**[redacted] Introduction to Data Structures**

**Addison L. Sears-Collins**

[redacted]

**Lab 1 Analysis**

**Due Date: June 29, 2017**

**Dated Turned In: June 28, 2017**

# Lab 1 Analysis

## Commentary

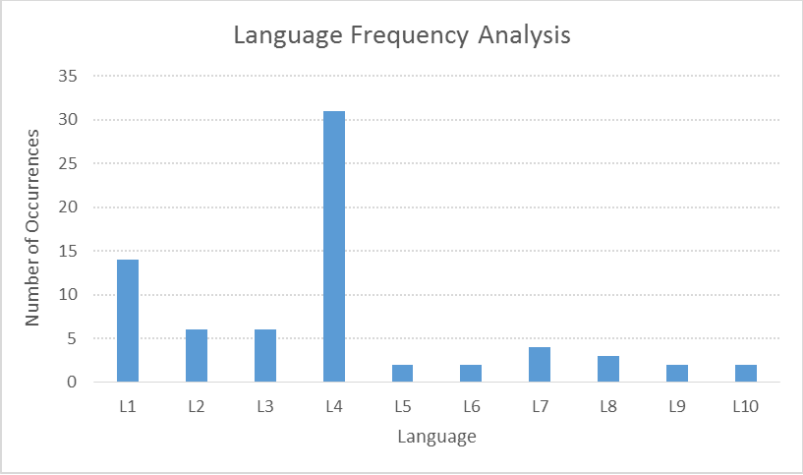
The application that I developed defines the "string rules" for 10 different languages and uses stacks in order to determine if a given string is valid in one or more of those languages.

An analogous real world example to the program would be the "Detect Language" feature on Google Translate. You input a string, and the program returns what language the string is in.

Consider the word "excellent", for example. This string is valid in both English and French but not in German. The program I developed works in the same way, except that it uses stacks as the data structure and tests for inclusion in one or more of 10 fictitious languages.

Stacks were highly effective for this purpose, and the language identification task too full advantage of the last-in-first-out nature of this data structure. The result was not surprising since the language identification problem is similar to the delimiter matching example that was presented in class. The difference is that, instead of left and right delimiters, we are (for most of the languages that were tested) matching As and Bs.

In total, I tested 65 strings, including both the required and supplemental cases. The graph below shows the results, the number of occurrences of each language (L1 through L10) vs. the language. L4 was by far the most popular language of the group, while four languages tied for being the least popular language (L5, L6, L9, and L10).




The following sections of this analysis cover the ten enhancements that were implemented beyond the requirements for this project, the justification for the design decisions that were made, what I learned in this project, issues of theoretical vs. observed efficiency, a comparison of an alternative recursive solution to the iterative solution, what I would do differently next time, and known problems and future work.

## Enhancements

The following enhancements were made during this project, above what was required in the original lab specification:

**1. 10 languages** in total were tested (5 required and 5 extra languages)

Required Non-Trivial Language
L5 = {w: w is of the form (A^n)(B^(5n)) for some n > 0}

Extra Languages
L6 = {w: w is of the form (A^(3n))(B^n) for some n > 0} // e.g. AAAB
L7 = {w: w is of the form (AB)^p for some p > 0} // e.g. ABAB
L8 = {w: w is of the form (JAVA)^p for some p > 0} // e.g. JAVAJAVAJAVA
L9 = {w: w is of the form (JHU)^p for some p > 0} // e.g. JHUJHUJHU
L10 = {w: w is of the form (A^(10n))(B^n) for some n > 0} // e.g. AAAAAAAAAAB

