Notes on how to solve Advanced contest problems of 3-21-15

A. The Fly

(a) Ignore the fly
We just need to calculate the time at which the bicycles collide

(b) Convert the problem to one bicycle by adding the speeds of the two bicycles.

$$T = distance/speed$$

$$T = D/(A + B)$$

Then compute the distance of the fly:

$$fly\ distance = T * F$$

B. Dijkstra Maze
Consider the following:

0	3	2	2	9
7	8	3	9	9
1	7	5	4	3
2	3	4	2	5

0	3	∞	∞	∞
7	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

The grid equates to a 4-connected undirected graph. Two neighboring grid cells with costs a and b represent two graph edges, one from a to b with weight a, the other from b to a with weight b.

Dijkstra's algorithm is **Greedy**. It adds 1 one graph vertex, or in this case, one grid cell to the *known set* on each iteration. Once a cell has been added to the known set, the shortest path to that cell never changes again - that's the algorithm's greediness property.

Call the grid on the left the *input grid*. Start with a *cost grid* as above on the right. I've inserted the cost of the top left cell, simply the value from the input grid. This cell is the new *known* cell added in this step. Each time a new known cell is selected, examine the costs of all its unknown neighbors. In this case, the top left cell has two unknown neighbors whose costs were infinity. If the cost of getting to any of those cells is smaller via the new known vertex, update that cost. That's how the 3 and 7 got inserted. The 3 is the cost of getting to the top left cell plus the cell's own input value of 3 from the grid on the left. The cells shown in light font are *candidates* to become the next cell added to the known set.

At each step choose one cell to add to the known set. The cell chosen is the candidate cell with the smallest cost.

We add the 3-cell to the known set and update the costs of getting to its neighbors:

0	**3**	5	∞	∞
7	11	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

The 5-cell has the smallest cost and is added next:

0	**3**	**5**	7	∞
7	11	8	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

Now we choose one of the 7's to add to the known set next (either may be chosen):

0	**3**	**5**	7	∞
7	11	8	∞	∞
8	∞	∞	∞	∞
∞	∞	∞	∞	∞

The other 7 is added next:

0	**3**	**5**	**7**	16
7	11	8	16	∞
8	∞	∞	∞	∞
∞	∞	∞	∞	∞

The process continues in this way, adding the smallest cost candidate cell to the known set, and recalculating the distances to its unknown neighbors

0	**3**	**5**	**7**	**16**
7	**11**	**8**	**16**	**25**
8	**15**	**13**	**17**	**20**
10	**13**	**17**	**19**	**24**

The result is 24.

C. Family Forests

You can use depth first search or breadth first search on the forest, restarting each time the search fails to find a successor, or you can use the Union-Find (Disjoint Set) data structure. It is very simple and quick to code. Here are some of my notes:

The disjoint set data structure has the following interface:

```
interface DisjointSet {
    void union(int x, int y);
    int find(int y);
    int numberOfSets(); // returns the number of disjoint sets remaining

}
```

The range of the integer keys is $[0, numElements-1]$, where there are $numElements$ disjoint sets. The idea is simple. The data structure is initialized with $numElements$ disjoint sets. Each call to $union(x, y)$ joins two sets together and each call to $find(x)$ indicates which set x belongs to. Once two sets are joined, they cannot be separated. Calls to $union()$ merge sets of elements to form larger and larger sets.

Example

```
disjointSet mySets = new disjointSet(10);  // creates a disjointSet object
   // for 10 elements numbered 0,1,...,9.
union(1,2);
union(3,4);
union(5,6);
union(2,5);
union(7,3);
```

After this sequence of unions, the data structure holds sets $\{0\}, \{1, 2, 5, 6\}, \{3, 4, 7\}, \{8\}, \{9\}$.

A call to $find(5)$ will return the same key as calls to $find(1)$, $find(2)$, and $find(6)$.

By convention, those calls all return the same key, a value from $\{1, 2, 5, 6\}$.

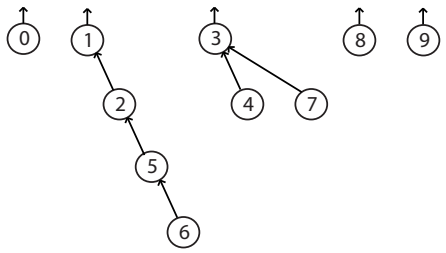
Similarly, calls to $find(3), find(4), find(7)$ all return the same key from $\{3, 4, 7\}$.

Implementation

The implementation is an array of integers. For example, the above disjointSet could be represented as follows:

Index	0	1	2	3	4	5	6	7	8	9
Contents	-1	-1	1	-1	3	2	5	3	-1	-1

And here is the data structure represented by the array:



The sets are represented by trees. Tree root elements contain -1 in the array, while children elements contain the index of their parents.

A call to $union(8, 5)$ may result in 1 (and its subtree) becoming a child of 8 or vice versa.

Similarly, a call to $union(4, 5)$ may result in 3 becoming a child of 1 or vice versa.

A Call to $find(x)$ returns the array index of the root of the subtree containing x. So, a call to $find(6)$ returns 1, a call to $find(1)$ returns 1, and a call to $find(7)$ returns 3.

A call to $union(x, y)$ first requires calls to $find(x)$ and $find(y)$ to discover the indices of the root nodes of the subtrees containing x and y. If those two values are different, the union process is completed with one change of an array index.

