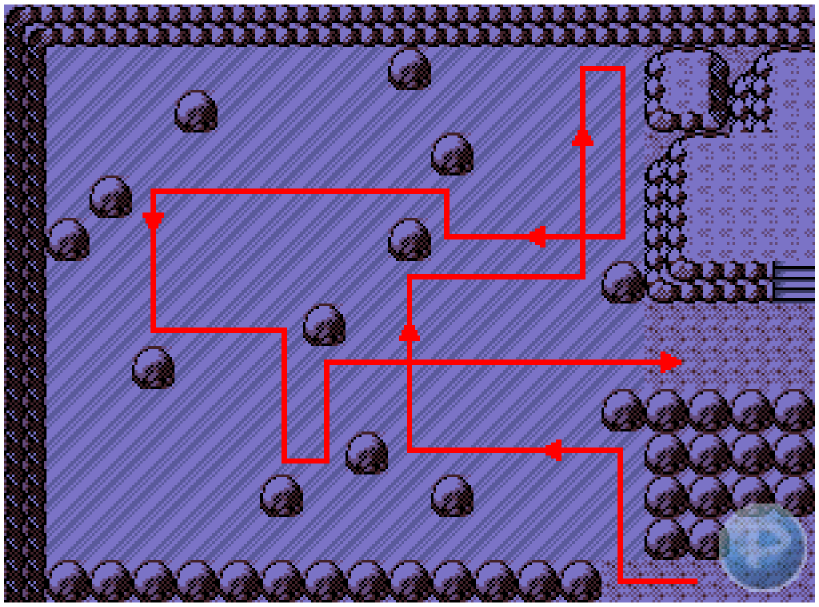
CS 61B Spring 2017 Guerrilla Section 7 Solution

29 April 2017

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 Slip and Slide



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!

Given an ice cave and a starting position, return a sequence of positions that will take you from the starting position to the goal position in the fewest inputs. Since we move until we hit something, we can ignore the intermediate steps between rocks and walls.

To refer to the left direction, for example, use `Direction.LEFT`. To iterate over all four directions, use `Direction.values()`.

```java
public interface IceCave {
    /** Return the terrain of the level, a matrix of positions. */
    Position[][] getTerrain();

    /** Return whether the position P is in bounds. */
    boolean inBounds(Position p);

    /** Slide in the given direction until an obstacle is hit. */
    Position move(Position p, Direction d);
}

public interface Position {
    boolean isIce();
    boolean isRock();
    boolean isGoal();
}

public enum Direction {
    LEFT, UP, DOWN, RIGHT
}

public LinkedList<Position> solve(IceCave cave, Position start) {
    Queue<LinkedList<Position>> fringe = new LinkedList<>();
    Set<Position> seen = new HashSet<Position>();

    LinkedList<Position> startList = new LinkedList<>();
    startList.add(start);
    fringe.add(startList);

    while (!fringe.isEmpty()) {
        LinkedList<Position> path = fringe.remove();
        Position last = path.pop();
        if (last.equals(goal)) {
            return path;
        } else if (!seen.contains(last)) {
            for (Direction d : Direction.values()) {
                LinkedList<Position> newPath = path.clone();
                newPath.add(cave.move(last, d));
                fringe.add(newPath);
            }
            seen.add(last);
        }
    }
    return null;
}
```

STOP!

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

2 Data Structures & Algorithms 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
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 search tree, but remember the number of nodes in the subtree for each node.

(d) Given an unordered list of N `Comparable` objects, construct a binary search tree 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:

```java
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 slight speedup

STOP!

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

4 Synthetic Trees

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

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

    /** Insert X, if not already present, and return the resulting tree. */
    Tree insert(int x);

    /** The empty tree, containing nothing. */
    static Tree emptyTree() {
        return new EmptyTree();
    }
}

public final class RegularTree implements Tree {
    /* NOTE: Final classes cannot be extended. */
    private int label;
    private Tree left, right;

    /** 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) {
        this.label = lab;
        this.left = L;
        this.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;
    }
}
```

Add any additional code on the next page. It **may not** contain any `if` statements, `while` statements, `switch` statements, conditional expressions, or `try` statements.

```java
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);
  }
}
```

5 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 or 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.

STOP!

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

6 Sorting Mechanics

Below, the **leftmost column** is an unsorted list of strings. The **rightmost column** gives the same strings in sorted order. Each of the remaining columns gives the contents of the list during some intermediate step of one of the algorithms listed below. Match each column with Merge sort, Quicksort, Heap sort, LSD radix sort, or MSD radix sort. For quicksort, choose the topmost element as the pivot. Use the recursive (or top-down) implementation for merge sort.

