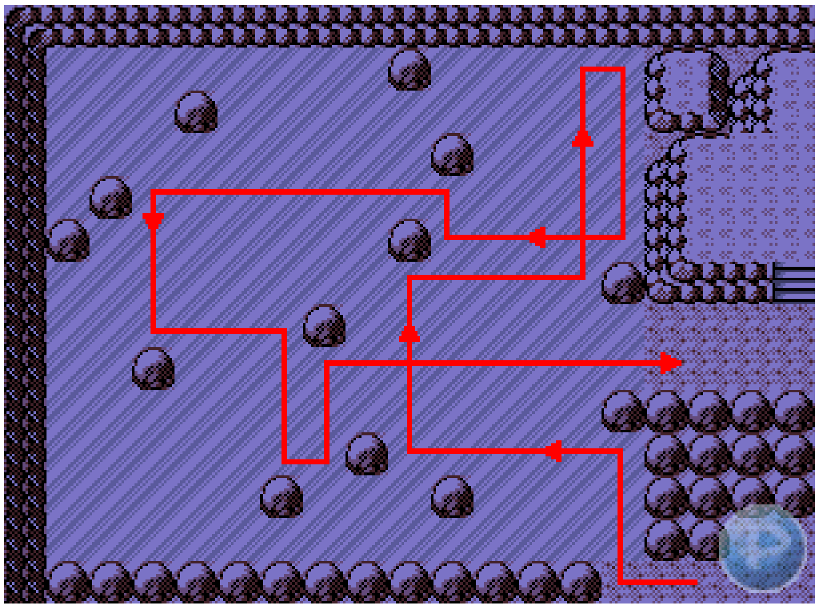
CS61B Spring 2016 Guerrilla Section 6 Worksheet Solutions

5 May 2016

Directions: In groups of 4-5, work on the following exercises. Do not proceed to the next exercise until everyone in your group has the answer and *understands why the answer is what it is.* Of course, a topic appearing on this worksheet does not imply that the topic will appear on the exam, nor does a topic not appearing on this worksheet imply that the topic will not appear on the midterm.

1 Pokemon Ice Cave Puzzle



You are Hash Ketchum. On your way to find the move tutor who can teach your Pikachu Mergesort, you find yourself lost in an ice cave! You need to figure out how to navigate from your starting position to the goal position with the fewest number of inputs. When you take a step (up, down, left, or right) you slide in that direction until you hit a rock or wall. Since we move until we hit something, we can ignore the intermediate steps between rocks/walls. Fill in the blank spaces below to find a sequence of moves to make that will allow you to make it through the cave!
Note: Enums can be considered as a set of user defined constants, and can be compared with the == operator.

```
// Represents an Ice Cave.  Has a fixed height and width as well as a double array of Squares
// describing what fills the cave.
public abstract class IceCave {
    Int height;
    Int width;
    Square[][] terrain;
    public boolean inBounds(Pos p);
}
// Represents a square of area in the ice cave.  You can travel over ice, but are stopped by rock.
// A square can be ice or rock, but never both.
public abstract class Square {
    boolean isIce();
    boolean isRock();
}
public class Position {
   int x;
   int y;
   public boolean equals(Position p);
}
public enum Direction {
    LEFT, UP, DOWN, RIGHT
}
// Given an ice cave, a position, and a direction, returns your new position after taking a step in
// that direction.
// Since we move until we hit something, we can ignore the intermediate steps between rocks/walls.
private static Position takeStep(Position pos, IceCave cave, Direction dir) {
    Position newPos = new Position();
    newPos.x = pos.x;
    newPos.y = pos.y;
    Position lastPos = new Position();
    while(cave.terrain[newPos.x][newPos.y].isIce() && cave.inBounds(newPos)) {
            lastPos.x = newPos.x;
            lastPos.y = newPos.y;

            if (dir == UP)
                newPos.y -= 1;
            else if (dir == DOWN)
                newPos.y += 1;
            else if (dir == LEFT)
                newPos.x -= 1;
            else
                newPos.x += 1;
    }
    return lastPos;
}
```

```
// Given an ice cave, starting position, and goal position, returns a sequence of positions that will
// take you from the starting position to the goal position with the fewest inputs.
// Since we move until we hit something, we can ignore the intermediate steps between rocks/walls.
public static LinkedList<Position> findInputs(IceCave cave, Position startPos, Position goalPos) {
    HashSet<Position> seen = new HashSet<Position>();
    seen.add(startPos);
    LinkedList<LinkedList<Position>> fringe = new LinkedList<LinkedList<Position>>();
    LinkedList<Position> startList = new LinkedList<>();
    startList.add(startPos);
    fringe.add(startList);
    while(fringe.size() > 0) {
        LinkedList<Position> path = fringe.pop();
        Position last = path.pollLast();
        if(seen.contains(last) {
            continue;
        }
        seen.add(last);
        if(goalPos.equals(last)) {
            Return path;
        }
       for (Direction dir : Direction.values()) {
            LinkedList<Position> newPath = path.clone();
            newPath.add(takeStep(last, cave, dir));
            fringe.add(newPath);
       }
    }
    return null;
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

2 Data Structures Potpourri

For each of the following, give a data structure or algorithm that you could use to solve the problem or write "impossible" if it is impossible to meet the running time given in the question. For each question, we have one or more right answers in mind, all of which are among the data structures and algorithms listed below. For each answer, provide a brief description of how the algorithm or data structure can be used to solve the problem.
Possible answers:

DFS	Heap	Quick sort	Impossible
BFS	Trie	Merge sort	
Dijkstra's	Hash table	Insertion sort	
Topological Sort	Balanced BST	Radix sort	
Kruskal's	Linked list	Selection Sort	

(a) Given a list of N words with k characters each, find for each word its longest prefix that is the prefix of some other word in the list. Worst-case running time $O(Nk)$ character comparisons.
(1) Trie: construct a trie of the words then find the longest prefix for each word by finding the parent of each special end node for a word (or last node with more than one child along path to end of word) or
(2) Radix sort all the words then compare each word with the one sorted before and after it to find the longest prefix

(b) Given an undirected, weighted and connected graph with $|E|$ edges, find the heaviest edge that can be removed without disconnecting the graph. Worst-case running time: $O(|E|log|E|)$.
Run Kruskal's to find an MST, then take the heaviest edge not in that MST

(c) We would like to build a data structure that supports the following operations: add, remove and find the k^{th} largest element for any k. Worst-case running time: $O(log(N))$ comparisons for each operation (where N is the number of elements currently in the data structure).
You can use a Balanced BST, but remember the number of nodes in the subtree for each node.

(d) Given an unordered list of N Comparables, construct a BST containing all of them. Running time: $O(N)$ comparisons.
This is impossible, because this would be tantamount to performing a comparison sort in linear time.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

3 Hashing

The (unmemoized) hashCode method for the String class is as follows:

```
public int hashCode() {
    int h = 0;
    for (int i = 0; i < length() ; i++) {
        h = 31 * h + charAt(i);
    }
    return h;
}
```

For parts (a) and (b), assume that a HashSet uses the value of this function by taking it modulo the number of buckets, after first masking off the sign bit to make the number non-negative (The actual HashSet and HashMap implementations do something more sophisticated.). The other parts of the problem are disjoint from (a) and (b).

(a) Given a String of length L and a HashSet that contains N Strings, give the worst- and best-case running times of inserting the String into the HashSet.
Best: O(L)
Worst: O(NL)

(b) In Java, HashSets always use arrays whose size is a power of two. If this were not the case, the hashCode method shown above could be a very poor hash function. Give an example of an integer N such that hashCode would be a very poor choice of hash function for a HashSet whose array had size N. Give a brief explanation of your answer. Assume that the Java HashSet uses external chaining to resolve collisions.
31 would be bad, since modding this by 31 would lose all information except that of the last character.

(c) What is a collision in a hash table? Select the best definition below.
• Two key-value pairs that have equal keys but different values.
• Two key-value pairs that have different keys and hash to different indices.
• **Two key-value pairs that have different keys but hash to the same index.**
• Two key-value pairs that have equal keys but hash to different indices.

(d) Suppose that your hash function does not satisfy the uniform hashing assumption. Which of the following can result? Select all that apply. For each one selected, give a brief explanation of why it would result.
• Poor performance for insert. True, because long external chains would take a while to check if the key was already present.
• Poor performance for search hit. True, because you would have to search long external chains.
• Poor performance for search miss. True, because you would have to search long external chains.

(e) Suppose that instead of using a linked list for our external chaining, we instead use another HashMap. List some disadvantages of this approach.
- More overhead
- It would suffer from the same problems as the outer hashmaps for bad hashcodes
- On average it would have very little speedup, since our standard HashMap already has an expected constant length to each external chain
- The memory requirements for a HashMap in every bin far outweigh any slightly speedup you would get

(f) What if we don't trust the user to write a good hashCode function for their objects? What improvements could we make to try and mitigate this problem?
We could run their hashCode output through a hashcode of our own designed to distribute integers. See the (private) hash() function of java's HashMap on grepcode for an example.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

4 Symmetric Tree

Complete the following program by filling in the blank line on this page and adding any necessary code to the next page.

