Sample Interview Problems and Solutions

September 14, 2016

Problem 1

Verify if a binary tree is also a binary search tree. (Naren Manoj)

A binary tree is a BST iff, for a given node in the tree, the following are true:

- The value of the left child is less than or equal to that of the given node.
- The value of the right child is greater than that of the given node.
- The trees rooted at the left child and right child are BSTs.

Hence, our solution just involves encoding this. In C, we can write:

```
int isBST(struct TreeNode* root) {
    if(root == 0) return 1;
    if(root->left && root->left->val > root->val) return 0;
    if(root->right && root->right->val <= root->val) return 0;
    return (isBST(root->left) && isBST(root->right));
}
```

Problem 2

Suppose a set contains exactly $n-1$ of the integers $1, 2, ..., n$. If one more integer is added to the set, then the set is equal to $\{1, 2, ..., n\}$. Determine this integer. What if our set originally contains $n-2$ integers? (Anonymous course at UT)

Notice that the sum of the first n positive integers is $\frac{n(n+1)}{2}$. Hence, if we add the elements of the given set and subtract the sum from this value, then we should get our answer for just one element. A C function for this could be

```
int findMissing(int* inputArr, int n) {
    int sum = 0;
    for(int i = 0; i < n; i++ ) {
        sum += inputArr[i];
    }
    return n * (n + 1) / 2 - sum;
}
```

To solve the other problem, we can use the insight from the previous solution. Notice that by adding all the elements in the set and subtracting from the total sum, we obtain the sum of the two missing numbers. Similarly, by computing $n!$ and dividing out the product of the integers in the set, we obtain the desired product of the two missing numbers. Notice, then, that given the product and sum of the two missing numbers, we can write a quadratic whose roots are the two missing numbers. Solving this using the quadratic formula gives us our answer.

Problem 3

Implement a stack using queues. (Anonymous company)

Describing the solution is too annoying here. Please visit
`http://www.geeksforgeeks.org/implement-stack-using-queue/` for the solution.

Problem 4

Suppose there is an integer array with m rows and n columns. Suppose that every entry in this matrix is either 0 or 1. Define a path *to be a sequence of distinct pairs $(r_0, c_0), (r_1, c_1), ...$ such that $c_i - c_j$ and $r_i - r_j$ are nonnegative and $i > j$. How many paths exist between $(0, 0)$ and $(m-1, n-1)$ such that none of the pairs in the path index an entry marked with a 0? (Naren Manoj)*

Notice that the number of paths from `matrix[i][j]` is simply `solve(matrix[i + 1][j]) + solve(matrix[i][j + 1])` or 0 if `matrix[i][j]` is 0. Since we will run into many overlapping subproblems, memoizing (storing) the answers to `solve(matrix[i][j])` is useful. Hence, a C program to do this might look like:

```
int memoTable[M_MAX][N_MAX]; // define these constants somewhere

int fillTable(int** matrix, int i, int j, int m, int n) {
    if(i >= m || j >= n) {
        return 0;
    }
    if(i == m - 1 || j == n - 1) {
        return 1;
    }
    if(matrix[i][j] == 0) {
        memoTable[i][j] = 0;
        return 0;
    }
    if(memoTable[i][j] != 0) {
        return memoTable[i][j];
    }
    memoTable[i][j] = fillTable(matrix, i + 1, j, m, n) +
                      fillTable(matrix, i, j + 1, m, n);
    return memoTable[i][j];
}

int solve(int** matrix, int m, int n) {
    return fillTable(matrix, 0, 0, m, n);
}
```

Problem 5

Suppose N people are in a tennis tournament. Each player has a fixed, distinct skill level. For any players x and y, x always beats y if and only if the skill level of x is greater than that of y. How many games are required to determine the best player? What about the second best player? (Anonymous company)

This is equivalent to drawing a binary tree with each node representing a player and each pair of children representing a matchup. Notice that the victorious player (also the best) will emerge at the root of the tree. Since there are n nodes, there must then be $n - 1$ edges and therefore $n - 1$ games to determine the best player. Now, notice that at some point, the best player must have played against the second best player, since nobody can beat the second best player other than the best player. The height of our binary tree (given that we construct it optimally) is $O(\log_2 n)$, and the height represents the matches that the first player plays. Hence, we filter among the players that the first player played, and we obtain $\lceil \log_2 n \rceil - 1$. Implementing a program to do this is trivial, and I will therefore not do this here.