Fall '15 CS 5800 Algorithms Hackerrank Group: akashsingh245

Problem Set 2

Surekha Jadhwani, Akash Singh, and Ayush K. Singh

1 Part A

Answers

1. Consider the problem of sorting an array $A[1, ..., n]$ of integers. We presented an $O(nlogn)$-time algorithm in class and, also, proved a lower bound of $\Omega(nlogn)$ for any comparison-based algorithm.

   (a) Give an efficient sorting algorithm for a boolean array $B[1, ..., n]$.
   $*$In a boolean array $B[1, ..., n]$, each element $B[i]$ (for $i = 1, ..., n$) is either 0; or 1.
   Solution:
   Input: $Boolean[1, \cdots, n] : B$
   Output: $Boolean[1, \cdots, n] : B$ sorted in increasing order
   Psuedocode:

```
Create an array $count[2]$
Initialize the elements of $count$ array to 0
$current\_pointer = 1$
for i ← 1 to $B.size$ do
  $count[B[i] + 1] + +$
end for
for j ← 1 to $count.size$ do
  while $count[j] > 0$
    $B[current\_pointer] = j - 1$
    $current\_pointer + +$
    $count[j] - -$
  end while
end for
```

   Complexity: The complexity of above algorithm is $\mathbf{O(n)}$ for an input array of size n which is linear time.

   (b) Give an efficient sorting algorithm for an array $C[1, ..., n]$ whose elements are taken from the set $\{1, 2, 3, 4, 5\}$.
   Solution:
   Input: $Integer[1, \cdots, n] : C$
   Output: $Integer[1, \cdots, n] : C$ sorted in increasing order
   Psuedocode:

```
Create an array $count[5]$ //where '5' is number of elements in set $\{1, 2, 3, 4, 5\}$
```

```
 Initialize the elements of $count$ array to 0
 $current\_pointer = 1$
 for i ← 1 to $C.size$ do
   $count[C[i]]$++
 end for
 for j ← 1 to $count.size$ do
   while $count[j] > 0$
     $B[current\_pointer] = j$
     $current\_pointer + +$
     $count[j] - -$
   end while
 end for
```

Complexity: The complexity of above algorithm is **O(n)** for an input array of size n which is linear time.

(c) Give an efficient sorting algorithm for an array $E[1, ..., n]$ whose elements are distinct ($E[i] \neq E[j]$, for every i $\neq j \in 1, ..., n$); and are taken from the set $\{1, 2, ..., 2n\}$.

Solution:

Input: $Integer[] : E$

Output: $Integer[] : E$ sorted in increasing order

Psuedocode:

```
 Create an array $count[2 * E.size]$
 Initialize the elements of $count$ array to 0
 $current\_pointer = 1$
 for i ← 1 to $E.size$ do
   $count[E[i]] = 1$
 end for
 for j ← 1 to $count.size$ do
   if ($count[j] == 1$) then
     $E[current\_pointer] = j$
     $current\_pointer + +$
   end if
 end for
```

Complexity: The complexity of above algorithm is **O(n)** for an input array of size n which is linear time.

(d) In case you designed linear-time sorting algorithms for (a-c) above, does it mean that the lower bound for sorting of $\Omega(nlogn)$ is wrong? Explain.

Solution: Lower bound for sorting of $\Omega(nlogn)$ is for comparison based sorting. However, sorting technique used in $(a - c)$ above is counting sort which uses key values as indexes for array. Counting sort uses simple for/while loops while comparison sort uses recursion. Hence, the linear-time sorting of algorithms is possible.

2. A Fibonacci number is a number that appears in the Fibonacci Sequence 1,1,2,3,5,8,13,.... The next number is found by adding the two numbers before it. Formally, $F_0 = 1, F_1 = 1$ and $F_{n+1} = F_n + F_{n-1}$, for $n \geq 2$. Define the set

$$\text{FIBO} = \{n \mid n \text{ is Fibonacci number}\}$$

. Give an efficient algorithm for checking if a given number n is a Fibonacci number (i.e., an algorithm that return 1, if $n \in FIBO$ and 0, otherwise). Analyze the correctness and the running time of your algorithm.

