EN.605.202.81.SU18 Data Structures**: Introduction to Data Structures**

Aravind Ravindranath

Language Verifier Project Analysis (LAB 1)

Due Date: July 3, 2018

Dated Turned In: July 2, 2018

Language Verifier Analysis

The current requirement deals with analyzing a string and to check whether certain defined languages can accept it as valid and legal. String functions cannot be used and to build the string parser and language verification from scratch it was required to use character based manipulation. Stacks as a simple data structure helps in storing the characters and then churning them out on demand. Since there is no restriction on the number of stacks or the operations that can be used this serves the purpose of doing the verification in any form and way.
As a side remark, queues or linked lists could have also been used for this purpose but there could have been overhead to maintain those data structures. Stacks as a simple data structure following a LIFO pattern serves the purpose with minimal management. It is just required to keep a pointer to the latest member or element of the stack and then provide operations to move through the stack by executing pop operation.

In my implementation I chose to go with a linked stack and not an array based one. With a linked stack, I did not have to think of allocating memory at design time thus theoretically removing the limit in input data sets other than that put by the OS. With a definition of a stack where we are supposed to have a view only on the last element pushed in, a linked stack makes senses as we do not leverage the random access associated with an array implementation. With an array implementation, if there was a need to create more stacks, one would end up allocating memory to these arrays and most of them would be unutilized as well. There was a chance to use a single array and then to manage many stacks on it but that too has a defined memory limit and the management needs to be robust preventing underflow and overflow.

1. Brief description of the project and the objects:

I structured the entire project into three different packages logically with one package holding all the necessary aspects of stack management, the second package containing all the aspects for the language verifier and the last one which had a class dealing with Input/Output tasks.

Package **com.jhu.ds.lab1.reuse**: An interface CharStackable holding the methods to operate a stack is implemented by class LinkedStack. The SNode class provides the structure and methods to manage elements of a stack. These classes can probably be reused for some other project if it is required to have a stack of characters.

Package **com.jhu.ds.lab1.langcheck**: All the language types against which the string gets validated is defined by an enumeration type EnumLanguageType. This helps avoid literals in the code and allows traceability and readability. The main logic to verify the language is in class LangParser. In the current state comparisons are done against 7 languages with enumerated types staring from L1 until L7. The class LangParser has a method which fills stacks with the characters from the string.

Package **com.jhu.ds.lab1.IOoperations**: The class ProcessInputCheckLang is the driver class which interacts with the end user and orchestrates the call to language verifier.

In addition, this project consists of JUNIT test cases which provides a guard rail for the functions and features developed in this project. The test cases test the main algorithms in SNode, LinkedStack and LangParser class. The tests reside in package com.jhu.ds.test . This allows any other developer to come in and enhance the code without breaking the current working functionality. The initial effort is a tad high but later the tests which can be run on demand ensures that further enhancements have not broken the expected behavior.

2. Algorithms used and comparison with recursion:

To ascertain whether a string is valid for language L1, a method checkCountEqual is implemented. Here two stacks are used one which holds all the characters of the string in the same order as is maintained and an empty stack. A loop is executed on the first stack until its empty. The second stack gets pushed with character from the string. We are expecting equal number of A's and B's. It pushes A or B in, continues to push if it still finds A or B respectively in the subsequent character positions of the string. Now when the next character encountered is not the same as the last one, a pop is performed. Therefore, if there are equal number of A's and B's there will be an equal amount of push and pop operations leading to an empty stack.
Now if we do this recursively the pseudo code explains how this is done. Stack s contains all the pushed characters from the string:

```
LinkedStack t = new LinkedStack;
public boolean is_L1( LinkedStack t){
        if(s.isEmpty() && t.isEmpty()){
                return true;
        }else if( s.isEmpty() && !t.isEmpty() ) {
                return false;
        }else{
                char c = s.peek();
                if (t.isEmpty() || c == t.peek()) {
                        t.push(s.pop());
                }else{
                        s.pop();
                        t.pop();
                }
                return is_L1(t); // recursive call
        }
}
```

The recursive method is is_L1(). Base case is when both s and t are empty the string is a valid expression in L1 and when s is empty but t is not then it is not L1.

