Working with Languages
CS142 Unit 1 Assignment

Fall Semester 2022

1 Warm Up (#Automata, #Logic)

I will prove both 1.1 and 1.2 simultaneously by not using any features of determinism or non-determinism in my reasoning. The proof asks to prove that a string exists. I will focus my analysis on the shortest string that any DFA or NFA would accept and how we can determine its length.

Let's assume that the string s is the shortest string that is accepted. I will use proof by contradiction. Let's assume that $|s| > n$, where n is the number of states in the FA. As we read the input string s the FA will have to visit one of the states at least twice even if it goes through all other states.

The string s can be represented as $s = s_1 s_2 \dots s_i \dots s_j \dots s_n$. Each of s_m represents a symbol in the string. Now let's say that when the FA reads s_i and s_j it goes through the same state.

If the state we repeat was the first state ($i = 1$), then the string is $s = s_i \dots s_j \dots s_n$. This means that the right-hand chunk of this string is in fact a shorter string that is accepted by the FA. This leads to a contradiction with our original assumption that $|s| \geq n$.

The second scenario is that we repeat the state later on in the string ($i > 1$). Then the string is $s = s_1 \dots s_i \dots s_j \dots s_n$. But there would still be a shorter string accepted by the FA: $w_1 \dots w_{i-1} w_j \dots w_n$. We just get rid of the part that makes the string longer (and repeat state it does not have to). Now it directly proceeds to the final accept state. This case also leads to a contradiction with our original assumption.

Therefore, the shortest string s accepted by any FA is of length smaller than the number of states. Let's discuss the reason why this makes sense. A string of length n can be represented by a path in any FA that goes through at least $n+1$ of the states. We can further prove this by induction.

If the length of the string s is $|s| = 0$, then it will go through 1 state, which is the start state.

If $|s| = 1$, then the FA will go through 2 states (not necessarily 2 different states).

The pattern is that, if $|s| = n$, then the FA goes through $n + 1$ states. In the worst-case scenario to go to the accept state an FA needs to go through all other states before being able to accept. That means that we go through n states, which corresponds to a string of size $n - 1$, which is smaller than n. Hence, the claim is true.

For the NFAs the bound of 2^n is actually bigger than $n-1$ so that statement still holds. I assume that this bound comes from the fact that in a computation of an NFA there are parallel branches of computation taking place simultaneously. However, this does not change the fact that the shortest string that an NFA accepts is the string that takes the shortest path from the start state to its accept state. This path in the worst case scenario is of size n as I already discussed. Therefore, this statement holds true for both DFAs and NFAs.

Total Time to complete 1 hour to fully complete. This assignment took me extraordinarily long to complete so I wrote the time each task took me after I completed it to provide a general idea of the workload. I still really liked solving all the problems and enjoyed the process. Still, it was quiet time consuming. I dedicated 26 hours in total to solving the problems and typing up all solutions. If I take into consideration the time I spent double-checking the paper, time in OH, and discussing with other students it is about 30 hours of pure work. I have never had an assignment this long before.

2 Regular Langauges

2.1 Homomorphisms (#Automata, #Logic)

2.1.1 Solution:

Let's first explain what the definition says, which will help us come up with an example. The definition states that f is a function that takes any string w and outputs $f(w_1)f(w_2)\dots f(w_n)$. We can see that the function f concatenates the transformation of each symbol in the original string w.

For the example, let the alphabet be: $\Sigma = 0, 1$.

For the function let: $f(0) = 1, f(1) = 0$.

If the original string $w = 010$, then:

$$\begin{aligned} &f(010) = \\ &= f(0)f(1)f(0) = \\ &= 101 \end{aligned}$$

So let's define the language A to be: $A = \{0^n 1 | n \geq 0\}$.
Let $f(0) = 1$ and $f(1) = 0$.
Then, the language $B = f(A) = \{1^n 0 | n \geq 0\}$.

2.1.2 Solution:

Let's follow the logic from the previous question.

Let the alphabet be: $\Sigma = \{0, 1\}$.
Let the other alphabet be: $\Gamma = \{a, b\}$.

For the function let: $f(0) = a, f(1) = b$.

If the original string $w = 010$, then:

$$\begin{aligned} & f(010) = \\ & = f(0)f(1)f(0) = \\ & = aba \end{aligned}$$

So let's define the language A to be: $A = \{0^n 1 | n \geq 0\}$.
Let $f(0) = a$ and $f(1) = b$.
Then, the language $B = f(A) = \{a^n b | n \geq 0\}$.

2.1.3 **Solution:**

For the class of regular languages to be closed under homomorphism, it means that any time the homomorphism operation is used the output still belongs in the set of regular languages. In other words, it is guaranteed that using homomorphism on a regular language will produce again a regular language. A language is regular if it is recognized by a Finite Automaton.

Proof Idea: I will use the main strategy for proving regular languages are closed under the regular operations discussed in the textbook and during class (Session 5) to prove regular languages are also closed under homomorphism. In order to prove that that is the case, we use proof by construction. We need to construct a DFA or NFA that recognizes the language.

So let L be a regular language and N be an NFA that recognizes it. We need to construct another machine N' by modifying N. The new N' is such that it recognizes $f(L)$ (applying homomorphism to the entire language L). For the purposes of clarity let's use an example. Let's say this is N:

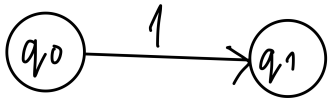

Figure 1: Starting NFA N that recognizes $L = 1$

So what we want is a NFA N':

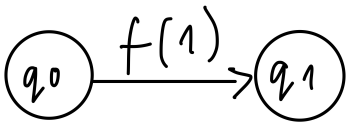

Figure 2: Equivalent NFA N' that recognizes $f(L)$

A NFA processes a single symbol at a time. Since the output of $f(1)$ can map to a string rather than a single symbol, the length of $f(1)$ can be longer than one so we need to break up the

transition between the states q_0 and q_1. What we need to do is to read each of the individual characters of the output $f(1) = w$, of length n, by adding additional states and transition between q_0 and q_1. The concatenation of the transitions between these newly added states is equivalent to the output of $f(1)$: $f(1) = w = w_1 w_2 \dots w_n$. The figure 3 below describes that:

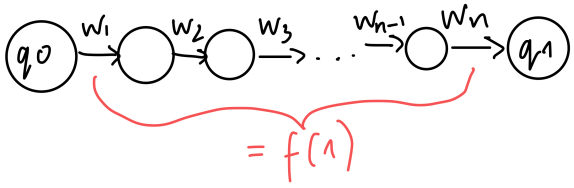

Figure 3: NFA N' that recognizes $f(L) = w$

This makes sense because in the original NFA N when the read symbol at $q0$ is 1, then it transitions to the state q_1. In the constructed NFA N' we want the same thing to happen: whenever $f(1)$ is read at state q_0, it should transition to the state q_1. Looking at the diagram above this happens as we read w_i one by one to follow the path of the newly added states whose concatenation is equal to $f(1) = w_1 w_2 \dots w_n$. So now if N accepts an input, like 1 so does N' and vice versa.

So this is the idea for constructing N' from N described in the diagram below.

