COP 4516 Spring 2026 Week 10 Individual Contest #3: DFS/BFS
Solution Sketches

Affine Cipher

Consider the question: what would have to happen if two different values, $0 \le x_1 < x_2 < n$, with $f(x_1) = f(x_2)$? Set these expressions equal to each other and we get:

$ax_2 + b \equiv ax_1 + b \pmod{n}$
$a(x_2 - x_1) \equiv 0 \pmod{n}$

Thus, in order for an ordered pair (a, b) to be an invalid key, it must be the case that $n \mid a(x_2 - x_1)$, where $0 < x_2 - x_1 < n$. Notice that if gcd(a, n) = 1, then this can not happen.

Alternatively, if gcd(a, n) > 1, then we can set $x_2 - x_1 = \frac{n}{\gcd(a,n)}$ and get a solutions for x_1 and x_2 where $0 \le x_1 < x_2 < n$, with $f(x_1) = f(x_2)$. To prove that a specific solution exists, a valid solution is simply $x_1 = 0$ and $x_2 = \frac{n}{\gcd(a,n)}$.

Thus, we can conclude that a can only take on values where gcd(a, n). Specifically, a can be one of φ(n) values. We can choose b independently and there are no restrictions on b, thus, b can be one of n values, ranging from 0 to n-1, inclusive. Thus, the total number of valid keys is nφ(n). Since the input value of n ranges up to 10^9 and it's possible that φ(n) ~ n, while both n and φ(n) can be stored in ints, their product can be close to 10^{18} and must be calculated using longs (so cast one item to long or just store both as longs from the get go.) Also, we must use a reasonably efficient algorithm to calculate phi(n) which runs in $O(\sqrt{n})$ time. A program would get a time limit exceeded if it did a for loop from 1 to n checking the gcd of each value with n. Rather, the prime factorization formula given in class should be used. The prime factorization and the corresponding calculation of phi can be executed in $O(\sqrt{n})$ time by stopping looking for unique prime divisors when reaching the square root of n and taking care to not forget the last prime factor if it wasn't yet discovered.

Get Out of this Maze!!!

This problem is a more traditional BFS (than Eight Puzzle). Each of the vertices of the graph are grid squares, and the edges only connect adjacent grid squares up, down, left or right. We can simply create a two-dimensional integer array the same size as the grid to store distances for our BFS, initializing values to -1 (not visited). We want to cut out of our BFS as soon as we reach any border square. Whenever we dequeue a square, we want to process it by enqueuing any unvisited neighbors. For grids, using DX/DY arrays is the best way to iterate through all the neighbors of a square. One example was live coded in class: a Kattis problem called Grid. This reduces the chance of error since you don't have to do bounds checks four times over. (Though, in this problem you can get away without doing bounds checks because the whole border is guaranteed to be tilde characters…) One other key detail about a BFS is that you **have to mark the distance when you enqueue an item, not when you dequeue it.**

8 Puzzle

This is a classic game BFS. For each board position, we can calculate the other reachable board positions in one move. Then, for each position, we must store the distance (fewest number of moves to reach it), from the starting board. Instead of storing this distance in an array, it's easier to store it in a HashMap. Two ideas for the HashMap are as follows: (1) Store each board position as a string of 9 characters '0' - '8', where '0' is the empty square. (2) Store each board position as a single integer of 9 digits, again treating the empty square as 0. Thus, it makes sense to have a function that takes in the 2D array board and turns it into the string/integer or vice versa. Or, alternatively, just do all the index math and swap positions in the string or integer.

So, even after doing everything stated above, if you were to submit, you get Time Limit Exceeded. If you do the math, you find out that a BFS to solve one puzzle might take close to 4 x 9! steps. (This is because there are a maximum of 9! possible board positions to reach, and you have to check out 4 possible moves when exploring from each board position.) More accurately, since half of the board positions aren't reachable, it's really close to 2 x 9! steps. This is roughly a half million steps, which is fine for a single case. But, the input format explicitly states the following: "The first line of input will contain an integer N ($1 \leq N \leq 10000$) which represents the number of test cases. So, now, our total run time is $10000 \times 2 \times 9! \sim 7 \times 10^9$ steps, which is way too many for a programming contest.

But, one key realization is this: ***every move while solving a puzzle is reversible!!!*** So, what if, instead of starting at the initial puzzle, we start at a solved board, and try to make moves to all other board positions? Doing so is a single BFS. But in that BFS, we will find how far away each board position is from the ***ending board position.*** Thus, the key to solving this problem is the following:

Before you read in the input, run a **SINGLE** BFS from the ending board position to all other reachable positions. Store the distances in a HashMap as mentioned previously.
Then, when you read in the input, just provide the answer stored in the HashMap for how far from the solution board each input board is.

This is a critical idea in many shortest distance problems - if you want the shortest distance from many different places to one end place AND the edges are undirected, then flip your search and start from the ending spot and in one fell swoop you'll get all the distances from that place to all possible beginning locations, which is what you want!!!

Same Letters

Many ways to solve this using elementary techniques. The standard way would be to use a frequency array of size 26 keeping track of how many of each letter is in each word and then compare the frequency arrays. An even easier way would be to store each word as a character array, sort them using an API call, and then check if the two resultant strings are equal or not (another API call).