Solution:

Input: A number n

Output: 1 if n is a Fibonacci number, 0 otherwise

Psuedocode:

```
//Since, $n$ is an integer, we will use following numbers $(n_1, n_2, s_1, s_2)$ as integers.
 $n_1 \leftarrow (5n^2 - 4)$
 $n_2 \leftarrow (5n^2 + 4)$
 $s_1 \leftarrow (sqrt(n_1))$ //Decimals, if any, will be truncated as $s_1$ is integer.
 $s_2 \leftarrow (sqrt(n_2))$ //Decimals, if any, will be truncated as $s_2$ is integer.
 //Check for perfect squares
 if $(((s_1 * s_1) == n_1) or ((s_2 * s_2) == n_2))$ then
   return 1
 else
   return 0
 end if
```

Correctness: According to Gessel's test, a number n is a Fibonacci number iff $(5n^2-4)$ or $(5n^2+4)$ is a perfect square.
Above algorithm uses same test, hence, it will return 1 or 0 for every n value.
This program doesn't have any loops, hence, we know that it will terminate.
As, the program correctness is not dependent upon the value of the n and it terminates, we can say that this algorithm is correct.

Complexity: This algorithm doesn't involve any loops or recursive calls. Also, it is executed for constant number of steps which are independent of n
Hence, the recurrence relation is:
$T(n) = c$
So, the runtime complexity is $O(1)$.

3. **Graph Coloring.** A vertex coloring is an assignment of colors (or labels) to each vertex of a graph such that no edge connects two identically colored vertices. More formally, a proper vertex coloring is an assignment $c : VS$ such that $c(v) \neq c(u)$, for every edge $(u, v) \in E$. The elements of S are called colors; the vertices of one color form a color class. If $|S| = k$, we say that c is a k-coloring (often we use $S = \{1, ..., k\}$). A graph is k-colorable if it has a proper k-coloring. The chromatic number $\chi(G)$ is the smallest value of k such that G is k-colorable.

   (a) Let G be a graph where no node has degree larger than Δ. Prove that $\chi(G) \leq \Delta + 1$. (Hint: design a $\Delta + 1$ coloring algorithm and prove its correctness.)
   Solution: We will use greedy algorithm for coloring a graph with n-nodes having maximum degree as Δ
   The algorithm is as follows:

      i. Sort the vertices of the graph as $v_1, v_2, \cdots, v_n$ using smallest vertex ordering.

      ii. Order the colors as $c_1, c_2, \cdots, c_{\Delta+1}$

      iii. Assign the first color to vertex v_1.

      iv. For each of the remaining v_{n-1} vertices, ie., for i=1...n-1, repeat the following steps:

         A. Visit v_i and assign the least available color to it which is not yet used by any of it's neighbors from $v_1, \cdots, v_{i-1}$.

         B. If all previous colors are used by v_i's neighbors, assign a new color to it.

   Correctness: Let $P(n)$ be: If every node in a n-node graph G has at most degree of Δ, then maximum $(\Delta + 1)$ colors are used in the coloring of graph G.
   Base Case: For $n = 1$, the number of edges in graph is 0, that is degree(Δ) is 0. Also, it requires 1 color for the graph which is equal to $(\Delta + 1)$.
   Hence, $P(1)$ holds true.
   Induction Step: Since, $P(1)$ is true, we assume $P(n)$ is true.
   Let G(V, E) be any $(n + 1)$-node graph having maximum degree as Δ.
   As per the algorithm, we order the vertices of graph G as $v_1, v_2, \cdots, v_{n+1}$
   Now, we consider a graph G', which is a sub-part of G but has n-nodes and maximum degree as Δ
   By induction hypothesis, G' uses maximum $(\Delta + 1)$ colors.
   We know that, maximum degree of v_{n+1} is Δ, i.e., it has at most Δ neighbors. Thus, at most Δ colors are assigned to its neighbors and we cannot use it again.

So, we need to find the next smallest color which is not used by it's neighbors. Hence, there exists a color in a set of $(\Delta + 1)$ colors which is not used by any Δ neighbors and it can be used for v_{n+1} node.
Thus, it uses at most $(\Delta + 1)$ colors on G, which implies $P(n + 1)$ is true.
Termination: This algorithm visits each vertex of the graph exactly once. This implies that the algorithm terminates.
Hence, the above algorithm is correct.