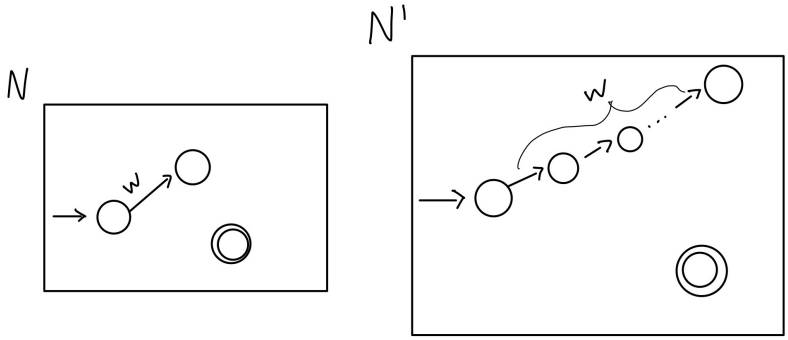

Figure 4: Construction of N' to recognize $f(L)$

Let's finalize the example from Figures 1 and 2. Let's say $f(1) = aaa$. By following this procedure we outlined above, the resulting NFA will be:

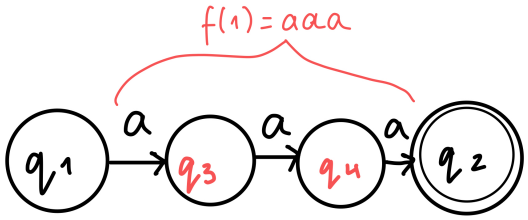

Figure 5: NFA N' that recognizes $f(L) = aaa$

Let the original NFA in Figure number 1 on page 3 be $N = (Q_1, \Sigma, \delta_1, q_0, F)$.

(a) $Q_1 = \{q_0, q_1\}$

(b) The start state q_0.

(c) The accept states q_1.

(d) The alphabet is $\Sigma = \{1\}$..

(e) The transition function δ is described as $\delta(q_0, 1) = q_1$

The newly constructed NFA's in 5 formal definition is:

(a) $Q = \{q_3, q_4\} \cup Q_1$

(b) The start state is the same q_0.

(c) The accept state is the same q_1.

(d) The alphabet is $\Gamma = \{a\}$.

(e) The transition function takes into account the newly added states:

	1
q_1	q_3
q_2	
q_3	q_4
q_4	q_2

Proof

Let $N = (Q_1, \Sigma, \delta_1, q_0, F)$.
Construct $N' = (Q, \Gamma, \delta, q_0, F)$ to recognize $f(L)$.

(a) $Q = q_{w_1}, q_{w_2}, \ldots, q_{w_n} \cup Q_1$.
The states of N' are the states of N plus all the new states that are added between the existing states to simulate transitions like the one described above (when the $f(symbol)$ maps to a string rather than an individual symbol).

(b) The start state remains the same q_0.

(c) The accept states remain the same F.

(d) The alphabet is different (but could also remain same) and is such that $f(w) = f(w_1)f(w_2)\dots f(w_n) \in \Gamma^*$.

(e) The transition function is such that it takes into account the newly added states but still ultimately leads from the same original state to the same original endpoint. It might be bigger since it needs to take into account all these sub transitions.

This has finalized the proof by construction.

2.1.4 **Answer** The class of nonregular languages is not closed under homomorphism.

If the class of nonregular languages is not closed under homomorphism, then applying this operation on a nonregular language can output a language that is not nonregular. The fastest way to prove this is not the case is with proof by contradiction.

Proof Idea My strategy is to get a non-regular language, show that it is a non-regular language using the pumping lemma. Then, apply the homomorphism operation, show that the resulting language is regular (not non-regular) with the help of the pumping lemma.

I will define the pumping lemma here as it is defined in the textbook for reference later on in the proof.

Pumping Lemma for regular languages: If L is a regular language, then there exists an integer p (pumping length), which allows if w is any string in L with $|w| \geq p$, then w can be divided in three parts $w = xyz$, satisfying the following conditions:

(a) for each $i \geq 0, xy^i \in L$

(b) $\|y\| > 0$

(c) $\|xy\| \leq p$

Proof

Let's take the non-regular language $L = \{0^n1^n | n \geq 0\}$, $\Sigma = \{0, 1\}$ from Example 173 (Chapter 1) in the textbook.

Let's first prove L is non-regular using the pumping lemma. The approach in Example 173 explains that we can decompose the string and use the third condition to prove L is not regular.

Let's assume L is regular. There must exist a number for the pumping length p to satisfy the conditions in the Pumping Lemma above.

Let's choose our string $w = 0^p1^p$ which guarantees that the length is always greater than p. To prove this language is not regular we need to show this for every possible value of p and decomposition of w.

From the third condition c) $|xy| \leq p$ we know that x and y must contain only 0s, so we have:

$$x = 0^a$$
$$y = 0^b$$
$$a + b \leq p$$

And from the second condition b) of the pumping lemma we know that $b \geq 1$.

Logically, z must be the rest of the string:

$$z = 0^{p-(a+b)}1^p$$

Now our goal is to use the first condition a) to show that the outputted string $\notin L$:

$$xy^iz \notin L$$
$$xy^iz = 0^a0^{i\times b}0^{p-(a+b)}1^p =$$
$$= 0^{p+i\times b-b}1^p$$

Now if we want the output string to be in L, then we need the exponent of 0: $p+i\times b-b = p$. The only value of i that would meet this requirement is if $i = 1$. For any other value, $i \neq 1$, the string is $\notin L$. The next possible value is $i = 2$ so let's choose it. This will contradict the first condition a) in the Pumping lemma that for all values of i, $xy^i \in L$.

When $i = 2$, then the string is:

$$0^{p+2\times b-b}1^p =$$
$$= 0^{p+b}1^p$$

Above we also established that $b \geq 1$. This is a contradictions as it means that there will be more 0s than 1s. Therefore, the language L is non-regular.

Now let's define the homomoprhism function: $f : \Sigma \longrightarrow \Gamma^*$. Let's take $\Gamma = a, b$ and:

$$f(0) = \epsilon$$
$$f(1) = ab$$

Now the new language $B = f(L) = \{(ab)^n | n \geq 0\}$.

Now let's use the Pumping lemma to find the pumping length of B to show it is a regular language.

Some of the strings in the language B are: $\{\epsilon, ab, abab, ababab, \dots\}$.

The smallest value for p is $p = 1$.

If $p = 1$, we can take the smallest string in the language of length greater than the pumping length, which is $w = ab$.

Let's decompose the string in the following way:

$$x = \epsilon$$
$$y = ab$$
$$z = \epsilon$$

Condition b) $|y| > 0$ is met since $|y| = 2$. However, condition c) $|xy| \leq p$ does not hold since $|xy| = 2$ which is greater than $p = 1$. So let's take the next possible value for p.

If $p = 2$, conditions b) and c) still hold as we shown above.

For condition a):

- $i = 0$, yields the empty string, which $\in B$
- $i = 1$, yields 0101, which $\in B$
- $i = 2$, yields 010101, which $\in B$
- ...
- $i = n - 1$, yields $(01)^{n-1}$, which $\in B$
- $i = n$, yields $(01)^n$, which $\in B$

We have shown by induction that the pumping length of B is $p = 2$, which satisfies the Pumping lemma.

Let's circle back to the beginning of the proof. We applied homomoprhism to the non-regular language L and got the output language B, which is regular. The output is not in the class of non-regular languages. Therefore, the class of non-regular languages is not closed under the operation of homomorphism.

Total time for Problem 2.1 Homomorphisms: 5 hours to fully complete.