**2. 50 additional input strings** were tested above the 15 required strings (see language_inclusion.xls)

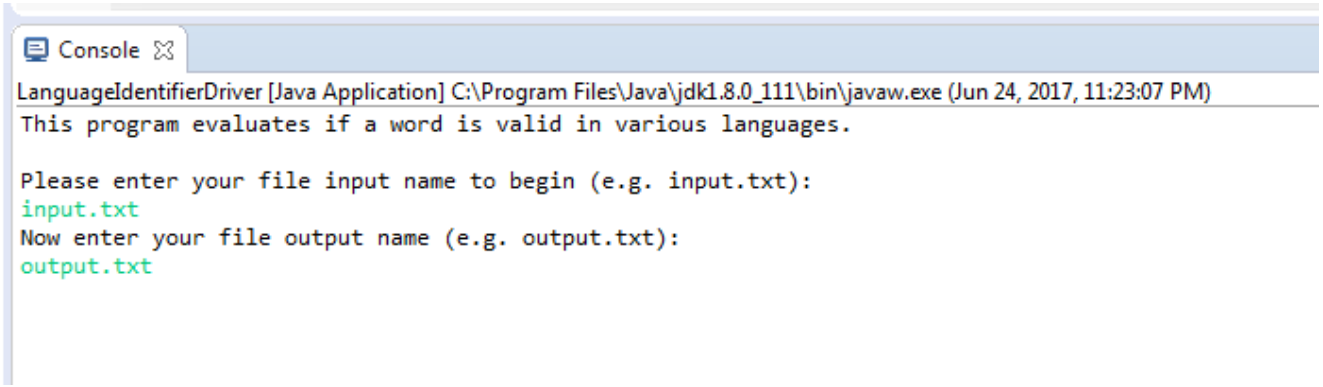
**3. Summary statistics of the results** are displayed in the output.txt file

```
67 ----------------------------------------
68 DESCRIPTIVE STATISTICS ABOUT THE RESULTS
69 ----------------------------------------
70 Number of Languages Tested: 10
71 Number of Strings Tested: 65
72 Number of Valid L1 Strings: 14
73 Number of Valid L2 Strings: 6
74 Number of Valid L3 Strings: 6
75 Number of Valid L4 Strings: 31
76 Number of Valid L5 Strings: 2
77 Number of Valid L6 Strings: 2
78 Number of Valid L7 Strings: 4
79 Number of Valid L8 Strings: 3
80 Number of Valid L9 Strings: 2
81 Number of Valid L10 Strings: 2
82
```

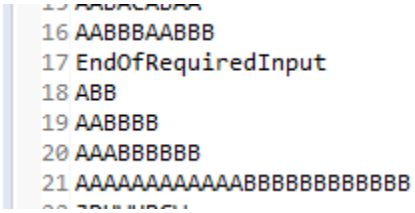
**4. Program runtime displayed** in the output.txt file

```
83 ----------------------------------------
84 OBSERVED EXECUTION TIME
85 ----------------------------------------
86 Execution Time: 8 milliseconds
```

**5. Prompt for file input** enables flexibility




**6. Separator** for the required input and supplemental input



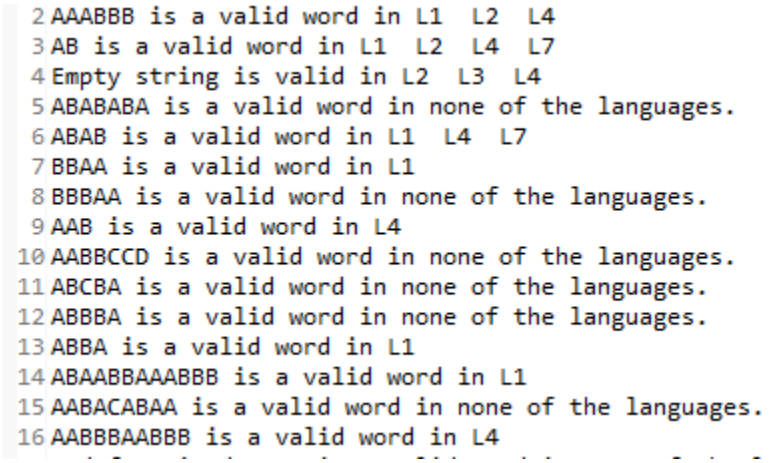

**7. Excel worksheet (language_inclusion.xls)** that shows **language statistics** related to the output.txt file (table and graph format)

**8. Excel worksheet that shows time performance** results at different input sizes (table and graph format)

**9. Input text files used to test time performance** and extreme cases of data size

**10. User friendly output** format

Below is the screenshot of output.txt for the 15 required strings



## Justification for Design Decision (incl. Data Structures and Implementation Used)

The application uses three different classes:

1. *LanguageIdentifierDriver.java:* the driver class that contains the main method. Creating a separate driver class enables me to reuse the stack code and language identification code for other problems.
2. *Stack.java*: the stack code.
3. *LanguageIdentifier.java*: the class that contains the variables, constructor, and methods that perform the language identification on each input string.

**Why was a stack a reasonable choice to solve this problem?**

Stacks were highly effective for solving the language identification problem. As we learned in the class lecture when we were matching delimiters, stacks are useful when you need to store data and then manipulate that data in the reverse order (i.e. last-in-first-out).

