SIG Algorithm Challenges

Week 1: SIG Introduction + String Manipulation

Dane's Contact Email: dziema2@uic.edu

Join our Slack channel, #sig_algorithm_chall on the UIC ACM slack

SIG Introduction

General meeting structure: Introduce a type of common coding challenge, present an example problem of this type, then have everyone work through a different problem of this type on their own or in groups. A solution will be presented and discussed at the end of the meeting. Example of types of challenges include:

- String Manipulation
- Hashtable
- Recursion
- Trees
- Greedy Algorithms
- Dynamic Programming
- Graphs

All experience levels are welcome, but completion or current enrollment in CS 251 is recommended.

Mock technical interviews will be available by appointment.

All materials from meetings will be posted on github.com/dane8373/SIG_Algorithm_Challenges

Best Practices

First rule of coding challenges: There is no “I can’t solve this problem”, only “I can’t solve this problem yet”.

DON’T:

- Come in just to see solutions to problems
  - Or even worse: not attend and just look at the solutions online after they are posted
- Try to memorize solutions
- Look at a problem, think “I could totally do that” and then not attempt it
- Attempt a problem for 2 minutes, give up, and search for an answer on google

DO:

- Work through the problems
- Write out a solution on paper first
  - Leads to fewer errors
  - Better practice for interviews
- Struggle through difficult problems
  - You learn the most from problems you cannot solve yet

On Difficulty

Often times I will refer to a problem as “easy”, “medium” or “hard”, here is what I mean when I say that.

Easy: Someone who has been exposed problems of this type will be able to quickly identify what type of problem it is, and write an errorless implementation in 5-10 minutes

Medium: Either someone who has been exposed to problems of this type will be able to quickly identify what type of problem it is, but it will take 15-20 minutes of troubleshooting to write a good implementation, or it would take 5-10 minutes of thinking about the problem to identify an approach to solve it, but after finding this approach it could be coded rather quickly.

Hard: These problems either have underlying algorithms that take care to implement correctly, or they are very difficult to break down and create an approach, but can be solved in a moderate amount of time if you are experienced with that type of approach.

All problems are hard if you don’t know how to solve them

- Don’t be discouraged if you can’t solve an “easy” problem!

Detailed meeting structure

Introduction (Presented by me, 10-20 minutes)

- Introduction to a class of coding challenges
- Work through an easy-medium example of this type of challenge

Application (Done by everyone, 20-30 minutes)

- Everyone works through a problem of this type (either individually or in a small group)
    - I will generally prepare 2-3 questions with at least one easy-medium problem and one medium-hard one
    - I will be modulating the difficulty based off how everyone performs and hopefully be able to challenge everyone

Solution presentation (Presented by me, 10-20 minutes)

- Present and work through a solution to the problems we worked on

Mock Technical Interviews

- Held by appointment
- Currently, Me or Karol will administer them
- Performed on a whiteboard
    - Location will vary based off what is available
- Held in 30 minutes time blocks
    - Generally will either be 1 problem that is expected to be 30 minutes, or multiple problems that take 10 minutes each
    - May be shorter for non-seniors
- Will take some time to get them up and going (need to build up a base of questions to ask)
- Please do not share the questions asked with other students
- Please Indicate on the sign in sheet if you would potentially be interested in this service

String Manipulation

Identifying String Manipulation Problems

Compared to most other problem types, string manipulation problems are very easy to identify

- Generally the prompt will be of the form“Given a string with X property do task Y”
- Popular with interviews due to their short problem descriptions

More often than not, these problems are accompanied with runtime requirements

- Basically every string manipulation questions has a brute force O(n^2) or O(n^3) algorithm
- A lot of string manipulation questions have clever O(n) algorithms, rarely faster

Often times string manipulation problems are combined with a more advanced concept, and this allows their difficulty to range from easy to hard

- A lot of string manipulation problems use HashMaps, some use dynamic programming
- This property also makes them popular with interviewers
- For this meeting, we will focus on ones that don’t explicitly require those techniques

General Tips for String Manipulation

- Know the various properties of strings in your preferred language
  - Strings are mutable in C++, not in Java or Python
  - Need to use operators like getCharAt() in Java
  - May need to use the StringBuilder class in Java

- Know the available methods for you language (the docs for classes are a great help!)
  - Substring(), find() etc. in java
  - Beware of the runtimes of these functions (find is O(n) for instance)
  - Other useful functions such as Character.toLowerCase() in Java or tolower in C/C++

- Starting with brute force can sometimes help (just remember to optimize later)
- Always try to think if there is a way to do the problem in a single pass

Sample String Manipulation Problem

Given a string `S`, return the "reversed" string where all characters that are not a letter stay in the same place, and all letters reverse their positions.