2.2 Pattern Finding (#Automata, #Machines)

2.2.1 This is the main function of the algorithm I wrote. It is well commented. Each of the separate operations is defined in its own separate function. Please refer to the Jupyter Notebook for the whole code and multiple assert statements that test various test cases and show the code works.

```python
# The only symbols allowed are the lowercase letters.
import string
import re

alphabet = string.ascii_lowercase

def validate_rex_to_NFA(rex_input, string_input):
    # Store all NFAs as we build them.
    NFA_stack = []

    # Create alphabet.
    sigma = set()
    # Get the smallest building blocks: single characters.
    for symbol in rex_input:

        if symbol in alphabet:
            sigma.add(symbol)
        if symbol == ".":
            sigma = {'a','b','c','d','e','f','g','h','i','j','k',
                     'l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'}

    for char in rex_input:
        # If character is a symbol, create its NFA.
        if char in sigma or char==".":
#             print("Added new symbol NFA🚀")
            new_NFA = generate_nfa_symbol(char, sigma)
            NFA_stack.append(new_NFA)

        # If character is concatenation symbol &, combine the two NFAs.
        elif char == "&":

            NFA_second = NFA_stack.pop()
            NFA_first = NFA_stack.pop()
#             print("Added new & NFA🌱")
            new_NFA = generate_nfa_concatenation(NFA_first, NFA_second)
            NFA_stack.append(new_NFA)

        # Union.
        elif char == "U":
            NFA_second = NFA_stack.pop()
            NFA_first = NFA_stack.pop()
            new_NFA = generate_nfa_union(NFA_first, NFA_second)

            NFA_stack.append(new_NFA)
        # Star
        elif char == "*":
            NFA = NFA_stack.pop()
            new_NFA = generate_nfa_star(NFA)
            NFA_stack.append(new_NFA)

    final_NFA = NFA_stack.pop()
#     print(final_NFA.transition)
    return final_NFA.validate_input(string_input)
```

Figure 6: Code Implementation

Briefly, the algorithm gets the regular expression and uses it to make an NFA. It creates smaller NFAs based on the operation, stores them in a stack and then pops them whenever they need to be combined based on the operation.

First it creates a small NFA for each symbol in the alphabet of the regular expression. Each operation (union, star, concatenation) is in its own separate function. The functions take the NFAs that need to be combined and do so according to the operation. For example, as star operation on a single symbol adds a self loop to the final state. A union operation between two NFAs creates a new start state and epsilon transitions to the existing start states.

I spent a lot of time developing the code and it is all in the Python notebook. It is very lengthy so I decided not to include it in the main PDF but please refer to the Jupyter notebook.

In my implementation the regular expression needs to be inputted in a special way which is a small drawback but it made it easier to implement. Concatenations are shown with the & symbol. Any operation needs to be written one place to the right. For example, $a \cup b = ab\cup$, $ab = ab\&$, etc. Another thing that can be improved is to also consider parenthesis and the empty string. Other than that it has all necessary functionality mentioned in the prompt.

2.2.2 I chose a recursive implementation. When we have only concatenation (or the . symbol) we simply need to check the string from left to right whether it matches with the regular expression.

When we consider the star operation, we need to spot it before using up the symbol so it will be in the second position of the expression. In this case there are several options for a match: either drop that part since it can result in the empty string, or remove each symbol that matches in the original string. We can infer that the input string was validates by the expression if any of its sub-strings after choosing either of the two options, is validated.

When we consider the union operation, we also need to spot it before using up the symbol so again it will be in the second position of the expression. There are several options. We can expand the left side of the rule or the right-hand side of the rule. We do this by removing the symbol of the original input that matches the symbol in the regular expression. If any of the substrings match, again we validate the string.

This is my pseudocode:

‘‘‘ Given a regular expression and an input string.

if regular expression is empty
.... return True if string is also empty (completely matched string)
Otherwise return False (some of the string is left)

if there are symbols left in the input and the next symbol in the regular expression matches with the next symbol in the input:
... substring_isvalidated = True

if len(regural expression) ¿ 1 and next rule is *:
... either drop that part since it can result in the empty string, or remove each symbol that matches in the original string.
elsif len(regural expression) ¿ 1 and next rule is union:
... expand the left side of the rule or the right-hand side of the rule. We do this by removing the symbol of the original input that matches the symbol in the regular expression. If any of the substrings match, again we validate the string.
else:
... perform concatenation by checking each symbol by slicing from left to right and looking for a validated substring
‘‘‘

The time complexity depends on both the size of the input string n and the size of the regular expression r, each expressed in terms of number of symbols. This is because the number of

recursive calls depends on both of them as we slice both from left to right while we process the string. This is a rather complex recursion call so it is hard to analyze as it is outside of the scope of CS110. My intuition is that every time we split the string or the regular expression we generate twice as many combinations of sub-strings and sub-expressions as there were in the previous level. So the time complexity will be $O(2^{n+r})$. This shows the runtime will scale exponentially, it will double every time the input grows. This is in case we perform bad slices every time the algorithm goes (worst-case scenario).

```
def validate_string_from_rex(rex, input_string):
        """Implementation of recursive algorithm to validate string from rex"""

        # For the base case when they are empty.
        if not rex:
            return not input_string

        # If the input string is at least of length 1 and the first rule matches
        # with the first character of the inout, then store that as True.
        # Otherwise false (no validated substring).
        is_substring_validated = bool(input_string) and rex[0] in {input_string[0], '.'

        # If the next character of the Reg Expr has the star operation.
        if len(rex) > 1 and rex[1] == '*':
            # Call function recursively.
            # If the rest of the input_string matches, AND either we matched
            # the left side already or we match the string with another part
            # of the expression, validate the string.
            output = (validate_string_from_rex(rex[2:],input_string) or
                    is_substring_validated and
                    validate_string_from_rex(rex, input_string[1:]))

            return output

        # If the next character of the Reg Expr has the union operation.
        elif len(rex) > 1 and rex[1] == 'U':
            # If we match the string with whatever is on the right side of the union
            # or we march it with the left side, accept the string.
            output = (validate_string_from_rex(rex[2:],input_string) or
                      validate_string_from_rex(rex[0:1], input_string[0:1]) and
                      is_substring_validated)
            return output
        # If no special characters, then perform concatenation.
        else:
            # Check if each of the symbols in the input string match the pattern.
            output = (is_substring_validated and
                        validate_string_from_rex(rex[1:],input_string[1:]))
            return output
```

```
assert (validate_string_from_rex("a*", "aaaaaaaaa") == True)
assert (validate_string_from_rex( ".", "g") ==True)
assert (validate_string_from_rex( "....", "gggg") ==True)
assert (validate_string_from_rex( "....", "abcd") ==True)
assert (validate_string_from_rex( "a*", "") == True)
assert (validate_string_from_rex( "a*", "aa") == True)
assert (validate_string_from_rex( ".*", "aca") ==True)
assert (validate_string_from_rex("a*b", "aaaaaaaaab") == True)
assert (validate_string_from_rex("a*b*c", "aaaaaaaaabc") == True)
assert (validate_string_from_rex("a*b*c", "aaaaaaaaab") == False)
assert (validate_string_from_rex("aabc", "aabc") == True)
```

```
assert (validate_string_from_rex("aub", "a") == True)
assert (validate_string_from_rex("abuc", "ab") == True)
assert (validate_string_from_rex("abuc", "ac") == True)
assert (validate_string_from_rex("aucd", "a") == True)
assert (validate_string_from_rex("aucd", "cd") == True)
```

Figure 7: Code Implementation

2.2.3 For the experimental runtime complexity I chose to use time rather than steps as a measure since defining what a step is is rather subjective. I produced two plots: one that varies the size of the input string and one that varies the size of the regular expression. Please refer to the Jupyter notebook for more details. I use 100 trials and take the average time to make sure the values are true.

```
# Store the tuple of the x and y-axis lists.
dynamic_plot_values = graph_time(1000, validate_string_from_rex)
# Plot the function and set title, labels, units, and legend.
plt.plot(dynamic_plot_values[0], dynamic_plot_values[1])
plt.title('Runtime of Recursive String Validator')
plt.xlabel('Size of input string (# of characters)')
plt.ylabel('Time (s)')
plt.show()
```

