Computer Science 345
Test 4 Practice problems
April 20, 2023

The test itself will have three questions, one each from the categories of dynamic programming (Chapter 6), hash tables (Chapter 7), and strings (Chapter 8). This set of practice problems includes two problems from each category.

Pull from the repository. You will find the starter code for these practice problems in a folder called `test4-practice`, and I recommend you make an Eclipse project in that folder. A with the practice problems for test 2, code for each problems will be in its own package.

A solution to these problems can be found in the folder `test4-practice-soln`.

1. [**Dynamic programming.**] See Project 6.1.a in the textbook (pg 591). This is the "Hero-Hall" problem. The method to finish is `HeroHall.bestTreasure()` in the `herohall` package. Use `herohall.HHTest` to test.

2. [**Dynamic programming.**] Ronald Howard introduced the concept of a *micromort*, which is a one-in-a-million chance of dying[1]. Micromorts (abbreviated μ☠) are used to assess the risk involved in various activities. For example, skydiving carries a risk of 8μ☠ per jump, while climbing the Matterhorn carries a risk of 2840μ☠ per ascent attempt.

Suppose you want to minimize your risk of dying while traveling about a city whose streets are laid out in a rectangular grid. Using data based crime crime rates, traffic accidents, etc, you have calculated the risk that each intersection carries. For example, suppose you wanted to travel from point (0,0) (lower corner) to point (3, 2) (upper corner) in the grid below. The safest route, which is shown, has a risk of 2.3 μ☠.

	0	1	2	3
2	3.8	1.75	0.12	0.1
1	0.01	2.9	1.1	0.8
0	0.2	0.7	0.08	1.5

Assume that you are considering routes that always move you closer to your destination—you are unwilling to take a roundabout route even if it is safer. Thus your route will consist only of moves to the right and up in the grid.

Let M be a table that represents the risk of each intersection such that $M[i][j]$ is the number of μ☠ carried by intersection (i, j). Let $R[i][j]$ be the number of μ☠ on the safest route from starting point $(0, 0)$ to intersection (i, j), including the risk of intersection (i, j) itself. The recursive characterization of this problem is

[1] Ronald Howard, *On Making Life and Death Decisions*, 1980.

$$R[i][j] = M[i][j] + \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ R[0][j-1] & \text{if } i = 0 \text{ and } j > 0 \\ R[i-1][0] & \text{if } i > 0 \text{ and } j = 0 \\ \min(R[i-1][j], R[i][j-1]) & \text{otherwise} \end{cases}$$

Assume the destination is the top right corner. Thus if the grid is $n \times m$, then the top-level problem is $R[n-1][m-1]$.

Finish the method `micromort.Micromort.findFewestMicromorts()`, which takes a two-dimensional array of `double`s representing the risks at each intersection and returns the risk of the safest path. (The risks tend to be very small—less than 1μ☠ for many intersections.)

Use `micromort.MRTest` to test.

3. [**Hashing.**] In an open-addressing hashtable, define a key's *penalty* to be the number of positions beyond its ideal place that need to be inspected in order to find that key. For example, in the hash table below, suppose all three keys `A`, `B`, and `C` hash to position 3:

			A	B	C														

In this case, `A`'s penalty is 0, `B`'s penalty is 1, and `C`'s penalty is 2.

Complete the function `averagePenalty()` in class `avePenalty.StatisticalHashtable` which returns the average penalty for all keys in the table. Assume linear probing and that there are no breaks in any chain (the class, in fact, does not even support removal). For example, in the table shown above, the average penalty is $(0 + 1 + 2) \div 3 = 1$.

Use `avePenalty.SHTest` to test.

4. [**Hashtables.**] Consider a hashtable that uses an approach similar to open addressing but with a *two-dimensional* table for keys (and a parallel table for values). A key is stored in a position indicated by two indices (i, j). For example, in the key table below of size 5×5, the keys `AXB`, `AXC`, `BXA` are stored in positions $(1, 3)$, $(2, 3)$, and $(2, 4)$, respectively. Their values are stored in the same positions in a separate table, not shown.