Several of the languages involved a series of As followed by a series of Bs. The rules of each language specified the ratio of As to Bs. From a stack perspective, identifying the language often involved pushing As to the stack and then popping those As with a series of Bs.

For example, in L2 where the word is of the form $A^nB^n$, we pushed with As and popped with Bs. An empty stack at the end of the input string is indicative that there are equal numbers of As and Bs. Combining if-statements and boolean expressions with those stacks ensured that I cannot push any As to the stack once I start popping with Bs. A string that violated this behavior was not valid in L2.

**What implementation of a stack did I choose and why?**

The array implementation (specifically a character array) worked well for this application. We know the maximum length of the input strings, so we are not concerned about the size limitations of arrays.

Arrays have a couple of advantages over the linked implementation of stacks. They offer random access because the items are contiguous in memory (not too unlike how words contain characters that are contiguous). Arrays also have less memory overhead for each element because you don't need the extra space for a pointer.

## What I Learned

Starting with the big picture and narrowing down to language and structure specific concepts, here is what I learned in this project:

1. How to use stacks to evaluate if a given string is in a Language L

- As mentioned in the prior section, stacks turned out to be useful for this purpose.

2. Theoretical and observed efficiency do not always match up

- I will elaborate on this discrepancy in the next section.

3. There can be significant variation in calculated runtime from run to run
   - I was surprised how the calculated runtime can have substantial variation (e.g. 15ms+) from run to run.

4. How to prompt for the input and output file name

   - Having never read an input file to a program before, the whole process of developing the prompt for the file name and then reading the file line-by-line was new to me.

5. Even with large input size (>100,000 strings) the execution time is on the order of milliseconds

   - It would take a human several months to analyze this data.
   - It is so impressive how a computer can complete the same task in milliseconds.

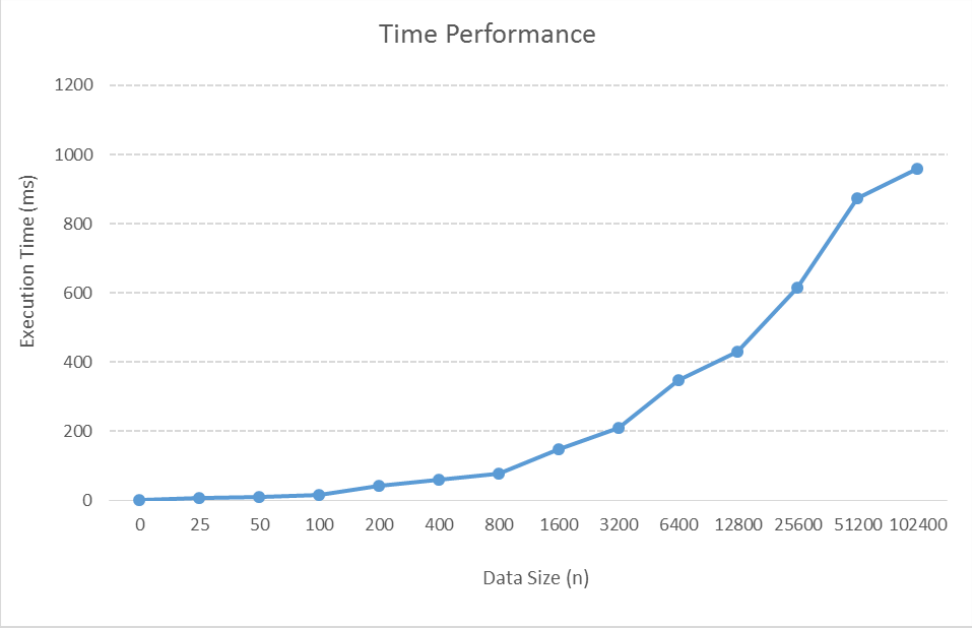
6. Proper error checking

   - The pdf document written by Professor [redacted] as well as the Project0 example served as useful guidelines for how to properly error check.

## Issues of Efficiency (Theoretical vs. Observed)

From a theoretical perspective, the cost to process a string is O(k) where k is the size of a string. Therefore, the total cost is nk if we assume that the average string size is k and that the processing of each string is linear (both in time and space).