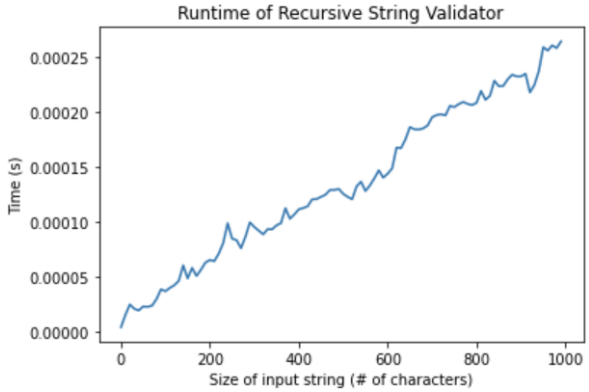

Figure 8: Runtime in terms of string size

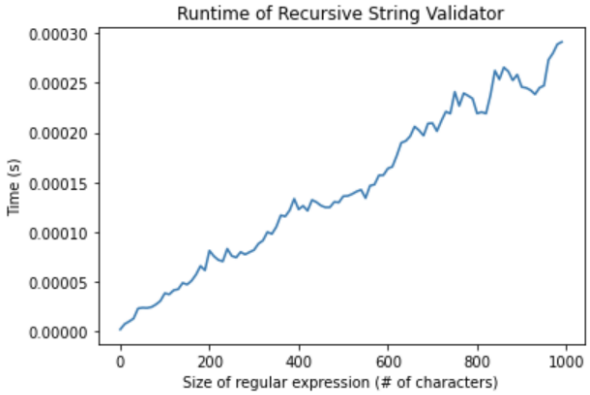

Figure 9: Runtime in terms of regular expression size

Both plots look like they are scaling linearly. This does not really confirm my theoretical derivation. The experimental result is suggesting that the runtime complexity scales $O(nr)$. However, I strongly doubt that because my implementation is far less efficient. Perhaps this is because in the average case it scales $O(nr)$ as the good and bad slices of the string and expression balance out. And the worst case scenario would be $O(2^{n+r})$ resulting in mostly poor slices. I would not find this scenario in my plots because I generate 100 trials for each input size and take their average, which leads to representing the average case.

Total Time to complete: 8 hours to fully complete. There was a lot of coding, writing, several algorithms, and the PCW did not prepare me for this problem. It was super fun! Just very time consuming to fully figure out since we have not discussed any Python implementations, time complexity, constructing these algorithms, etc.

2.2.4 *(Optional)*

Another option is to use dynamic programming because this problem exhibits optimal substructure! We can store each sub-string in a table as we process it use it later on to speed up the computation. Unfortunately, I will not have time to implement this. However, this would be that best scaling algorithm because it will only traverse each pattern symbol for each input symbol, leading to $O(nr)$.

3 Context-free Languages

3.1 Combining Langauges (#Automata, #Logic)

3.1.1 Let's show that $L = \{w | w = 0^m 1^n 0^p \text{ or } w = 1^m 0^n 1^p, n = m + p\}$ is context-free.

$L = \{w | w = 0^m 1^n 0^p \text{ or } w = 1^m 0^n 1^p, n = m + p\} = \{w | w = 0^m 1^n 0^p, n = m + p\} \cup \{w | w = 1^m 0^n 1^p, n = m + p\}$.

According to the #breakitdown method we should first construct the grammar for $L_1 = \{w | w = 0^m 1^n 0^p, n = m + p\}$ and then for $L_2 = \{w | w = 1^m 0^n 1^p, n = m + p\}$.

Constructing a CFG from a language that is regular is easy if we can use the DFA of the language. I wanted to use this strategy but intuitively the languages above seem non-regular. This is a brief justification using the pumping lemma to show that they are non-regular. I will only show it for $L_1 = \{w | w = 0^m 1^n 0^p, n = m + p\}$ but it is analogous for the other language just with mirrored symbols.

Let the pumping length $p = k$. Then the string we can take is $w = 0^k 1^{2k} 0^k$. Let's decompose it the following way:

$$x = 0^a$$
$$y = 0^b$$
$$z = 0^{k-a-b} 1^{2k} 0^k$$

Then, we need to find a value for i that will break the conditions. We know that $w \in 0^k 1^{2k} 0^k$ if and only if:

$$\begin{aligned} &a + i \times b + k - a - b + k = 2k \\ &i \times b - b = 0 \\ &i = 1 \end{aligned}$$

So we can choose any $i \neq 1$. Let's choose $i = 2$:

$$\begin{aligned} w &= xyyz \\ w &= 0^a 0^{2b} 0^{k-a-b} 1^{2k} 0^k \\ w &= 0^{k+b} 1^{2k} 0^k \end{aligned}$$

This leads to a contradiction as we get more 0s. Analogously, it can be shown that the other language is also non-regular.

My strategy to construct the CFGs now is to think about the patterns that occur when generating new strings. Let's look at some of the strings in $L_1 = \{w | w = 0^m 1^n 0^p, n = m+p\}$: $\{\epsilon, 01, 10, 0110, 0011, 1100, 111000, 000111, 001110, 0111000, \dots\}$. So what I notice is that we can think about the strings in the following way: $w = 0^n 1^{n+m} 0^m = 0^n 1^n 1^m 0^m$, where $0^n 1^n$ is the left-hand side of the generated string and $1^m 0^m$ as the right-hand side. This gives us our first rule:

$$\mathrm{S}_1 \rightarrow \mathrm{A}_1 \mathrm{B}_1$$

In the LHS A_1 we want to generate a 0 on the left and a 1 on the right. Another option for this side is for it to be empty if we want to terminate or simply add more 0s on the RHS rather than the LHS. Following the same logic for the RHS, we want to generate a 0 on the right and a 1 on the left, or the empty string. This gives us the following grammar G_1:

$$\begin{aligned} \mathrm{S}_1 &\rightarrow \mathrm{A}_1 \mathrm{B}_1 \\ \mathrm{A}_1 &\rightarrow 0 \mathrm{A}_1 1 \mid \epsilon \\ \mathrm{B}_1 &\rightarrow 1 \mathrm{B}_1 0 \mid \epsilon \end{aligned}$$

For the other part of L $\{w | w = 1^m 0^n 1^p, n = m+p\}$, we can follow the same logic but reverse the terminals for the symbols 0 and 1 to get the following CFG G_2:

$$\begin{aligned} \mathrm{S}_2 &\rightarrow \mathrm{A}_2 \mathrm{B}_2 \\ \mathrm{A}_2 &\rightarrow 1 \mathrm{A}_2 0 \mid \epsilon \\ \mathrm{B}_2 &\rightarrow 0 \mathrm{B}_2 1 \mid \epsilon \end{aligned}$$

To combine these two grammars into a single CFG G for the language L we simply need to add the rule $\mathrm{S} \rightarrow \mathrm{S}_1 \mid \mathrm{S}_2$:

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{S}_1 \mid \mathrm{S}_2 \\ \mathrm{S}_1 &\rightarrow \mathrm{A}_1 \mathrm{B}_1 \\ \mathrm{S}_2 &\rightarrow \mathrm{A}_2 \mathrm{B}_2 \end{aligned}$$

$$\begin{aligned} \mathrm{A}_1 &\rightarrow 0\mathrm{A}_1 1 \mid \epsilon \\ \mathrm{B}_1 &\rightarrow 1\mathrm{B}_1 0 \mid \epsilon \\ \mathrm{A}_2 &\rightarrow 1\mathrm{A}_2 0 \mid \epsilon \\ \mathrm{B}_2 &\rightarrow 0\mathrm{B}_2 1 \mid \epsilon \end{aligned}$$

If we want to generate a string in L_1 we would have to apply the $\mathrm{S} \rightarrow \mathrm{S}_1$ rule; if we want to generate a string in L_2 we would have to apply the $\mathrm{S} \rightarrow \mathrm{S}_2$ rule.