(b) Planar graphs are graphs that can be drawn on the plane such that no edges cross each other. Planar graphs are known to satisfy certain properties. For example, the number of edges in a connected planar graph will be proportional to the number of nodes. In particular, it is known that in a planar graph there are at most $3n$ edges ($|E| < 3n$) and hence $\sum_{v \in V} deg(v) < 6n$.
Give an efficient 6-coloring algorithm for planar graphs. Prove its correctness and analyze its running time.
Solution: For any planar graph with $n \geq 3$, it is given that ($|E| < 3n$).
Therefore, the degree of such graph can be derived as:
$d(G) = \dfrac{2|E|}{|V|} < \dfrac{2 * 3n}{n} < 6.$
Thus, any vertex in a planar graph can have at most 5 neighbors.
Now, greedy algorithm can be used for this problem as in (Part 3(a)) which will color the graph using at most 6 colors. The algorithm is as follows:

i. Sort the vertices of the graph as $v_1, v_2, \cdots, v_n$ using smallest vertex ordering.

ii. Order the colors as $c_1, c_2, \cdots, c_{\Delta+1}$

iii. Assign the first color to vertex v_1.

iv. For each of the remaining v_{n-1} vertices, ie., for i=1...n-1, repeat the following steps:

   A. Visit v_i and assign the least available color to it which is not yet used by any of it's neighbors from $v_1, \cdots, v_{i-1}$.

   B. If all previous colors are used by v_i's neighbors, assign a new color to it.

Correctness: Let $P(n)$ be: For any n-node planar graph G, at most 6-colors are used in the coloring of graph G.
Base Case: For $n \leq 6$, the number of edges in graph at most 5, that is degree 5. So, it will require at most 6 colors for the graph.

Hence, $P(1)$ holds true.
Induction Step: Since, $P(1)$ is true, we assume $P(n)$ is true.
Let G(V, E) be any simple planar graph with $(n+1)$ vertices. We know that maximum degree in graph G is 5.
As per the algorithm, we order the vertices of graph G as $v_1, v_2, \cdots, v_{n+1}$
Now, we consider a graph G', which is a part of G but has n-nodes and maximum degree as 5
By induction hypothesis, G' uses maximum 6 colors.
We know that, maximum degree of v_{n+1} is 5, i.e., it has at most 5 neighbors. Thus, at most 5 colors are assigned to its neighbors and we cannot use it again. So, we need to find the next smallest color which is not used by it's neighbors. Hence, there exists a color in a set of 6 colors which is not used by any 5 neighbors and it can be used for v_{n+1} node.
Thus, it uses at most 6 colors on G, which implies $P(n+1)$ is true.
Termination: This algorithm visits each vertex of the graph exactly once. This implies that the algorithm terminates.
Hence, the above algorithm is correct.
Complexity: In the above algorithm, the runtime complexity will include time taken for ordering of vertices ,i.e., $O(nlogn)$ time and the time taken 'for' loop execution, i.e., $O(n)$ time.
Hence, the running time complexity is $O(n)$.