4			BXA		
3		AXB	AXC		
2					
1					
0					
	0	1	2	3	4

To find a key's ideal place in this array, the two-dimensional hash table uses two hash functions, a "horizontal hash" and a "vertical hash," which compute the i-coordinate and j-coordinate. In this example, suppose the keys mentioned above have the following horizontal and vertical hashes:

key	**horizontal hash**	**vertical hash**
`AXB`	1	3
`AXC`	2	3
`BXA`	1	3

In the case of a collision, the probe strategy searches for an alternative location by alternating between incrementing i and j (and wrapping around as necessary) until finding the key being searched for or an empty position. In this example, `BXA` collides with `AXB` in position $(1, 3)$; the probe strategy then looks at $(2, 3)$; since that position is filled by `AXC`, the probe strategy then looks at $(2, 4)$, which is where `BXA` is. If $(2, 4)$ were filled with a different key, then the probe would have continued with $(3, 4)$, and then wrapped around vertically to $(3, 0)$.

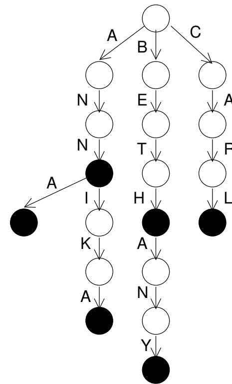
Implement this approach by writing the methods `put()`, `get()`, and `containsKey()` in the class `twoDHash.TwoDHashMap`. (For simplicity, there are no `remove()` or `iterator()` methods.) It is recommended that you finish the private method `find()` which can then be used as a helper method for `put()`, `get()`, and `containsKey()`.

Assume that the table is square, $n \times n$; the dimension size is stored in instance variable `n`. The horizontal and vertical hashes are computed by methods `hashHorz()` and `hashVert()`. Note that although these methods return indices in $[0, n)$, the code implementing the probe strategy will still need to mod when wrapping around.

Use `twoDHash.TDHMTest` to test.

5. [**Strings.**] Consider the reduced form of the class `TrieSet` provided in package `trie2Arry`. Specifically, it has no `remove()` method and no separate `TrieNode` class. Instead of internal nodes, a `TrieSet` is itself a node, having other `TrieSet`s as children.

Consider the problem of converting a set implemented as a trie into a sorted array of its keys. For example, the following trie



would produce the array `[ANN, ANNA, ANNIKA, BETH, BETHANY, CARL ]`.

This can be done recursively if the method is given the array to be filled (the code calling the method would create a big enough array to begin with), a starting position in the array (indicating how much of the array has already been filled with keys that come before the keys in the current subtrie), and a prefix for the keys in the subtrie (indicating the characters on the path to this subtrie), as in this method signature:

```
int trie2Array(String[] keys, int start, String prefix)
```

For example, for the node in the trie above with incoming link labeled `T`, the method would be called with `start` having value 3 (because the strings `ANN`, `ANNA`, and `ANNIKA` have already been put in the array) and `prefix` having value `BET` (because all the keys in that subtrie begin with `BET`).

Write the method `trie2Array.TrieSet.trie2Array()`, which populates the `keys` array and returns the number of keys added to the array by that subtrie.

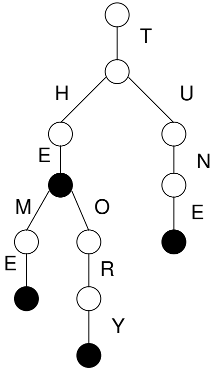
(Hint, for what it is worth: Although this requires a fair amount of thinking, it does not require very much coding. My solution is only six lines long.)

Use `trie2Array.T2ATest` to test.

6. [**Strings.**] Consider the class `trieTermRatio.TrieNode` which represents a nnode such as could be used in a trie implementing a set. In particular, the class has instance variables `children`, which contains links to children indexed by character adjusted to 0, and `isTerminal`, which indicates whether this node is the end of a route for a key in the set. All other set or map operations (besides `add()`) have been removed.

Write the static method `terminalRatio())` in class `triTermRatio.TermRatio` which takes a `TrieNode` and computes what proportion of nodes are terminal in the trie

rooted at the given node. In other words, it computes the *ratio* of terminal nodes to all nodes. For example, the trie below contains keys `THE`, `THEORY`, `THEME`, and `TUNE`:



Since this tree has four terminal nodes out of twelve nodes total, its terminal ratio is $\frac{4}{12} = \frac{1}{3} = 0.33333333333$.

(Hints: I can think of at least three equally good ways to do this. Also, since the number of terminals and the number of nodes are `int`s, make sure you cast correctly when computing the ratio, which is a `double`.)

Use `trieTermRatio.TRTest` to test.