Example 1:

```
Input: "ab-cd"
Output: "dc-ba"
```

Example 2:

```
Input: "a-bC-dEf-ghIj"
Output: "j-Ih-gfE-dCba"
```

Source: https://leetcode.com/problems/reverse-only-letters/

Problem Approach

Ideas for how to finish this problem

1: Keep track of two locations, one at the front of the string, one in the back

2: Check the character at each location, if they are both letters, swap them, otherwise move to the next location

3: Continue doing this until the two locations are equal

Pseudocode for this problem

```
startIndex = 0

endIndex = s.length()

while (endIndex > startIndex)

        if (s[startIndex] and s[endIndex] are both letters)

                swap(s[startIndex], s[endIndex])

                increase startIndex and decrease endIndex

        else if (s[startIndex] is not a letter)

                increase startIndex

        else

                decrease endIndex

return s
```

Java Code for this problem

```
class Solution {
    public String reverseOnlyLetters(String S) {
        StringBuilder sb = new StringBuilder(S);
        int front = 0;
        int back = S.length() - 1;
        while (front < back) {
            if (!Character.isLetter(sb.charAt(front))) {
                front++;
            }
            else if (!Character.isLetter(sb.charAt(back))) {
                back--;
            }
            else {
                char temp = sb.charAt(front);
                sb.setCharAt(front++, sb.charAt(back));
                sb.setCharAt(back--, temp);
            }
        }
        return sb.toString();
    }
}
```

String Manipulation Problems to work on

Links to level 1-3 can be found at https://github.com/dane8373/SIG_Algorithm_Challenges/week1

Level 1) Write a function that takes a string as input and reverse only the vowels of a string.

Level 2) Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

Level 3) Given a string s, find the longest palindromic substring in s (palindromic = word is same spelled backwards and forwards)

Level 4) Same question as Level 3 but with an O(n^2) runtime requirement

Level 5) Same question as Level 3 but with an O(n) runtime requirement (Super hard)

Problem Approach for Level 1 problem

This problem is very similar to the problem we worked through

1: Keep track of two locations, one at the front of the string, one in the back

2: Check the character at each location, if they are both vowels, swap them, otherwise move to the next location

- Slightly more difficult, as there is no isVowel() function in most libraries

3: Continue doing this until the two locations are equal

Java Code level 1 problem

```
class Solution {
    public String reverseVowels(String s) {
        Set<Character> vowels = new HashSet<Character>();
        vowels.add('a');
        vowels.add('e');
        vowels.add('i');
        vowels.add('o');
        vowels.add('u');
        int front = 0;
        int back = s.length() - 1;
        StringBuilder sb = new StringBuilder(s);
        while (front < back) {
            //keep advancing the front pointer until we get to a vowel
            while (front < back && !vowels.contains(Character.toLower(sb.charAt(front))) {
                front++;
            }
            //keep advancing the back pointer until we get to a vowel
            while (front < back && !vowels.contains(Character.toLower(sb.charAt(back))) {
                back--;
            }
            //always swap, we either got to a vowel and want to swap
            //or there were no vowels and front == back and we do a harmless swap
            char temp = sb.charAt(front);
            sb.setCharAt(front++, sb.charAt(back));
            sb.setCharAt(back--, temp);
        }
        return sb.toString();
    }
}
```

Problem Approach for Level 2 problem

Observation 1: The fewest possible repetitions is 2, therefore, we do not have to test any substrings longer than n/2

Observation 2: The length of the original string must be divisible by the length repeated string

Observation 3: if the string is a repeated string, then that repeated string must be a preffix of the original string

1: Starting with the prefix of length 1, see if the original string is a repetition of this prefix

2: If it is, return true. If it is not, add one more character to the prefix and check again

3: Repeat up until the prefix of length n/2, if all attempts fail then return false