(c) Give an efficient algorithm that takes as input a graph $G(V, E)$ and returns a proper 2-coloring assignment if G is 2-colorable; otherwise returning the message "G is **not** 2-colorable". Prove the correctness and analyze the running time of your algorithm.
Solution:
Algorithm: In order to check if the graph is 2-colorable or not, we will use a string array called Color whose size is equal to number of vertices in the graph. Initially, all the elements will be set to null and vertices would be unvisited. To check graph coloring, we will traverse the graph using BFS for all unvisited vertices one by one. First vertex in graph is assigned Red color and all its adjacent vertices are assigned Blue color. We will continue this process of assigning alternate colors till all the vertices are covered. If a vertex is colored same as it's predecessor, then graph is not 2-colorable.
Psuedocode:
```
Create a string array Color[V.size]
Initialize all elements of Color to null
Mark all vertices as unvisited
for v in V do
  if v is unvisited do
    frontier = new Queue() //initialize an empty queue frontier
```

```
      mark v visited (set v.distance = 0)
      Color[v] ← Red
      frontier.push(v)
      while frontier not empty do
        Vertex v' = frontier.pop()
        for each successor u of v'
          if u unvisited
            if color[u] = null do
              if color[v'] = Red
                Color[u] ← Blue
              else
                Color[u] ← Red
               end if
              frontier.push(v')
              mark v' visited (v'.distance = v.distance + 1)
            else if Color[v'] = Color[u]
              output "G is not 2-colorable"
            end if
        end for
      end while
    end if
  end for
  output Color[]
```
Correctness: Here, we will prove the correctness of above algorithm by using mathematical induction:
Let $P(n)$: For any vertex w in a graph G which is at a distance of n units from root v, w must receive a color that is part of 2-coloring in G.
Base case: For $P(0)$, distance between w and v is 0,i.e., vertex w = vertex v, it will get a red color.
Hence, P(0) holds true.
Induction Step: Since, $P(1)$ is true, we assume $P(n)$ is true.
For $P(n+1)$, the distance between w and v is $(n+1)$. Let u be the vertex which is at a distance of n from v. Hence, by hypothesis, u has a color that is part of 2-coloring.
As per the above algorithm, w will get either of the color depending on the color of u.
Hence, $P(n+1)$ also holds true.
Also, in the last step of this algorithm, we check if the two adjacent vertices have same color. If yes, the algorithm outputs "G is not 2-colorable" as endpoints of an edge must have different colors.
Termination: This algorithm visits each vertex of the graph exactly once. This implies that the algorithm terminates.

This ensures correctness of the algorithm.
Complexity: Since, this method uses BFS, the total running time for the above algorithm is $O(|V| + |E|)$

4. Prove that in any tree, there exists a node which if removed breaks the tree into connected components such that no connected component contains more than half the original nodes.
   Solution: Let there exists such a node in a tree which on removal breaks the tree into connected components such that no connected component contains more than half the original nodes.
   We can use following algorithm to find such vertex.
   Psuedocode:

```
$n \leftarrow$ Number of nodes in the tree
$current\_vertex \leftarrow$ Any vertex v
while true
  if (size of any connected component of $current\_vertex$) $> n/2$
    $current\_vertex =$ Any vertex in that largest component adjacent to $current\_vertex$
  else return $current\_vertex$
  end if
end while
```

   Correctness: We can prove the correctness of this algorithm by showing that it terminates. Here, the loop executes without any constraint but it never visits any old vertex back as the size of that component would be less than $(n/2)$. The program exits whenever it finds a vertex whose size of components is less than $(n/2)$. Also, the tree has finite number of vertices. Hence, the algorithm terminates.
   Since, the correctness can be devised for the above algorithm, we can say that, there exists a node which if removed breaks the tree into connected components such that no connected component contains more than half the original nodes.

5. For a given graph $G(V, E)$. The distance $d_G(v, u)$ between v and u is the length in hops of the shortest path between v and u. The diameter D_G of a graph G is the maximum distance among all pairs (of nodes) in V , i.e., $D_G = max d_G(u, v)|v, u \in V$. Let $a, b \in V$ be two nodes such that $d_G(a, b) = D_G$. Decide whether each of the following statements are correct and give a proof for each part.

   (a) For every node $r \in V$ either a or b is a leaf in a BFS tree of G rooted at r.
   Proof: The given statement is true.
   The algorithm to find diameter of a graph G is as follows:
   1. Start BFS on any node r and note the last vertex visited. Let the last vertex be u.
   2. Start BFS from node u and note the last vertex visited. Let the last vertex be v.
   3. d(u,v) is the diameter of the graph.
   The correctness of above algorithm can be proved as follows:
   Say we start at the node for BFS on the graph, we have an intermediate node s and node u as the leaf node on the graph G. The diameter of the graph is equal to $d(u, v)$
   Let us assume that there exists another diameter for the same graph between the nodes a and b.
   Let p_1 be the path from node s to node u and p_2 be the path from node a to node b.
   Now we have the following two cases:
   Case 1: If p_1 and p_2 have a common vertex.
   Then node t lies on path p_1. Since, u is the last node of the BFS search, $d(t, u) \geq d(t, a)$
   But as we know, p_2 is the largest path in graph G. This implies that, $d(t, a) \geq d(t, u)$. Hence we can say, $d(t, a) = d(u, a)$ and $d(u, b) = d(a, b)$.
   Case 2: If p_1 and p_2 do not have a common vertex, then we have
   $d(t, u) \geq d(s, u)$
   $d(t, u) \geq d(s, a)$
   $d(t, u) \geq d(t, a)$
   $d(b, u) \geq d(b, a)$
   Since, $d(a, b) \geq d(u, b)$, $d(a, b) = d(u, b)$
   Thus, $d(a, b) \geq d(u, v)$ and $d(u, v) \geq d(u, b)$. So, all three are equal and we can say that $d(u, v)$ is the diameter of the graph.
   Hence, from step 1 and 2 of above algorithm, we can conclude that if $d(a, b)$ is the diameter of graph G, then for every node $r \in V$ either a or b is a leaf in a BFS tree of G rooted at r.