The runtimes of both the iterative and recursive code is linear. There are on an average 8 operations in the iterative logic written in the project times the number of entries in the stack. The runtime 8n in the big O notation approximates to O(n). Even in the recursive case it is almost the same with an average of 8 to 9 operations per method call and the recursive call goes on until the stack empty case is encountered. So the runtime will be again a constant multiple of n, where n is the number of entries. Therefore, even for recursive case the runtime is O(n). Now

we must be careful with space allocation with recursion as each call will need to be stored in memory in a call stack and if we are analyzing a large string, recursion will lead to storage of such calls and there is a chance for memory overflow. If indeed we are analyzing a very large document and if there are many such documents requiring analysis concurrently then recursion would not be a suitable choice.

Now taking the next method to verify whether a string is valid in language L2, a similar approach is taken as for L1 verification. For this pattern to be valid, the count of A's and B's must match and this is verified before going on to check if the pattern follows AnBn.
In the iterative approach, there is a stack storing the characters in the same order as the string (achieved by copying from the stack built by reading from the string), then two more stacks one storing only A and the other B. The first expected character is A, if not it is not L2. If it is A, then the stack for A gets filled up until it encounters B, from then on only stack for B will be populated. At any point if it encounters an A again then it is violating the pattern.
This can be done recursively as well as depicted below:

An empty stack t, populated stack revBckUp and a Boolean bToggle= true is passed. The attribute s contains the string character in the reversed format.

```
public boolean is_l2( LinkedStack t, LinkedStack revBckUp, boolean bToggle){

        if(s.isEmpty() && t.isEmpty()){
                return true;
        }else if( ( s.isEmpty() && !t.isEmpty() ) ) {
                return false;
        }else{
                char b = s.peek();
                char a = revBckUp.peek();
                if ( bToggle && a!='A'){
                        return false;
                }
                if ((t.isEmpty() || ( b == t.peek() && a!=b )) && bToggle ) {
                        t.push(s.pop());
                }else{
                        bToggle = false;
                        s.pop();
                        if ( t.isEmpty() ){
                                return false;
                        }
                        t.pop();
                        revBckUp.pop();
                }
                return is_l2(t, revBckUp, bToggle); // recursive
        }
}
```

The base case is when both s and t are empty, the pattern is true else if s is empty but not t, then the string is not L2 relevant. The recursive call is made until the stack s is empty.
On the same lines as L1 verifier both iterative and recursive calls exhibit a runtime of O(n) and for smaller data sets both will respond the same way. But recursive algorithm is more elegant than the iterative one since in the project, check for L1 is made and then the pattern is checked for optimal performance. In the case of recursion, both the count and the pattern gets checked in the same logical unit.

In the same way the algorithm is written to check if a string is valid in L3. The only additional check which needs to be done is to see that a stack with A's is perfectly divisible by the stack of B's and the pattern stays AnB2n. Like in check for L2, this can also be recursively modelled.

Check for L4 is a bit involved, due to differing patterns of occurrence of A and B. This needs to be calculated in advance by taking the first occurrences of both and then using that to see if the stack of A and stack of B are following the pattern.

Check for L5, L6 and L7 follow the algorithm of L1 and L2.

3. Learnings:

In this project there are lots of stack copying done to preserve the original character stack. Sometimes additional character stacks are defined to verify repeating patterns. This does cause unwanted runtime based on the number of characters in the strings. Since the project is working functionally fine, the change of the code in a shorter time could have led to bugs. A smarted approach to minimize copying and using already built stacks can process larger strings much faster. It is also observed that if a parallelization approach is introduced the stack manipulations can be done much faster. In summary, there is definitely good scope for optimization and refactoring.