Let's try out an example with the string 0110. We start at S, then S_1 since the string belongs to L_1. From there we use the rules for $\mathrm{A}_1 \rightarrow 0\mathrm{A}_1 1$ and $\mathrm{A}_1 \rightarrow \epsilon$. Do the same for B_1: $\mathrm{B}_1 \rightarrow 1\mathrm{B}_1 0$ and $\mathrm{B}_1 \rightarrow \epsilon$ to output 0110.

3.1.2 For the class of context-free languages to be closed under union it means that if we have any two context-free languages L_1 and L_2, after applying the union operation on them, we will get a third language L_3 that will also be a context-free language.

Proof Idea A context-free language is such if there is a CFG that generates it or a PDA that recognizes it. To prove this we can use use CFGs or PDA since they are equivalent. I will use a CFG because the proof is shorter. I will use a proof by construction and the formal definitions to construct a CFG G from two other CFGs G_1 and G_2.

The two sets of variables for G_1 and G_2 need to contain distinct because otherwise we will not be recognizing the union of the languages but rather some cross of the two languages. To construct a CFG G from two other CFGs G_1 and G_2, we need to combine the two sets of variables of G_1 and G_2 and their sets of rules. However, because we want to be able to produce both original languages we need to add a new start variable that can go to each of the start variables of G_1 and G_2.

Proof

Let's use the formal definition of CFGs to describe G_1 and G_2 as two 4-tuples:

$$G_1 = (V_1, \Sigma, R_1, S_1)$$
$$G_2 = (V_2, \Sigma, R_2, S_2)$$

Now we can make a CFG such that:

$$G = (V_1 \cup V_2 \cup S, \Sigma, R_1 \cup R_2 \cup \{S- > S_1|S_2\}, S)$$

The reason this makes sense is because when G begins at the start variable S, it can either go to S_1 or S_2. The one it goes to will lead to the set of rules that generate either G_1 or G_2.

Let's show this with the example from the previous question. We had two context-free languages L_1 and L_2, generated by two CFGs G_1 and G_2, where:

$$G_1 = (V_1 = \{S_1, A_1, B_1\}, \Sigma = \{0,1\}, R_1 = \{S_1 \to A_1B_1, A_1 \to 0A_11|\epsilon, B_1 \to 1B_10|\epsilon\}, S_1)$$
$$G_2 = (V_2 = \{S_2, A_2, B_2\}, \Sigma = \{0,1\}, R_2 = \{S_2 \to A_2B_2, A_2 \to 0A_21|\epsilon, B_2 \to 1B_20|\epsilon\}, S_2)$$

We then used them to construct a CFG grammar G that recognizes both, represented by the union:

$$G = (V_1 \cup V_2 \cup S, \Sigma, R_1 \cup R_2 \cup \{S- > S_1|S_2\}, S)$$

We had that:

(a) $V = \{S, S_1, A_1, B_1, S_2, A_2, B_2\}$

(b) $\Sigma = \Sigma$

(c) $R = \{S- > S_1|S_2, S_1 \to A_1B_1, A_1 \to 0A_11|\epsilon, B_1 \to 1B_10|\epsilon\, S_2 \to A_2B_2, A_2 \to 0A_21|\epsilon, B_2 \to 1B_20|\epsilon\}$

(d) Added a new start variable S.

3.1.3 Let $A = \{0,1\}$, then $A^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$.

Let G be:

$$\text{S} \ \to \ 0 \ | \ 1$$

Let G' be:

$$\begin{aligned} \text{S}' \ &\to \text{SS} \\ \text{S} \ &\to \ 0 \ | \ 1 \end{aligned}$$

So now if we wanted to generate the empty string, we would not be able to. In fact, we would not be able to terminate this and 0s or 1s will be infinitely added to either the LHS or RHS of the string. So we would not be able to generate strings like $0, 1, 00, \dots$. The next problem discusses a proof by construction for the proper way to construct the grammar G'.

3.1.4 **Concatenation** For concatenation let's use the same strategy that we used in 3.1.2 to prove that the class of context-free languages is closed under union. In the proof by construction, the whole construction of the new grammar G remains the same except for the new rule we add. The union operation represents an OR so we added the rule $\text{S} \to \text{S}_1 \mid \text{S}_2$. However, now concatenation represents linking two strings one after the other. So if $w_1 \in L_1$ and $w_2 \in L_2$, then their concatenation is w_1w_2. Therefore, now we will add the rule $\text{S} \to \text{S}_1\text{S}_2$ instead. This leads to the following formal definition:

$$G_1 = (V_1, \Sigma, R_1, S_1)$$
$$G_2 = (V_2, \Sigma, R_2, S_2)$$
$$G = (V_1 \cup V_2 \cup S, \Sigma, R_1 \cup R_2 \cup \{S- > S_1S_2\}, S)$$

The reason this makes sense is because the start variable always leads to the concatenation of S_1S_2, where S_1 leads to the rules that will generate the part of the string $\in L_1$ which is generated by G_1, and S_2 leads to the rules that will generate the part of the string $\in L_2$, which is generated by G_2.

Star For the star operation we again use proof by construction but this operation works differently since only a single language L is involved. Let's define a CFG G_1 that generates L:

$$G_1 = (V_1, \Sigma, R_1, S_1)$$

When the star operation is applied to L, the new language L^* always includes the empty string. Every other string w it generates can be represented as $w = w_1w_2w_3 \dots w_{n-1}w_n$. Each piece w_i of the string $\in L$. For example if $A = \{0, 1\}$, then $A^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$. We can see a pattern that each string is recursively build from the smaller blocks. This is the pattern we need to represent using CFG by adding a new starting point in the following way:

$$\mathrm{S} \ \rightarrow \ \mathrm{S_1S} \ \mid \ \epsilon$$

The first part of the rule allows us to recursively build up the string by adding the building blocks that were originally in L generated by G_1. Each building block is added by S_1, while S is the part that we can either use to add another building block recursively or terminate by going to ϵ.

Proof

$$G_1 = (V_1, \Sigma, R_1, S_1)$$
$$G = (V_1 \cup S, \Sigma, R_1 \cup \{S- > S_1S|\epsilon\}, S)$$

Let's continue the example with the language A. The rules of the grammar G_1 are:

$$\mathrm{S_1} \ \rightarrow 0 \ \mid \ 1$$

Then for A^*, we have the grammar G:

$$\mathrm{S} \ \rightarrow \ \mathrm{S_1S} \ \mid \ \epsilon$$
$$\mathrm{S_1} \ \rightarrow 0 \ \mid \ 1$$

For example, let's try to generate the string 001. First S goes to S_1S three times. Each of the three S_1 becomes 0, 0, 1 respectively and each of the three S becomes ϵ.

3.1.5 We can do this by using proof by contradiction. We can take two CFLs apply the operation to them and show that the output is not a CFL using the Pumping Lemma for CFLs.

Intersection In the textbook exercise 2.2 on page 154, asks to prove this using the following two languages $A = \{a^m b^n c^n | m, n \geq 0\}$ and $B = \{a^n b^n c^m | m, n \geq 0\}$ so I will use them as well.

Let's first show that they are CFLs by constructing a CFG for each of them. For A we can see that each string can be divided in two parts. The LHS contains some number of a-s. The RHS contains the same number of b-s and c-s. The following CFG G_1 follows this pattern and can generate A:

$$\begin{aligned} \text{S} &\rightarrow \text{ML} \\ \text{M} &\rightarrow \text{aM} \mid \epsilon \\ \text{L} &\rightarrow \text{bLc} \mid \epsilon \end{aligned}$$

Analogously we can create the CFG G_2 that generates B:

$$\begin{aligned} \text{S} &\rightarrow \text{ML} \\ \text{M} &\rightarrow \text{aMb} \mid \epsilon \\ \text{L} &\rightarrow \text{bLc} \mid \epsilon \end{aligned}$$

Therefore, the languages A and B are CFLs.

