CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000

Practice Homework # 3

1. Consider again the problem of 3-coloring a graph: given an undirected graph, you want to determine whether it is possible to assign one of three possible colors to each node of the graph, so that no two adjacent nodes have the same color.

   Find an algorithm that runs in time $O(2^n \text{poly}(n))$ where here, poly(n) denotes any polynomial function of n.

   Suggestion: Note that once a node is colored, there are only two possible colors left for its neighbors. Use an approach based on depth first or breadth first search to take advantage of this when exploring possible colorings of the graph.

2. Consider the following variation of the golf balls problem. You are given n golf balls, and you know that all but one are of equal weight. The remaining ball – call it the "odd ball" – is either slightly lighter or slightly heavier than the others, but you don't know which. The only operation you are allowed to do with the golf balls is to weigh them using a balance scale. Your job is to output the odd ball, and state whether it is heavier or lighter than the other balls.

   (a) Design an algorithm that solves this problem. Your algorithm should use as few weighings as possible. Keep your algorithm simple!

   (b) Prove a bound on the number of weighings your algorithm does in the worst case.

   (c) Prove a lower bound on the worst-case number of weighings required by *any* algorithm for this problem.

   (d) Suppose the problem is to output the odd ball, but you don't need to state whether it is heavier or lighter. Are there cases in which you can solve the problem with fewer weighings in this case?

3. We wish to sort a bag of n nuts and n bolts by size in the dark, so that we output n (nut, bolt) pairs that fit together.

   We can compare the sizes of a nut and a bolt by attempting to screw one into the other. This operation tells us that either the nut is bigger than the bolt; the bolt is bigger than the nut; or they are the same size (and so fit together). Because it is dark we are not allowed to compare nuts directly or bolts directly.

   Design and analyze an algorithm for this problem for which the expected number of "comparisons" is $O(n \log n)$.

4. Recall the linear-time deterministic selection algorithm described in class. The algorithm is as follows:

```
Select(A[left,right],k)
   {We assume that A[left,right] has at least k elements, where k
    is the rank of the element that we want to find}

   if (left = right) then output A[left]
   else
      Find-Pivot(A[left,right])
      middle <- Partition(A, left,right)
      {Partition is the algorithm from the ``Quicksort'' lecture}

      if (k = middle-left+1) then output A[middle]
      if (k < middle-left+1) then Select(A[left,middle-1],k)
      if (k > middle-left+1) then
            Select(A[middle+1,right],k-(middle - left + 1))
```

In the above pseudocode, Find-Pivot is a procedure that selects a pivot from the array elements and swaps the selected pivot with the leftmost array element. (The Find-Pivot procedure in class chooses the pivot using the "median of medians of 5" algorithm, but that does not concern us here.)

Now, suppose that the Find-Pivot algorithm works as follows. Intuitively, all the numbers in the array are summed, the average is taken, and then a scan through the array is done to find the array element closest to this average. This closest array element is chosen as the pivot. More precisely, here is the Find-Pivot pseudocode:

```
Find-Pivot(A[left,right])
    sum <- A[left} + A[left+1] + ... + A[right]
    avg <- sum/(right-left+1)
    p <- left
    {A[p] is the closest array element to avg found so far}
    for j <- left +1 to right do
       if |A[j] - avg| < |A[p] - avg| then p <- j
       {here |x| denotes the absolute value of x,
        which is x if x>=0 and -x otherwise}
    swap(A[left], A[p])
```

What can you say about the worst-case running time of this algorithm?