4. Measurements done on some of the methods:

A class utility has been developed to measure the runtime of some of the algorithms. The class `MeasureLangVeriferRun` does runtime measurements. It resides in com.jhu.ds.test package.

L1		L2		L3		L4		L7	
Size	**Time (nanos)**	**Size**	**Time (nanos)**	**Size**	**Time (nanos)**	**Size**	**Time (nanos)**	**Size**	**Time (nanos)**
2	33469	2	20974	2	9818	2	40608	2	21420
4	9817	4	2678	4	893	4	4462	4	1338
8	18296	8	2678	8	893	8	4463	8	1338
16	44625	16	4016	16	893	16	5802	16	1785
32	76754	32	7140	32	1339	32	8925	32	2231
64	128074	64	12049	64	1339	64	14726	64	4017
128	177606	128	22313	128	1785	128	28114	128	7140
256	120041	256	44178	256	3570	256	54442	256	12941
512	1925114	512	86572	512	6248	512	116025	512	25882
1024	373509	1024	177160	1024	21866	1024	213306	1024	52657
2048	536836	2048	262840	2048	42840	2048	260163	2048	78986
4096	1087060	4096	1273592	4096	66937	4096	555132	4096	152617
8192	1657812	8192	606004	8192	118256	8192	1094646	8192	312373
16384	3943941	16384	1449414	16384	227587	16384	2166981	16384	701056
32768	7266257	32768	2181707	32768	732740	32768	4711041	32768	1635053
65536	11292754	65536	3677977	65536	1169616	65536	8148936	65536	3363372

Table 1 Runtime of the methods in nano seconds, size is length of the string

When these measurements are plotted they display a linear trend as can be seen below (fitting not done). When trying to increase the size, there was a slowdown in the response, since the stack operations was taking too much time. I could not get a profiler to work in my eclipse environment to pin point the bottleneck. The graphs are plotted by using the free utility from the web desmos.com/calculator