Now let's construct their intersection $A \cap B = C = \{a^n b^n c^n | n \geq 0\}$. This is because if we pick any string from A it needs to have the same number of a-s and b-s by definition but because it is also in B it needs to have the same number of b-s and c-s too.

Now we need to show that outputted language C is not CFL by using the Pumping lemma, which will lead to a contradiction. Let's first define the pumping lemma for CFLs:

Pumping Lemma for context-free languages: If L is a CFL, then there exists an integer p (pumping length), which allows if w is any string in L with $|w| \geq p$, then w can be divided in five parts $w = uvxyz$, satisfying the following conditions:

(a) for each $i \geq 0, uv^i xy^i z \in L$

(b) $\|vy\| > 0$

(c) $\|vxy\| \leq p$

First, assume that C is a CFL. Let's choose the string $w = a^p b^p c^p$, where p is the value of the pumping length. This guarantees that the length of the string is always at least as big as the pumping length, in this case it is three times bigger. Let's decompose the string in the following way:

$$\begin{aligned} u &= a^m \\ v &= a^{p-m} \\ x &= b^p c^l \\ y &= c^q \\ z &= c^{p-l-q} \end{aligned}$$

This satisfies the second and third conditions. Not let's choose $i = 2$ to show that the fist condition breaks.

When $i = 2$ the outputted string is:

$$w = uv^2xy^2z$$
$$w = a^m a^{2p-2m} b^p c^l c^{2q} c^{p-l-q}$$
$$w = a^{2p-m} b^p c^{p+q}$$

This leads to a contradiction because there will be too many a-s and too many c-s. Therefore, C is not a context-free language. Therefore, the class of CFLs is not closed under intersection.

Complement

To prove this I will use Theorem 0.20 (De Morgan's Law) from the textbook that connect union, intersection, and complement:

Theorem 1 *Theorem 0.20 For any two sets A and B,*

$$\overline{(A \cup B)} = \overline{A} \cap \overline{B}$$

Let L_1 and L_2 be any two CFLs and let's assume they are closed under complement. That means that

$$\overline{L_1}$$

and

$$\overline{L_2}$$

are also CFLs. In the previous parts of the problem we already proved that CFLs are closed under union which means that

$$\overline{L_1} \cup \overline{L_2}$$

is also a CFL. Therefore, we can take the complement of that to get another CFL:

$$\overline{(\overline{L_1} \cup \overline{L_2})}$$

Now we can use Theorem 0.20 (De Morgan's Law) to show that the complement of their union also must be a CFL:

$$\overline{(\overline{L_1} \cup \overline{L_2})} = L_1 \cap L_2$$

This is a contradiction to what we proved before: CFLs are not closed under intersection. Since we got to a contradiction, it means that CFLs are not closed under complement.

Total Time to complete 3.1 5 hours and 20 minutes to fully complete.

3.2 Testing Grammars (#Automata, #Machines)

Recursive, Top-Down Approach

Let $G = (V, \Sigma, R, S)$ be a grammar in Chomsky normal form and $w = w_1 w_2 \dots w_n \in \Sigma^*$. Recall that if G is a CFG in Chomsky normal form, then for any string $w \in L(G)$ of length $n \geq 1$, exactly $2n - 1$ steps are required for any derivation of w.

3.2.1 As per the definition of Chomsky normal form in the textbook there are only two types of rules allowed in this form of the grammar:

$$\begin{aligned} \text{A} &\rightarrow \text{BC} \\ \text{A} &\rightarrow \text{a} \end{aligned}$$

In the definition a is any terminal, B, C are any variables other than the start variable, and the rule $\text{S} \rightarrow \epsilon$ is also permitted, where S is the start variable.

If the length of the string is $n \leq 1$, then that means the string $w = \epsilon \cup a$, where a is any terminal in the set of terminals Σ in the grammar G. This means that the length is $n = 0$ or $n = 1$.

If $n = 0$, then the rule does not hold and it means we are dealing with the empty string, which takes 1 step. For the empty string to be in the language then it means the grammar G in Chomsky normal form needs to contain a rule $\text{S} \rightarrow \epsilon$ since only the start variable can make the empty string.

If $n = 1$, then there is $2n - 1 = 1$ step we need to determine whether $w \in L(G)$. Since the length is 1, then $w = a$ where a is any terminal in the set of terminals Σ in the grammar G. Therefore, looking back at the two different types of rules, we cannot use $\text{A} \rightarrow \text{BC}$ because each of B and C will generate at least one terminal. But we can use the other type of rule $\text{A} \rightarrow \text{a}$ that produces exactly one terminal. Therefore, if $w \in L(G)$, there needs to be a rule $\text{S} \rightarrow \text{w}$ that generates w from the start variable.

3.2.2 As mentioned above, a rule of this type $\text{A} \rightarrow \text{BC}$ means that each of B and C will generate at least one terminal. This is because each of B and C will either output more variables or a single terminal.

If we only apply the other type of rule $\text{A} \rightarrow \text{a}$ or $\text{S} \rightarrow \epsilon$ then we would be stuck because it will terminate and no more rules will be applied. Therefore, we need to use the other type of rule so that the start variable produces two more variables $\text{S} \rightarrow \text{BC}$ and then each of them will produce at least one terminal resulting in length of $n \geq 2$. The string w that is produced is of length n. It can be broken into two parts. The first part $w_1 w_2 \dots w_i$ is produced by B and the second part $w_i w_{i+1} \dots w_n$ is produced by C.

3.2.3 For my recursive algorithm, I decided to implement a brute-force algorithm. It basically starts at the top of a parse tree from the start variable and then generates every possible parse tree. The recursive element is that it recursively call a function that expand variables of the string of size n, $n-1$, ... until the size is equal to 1 at which point we can reach a terminal. When the terminal is reached, it is added and the next variable is expended.

Since we do this by breaking up the string into all possible chunks, the time complexity depends on the size of the string. As we break the string in the following way $w_i, w_2 \dots w_j$

and $w_{j+1}, w_2 \dots w_p$ each of the indexes i, j, p can go up to the magnitude of n. Therefore, as we examine all the possible combinations the time complexity of the recursive algorithm is $O(n^3)$.

In the worst-case scenario it generates all possible parse trees before returning a rejection (the string cannot be generated) $O(n^3)$. In the best case scenario the string is the first one to be generated $O(n)$. This is the case if it can be generated by always following the left-most nodes (terminals or variables to be expanded).

Pseudocode for the Recursive Algorithm

Given a CFG $G = (V, \Sigma, R, S)$ in Chomsky normal form, $w \in \Sigma^*$:

Initialize a stack (using a list). We will use it to keep track of the variables to be expanded. Initially add the start variable S.

current = stack.pop (to get the next variable to be expanded)

permute(w, w[0:1]) - this is the recursive step. Call this function that will explore all possible combinations of sub-strings and how to generate them recursively.

if permute(w, w[0:1] returns True accept, otherwise reject

def permute(string, substring):

.... if len(substring) == 0 – if we have managed to generate the string

.... return True

.... if len(substring) == 1 this is the base case

.... get the terminal and remove it from the string

.... else for i in range(len(substring)):

.... permute(string, substring based on i) - this is the recursive call

.... return False

Dynamic, Bottom-up Approach

The CYK parsing algorithm uses dynamic programming to try to solve the this problem by storing the sets of rules that could generate w in a table.

