

## Test 2 Review

### PROBLEM 1 : (*Short Ones*)

- A. Why should you use a `HashMap` rather than a `TreeMap`? Choose the best answer.
- I. To conserve memory.
  - II. Because the performance of `get` and `put` is better
  - III. Because the Java library implementation is optimized.
  - IV. To guarantee that lookups (`get`) will be faster than inserts (`put`).
  - V. Because it supports a fast sort implementation.
- B. Suppose that I take a sorted linked list of  $N$  integers, and break it into a list of  $M$  equal-sized sorted lists of integers (that is, put the first  $N/M$  integers into the first list, the next  $N/M$  into the second, etc.). What is the worst-case time for finding whether an integer  $x$  is in anywhere in this list of lists? If for some fixed  $N$ , you can choose  $M$ , what  $M$  should you choose for maximum lookup speed?
- C. In order to use the class `Point` containing fields `x` and `y` in a `HashSet`, you are considering multiple hash functions. Of these hash functions, which one would give the best performance in a `HashSet`? Assume that your points are likely to be between  $(0, 0)$  and  $(1280, 1024)$  (the size of the average computer monitor).
- A. `public int hashCode () { return super.hashCode(); }`
  - B. `public int hashCode () { return 42; }`
  - C. `public int hashCode () { return x; }`
  - D. `public int hashCode () { return x + y; }`
  - E. `public int hashCode () { return x * 3 + y; }`
  - F. `public int hashCode () { return x * 1000 + y; }`
- G. Given the following recursive method `mystery`:

```
int mystery(int n)
{
    if (n < 0)
        return -mystery(-n);
    else if (n < 10)
        return n;
    else
        return mystery(n/10 + n % 10);
}
```

What are would following calls evaluate to?

- I. `mystery(7)`
  - II. `mystery(-512)`
- H. True or False** State whether the following statement is true or false. If false, you should give a specific counterexample.
- I. A certain hash table contains  $N$  integer keys, all distinct, and each of its buckets contains at most  $K$  elements. Assuming that the hashing function and the equality test require constant time, the time required to find all keys in the hash table that are between  $L$  and  $U$  is  $O(K \times (U - L))$  in the worst case.
  - II. Instead of using a heap, we use a *sorted ArrayList* to represent a priority queue. The worst-case big-Oh of *add* and *poll* do not change.

**PROBLEM 2 : (Hash (6 pts))**

In the Markov assignment, the `WordNgram` class encapsulated  $N$  words/strings so that the group of  $N$  words can be treated as a key in a map.

```
public class WordNgram implements Comparable<WordNgram>{

    private String[] myWords;

    [ methods deleted ]
    /**
     * Return true if this N-gram is the same as the parameter: all words the same.
     * @param o is the WordNgram to which this one is compared
     * @return true if o is equal to this N-gram
     */
    public boolean equals(Object o){
        WordNgram wg = (WordNgram) o;
        if (myWords.length != wg.myWords.length)
            return false;
        for(int k = 0; k < myWords.length; k++){
            if (!myWords[k].equals(wg.myWords[k]))
                return false;
        }
        return true;
    }
}
```

In this problem, you will answer questions about possible implementations of `hashCode`.

- A.** Will `get` and `put` still work in a `HashMap` if the `WordNgram`'s `hashCode` implementation is as follows? Explain why or why not. What will be the average run-time for `get` and `put` if there are  $n$  `WordNgrams` stored in the `HashMap`.

```
public int hashCode(){
    // TODO return a better hash value
    return 15;
}
```

B. Why is the following *hashCode* implementation flawed?

```
public int hashCode(){
    int result = 0;
    for(int k = 0; k < myWords.length; k++)
        result += myWords[k].hashCode();
}
```

**PROBLEM 3 : (Reverse (9 points))**

Each of the Java functions on the left take a string *s* as input, and returns its reverse. For each of the following, state the recurrence (if applicable) and give the big-Oh complexity bound.

Recall that concatenating two strings in Java takes time proportional to the sum of their lengths, and extracting a substring takes constant time.