In order to calculate the observed efficiency, I executed the algorithm many times and collected timing information. What was observed turned out to be different than my theoretical expectations.



| Data Size (n) | Execution Time (milliseconds) |
|---|---|
| 0 | 0 |
| 25 | 6 |
| 50 | 8 |
| 100 | 15 |
| 200 | 41 |
| 400 | 59 |
| 800 | 77 |
| 1600 | 146 |
| 3200 | 209 |
| 6400 | 348 |
| 12800 | 429 |
| 25600 | 615 |
| 51200 | 874 |
| 102400 | 958 |

As you can see in the table above, there were flashes of O(n) behavior during this analysis (such as from n = 1600 to n = 3200), but it did not hold. O(n) in this case is not a reliable estimator of performance.

## Recursive Algorithm vs. My Iterative Solution

**Sample recursive algorithm**

The recursive algorithm I would develop would depend on the language I am testing.

Let's consider L2 as an example. Strings that are valid in L2 are of the form $A^nB^n$ for some n >= 0. Examples of strings that are valid in L2 include AB, AABB, and AAABBB.

To handle this problem recursively, I would read each string one-by-one into a character array. For each character array, I would start at the first character in the array and the last character in the array. I would compare those two values and then, if the comparison is true, I would compare the second character in the array and the next-to-last character in the array, and so on. There could be four stopping cases:

1. If the character array is empty (in this case zero occurrences of A and zero occurrences of B)
2. If we have no more character comparisons to do
3. If there are any characters other than A or B
4. If the compared values do not fit the required criteria

The pseudocode for the recursive algorithm would be as follows. It takes advantage of the fact that the ASCII value of A is 65 and B is 66:

```
// A recursive function that checks a charArray[start….end]
// is valid in L2 or not
boolean isL2(char charArray[], int start, int end) {
```

```
    // Stopping case: if the character array is empty (this means n = 0)
    if (charArray[] is empty)
            return true;

    // Stopping case: if we have no more inputs
    if (charArray[] has no more elements to compare)
            return true;

    // Stopping case: if the letter is neither A nor B, return false //
    if (!(charArray[start] == 'A' || charArray[start] == 'B')) {
            return false;

    // Stopping case:
    // Check if the start character is an A and the end character is a B
    // Recall that the ASCII value of A is 65, and the ASCII value of B is 66
     if (ASCII value of charArray[start] - ASCII value of charArray[end] != -1)
            return false;

    // Recursive case:
    // Check if the start character is an A and the end character is a B
    // If true, check the inner substring
     if (ASCII value of charArray[start] - ASCII value of charArray[end] == -1)
            return isL2(charArray[start], start + 1, end - 1);

    return true;
}
```

A recursive algorithm for the other languages -- L1 and L3 through L10 -- would use ASCII values of the characters as well as if-statements and Boolean expressions to evaluate the validity of each string in each language.

**Comparison of the iterative solution to the recursive solution**

Like in the iterative solution I implemented in the program, the recursive solution uses if-statements to check if we have any characters other than A or B. We also use if-statements to check if we have an empty string. The fundamental difference between both solutions is that the recursive solution starts at both ends of the string and moves inward towards the middle, incrementally. The stack-based iterative solution checks each character one-by-one, moving from left-to-right, pushing to and popping from the stack according to the rules of the language.

However, unlike the classic Tower of Hanoi and Factorial problems that are naturally recursive programs, the language identification program is not naturally recursive. The iterative solution is better because it is easier to understand (for example should someone else pick up your code and review it), less redundant, and uses less space because you do not have to save all instances to the stack.

## What I Might Do Differently Next Time

There are several things that I might do differently next time. These include:

1. Use the Java Stack class and compare the efficiency with the class I coded in the application.
2. Test even larger input sizes (> 100,000) to see if there is a point where I start to get more consistent time performance behavior.
3. I would create another program that uses counting of the characters in the input string and compare that program to the one I created in this lab.

## Known Problems and Future Work

Stacks may or may not be the most efficient way to solve this problem. For example, when you consider L1 where we have equal numbers of A's and B's, I would be interested to see what performance we would get with an algorithm that used counting of the input string. For example, we count the number of As and count the number of Bs using two counter variables (int a and int b). Each time we read an A, we increment by 1 (i.e. a++). Each time we read a B, we increment by 1 (i.e. b++). If a and b are equal, we know we have a valid string of L1 (as long as we have no other characters other than A and B).

Other future work could include:

1. Enabling the user to create his or her own language
2. Creating a separate class that handles the runtime metrics
3. Replacing the stack I developed with the built-in Java Stack class, and comparing the performance between the two (https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html).