For most uses of this data structure you need not read any further, but two simple optimizations make its runtime incredibly small. The runtime of both $union()$ and $find()$ depend on the depth of the trees in the data structure. There are simple techniques that can be applied

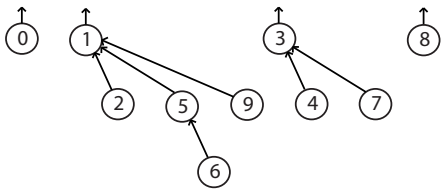
to the *union*() and *find*() functions that minimize tree heights. First we consider **Smart Union** algorithms.

Union by size

In this algorithm, a call to $union(x, y)$ causes the subtree with the fewest elements to become a subtree of the other. For example, with the trees in the above example, a call to $union(2, 4)$ would cause node 3 to become the child of node 1. The algorithm must know the number of elements in each of the trees. This is easily accomplished by making each tree root hold the negative of its size. Consider the following example:

```
disjointSet mySets = new disjointSet(10);  // creates a disjointSet
   \\ object for 10 elements numbered 0,1,...,9.
union(1,2);
union(3,4);
union(5,6);
union(2,5);
union(7,3);
union(6,9);
```

Index	0	1	2	3	4	5	6	7	8	9
Contents	-1	-5	1	-3	3	1	5	3	-1	1



When the two trees in a union operation are the same size (have the same number of elements), either root node may be chosen to be the new root node. This is the only occasion when the union function increases the height of a tree. In the worst case, trees are built in pairs, then those trees are unioned to form trees of size four and so on, finally resulting in a single tree (or set) of height $1 + p = 1 + log\, n$ for $n = 2^p$ nodes in the set.

This simple statement is an informal proof that both union and find operations have $O(log\, n)$ running time when union-by-size is used.

Path Compression

Path compression is a technique applied during a find operation. When a node, say x, is sought during a find operation, all parent pointers of the nodes along the path from x to the root are changed to point to the root.

In the worst case it is still possible to have nodes as deep at $O(log\, n)$ (when a sequence of unions occurs with no intervening find operations).

Theorem: *The amortized runtime of M union and find calls with smart union and path compression is* $O(Mlog^*\, n)$

Here, $log^* n$ is the number of times that the log of n may be taken before the result is 1 or less. For $n = 10^6 \approx 2^{20}, log^* n = 4$. For $n = 10^{30} \approx 2^{100}, log^* n = 5$.

Since $log^* n$ is such a small number it is regarded as a constant and the amortized running time of M union and find operations is considered to be $O(M)$.

D. Heads and Tails

The solution is straightforward. Use an array of 8 ints to collect the results. Develop a function `int convert(String s, int i)` that converts the three characters in s from position i to position i+2 to an int in [0,7]. Call this function in a for-loop 38 times, each time incrementing the result array in the position indicated by the function.

E. Adding Without Carries

Straightforward. Read each number as a decimal int. Convert each number to the given base, resulting in two arrays of ints. Do this by repeatedly dividing by the new base and saving the remainders in the result array:

Example:
to convert 1234_{10} to base 16 resulting in the int array A[ ]:

1234/16	$\rightarrow$	77 rem 2	A[0] = 2
77/16	$\rightarrow$	4 rem 13	A[1] = 13
4/16	$\rightarrow$	0 rem 4	A[2] = 4

Then add the two values digit by digit mod the new base;

A = [4, 13, 2]
B = [12, 6 8]
C = [0,3,10]

Convert the result (C[ ]) back to decimal and print the result: $0 * 16^2 + 3 * 16 + 10 = 58$

F. Dividing the Ranch

Fairly straightforward geometry:

Line Intersection

Two lines intersect unless they are parallel. If they are parallel, they have the same slope.

The intersection point, if $m_1 \neq m_2$, is found by using the formulation $y = m_1 x + b_1$, $y = m_2 x + b_2$:

$$x = \frac{b_2 - b_1}{m_1 - m_2}, \qquad y = m_1 x + b_1$$

Shoelace Theorem: Area of a Polygon

Given any simple polygon (no holes, edges do not cross each other) traverse its perimeter in clockwise or counter clockwise order placing the vertex coordinates into two arrays of doubles, one for x values, one for y values. Repeat the start point at the end:

$$\begin{array}{ll} x_1 & y_1 \\ x_2 & y_2 \\ \cdots & \\ x_n & y_n \\ x_1 & y_1 \end{array}$$

Then form products and sum them as follows:

$$\text{Area } = \frac{1}{2}\left[(x_1y_2 + x_2y_3 + ... + x_ny_1) - (y_1x_2 + y_2x_3 + \cdots + y_nx_1)\right]$$

G. Islands

Each time the new value read is larger than the previous value, increment a count. At the end of the input sequence, output that count.

```
// code for one dataset.
        int nisland = 0;
        int cur = 0;
        cur = scanner.nextInt();  // read from F.txt
        prev = cur;
        for(ind = 1; ind < N_TERMS ; ind++) {
            cur = scanner.nextInt();
            if(cur > prev)
                nisland++;
            prev = cur;
        }
        System.out.println( index + " " + nisland);
```

Problems solved:

Advanced							Novice							
A	B	C	D	E	F	G	H	I	J	K	L	M	N	Total
15	6	4	18	6	0	17	27	29	18	3	7	24	18	192