A. 

```
public static String reverse1(String s) {
    int N = s.length();
    String reverse = "";
    for (int i = 0; i < N; i++)
        reverse = s.charAt(i) + reverse;
    return reverse;
}
```

B. 

```
public static String reverse2(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse2(right) + reverse2(left);
}
```

C. 

```
public static String reverse3(String s) {
    int N = s.length();
    char[] a = new char[N];
    for (int i = 0; i < N; i++)
        a[i] = s.charAt(N-i-1);
    return new String(a);
}
```

**PROBLEM 4 : (Boggle)**

The game of Boggle is usually played using sixteen letter cubes. A letter cube can be represented as a string of length 6, one character for each face on the cube. Given a list of letter cubes and a target word, you want to determine whether it is possible to spell that word using those letter cubes, where each cube can be used at most once in spelling the word.

Examples: given the list of cubes {"etaoin", "shrdlu", "qwerty"}, it is possible to spell the words "as" and "law" but not "weld" or "toe".

Write a recursive, backtracking method `canSpell` that takes a target word along with an `ArrayList` of letter cubes. The function should return `true` if it is possible to spell the word using the cubes in the list and `false` if otherwise. Assume that the word and letter cubes will only contain lowercase letters.

```
public static boolean canSpell(String word, ArrayList<String> cubes)
{
```

**PROBLEM 5 : (Puzzle Hunt)**

You are given a matrix of positive integers to represent a game board, where the (0, 0) entry is the upper left corner. The number in each location is the number of squares you can advance in any of the four primary compass directions, provided that move does not take you off the board. You are interested in the total number of distinct ways one could travel from the upper left corner to the lower right corner, given the constraint that no single path should ever visit the same location twice.

Consider the initial game board to the left, and notice that the upper left corner is occupied by a 2. That means you can take either two steps to the right, or two steps down (but not two steps to the left or above, because that would carry you off the board). Suppose you opt to go right so that you find yourself in the configuration to the right.

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

After that, you could continue along as follows:

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

  

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

This series of moves illustrates just one of potentially several paths you could take from upper left to lower right. Your task is to write a method called `numPaths`, which takes a 2-d array of integers and computes the total number of ways to travel to the lower right corner of the board. Note that you never want to count the same path twice, but two paths are considered to be distinct even if they share a common sub-path. And because you want to prevent cycles, you should change the value at any given location to a zero as a way

of marking that you've been there. Just be sure to restore the original value as you exit the recursive call. You may want to write a helper function to handle the recursion and a utility function to decide if you are on the board or not.

A. Write numPaths below.

```
/**
 * Calculates total number of distinct ways one could travel from the
 * upper left corner of grid to the lower right corner, given the
 * constraint that no single path should ever visit the same location twice.
 *
 * @param board square matrix board[i][j] is the number of squares
 * one can advance vertically or horizontally from (i,j)
 *
 * @return the number of possible paths from (0,0) to the lower
 * right corner of board (board.length-1, board[0].length - 1)
 */
public static int numPaths(int[][] board)
{
```

```
// HELPER FUNCTIONS
/**
 * @return true if (row,col) is within the bounds of the board
 * (i.e. 0 <= row < board.length and 0 <= col < board[0].length)
 * false otherwise
 */
public static boolean onBoard(int[][] board, int row, int col)
{
```

```
/**
 * @return the number of possible paths from (row,col) to the lower
 * right corner of board (board.length-1, board[0].length - 1)
 */
public static int numPaths(int[][] board, int row, int col)
{
```

**PROBLEM 6 :** (*Nothing in Common (15 points)*)

The *Longest Common Subsequence* or *lcs* of two strings is the longest sequence of characters in order, not necessarily adjacent, that is in common to both strings. This has applications in text processing, genomics, and web searching.

For example, the *lcs* of the strings “sorting” and “describe” is “sri” and the *lcs* of the strings “human” and “chimpanzee” is “hman”.

The code below on the left correctly returns the longest common subsequence of two strings.

```
public String lcs(String a, String b){
    if (a.length() == 0 || b.length() == 0){ // CASE 1
        return "";
    }

    // placeholder A

    if (a.charAt(0) == b.charAt(0)){ // CASE 2
        String after = lcs(a.substring(1),b.substring(1));
        after = a.charAt(0) + after;

        // placeholder B

        return after;
    }
    // CASE 3
    String t1 = lcs(a.substring(1),b);
    String t2 = lcs(a,b.substring(1));
    if (t1.length() > t2.length()) return t1;
    return t2;
}
```

31	ting	ibe
37	ing	be
37	ting	be
46	ting	e
48	ing	ribe
51	ng	cribe
78	g	cribe
79	ing	ibe
83	ing	e
99	ng	ibe
99	ng	ribe
177	g	ribe
215	ng	be
276	g	ibe
298	ng	e
491	g	be
789	g	e

**Part A (3 points)**

Describe in words what the three cases in the code are and why these cases make the code correct.