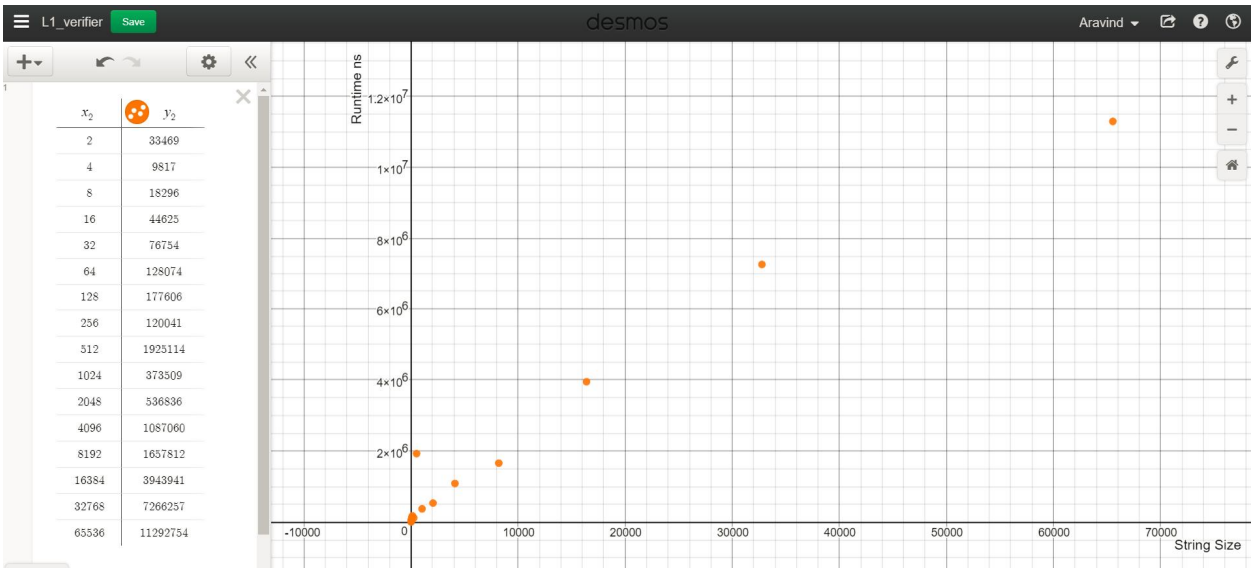

Figure 1 Point plot of L1 verifier

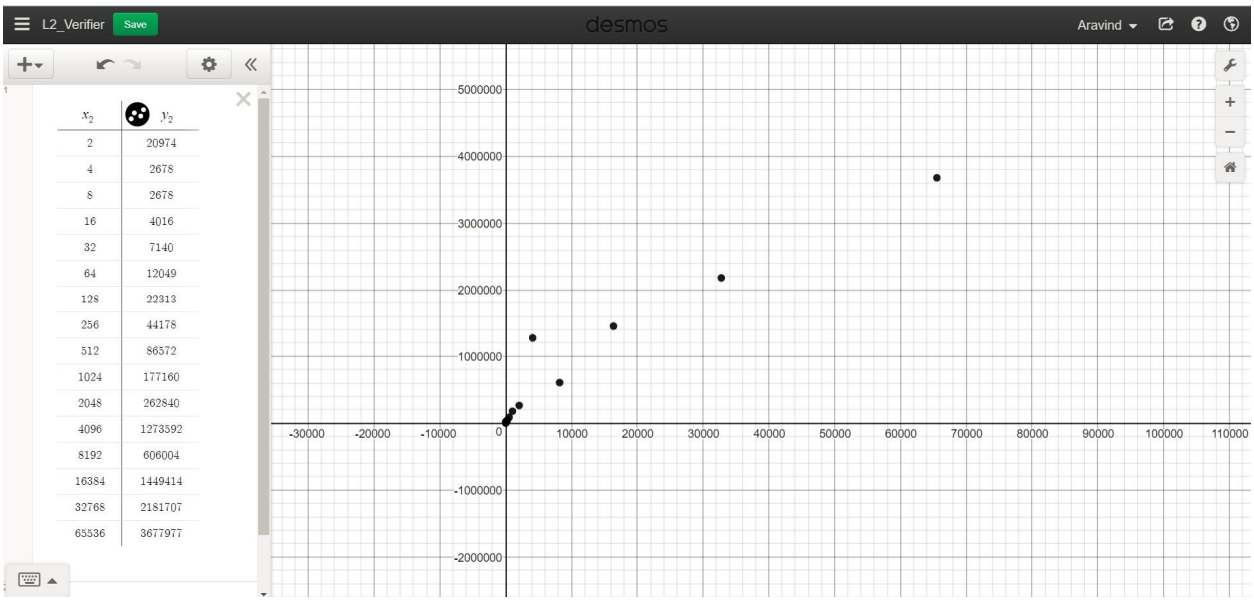

Figure 2 Point plot of L2 verifier

