Assignment 1 - Coding and Complexity

Due: Sat Jan 25, 2025 @ 9:59pm

Jayden Cruz

2. (text) Common Substring [15 points]

```
//Problem 2: Common Substring (text)
public static String commonString(String text1, String text2) {
  StringBuilder sb = new StringBuilder(); //a string is immutable, StringBuilder allow you to add.
  int maxLength = 0;

  //System.out.println(sb); //make sure that cS is empty
  //boolean[] isLength = new boolean[text2.length()];

  //reusing loops from problem1
  for (int i = 0; i < text1.length(); i++) { // O(n)
    for (int j = 0; j < text2.length(); j++) { // O(m)
      int length = 0;
      while (i + length < text1.length() && j + length < text2.length()
          && text1.charAt(i + length) == text2.charAt(j + length)) {
        length++;
      }
      if (length > maxLength) {
        maxLength = length;
        sb = new StringBuilder(text1.substring(i, i + length));
      }
      //isLength[i] = true;
      // break; not need bc of while loop;
    }
  }
  return sb.toString();
}
```

- modify string
- reused code from Problem one
- add while loop with logic that check how fare the two strings match starting at positions
- want the longest common Substring

text1.indicies

[While loop logics: i + length < text1.length() & j + length < text2.length() & text1.charAt(i+length) == text2.charAt(j+length)]

- Similar to problem1: just need to add code to meet extra conditions.

6. Algorithm Analysis [20 points]

outer loop: O(n) — inner loop: O(m)

Problem 1: Worst-case Big-O: $O(n \cdot m)$, where n = text1.length(), m = text2.length, nested loop but they dont rely on each other. 3 operations → O(1) constants

Best-case: Big-Ω: $\Omega(1)$, when both text1 & text2 match in every character so text1 = "abc" & text2 = "abc".

Problem 2: Worst case: $O(n \cdot m)$ (know this from Problem 1.) • min amount while loop checks (i & j) — 3 nested loop but NOT n^3 bc it what we know from 1.

Best Case: $\Omega(n \cdot m)$, these are not common letter so while loop breaks early but we still need to check all pairs i & j (text1 & text2)

Problem 3: I used 2 methods. notFib(): worst case $O(2^n)$ → we have 2 guaranteed recursive calls notFib(n-1) & notFib(n-2) the recursive tree is built each call. each level the number of call doubles. $O(2^n)$ n is the input size, # of sequence you want.

Base Case: $\Omega(2^n)$, no matter what the input is it still has to go throug return notFib(n-1) + notFib(n-2) → recursive method: notFibonacci(). same as notFib() bc it just calling this recursive method to save to long[] resultArr

↳ $O(2^n) == \Omega(2^n)$ so we can say $\Theta(2^n)$ Big theta:)

Problem 4: does a recursive call to notFib(i) → has time complexity of $O(2^n)$, while goes until notFib(i) is > or == n. $T(n) = O(2^n) + O(2^1) + O(2^2) + \dots O(2^x)$ just a Σ (sum-series) while loop O(n). Best Case is O(1) when input = 0 loop only goes once. Worst Case is O(n) loop goes until notFib(i) > or = n.

Problem 5: just has 1 for loop where it runs the length of the array given. Best Case: O(n), none of elements == target, one pass through array. (not elements copied)

Worst Case: O(n) where n = elements in the array. whole array is gone through

↳ $O(n) == \Omega(n)$ so we can say $\Theta(n)$ Big theta:)

Problem: Extra Credit (text)

code:

```
long[] extra = notFibonacci(1000)
System.out.println("Extra Credit: ") + Arrays.toString(extra));
```

(# of the notFib sequence you want.) → code needed for this BUT will not work!

each time the notFib is called it creates a recursive tree, as seen in class, creats over & over again but this case 1000 times that's why it would run forever if uncommented. & the tree depths get bigger/deeper each time. Output also grow exponentially In CompOrg we learned about integer overflow, this occurs when int is larger than the max value variable type can hold.

- java int (-2147483648 to 2147483647)
- java long (-9223372036854775808 to 9223372036854775807)

- additional comment about time complexity also left in code file.)