**Part B (4 points)**

What is the running time of this code for two strings of *N*-characters? Use big-O and justify your answer.

**Part C (8 points)**

The list of words on the left of the previous page shows some of the calls for the method `lcs` for the words “sorting” and “describe” – the list shows the two parameters passed to `lcs` and the number of calls with these parameters. For example, there are 491 calls with parameters “g” and “be”. For the strings “sorting” and “describe” there are 6,238 calls made to find the longest common subsequence of “sri”.

To make the method faster you will *memoize* so that results are stored and retrieved rather than being recomputed. For this problem you’ll describe how to implement memoization and how to make it work. The idea is to make at most one recursive call for each pair of parameters. To do this you’ll use the class `Pair` below.

The idea is to modify `lcs` so that the result returned for parameters `(a,b)` as a `Pair` is stored in a map and retrieved if it is stored rather than recomputed recursively.

You'll do three things for this part of the problem:

- Comment on the definition for the map which would be an instance variable.
- Show how to check the map and return the stored value if it's present.
- Show how to store a value in the map so it will be available for subsequent calls.

Using this memoization technique reduces the calls from 6,238 to 100 for the strings “sorting” and “describe”.

```
public class Pair implements Comparable<Pair>{
    String a;
    String b;
    public Pair(String s, String q){
        a = s;
        b = q;
    }
    public int hashCode(){
        return a.hashCode()*100+b.hashCode();
    }
    public boolean equals(Object o){
        Pair p = (Pair) o;
        return a.equals(p.a) && b.equals(p.b);
    }
    public int compareTo(Pair o) {
        int f = a.compareTo(o.a);
        if (f == 0) return b.compareTo(o.b);
        return f;
    }
    public String toString(){
        return "("+a+", "+b+")";
    }
}
```

The definition for the map follows:

```
Map<Pair,String> myMemo = new HashMap<Pair,String>();
```

### C.1

Can you replace `HashMap` with `TreeMap` in the definition above and have the rest of the code work (after modifying `lcs` to use the memo/cache)? Justify your answer.

### C.2

Put code in `lcs` at the location marked *placeholder A* to check the cache and return the result for a `Pair p` defined as:

```
Pair p = new Pair(a,b);
```

### C.3

Put code at the location marked *placeholder B* so that a value is stored properly in the memo/cache for subsequent retrieval. Where else in the code would you also have to store values in the map (label on the `lcs` code page).

**PROBLEM 7 :** (*Stacks and Queues (6 points)*)

- A.** Describe the contents of stack `s` after the method `convert` executes. That is, describe the contents in a general manner based on what's in `s` before the code executes.

```
public void convert(Stack<Object> s){
    ArrayList<Object> list = new ArrayList<Object>();
    while (s.size() > 0) {
        list.add(s.pop());
    }
    for(Object o : list) {
        s.push(o);
    }
}
```

- B.** What happens if a queue is used instead of a stack in the code above, e.g.,

```
public void convert(Queue<Object> q){
    ArrayList<Object> list = new ArrayList<Object>();
    while (q.size() > 0) {
        list.add(q.remove());
    }
    for(Object o : list) {
        q.add(o);
    }
}
```

**PROBLEM 8 :** (*Tradeoffs (8 points)*)

You are given an array of  $n$  ints (where  $n$  is very large) and are asked to find the largest  $m$  of them (where  $m$  is much less than  $n$ ).

- A.** Design an efficient algorithm to find the largest  $m$  elements.

You can assume the existence of all data structures we discussed in class. You *do not* have to explain how any of the standard methods (e.g. constructing a heap) work. Be specific, however, about which data structures you are using and how these data structures are interconnected.

Your algorithm should work well for all values of  $m$  and  $n$ , from very small to very large.

- B.** What is the running time of your algorithm? What is it for small  $m$ ? What is it as  $m \rightarrow n$  (i.e. as  $m$  approaches  $n$ )?