Java Code for level 2 problem

```java
class Solution {
    public boolean repeatedSubstringPattern(String s) {
        String testString = ""; //candidate for repeated string
        for (int i=0; i<s.length()/2; i++) { //try all repeated strings of length <= n/2
            testString += s.charAt(i); //append a character to the candidate string
            if(s.length() % testString.length() != 0) { //if the strings original length isn't divisible by the candidate length
                continue;
            }
            int currentLetter = 0; //store the location in the repeated string we are checking
            boolean ret = false;
            for (int j=i+1; j<s.length(); j++) { //check all the letters after the candidate string
                ret = true;
                if(s.charAt(currentLetter) != s.charAt(j)) { //if any character breaks the pattern of the repeated string
                    ret = false;
                    break;
                }
                currentLetter++;
                if (currentLetter >= testString.length()) { //if we reach the end of the repeated string
                    currentLetter = 0;
                }
            }
            if (ret) { //if we didn't fail at any point in the above test
                return true;
            }
        }
        return false;
    }
```

Problem Approach for Level 3 problem

This problem can be solved using a brute force approach

1: Starting from the front, find the longest palindromic substring starting at that index

2: If this substring is the longest we have seen so far, store it

3: Continue doing this until we have tried starting from all positions

Java Code for level 3 problem

```
class Solution {
    public String longestPalindrome(String s) {
        String testString = ""; //string we will check to see if is palindrome
        String maxString = ""; //longest palindrome so far
        for (int i=0; i< s.length(); i++) {
            testString = ""; //reset the palindrome for every start index
            for (int j=i; j<s.length(); j++) {
                testString+=s.charAt(j); //add one chacaracter to the test string
                if (testString.length() > maxString.length()) { //don't bother testing if the test string wouldnt be longest anyway
                    boolean isPalindrome = true;
                    for (int k=0; k<testString.length(); k++) { //this for loop checks to see if the teststring is a palindrome
                        if (testString.charAt(k) != testString.charAt(testString.length()-1-k)) {
                            isPalindrome = false;
                            break;
                        }
                    }
                    if (isPalindrome) { //if this is a plaindrome it is the longest
                        maxString = testString;
                    }
                }
            }
        }
        return maxString;
    }
}
```

Improving Level 3 solution to Level 4

The presented level 3 solution is O(n^3), but this can be improved to O(n^2) by observing the following

Observation: if you know that a string from index i to index j is a palindrome, then a longer palindrome can be formed by adding the character at index i-1 and index j+1 if the characters at index i-1 and j+1 are equal

- This make this problem a great candidate for dynamic programming

Pseudocode

For every ending index j from 1 to s.length

    For every i from 0 to j-1

        If the substring from index i+1 to j-1 is a palindrome and the characters at index i and j are equal

            Store the fact that the substring from i to j is a palindrome

            If this is the longest palindrome we have seen store it

Java Code for level 4 problem

```java
//I wrote out a top down recurrence and stole this code from one of the comments because I was lazy
class Solution {
    public String longestPalindrome(String s) {
    int length = s.length();

    if (s == null || length < 2) { //input checking, any string with length <2 is a palindrome
        return s;
    }
    //boolean array used to store whether or not the substring starting and index i and ending at index j is a substring
    boolean[][] isPalindromic = new boolean[length][length];

    int left = 0, right = 0; //the left and right indecies of the longest substring we have found

    for (int j = 1; j < length; j++) {
        for (int i = 0; i < j; i++) {
            //check to see if the substring from index i+1 to j-1 is a palindrome
            boolean isInnerWordPalindrome = isPalindromic[i + 1][j - 1]
                    || j - i <= 2;
            //if the substring from index i+1 to j-1 is a plaindrome and the characters at i and j are equal
            //then the substring from i to j is a palindrome
            if (s.charAt(i) == s.charAt(j) && isInnerWordPalindrome) {
                isPalindromic[i][j] = true;
                //if this palindrome is bigger than the biggest one we have found then store it
                if (j - i > right - left) {
                    left = i;
                    right = j;
                }
            }
        }
    }
        //java substring method gets cranky if you use an end index too high
        if (right == s.length()) {
            return s.substring(left);
        }
        //java syntax dictates the +1 one since substring(i,j) gives only the characters from i to j-1.
        return s.substring(left,right+1);
    }
}
```

Improving Level 3 solution to Level 5

Use Manacher's Algorithm

- This is done using a similar approach of expanding a known palindrome substring
- I didn't write a solution for this approach, nor did I expect anyone to solve it
- This is too hard for you to ever receive in an interview
- If you are curious there is a geeksforgeeks guide on this algorithm

https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/

Next Week: Recursion