Pseudocode for the CKY algorithm

Given a CFG $G = (V, \Sigma, R, S)$ in Chomsky normal form, $w \in \Sigma^*$:

Set $n = |w|$

Intialize an empty $n \times n$ table

for $i = 1$ to n:

....for each variable A in grammar:

.... if $A \to b$ is a rule and $b = w_i$, add A to cell $(1, i)$

for $l = 2$ to n:

.... for $i = 1$ to $n - l + 1$:

.... for $j = i + l - 1$:

.... for $k = i$ to $j - 1$:

.... for each rule $A \to BC$:

.... if $B \in (i, k)$ and $C \in (k + 1, j)$, put $A \in (i, j)$

if $S \in (n, 1)$, accept string

3.2.4 Let G be the context-free grammar in Chomsky normal form with the following rules.

$$S \to TT|UB|AB|\varepsilon; T \to TT|UB|AB; U \to AT; A \to a; B \to b.$$

(a) If we perceive the symbol a as a left parenthesis (and b as a right parenthesis), then this grammar generates the language L of all matched sets of balanced parenthesis.

It is the Chomksy normal form the of the following grammar:

$$\text{S} \ \to \ \text{SS} \ \mid \ \text{aSb} \ \mid \ \epsilon$$

(b) I believe there are some slight errors in the pseudocode provided above. There should not be a for-loop with the variable j. I assume that line strictly initializes j rather than beginning another for loop.

Another error is with the indexing of the table. The way it is set up currently, in the first nested for loop:

for $i = 1$ to n:

....for each variable A in grammar:

.... if $A \to b$ is a rule and $b = w_i$, add A to cell $(1, i)$

We always add the variables in the first row, but one space to the right every time the loop goes. However, later on when we check this condition:

.... if $B \in (i, k)$ and $C \in (k + 1, j)$, put $A \in (i, j)$

Let's look at C. We check if $C \in (k + i, j)$. The minimum value of k is $k = i = 1$. So we will always check at least the second row of the table. This does not align with the way we have set up the table earlier. A way to fix this is to not always add the variables in cell $(1, i)$, but also move them one row down. So I will add them to cell (i, i). So this is the modified line of code:

.... if $A \to b$ is a rule and $b = w_i$, add A to cell (i, i)

The other small thing is that when we check for the starting variable, we check for it at $(n, 1)$ so the n-th row of the first column. The time we would add the S variable can occur here:

.... if $B \in (i, k)$ and $C \in (k + 1, j)$, put $A \in (i, j)$

So we should actually check the first row of the n-th column $(1, n)$

Modified pseudo code:

Pseudocode for the CKY algorithm

Given a CFG $G = (V, \Sigma, R, S)$ in Chomsky normal form, $w \in \Sigma^*$:

Set $n = |w|$

Intialize an empty $n \times n$ table

for $i = 1$ to n:

....for each variable A in grammar:

.... if $A \to b$ is a rule and $b = w_i$, add A to cell (i, i)

for $l = 2$ to n:

.... for $i = 1$ to $n - l + 1$:

.... $j = i + l - 1$:

.... for $k = i$ to $j - 1$:

.... for each rule $A \to BC$:

.... if $B \in (i, k)$ and $C \in (k + 1, j)$, put $A \in (i, j)$

if $S \in (1, n)$, accept string

I will choose $w = ab$ and $v = aa$ and parse both of them simultaneously.

First we are given CFG from the previous problem. Then we set $|n| = 2$ for both strings. We initialize an empty 2x2 table.

Now let's enter the first loop $for i = 1 to n$. It goes twice. When $i = 1$, we traverse the set of variables $\{S, T, U, A, B\}$ in the grammar. For S, T, U there is no rule that satisfies the if-statement. For B there is, but $b \neq w_1 = v_1 = a$. For A the if-statement is satisfied so we update the table by adding A in cell $(1, 1)$. Then we repeat the same for $i = 2$. For the string w_2, the variables S, T, U, A do not satisfy the if-statement, but B does because of $B \to b$ and $w_2 = b$. Therefore, we update that table by adding B to cell $2, 2$. For the string v, we follow the analogous logic and add A to cell $(2, 2)$.

Now we enter the second main for-loop "'for l=2 to n"'. So the loop only goes once since $n = 2$ as well.

The first nested loop "'for i=1 to n-l+1"' also loops only once since $n-l+1 = 2-2+1 = 1$. Then we set $j = i + l - 1 = 1 + 2 - 1 = 2$. Now we enter the next nested for loop "'k=i to j-1"', which again will only go once since $j - 1 = 2 - 1 = 1 = k$.

Now we start traversing the set of rules. Particularly the rules of type $A \to BC$. Let's first show how it works for the string $w \in L$.

Every time the loop goes it inspect a particular rule. It checks whether the first variable is in cell $(i, k) = (1, 1)$, and the second variable in cell $(k + 1, j) = (2, 2)$.For the following rules nothing happens since it does not find B, C in the respective cells: $S \to TT|UB, T \to TT|UB, U \to AT$. However, let's see what happens when it checks

this rule: $S \to AB$. It checks for A in cell $(1,1)$ of the table and finds it. It checks for B in cell $(2,2)$ and also find it. So it adds S to cell $(i,j) = (1,2)$. The same process is repeated for the rule $T \to AB$, so now cell $(1,2)$ of the table contains (S,T). Once all rules are checked, all loops are exited because they are done.

We go to the exit condition. In this case $S \in (1,n) = (1,2)$, which is true and the string $w = ab$ is accepted.

The same is repeated for the other string v. However, there the last loop that traverses the rules of type $A \to BC$ never satisfies the if-condition since there is no rule that has AA on the right-hand side. Therefore, when we go to the exit condition the start variable S is not found and the string is rejected.

3.2.5 Implement the CYK Algorithm in Python.

First I defined a class to be able to create instances of a CFG.

```
class CFG():
    """
    A class to define CFGs.
    Parameters:
        variables: a set strings
            The set of distinct variables in the CFG.
        terminals: a set strings
            The set of distinct terminals in the CFG.
        rules: dictionary with strings for keys and lists for items
            Dictionary that stores the rules of the grammar.
        start_var:
            The start variable of the grammar.
    """
    def __init__(self, variables, terminals, rules, start_var):
        self.variables = variables
        self.terminals = terminals
        self.rules = rules
        self.start_var = start_var
```

Figure 10: CFG class Python Implementation

Then, I defined the grammar G that was given earlier in the problem in the following way:

```
rules = {
     "S": [["T", "T"], ["U", "B"], ["A", "B"], [" "]], # epsilon is represented as " "
     "T": [["T", "T"], ["U", "B"],["A", "B"]],
     "U": [["A", "T"]],
     "A": [["a"]],
     "B": [["b"]]
    }

variables = {"S", "T", "U", "A", "B"}
terminals = {'a', 'b'}


G = CFG(variables, terminals, rules, "S")
```

Figure 11: Python definition of CFG generating balanced parenthesis language

Then, I turned the pseudo code into Python and also added several tests with assert statements to make sure that my code works correctly.

```python
def CYK(input_string, CFG):
    """
    A function that represents the CYK algorithm (bottom up approach) parser.

    Parameters:
    input_string: string
        The string we would like to check
    CFG: class CFG
        The CFG in Chomsky normal form.
    """

    # Initialize length of string.
    n = len(input_string)

    # Initialize empty nxn table.
    # Each cell is a set containing distinct variables.
    table = [[set([]) for i in range(n)] for j in range(n)]

    # Update the table.
    for i in range(n):
        # Add each variable to the table if it meets the conditions.
        for rule in G.rules.items():
            # If the rule only contains a terminal and it matches the symbol of the string.
            for rule_ in rule[1]:
                if len(rule_) == 1 and rule_[0] == input_string[i]:
                    # Update table.
                    table[i][i].add(rule[0])

    # Break up the input string
    for l in range(2,n+1):
        for i in range(1, n-l+1+1):
            j = i+l-1
            for k in range(i, j):

                # Find the rules of type A->BC.
                for rule in rules.items():
                    for rule_ in rule[1]:
                        if len(rule_) >= 2:

                            # Check if B and C are in the respective table cells.
                            if rule_[0] in table[i-1][k-1] and rule_[1] in table[k][j-1]:
                                # If so, update table, add the variable.
                                table[i-1][j-1].add(rule[0])

    # If starting variable in top right-corner, accept.
    if G.start_var in table[0][n-1]:
        return True
    # Otherwise, reject.
    else:
        return False
```

```python
assert CYK(' ', G) == True # empty string
assert CYK('a', G) == False # buidling blcok
assert CYK('b', G) == False # bulding block
assert CYK('ab', G) == True # shortest string
assert CYK('aaaaaabbbbbb', G) == True # longer string a^nb^n
assert CYK('ababaababb', G) == True # longer string with harder format
```

