CS361 Homework #4 Solutions

1. Consider an implementation of Union-Find where I use Union-by-Size: that is, I look at the size of the two components, and always have the representative of the larger one become the parent of the representative of the smaller one. Prove inductively that the depth of a component that contains ℓ items is at most $\log_2 \ell$, even without any path compression. (The depth of a component consisting of a single isolated item is zero.)

   Proof. The base case is given by an isolated item, since then $\ell = 1$ and the depth is $d = 0 = \log_2 1$.

   For the induction step, we assume that the claim is true for all smaller sizes: that is, for all $k < \ell$, a component with k items has depth at most $\log_2 k$. Now consider what happens when we merge two components of size $j, k < \ell$ such that $j + k = \ell$; call the depths of these components d_j, d_k.

   Now, suppose that $j \leq k$ (the case $j > k$ is similar). Then we make the representative of the component of size k the parent of the representative of the component of size j (we are free to break ties this way). Then the depth of the new component is

   $$d_\ell = \max(d_j + 1, d_k)$$

   Now inductively, $d_j \leq \log_2 j$ and $d_k \leq \log_2 k$; but since $j \leq k$ and $j + k = \ell$, we have $j \leq \ell/2$. Thus

   $$d_j \leq \log_2 j \leq \log_2 \ell - 1$$

   and

   $$d_k \leq \log_2 k < \log_2 \ell \ .$$

   Thus $d_j + 1, d_k \leq \log_2 \ell$, and $d_\ell = \max(d_j + 1, d_k) \leq \log_2 \ell$. □

2. Consider a binary tree class defined as (in C++; in Java these pointers would be references instead)

```
class node {
  node *left; // NULL if no left daughter
  node *right; // NULL if no right daughter
  int key;
};
class tree {
  node *root; // NULL if empty
};
```

Write C++ or Java code (or clear pseudocode) for a recursive function `bool isSearchTree(tree t)` that determines whether `t` is a valid binary search tree.

Answer. A common mistake is to think that all we need to do is check for each node that its left daughter has a smaller key, and its right daughter has a larger key. But this isn't enough: we need to check for each node that the rightmost left descendant is smaller and its leftmost right descendant is larger.

Here's C++ code that works; there are, of course, many ways to write this. Recall that `if(pointer)` is the same as `if(pointer != NULL)`. Note that an empty tree is a valid binary search tree!

```
// return the key of n's leftmost descendant
int leftmost(node *n) {
  if (n->left) return leftmost(n->left);
  else return(n->key);
}

// return the key of n's rightmost descendant
int rightmost(node *n) {
  if (n->right) return rightmost(n->right);
  else return(n->key);
}

bool isSearchTree(node *n) {
  if (!n) return true; // empty trees are valid
  // check that this node has smaller stuff to its left and larger stuff to its right
  if (n->left && rightmost(n->left) > n->key) return false;
  if (n->right && leftmost(n->right) < n->key) return false;
  // now recursively check that the left and right subtrees are valid
  return (isSearchTree(n->left) && isSearchTree(n->right));
}

bool isSearchTree(tree &t) {
  return isSearchTree(t.root);
}
```

3. Suppose I have a binary tree. Define its *average depth* as the average depth of a leaf, where all the leaves are equally likely. (Let's say that the depth of a 1-node tree, consisting just of the root, is 0).

   Prove by induction, by using the fact that a binary tree consists of two subtrees, that the average depth of any binary tree with n leaves is at least $\log_2 n$.

   Answer. The base case is obvious, since a 1-node tree has (average) depth $0 = \log_2 1$.

   The induction step is a little challenging. First, suppose that the two subtrees have size n_1 and n_2, so the total number of leaves in the combined tree is $n = n_1 + n_2$. Inductively, we can assume that the subtrees' depths are $d_1 \geq \log_2 n_1$ and $d_2 \geq \log_2 n_2$. Then the average depth of the combined tree is the average weighted by the size of the subtrees, plus one:

$$\begin{aligned} d &= \frac{n_1}{n} d_1 + \frac{n_2}{n} d_2 + 1 \\ &\geq \frac{n_1}{n} \log_2 n_1 + \frac{n_2}{n} \log_2 n_2 + 1 \end{aligned}$$

   I found it useful to rearrange this a little. By adding and subtracting

$$\log_2 n = \frac{n_1}{n} \log_2 n + \frac{n_2}{n} \log_2 n$$

   on the right-hand side, we get

$$d \geq \log_2 n + \frac{n_1}{n} \log_2 \frac{n_1}{n} + \frac{n_2}{n} \log_2 \frac{n_2}{n} + 1 \ .$$

   Let's define $a = n_1/n$, and define the function

$$h(a) = -a \ln a - (1-a) \ln(1-a) \ .$$

   Then we have

$$d \geq \log_2 n + 1 - h(a)$$

   so the proof will be complete if we can show that $h(a) \geq 1$ for all a.

