplicates from temp based on value:
  keep the entry with minimum key
sort temp based on key
construct array A from temp:
  A[i] = temp[i].value
return A
```

The field key helps in keeping the order of elements in output array. Two sort takes O($n \log n$). Duplicate removal considering key is O(n). Therefore the algorithm is O($n \log n$). If we don't consider maintaining the order of the original array in the output array the algorithm can be simply given as

```
function remove-duplicate(a[1...n])
Input:  An array of numbers a[1...n]
Output:  Array A with duplicates removed
sort a
construct array A from a:
  by removing duplicate entries from a
return A
```

4 Problem 3.5

Given a graph G = (V,E) we have th find another graph $G^R = (V, E^R)$ where $E^R = (v, u) : (u, v) \in E$. We assume that each edge e has a source vertex u and a sink vertex v associated with it.

```
function find-reverse(G)
Input:  Graph G = (V,E) in adjacency list representation
Output:  Graph G^R
Generate all edges e ∈ E using any traversal
Construct adjacency list G^R:
  vertex set = V
For each generated edge e
  temp = e.source
  e.source = e.sink
  e.sink = temp
  insert e into G^R
return G^R
```

Complexity Analysis: All edges can be generated in O($V + E$). Edges can be modified and new adjacency list can be populated in O(E). Therefore the algorithm is linear.

5 Problem 3.7

A bipartite graph G=(V,E) is a graph whose vertices can be partitioned into two sets (V=$V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$ such that there are no edges between vertices in the same set.Formally

$$u, v \in V_1 \Rightarrow (u, v) \notin E$$
$$u, v \in V_2 \Rightarrow (u, v) \notin E$$

(a)We use the property given in (b) to get a linear time algorithm to determine whether a graph is bipartite. The property says that an undirected graph is bipartite if it can be colored by two colors. The algorithm we present is a modified DFS that colors the graph using 2 colors. Whenever an back-edge, forward-edge or cross-edge is encountered, the algorithm checks whether 2-coloring still holds.

```
function graph-coloring(G)
Input:  Graph G Output:  returns true if the graph is bipartite
   false otherwise
for all v ∈ V:
 visited(v)= false
 color(v) = GREY
while ∃s ∈ V : visited(s) = false
 visited(s) = true
 color(s) = WHITE
 S = [s] (stack containing v)
 while S is not empty
   u = pop(S)
   for all edges (u,v) ∈ E:
     if visited(v) = false:
       visited[v] = true
       push(S,v)
     if color(v) = GREY
       if color(u) = BLACK:
         color(v) = WHITE
       if color(u) = WHITE:
         color(v) = BLACK
     else if color(v) = WHITE:
       if color(u) ≠ BLACK:
         return false
     else if color(v) = BLACK:
       if color(u) ≠ WHITE:
         return false
return true
```

(b)

Lemma 1 *An undirected graph is bipartite if and only if it contains no cyles of odd length*

Proof: $\Rightarrow$Consider a path P whose start vertex is s, end vertex is t and it passes through vertices $u_1, u_2, ..., u_n$ and the associated edges are $(s, u_1), (u_1, u_2), ..., (u_n, t)$. Now if P is a cycle, then s and t are the same vertices. Without loss of generality assume s is in V_1. Each edge (u_i, u_{i+1}) goes from one vertex set to other. Therefore a path must have 2·i edges to come back into the same vertex set where i ∈ N. Since s and t are in same vertex set, so the length of the cycle formed must be 2·i which is even.
$\Leftarrow$Suppose the graph has a cycle of odd length. Let the cycle be C and it

passes through vertices $u_1, u_2, ..., u_n$ where $u_1 = u_n$. The associated edges are $(u_1, u_2), ..., (u_{n-1}, u_n)$. We start coloring edges of using two colors WHITE and BLACK. Without any loss of generality u_1 is colored WHITE while u_{n-1} is colored BLACK since n is odd and therefore $n-1$ is even. Choosing color of u_n as WHITE conflicts with the color of u_{n-1} while choosing color as BLACK conflicts with the color of u_1. Therefore it is not possible to color an odd cycle with 2 colors which implies that the graph is not bipartite(using the property mentioned in (b)) □

(c)3. It follows from the $\Leftarrow$ proof of part (b).

6 Problem 3.13

(a)

Lemma 2 *In any connected undirected graph G=(V,E) there is a vertex $v \in V$ whose removal leaves G connected*

Proof: Consider a graph G=(V,E) and a Depth-first-search tree T constructed from the graph using DFS. Given any connected graph, generation of T is linear time O(V+E) and there exist one unique tree T. Denote the leaves of the tree T by L(T). If we choose any vertex v $\in$ L(T), removal of that vertex and the edges associated with it will keep T connected (by the definition of tree). Thus $\hat{T}$ = T - {v} is connected which implies that the graph $\hat{G}$=(V-{v},E-$\{(i,j) : i = v \vee j = v\}$) is connected since $\hat{T}$ and $\hat{G}$ are isomorphic. □

(b)See Figure 1(a)

(c)See Figure 1(b)

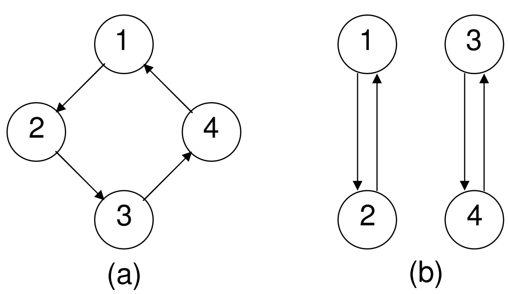

Figure 1: Examples for 3.13(b) and 3.13(c)