```
public interface Tree {
    /** Return true iff I contain X. */
    boolean contains(int x);

    /** Insert X into me, if not already present, returning the result. */
    Tree insert(int x);

    /** The empty tree, containing nothing. [NOTE: static methods
     *  in interfaces is a Java 8 feature.] */
    static Tree emptyTree() {

        return new EmptyTree();
    }
}

public final class RegularTree implements Tree {
    /* NOTE: Final classes cannot be extended. */
    /** A Tree containing label LAB and the contents of trees L and R,
     *  where all elements of L are < LAB and those of R are > LAB. */
    RegularTree(int lab, Tree L, Tree R) {
        _label = lab; _left = L; _right = R;
    }

    @Override
    public boolean contains(int x) {
      if (x == label) { return true; }
      else if (x < _label) { return _left.contains(x); }
      else { return _right.contains(x); }
    }

    @Override
    public Tree insert(int x) {
        if (x < _label) {
            _left = _left.insert(x);
        } else if (x > _label) {
            _right = _right.insert(x);
        }
        return this;
    }

    private int _label;   private Tree _left, _right;
}
```

Continued on next page

Add any additional code here. It **may not** contain any **if** statements, **while** statements, **switch** statements, conditional expressions, or **try** statements.

```
public class EmptyTree implements Tree {

        public boolean contains(int x) {
                Return false;
        }

        public Tree insert(int x) {
                Tree left = new EmptyTree();
                Tree right = new EmptyTree();
                return new RegularTree(x, left, right);
        }
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

5 Merging 2 Lists Iterator

Given 2 sorted arrays (all elements are integers), return an iterator that gives you the elements in sorted order.

```
public class MergingIterator {

    Int cur_index_0 = 0;
    Int cur_index_1 = 0;

    public MergingIterator(int[]  given0, int[] given1) {

    }

    public int next() {
        If (cur_index_0 >= given0.length) {
                cur_index1++;
                return given1[cur_index_1-1];
        } else if (cur_index_1 >= given1.length) {
                cur_index0++;
                return given1[cur_index_0-1];
        } else if  (given0[cur_index_0] < given1[cur_index_1]) {
                cur_index_0++;
                return given0[cur_index_0-1];
        } else {
                cur_index_1++;
                return given1[cur_index_1-1];
        }

    }

    public boolean hasNext() {
        return (cur_index_0 < given0.length) || (cur_index_1 < given1.length);
    }
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

6 Disjoint Trivia

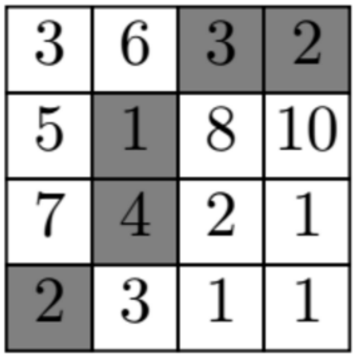
(a) Explain how the disjoint sets data structure could be used in an implementation of Kruskal's MST algorithm. Consider efficiency. Would you actually want to use this data structure in this algorithm? Explain.
Part of Kruskal's involves checking if an edge being added will connect two vertices that are already in the same component. Disjoint sets can be used to do this. Yes, this is a reasonable data structure to use, because weighted quick union objects have very good worst-case performance on union and find operations.

(b) [True/False] To improve the efficiency of joins on two trees, we keep track of the height of the two trees and always link the root of the shorter tree to the root of the taller tree.
False - We keep track of the number of elements in trees and link the root of the smaller tree to the root of the larger tree to improve efficiency.

(c) Extra - Assume we are using disjoint sets with path compression. How many calls to find() need to be made in order for each node to be directly connected to the root node?
We need to call find as many times as the # of leaves in the disjoint sets tree.

7 Dynamic Programming

Consider an $N \times N$ grid whose cells contain integer values. We wish to find a shortest path from the lower-left corner of the grid to the upper-right corner, where a path consists of a sequence of squares, each one of which is adjacent to the preceding square in the path and either above, to the right, or diagonally above and to the right of the preceding square. The length of the path is the sum of the integers in its squares. For example, the shaded path shown below is shortest:



Here's a pseudo-code function to find the length of the shortest path from row i, column j to row N, column N, for $1 \leq i \leq N$, $1 \leq j \leq N$, in an $N \times N$ grid G. $G(r, c)$ is the value at row r, column c of G (numbering from (0, 0) in the bottom left corner. Unfortunately, its running time of `lsp` is exponential.

```
def lsp(i, j):
    if i >= N or j >= N:
        return infinity
    if i == N-1 and j == N-1:
        return G(i, j)
    else:
        return G(i, j) + min(lsp(i+1, j), lsp(i, j+1), lsp(i+1, j+1))
```

a. Fill in the program below to compute the result in polynomial time into `L[0][0]`. You may assume you have a multi-argument `min` function available, as in Python.

```
int[][] L = new int[N][N];
L[N-1][N-1] = G(N-1, N-1)
for (int c = N-2; c >= 0; c -= 1) {
    L[N-1][C] = G(N-1, C) + L[N-1][C+1]
}

for (int r = N - 2; r >= 0; r -= 1) {
    L[r][N-1] = G(r, N-1) + L[r+1][N-1];
    for (int c = N-2; c >= 0; c -= 1) {
        L[r][c] = G(r,c) + min(L[r+1][c], L[r][c+1], L[r+1][c+1]);
    }
}
```

b. As a function of N, what is the worst-case execution time of this program? $\Theta(N^2)$