   (b) Node a is a leaf in any BFS tree of G rooted at b.
   Proof: The given statement is true.
   We know that the algorithm to find diameter of a graph G is as follows:

1. Start BFS on any node r and note the last vertex visited. Let the last vertex be u.
2. Start BFS from node u and note the last vertex visited. Let the last vertex be v.
3. d(u,v) is the diameter of the graph.
The correctness of above algorithm can be proved as follows:
Say we start at the node for BFS on the graph, we have an intermediate node s and node u as the leaf node on the graph G. The diameter of the graph is equal to $d(u,v)$
Let us assume that there exists another diameter for the same graph between the nodes a and b.
Let p_1 be the path from node s to node u and p_2 be the path from node a to node b.
Now we have the following two cases:
Case 1: If p_1 and p_2 have a common vertex.
Then node t lies on path p_1. Since, u is the last node of the BFS search, $d(t,u) \geq d(t,a)$
But as we know, p_2 is the largest path in graph G. This implies that, $d(t,a) \geq d(t,u)$. Hence we can say, $d(t,a) = d(u,a)$ and $d(u,b) = d(a,b)$.
Case 2: If p_1 and p_2 do not have a common vertex, then we have
$d(t,u) \geq d(s,u)$
$d(t,u) \geq d(s,a)$
$d(t,u) \geq d(t,a)$
$d(b,u) \geq d(b,a)$
Since, $d(a,b) \geq d(u,b)$, $d(a,b) = d(u,b)$
Thus, $d(a,b) \geq d(u,v)$ and $d(u,v) \geq d(u,b)$. So, all three are equal and we can say that $d(u,v)$ is the diameter of the graph.
Hence, from step 2 of above algorithm, we can conclude that if $d(a,b)$ is the diameter of graph G, then Node a is a leaf in any BFS tree of G rooted at b.

(c) The depth of every BFS tree of G is at least $D_G/2$. (The depth of a tree is the depth of its deepest leaf).
Proof: The given statement is true.
By definition, radius r_G of the graph G is equal to the minimum distance among all pairs of the nodes, while, diameter D_G of the graph G is equal to the maximum distance among all pairs of the nodes.
Also, the minimum height of every BFS tree is equal to the radius r of the graph.
From above definition,
$r_G \leq D_G$. (Equation 1)
Now, let us consider two vertices a,b such that $d(a,b)$ is the diameter of the graph.
$d(a,b) = D_G$. (Equation 2)
Let v be a central vertex of graph. Thus, eccentricity of vertex v is equal to the

radius of graph G
$\in(v) = r_G$ (Equation 3)
We know that,
$d(a,b) \leq d(a,v) + d(v,b)$
Since, vertex v is central vertex, $d(a,v)$ and $d(v,b)$ will be equal to eccentricity of v. $d(a,b) \leq 2\in(v)$
Using equation 3, we get,
$d(a,b) \leq 2r_G$ (Equation 4)
Combining equations 1,2 and 4, we get,
$r_G \leq D_G \leq 2r_G$
Thus, $r_G \geq \frac{D_G}{2}$
Hence, the depth of every BFS tree of G is at least $D_G/2$.

2 Part B

Group Hackerrank Name: akashsingh245

1. Surekha Jadhwani : surekha@ccs.neu.edu / jadhwani.s@husky.neu.edu
2. Akash Singh : singhaka@ccs.neu.edu / singh.aka@husky.neu.edu
3. Ayush K. Singh : singhay@ccs.neu.edu / singh.ay@husky.neu.edu