HW2 Philip Tierney 9/26/2023

1. **BSP (2 pt)**
   * Read <https://developer.valvesoftware.com/wiki/Binary_space_partitioning>
   * Write a program to construct a BSP tree for the following shape (from the reading above):  
       
     A white rectangular object with a red border

     Description automatically generated
   * Your program should do the following:
     + Allow a user to pick a starting line
     + Construct the tree
     + Print out the in-order traversal of the tree
   * Here is an example of the tree constructed from the shape above, starting from the top line:  
     A diagram of a diagram

     Description automatically generated
   * Explain your code in English on high-level
   * You can use Java or Python
   * Upload your code to a GitHub repo and include the link to your repo
   * **BONUS [up to 2pt]:** do the collision detection (this could be tricky… so don’t waste too much time if you think this is too hard or if you’re not interested)
   * Extra readings that might confuse you (but could also be fun):
     + <https://www.youtube.com/watch?v=nbi88hY9hcw>
     + <https://www.geeksforgeeks.org/binary-space-partitioning/>
     + <https://commons.apache.org/proper/commons-geometry/tutorials/bsp-tree.html>
     + <https://www.fotonixx.com/posts/2d-binary-spacing-partitioning-with-python-and-networkx/>
     + <http://groups.csail.mit.edu/graphics/classes/6.838/F01/lectures/BSP/BSP2D.pdf>
     + <https://www.cs.cornell.edu/courses/cs410/1999fa/Handouts/BSP.htm>
     + <https://twobithistory.org/2019/11/06/doom-bsp.html>

**Part 2: Rotating Strings (2pt)**

**Solution to Hackerrank problem via Clojure:**  
(defn rotations [s]

  (let [n (count s)

        concatenated-string (str s s)]

    (map (fn [i] (subs concatenated-string (+ i 1) (+ i n 1))) (range n))))

(defn main []

  (let [t (Integer. (read-line))] ; Number of test cases

    (doseq [\_ (range t)]

      (let [s (read-line)

            rots (rotations s)]

        (println (clojure.string/join " " rots))))))

(main)

**Solution to problem with Java:**  
 import java.util.Scanner;

public class RotateString {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int t = scanner.nextInt();

scanner.nextLine(); // Consume newline

for (int i = 0; i < t; i++) {

String s = scanner.nextLine();

printRotations(s);

}

}

public static void printRotations(String s) {

for (int i = 1; i <= s.length(); i++) {

System.out.print(s.substring(i) + s.substring(0, i) + " ");

}

System.out.println();

}

}

**Pseudo code:**

Function main:

Read t (number of test cases)

For i from 1 to t:

Read string s

Call printRotations(s)

Function printRotations(string s):

For i from 1 to length of s:

Print s starting from i concatenated with s up to i

Print newline

**Why algorithm is correct:**  
The algorithm takes a string and prints its rotations. The idea is to "cut" the string at every possible position and "swap" the two parts. For instance, for the string "abc", the first rotation cuts between "a" and "bc" to produce "bca". The second rotation cuts between "ab" and "c" to produce "cab". The final rotation doesn't cut at all and just produces the original string "abc".

By iterating over every position in the string and performing this "cut and swap", we generate all possible rotations. Since we handle every position and the string operations are reliable, the algorithm is correct.

**4. Runtime Analysis**

Let's analyze the time complexity:

* Reading the string takes time O(n), where "n" is the length of the string.
* For each rotation, we perform two substring operations. Each of these operations takes time O(n). Since we perform this operation "n" times (once for each rotation), the total time for all rotations of one string is O(n times n) or O(n^2).

For a single string, our algorithm runs in time O(n^2). If we have "t" test cases, the worst-case time complexity is O(t \* n^2). However, since each string can have a different length, it's more accurate to say the time complexity is O(n^2) for each string.

The space used by our algorithm is O(n), because we only store the original string and its rotations temporarily.

**Part 3: ID Search**

Alright, let's tackle this problem. Given the constraints, we can use the Boyer-Moore Voting Algorithm(Source: <https://medium.com/@tomas.svojanovsky11/boyer-moore-algorithm-mastering-efficient-string-searching-51312fe5098b> . It's a linear time algorithm that's perfect for this problem.

*Algorithm*:

1. Initialize a candidate ID as None and a count as 0.

2. Traverse through the list of IDs:

- If the count is 0, set the current ID as the candidate.

- If the current ID is the same as the candidate, increment the count.

- Otherwise, decrement the count.

3. The candidate ID at the end of the traversal is the ID that appears the most.

*Pseudocode*:

```

function findMajorityID(A: list of IDs) -> ID:

candidate = None

count = 0

for id in A:

if count == 0:

candidate = id

if id == candidate:

count += 1

else:

count -= 1

return candidate

```

Explanation:

The algorithm works by maintaining a count of the current candidate ID. If a new ID is encountered, the count is decremented. If the count reaches 0, a new candidate is chosen. Given the constraint that there exists an ID that appears more than 50% of the times, this algorithm will always result in that ID being the final candidate.

Why it's correct:

The key insight is that if there's an ID that appears more than 50% of the time, then it will have a net positive count after traversing the entire list. Every time another ID appears, it decrements the count of the majority ID, but since the majority ID appears more than all other IDs combined, its count will never reach zero after its first occurrence, ensuring it's selected as the candidate.

Efficiency:

The algorithm runs in O(n) time, where n is the number of IDs in the list, making it highly efficient. The space complexity is O(1) since it uses a constant amount of space regardless of the input size.