   There are several ways to prove this mathematically, but we can also see it just by graphing $h(a)$ with a ranging from 0 to 1. It has a peak where $a = 1/2$ (which means $n_1 = n_2 = n/2$, so the two subtrees are equal) where $h(1/2) = 1$, giving exactly $\log_2 n$ for the depth of the tree. For other values of a it is smaller than 1, and so the depth is greater than $\log_2 n$.

   (This function $h(a)$ is called the *entropy function*. It also showed up as 1 over the constant in front of $n \log_2 n$ in the solution to the recurrence

$$f(n) = f(an) + f((1-a)n) + n$$

   from homework #2.)

4. Let x, y, and z be three Boolean variables. Write the constraint

$$x \text{ OR } y = z$$

in SAT form, i.e., an AND ($\wedge$) of clauses, each of which is the OR ($\vee$) of two or three variables, which may be negated. Do the same thing for

$$x \text{ XOR } y = z \ .$$

Answer. We can translate x OR $y = z$ into SAT form as

$$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee z) \ .$$

The first clause forces z to be false if a and b are both false. The second and third clauses force z to be true if x or y is true.

Similarly, we can translate x XOR $y = z$ into SAT form as

$$(\overline{x} \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee y \vee z) \wedge (x \vee \overline{y} \vee z) \wedge (x \vee y \vee \overline{z}) \ .$$

There are four combinations of x, y, z which violate the equation x XOR $y = z$: namely, TTT, TFF, FTF, and FFT. Each one of these clauses prevents one of them.

5. Is this 2-SAT formula satisfiable?

$$(x \vee y) \wedge (\overline{y} \vee z) \wedge (\overline{x} \vee \overline{z}) \wedge (x \vee \overline{z})$$

Either prove that it is not, or find a solution. In the latter case, state whether or not it is the only solution, and justify your answer.

Answer. It is satisfiable, and the only solution is $x =$ true, $y =$ false, and $z =$ false. To see this, note that there is a path of implications from $\overline{x}$ to x, namely

$$\overline{x} \to \overline{z} \to \overline{y} \to x \ .$$

There's no path from x to $\overline{x}$, so this isn't a contradiction, but it does force x to be true. And there is a path from $x \to \overline{z} \to \overline{y}$, causing x and y to be false, and this satisfies all four clauses.

6. Write the multiplication table mod 7. For each nonzero element, what is its inverse? In other words, for each x, what is the "$1/x$" such that $x \cdot 1/x = 1$?

Answer. I'm using the professor's privilege to skip the multiplication table `;-)` but here is the list of inverses:

x	1	2	3	4	5	6
x^{-1}	1	4	5	2	3	6

7. What is $\log_3 4$ in the mod-7 world?

Answer. $3^4 \bmod 7 = 4$, and so $\log_3 4 = 4$.

8. An element x is called a *generator* for the integers mod p if its powers include all the nonzero integers mod p. Which integers are generators for the integers mod 13?

Answer. 2, 6, 7, and 11.

9. Describe how to calculate x^{1000} with much fewer than 1000 multiplications.

Answer. We can start by squaring x 9 times in order to get

$$x, x^2, x^4, x^8, x^{16}, x^{32}, x^{64}, x^{128}, x^{256}, \text{ and } x^{512} \ .$$

Then, since $1000 = 512 + 256 + 128 + 64 + 32 + 8$, we can calculate

$$x^{1000} = x^{512} x^{256} x^{128} x^{64} x^{32} x^8$$

with five more multiplications, for a total of 14.

10. Suppose you and I are doing Diffie-Helman key exchange. In order to exchange a 9-bit key, we publicly agree that we will operate mod 127, and that we will use 3 as a generator. I choose $a = 96$, and you choose $b = 40$. What numbers do we send back and forth, and what secret key do we end up with? Rather than just giving the answers, explain what computations we do, and try to do them with as few multiplications as possible.

    Answer. I wish to broadcast $g^a = 3^{96} \bmod 127$. Since $96 = 64+32$, I can do this with 7 multiplications, first squaring six times (taking mod 127 each time) to get

    $$3, 3^2 = 9, 3^4 = 81, 3^8 = 84, 3^{16} = 71, 3^{32} = 88, 3^{64} = 124$$

    and then multiplying 3^{32} and 3^{64} to get 117. Thus I send you 117.

    You in turn send me $g^b = 3^{40} \bmod 127$, which you can get with six multiplications by squaring five times and then combining 3^{32} with 3^8. This gives you 26, which you send me.

    Now, you take 117 and raise it to the 40th power in the same way, giving $117^{40} \bmod 127 = 47$. And I take 26 and raise it to the 96th power, giving $26^{96} \bmod 127 = 47$. Now we both share the key $(g^a)^b = (g^b)^a = 47$, without ever having revealed our secret exponents $a = 96$ and $b = 40$ to each other or to the eavesdropper.