	A	B	C	D	E	F	G
1	4873	1876	1874	1626	9573	2212	1626
2	1874	1874	1626	1874	7121	8917	1874
3	8917	2212	1876	1876	9132	7121	1876
4	1626	1626	1897	4873	6973	1626	1897
5	4982	3492	2212	4982	4982	9132	2212
6	9132	1897	3492	8917	8917	6152	3492
7	9573	4873	4873	9132	6152	4873	4873
8	1876	9573	4982	9573	1876	9573	4982
9	6973	6973	6973	1897	1626	6973	6152
10	1897	9132	6152	3492	1897	1874	6973
11	9587	9587	7121	6973	1874	1876	7121
12	3492	4982	8917	9587	3492	9877	8917
13	9877	9877	9132	2212	4873	4982	9132
14	2212	8917	9573	6152	2212	9587	9573
15	6152	6152	9587	7121	9587	3492	9587
16	7121	7121	9877	9877	9877	1897	9877

From left to right: unsorted list, quicksort, MSD radix sort, merge sort, heap sort, LSD radix sort, completely sorted.

MSD Look at the left-most digits. They should be sorted. Mark this immediately as MSD.

LSD One of the digits should be sorted. Start by looking at the right most digit of the remaining sorts. Then check the second from right digit of the remaining sorts and so on. As soon as you find one in which at least something is sorted, mark that as LSD.

Heap Max-oriented heap so check that the bottom is in sorted order and that the top element is the next max element.

Merge Realize that the first pass of merge sort fixes items in groups of 2. Identify the passes and look for sorted runs.

Quick Run quicksort using the pivot strategy outlined above. Look for partitions and check that 4873 is in its correct final position.

STOP!

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

7 Trieing to Find Partial Matches

Given a list of N input words all of length at most k and M query words, we would like to find, for each query word, the number of input words that match the first $k/2$ letters of the query word. Describe an algorithm that accomplishes this and give its running time as a function of N, M, and k.

Solution 1: Use a trie where every node of the trie tracks how many of its descendants are complete words. Insert all N input words into such a trie. This takes $O(Nk)$ time (assuming a constant alphabet size). Then for each query word, find the node in the trie corresponding to the word's first $k/2$ letters and output the number stored in that node (and output 0 if there is no such node). This takes $O(k)$ time for each query word. So the entire algorithm takes $O((N + M)k)$ time (note that in the best case, all of the query words begin with some letter that is not in the trie at all, in which case this solution will run in $O(Nk+M)$ time).

Solution 2: Use a HashMap where the keys are strings of length $k/2$ and the values are integers representing how many of the input words begin with those $k/2$ letters. For each input word x, look up the length $k/2$ prefix of x in the HashMap. If it is present, increment the corresponding value. If it is not present, add it as a key with corresponding value of 1. Then for each query word, check if its length $k/2$ prefix is present in the HashMap. If so, output the corresponding value. If not, output 0. In the worst case, inserting the input words into the HashMap takes $\Theta(N2k)$ (because in the worst case all input words hash to the same bucket and comparing the inserted word to all previously inserted words takes up to $O(Nk)$ time because we have to compare the first $k/2$ letters of each word) and similarly, looking up all the query words takes $\Theta(MNk)$ time. So in the worst case this solution will take $\Theta((N + M)Nk)$ time. If we assume however that the keys are distributed uniformly among the buckets then this solution is also $\Theta((N + M)k)$ time.

STOP!

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

8 T9

You're living in the future in the late 1990's. Everyone owns a cell phone — the handheld kind — and SMS is the bee's knees. However, cell phones only have 9 keys while there are 26 letters in the alphabet. Your company is on the verge of developing a new algorithm for faster texting called "Text on 9 keys", or "T9". Each keypress maps to any one of three or four letters in the alphabet.
Given a trie containing the dictionary of words, define a procedure, `getWords`, that returns the set of matching words for a given key-press sequence.

```
public interface TrieNode {
    public char getCharacter();
    public boolean isWord();
    public String getWord();
    public Map<Character,TrieNode> getChildren();
}
public class T9 {
    private static char[][] KEY_MAPPINGS = {
        {}, {}, // keys 0 and 1 don't map to any characters
        {'a', 'b', 'c'},
        {'d', 'e', 'f'},
        {'g', 'h', 'i'},
        {'j', 'k', 'l'},
        {'m', 'n', 'o'},
        {'p', 'q', 'r', 's'},
        {'t', 'u', 'v'},
        {'w', 'x', 'y', 'z'}
    };
    public static Set<String> getMatches(int[] keyPresses, TrieNode words) {
        return getMatches(keyPresses, words, 0);
    }
    private static Set<String> getMatches(int[] keyPresses, TrieNode wordNode, int
        index) {
        Set<String> matches = new HashSet<String>();
        if (index == keyPresses.length) {
            if (wordNode.isWord()) {
                matches.add(wordNode.getWord());
            }
            return matches;
        }
        Map<Character,TrieNode> children = wordNode.getChildren();
        for (Character c : KEY_MAPPINGS[keyPresses[index]]) {
            if (children.containsKey(c)) {
                matches.addAll(getMatches(keyPresses, children.get(c), index + 1));
            }
        }
        return matches;
    }
}
```

STOP!

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