Figure 12: CYK Python Implementation

3.2.6 **Theoretically**

The first part of the algorithm creates the table by looking at all variables. In my implementation the code actually looks at all rules. Either way let's call that number x. The outer loop goes for n times (the size of the string). This leads us to $O(n \times x)$. Either way, we will see that this does not matter because the second part of the algorithm has a higher magnitude so we will disregard this part completely.

After creating the table, the main part of the algorithm basically looks at every possible way to split the input string by using three indexes l, i, k. The splits are made this way: $w_l \dots w_i$ and $w_{i+1} \dots w_k$. For these splits there are n^3 combinations, where n is the length of the string, since each of the indexes l, i, k can be of order n. This gives us $O(n^3)$.

For each of these combinations we traverse the rules of the grammar and update the table. This allows us to store which part of the string can be generated by which variables. Therefore, we also need to take into account the size of the grammar which does not depend on the size of the string. This leads us to $O(n^3|G|)$, where $|G|$ is the size of the grammar.

We would have to add this factor $O(n^3|G|)$ to the term to create the table (discussed earlier) but since the latter because it is a lower-order term we can drop it. The final time complexity is $O(n^3|G|)$, where $|G|$ is the size of the grammar and n is the length of the string.

I will not differentiate between best, average, and worst-case because the algorithm goes through all the same steps every time it runs.

Experimentally

To test this out experimentally we can either use time or number of steps as a measure. For my approach I will choose time because a step can be defined in different ways and could be subjective.

I created a plot for the run time of the algorithm. On the x-axis I plotted the length of the input string (n) in terms of number of characters and on the y-axis I plotted the running time in seconds, which shows how long the algorithm took to execute. I varied the size of the string from $n = 1 to n = 500$ while keeping the size of the grammar constant. This means we expect the time to scale in a factor of $O(n^3)$. Reference the figure below:

```
# Store the tuple of the x and y-axis lists.
dynamic_plot_values = graph_time(503, CYK)
# Plot the function and set title, labels, units, and legend.
plt.plot(dynamic_plot_values[0], dynamic_plot_values[1])
plt.title('Runtime of CYK Bottom Up Parser')
plt.xlabel('Size of input string')
plt.ylabel('Time (s)')
plt.legend()
plt.show()
```

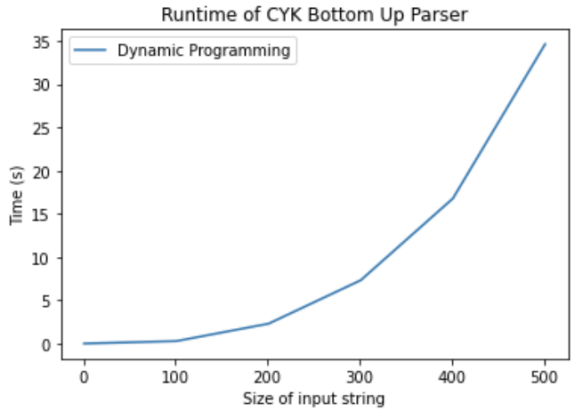

Figure 13: Runtime Complexity of CYK

I wanted to vary it up to at least 1000 but the plot took longer than an hour to generate so I had to terminate the Kernel. Nevertheless, as we can see from the plot the time grows exponentially as the input size increases. We can compare some milestone values of the x-axis to determine the schale factor of the graph itself. If we do that, we can see that it approximates $O(n^3)$.

In my approach I have 100 trials for each input size and take the average of the times. This is to make sure the runtime is accurate and we did not fall into a best or worst case scenario. The code can be seen below.

```python
import random
import matplotlib.pyplot as plt
import time


def graph_time(n, algorithm):
    """
    Produce 2 lists for the x and y-axis of an algorithm runtime graph.
    ---------
    Parameters:
    - n: int
        Maximum length of input string we want to reach.
    - algorithm: function
        The algorithm we want to test.
    Returns:
    - tuple: a tuple of two lists with values for the x-axis (number of activities) and
            y-axis the average runtime after 100 trials it took the algorithm.
    """
    # Take every 100 points.
    # At first the step was smaller at 10 to allow for more data points,
    # but they take an incredibly long time to compute so I ahd to icnrease it.
    x_axis = list(range(1,n,100))
    y_axis = []


    for i in range(2,n,100):
        temp_time = 0
        # Have 100 trials.
        for j in range(100):
            start = time.time()

            # Create input strings of different sizes.
            random_input = ''.join(random.choices(['a','b'], k=i))

            # Call algorithm.
            algorithm(random_input, G)

            end = time.time()
            temp_time += end - start

        # Take the average time.
        y_axis.append(temp_time/100)

    return (x_axis, y_axis)
```

Figure 14: Python code to Generate Runtime Complexity Graph

My experimental approach confirms my theoretic derivation.

To take this further we should create another plot where we keep the size of the string constant while we vary the size of the grammar. However, since I generating random grammars of different sizes will take a long time I have left this out. We would expect to see that the time scales linearly $O(g)$ in terms of the size of the graph g.

Total Time to complete 3.2 7 hours to fully complete

3.2.7 *(Optional Challenge)*

I would have loved to attempt this! I know I can probably solve it but I have already spent too long on the assignment.

4 Reflection

4.1 Creating the context-free grammars for languages is like art. However, I will discuss the Python implementation of constructing an NFA from a regular expression and then validating a string. There were many different components that go into all of the code: breaking down the problem effectively, writing separate functions to stay organized, and then seeing how all of them come together in the end to compute the entire process. It is like being a mastermind.

4.2 **#heuristics** I applied this HC while completing the code for 2.2.1. In order to complete it more efficiently, I applied abstractions: I separated all logical components of the algorithm into their own packages. For example, I created a class for the NFA that has its attributes and methods. I created a function for the main algorithm that calls all other relevant function when they are needed (e.g. update the transition function, construct an NFA for union, construct NFA for concatenation, validate input using the final NFA). This allowed me to reuse a lot of the code rather than having to type it again. All details were abstracted so the code is also easier to follow and read, and hence easier to debug. I also did the same when generating the graphs as I made a function that generates the x and y-axis. However, to improve my heuristic of abstraction, I can put all the code in a main function. Now I still have to copy paste it in order to manually change the random inputs. Rather then doing that, they can all go as parameters of the function, including title, axis-labes, and legend if needed.

#deduction I applied this HC throughout the whole assignment when constructing proofs. Let's focus on problem 2.1. To prove that the class of regular languages is closed under homomorphism I explained the procedure step by step using proof by construction, which effectively proved the claim. Then, I utilize proof by contradiction to show that the class of non-regular languages is not closed under homomorphism. To build the argument, I list several premises. First I assume that the claim is true. Then I utilize the pumping lemma to show a non-regular language, I apply the operation on it to get a regular language. This showed that the initial assumption was in fact wrong. The conclusion necessarily follows from the premises.