CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000

Solutions to Practice Homework # 3

1. For simplicity, assume that the graph $G = (V, E)$ is connected.

   The first stage of the algorithm simply numbers the nodes of the graph from 1 to n in the order that nodes are visited by either the dfs or bfs algorithms. Also, for each node x, the parent of x in the dfs (or bfs) tree is recorded and is denoted by parent(x). Note that parent(x) $\leq$ x for all nodes x $>$ 1 because a node is visited after its parent in either dfs or bfs order.

   The second stage of the algorithm uses another array, called color, to store the color (green, red, or blue) assigned to nodes of the graph during the algorithm. Initially, color(1), i.e. the color of node 1, is set to red and the remaining colors are undefined. This stage of the algorithm calls a procedure called find-3-colorable (below). This procedure has a single parameter, x. When find-3-colorable(x) is called, it is the case that nodes $1, \ldots n - x$ are already colored and moreover, this partial coloring is valid in the sense that no two nodes from the set $\{1, \ldots, n - x\}$ that are connected by an edge are colored with the same color. Procedure find-3-colorable(x) sets a boolean variable called 3-colorable to "true" if this coloring of nodes $1, \ldots n - x$ can be extended to a valid coloring of the whole graph. (Initially, 3-colorable is set to false.) Since node 1 is initially colored, this procedure is initially called with $x = n - 1$. Upon completion of find-3-colorable, the value of the variable 3-colorable indicates whether the graph is 3-colorable or not.

```
       find_3_colorable(x)
1         if (x = 0) then set 3-colorable to ‘‘true’’
2         else {try to extend the coloring to node n-x+1}
3            for each color c that is not color(parent(n-x+1)) do
4               check if there is a node i < n-x+1 of color c
5                                    with i adjacent to n-x+1
6               if there is no such conflicting node then
7                  color(x) = c
8                  find_3_colorable(x-1)
```

   Running time: The time to complete the initial dfs or bfs is linear, $O(n + m)$.

   The procedure find-3-colorable(x) makes at most two recursive calls to find-3-colorable(x-1), since there are at most two possible valid colors for node n-x+1. Other than the recursive calls (in line 7), the time taken by a single call to the procedure is at most linear in the size of the graph. Hence, we obtain the following recurrence for the running time $R(x)$ of the find-3-color procedure:

$$R(x) = 2R(x - 1) + O(n), n \geq 2, \quad R(0) = O(1).$$

   Solving this, say using the iteration method, we see that $R(n) = O(2^n n)$.

2. Assume $n \geq 3$.

   (a) Initially, divide the set of balls into three groups. Weigh two of them to decide which group has the odd ball. Do one further weighing of this group with a different group to determine whether the odd ball is heavier or lighter. If it is heavier, continue with the algorithm of lecture 4. Otherwise, continue with the same algorithm, modified to handle the case that the odd ball is lighter, not heavier, than the other balls.

   b) The number of weighings in the worst case is the same as in lecture 4, plus one extra to determine whether the odd ball is heavy or light. In the case that the number of balls n is a power of 3, the total number of weighings is therefore $\log_3 n + 1$.

c) How many solutions are there to the problem? There are n balls, and each may be heavy or light, so there are $2n$ possible solutions. We can model any algorithm as a decision tree with at least $2n$ leaves, in which nodes correspond to weighings. The fanout of each internal node is at most 3 (corresponding to the three cases that in the weighing the left side of the scale is heavy, the left side is light, or both sides are of equal weight). The height of the tree is the number of weighings needed. So the lower bound on the worst-case number of weighings required by any algorithm should be $\lceil \log_3(2n) \rceil$. This is at least $\log_3 n + 1$ when $n \geq 3$ is a power of 3.

d) Yes. For example, there n = 3*k+1 balls. The three subgroups have the same weight. We know the odd ball must be the left one. If we need to decide whether it's lighter or heavier, one more weighing is necessary; otherwise, just output this odd ball.

3. Here is an algorithm for solving the nuts and bolts problem, inspired by Hoare's quicksort algorithm.

```
Match_Nut_to_Bolt(set_of_nuts, set_of_bolts)
{
  if (set of nuts is empty) then nothing to do
  if (one nut in each set) then {We have found the match}
    put the nut on the bolt
  else
  {
     randomly select a nut from the set of nuts

     By comparing each bolt with the selected nut, divide the bolts
     into 3 groups: - big-bolts: bigger than the selected nut
                    - small-bolts: smaller than the selected nut
                    - matches the selected nut

     Then, by comparing each nut with the selected bolt (i.e. that matching
     the selected nut), divide the nuts into 3 groups :
                    - big-nuts: bigger than the selected bolt
                    - small-nuts: smaller than the selected bolt
                    - matching the selected bolt

     The selected nut and bolt are now matched.

     Match_Nut_to_Bolt(small-nuts, small-bolts)
     Match_Nut_to_Bolt(big-nuts, big-bolts)
  }
}
```

Correctness: At each iteration, a nut will always be in the same group as the bolt that matches that nut. Because the algorithm runs until the size of a group equals one, we are guarenteed to eventually match every nut to the correct bolt.

Running time: If the sets are of size n, the amount of time needed to divide the nuts and bolts into 3 groups is O(n). Therefore, the total running time is O(n), not counting the recursive calls.

Each matching nut-bolt pair is equally likely to be selected as the "splitter". Therefore, at each stage, the algorithm has the possiblity of splitting the data into subproblems of exactly the same sizes as does the randomized quicksort algorithm.

Putting the previous two fact together, we can see that the recurrence relation that models the running time of the randomized quicksort algorithm also models the running time for this problem. The recurrence is:

$$T(n) = O(n) + \frac{2}{n} \sum_{i=1} n - 1T(i)$$

The solution to this recurrence is $T(n) = O(n \log n)$.

4. No solutions were submitted for problem 4. We'll put a solution on-line as soon as anyone submits one.