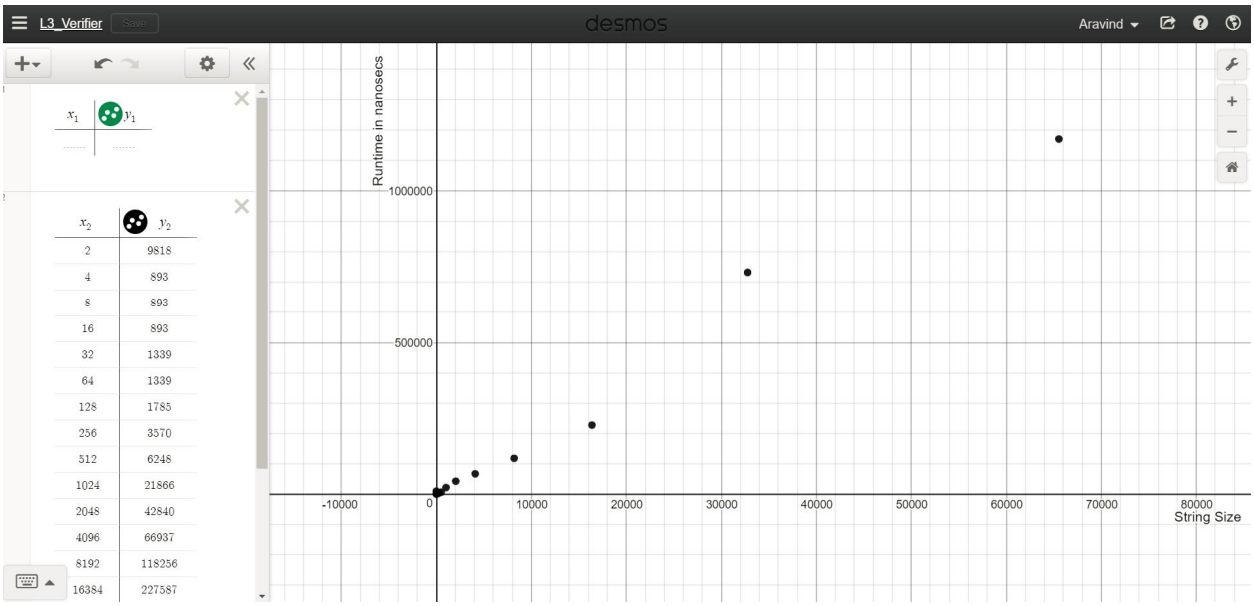

Figure 3 Point plot of L3 verifier

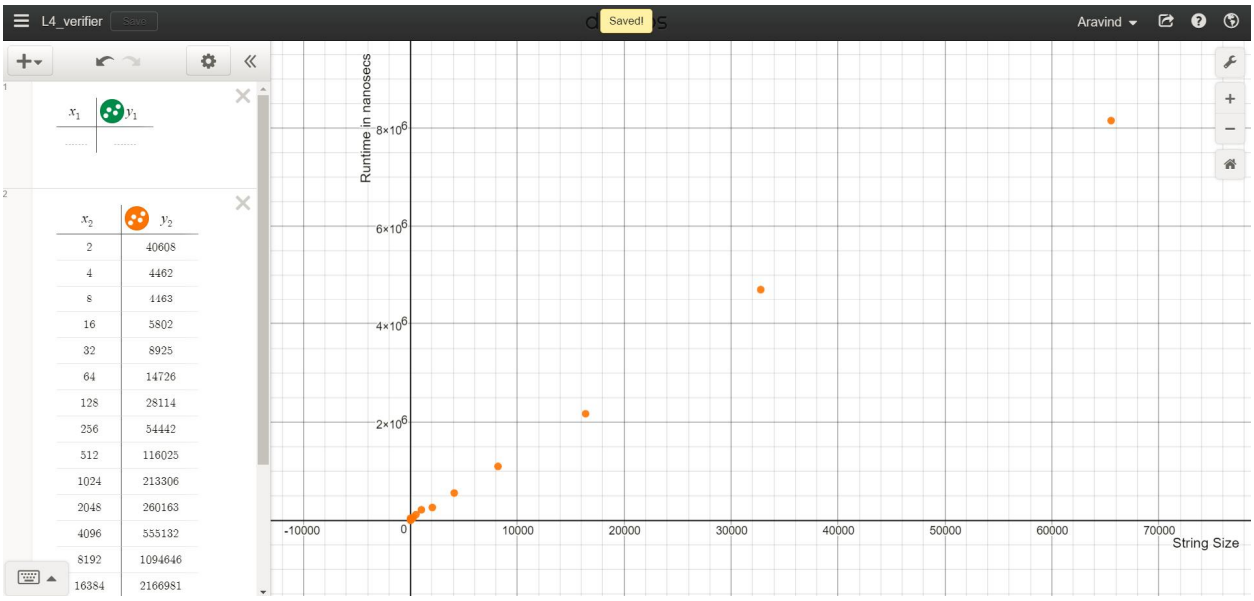

Figure 4 Point plot of L4 verifier