In this lab, we covered 2 algorithms to search for a pattern within a given string. The first algorithm is the naive approach and simply traverses the string trying to match a substring at the i-th position to the pattern. This approach works well for small texts but is rather slow for larger texts.

# Exercise 1: Naive Approach

```python
"""Simple method to compare the pattern against the text"""
def naive_pattern_match(text, pattern):
        match_idx = []

        # iterate over |text-pattern| indicies and check for the pattern one indicie higher each time
        for i in range((len(text) - len(pattern) + 1)):
                # compare the substring starting at the ith position with length pattern to pattern
                if str(text[i:len(pattern)+i]) == pattern:
                        match_idx.append(i) # add to matches

        if not match_idx:
                return -1
        else:
                return match_idx

print(naive_pattern_match('ccccabcdefabc','abc'))
```

## Output

```
/usr/local/bin/python3.9 /Users/aaronfeinberg/Projects/classWork/CMPSC413/lab3/naive_pattern_matching.py
[4, 10]

Process finished with exit code 0
```

# Exercise 2: Knuth-Morris-Pratt (KMP) Algorithm

```python
"""KMP Pattern Matching Algorithm"""

def lps_generator(pattern: str) -> []:
        pattern = list(pattern) # converts the string into a list for traversal

        lps = [0] # create a list to hold our lps table
        pattern_length = 0 # length of the current pattern
```

```
        idx = 1 # initial search position

        # search through the entire pattern
        while idx < len(pattern):

                # match
                if pattern[idx] == pattern[pattern_length]:
                        lps.append(pattern_length+1) # add length+1 to position
lps[idx]
                        idx+=1 # increment index
                        pattern_length+=1 # increment length

                # charecters don't match
                else:
                        if pattern_length!=0:
                        pattern_length=lps[pattern_length-1] # return len back to
the last matching char
                        lps.append(0) # start over again with pattern search
                        idx+=1

        return lps

def kmp_string_search(text, pattern, lps=None):
        if not lps:
        lps = lps_generator(pattern) # generate lps

        # printing data
        print("\n---- KMP Search -----")
        print(f"Text=[{text}]")
        print(f"Patt=[{pattern}]")
        print(f"\nLPS={list(pattern)}\n\t{[str(i) for i in lps]}\n")

        # declare tracking variables
        text_index, pattern_index = 0, 0
        match_idx = [] # list to store the start indices of matches

        while text_index < len(text):
                # chars match -> advance
                if text[text_index] == pattern[pattern_index]:
                        pattern_index += 1
                        text_index += 1

                        # check for a full match of the pattern
                        if pattern_index == len(pattern):
                                match_idx.append(text_index - pattern_index) #
append the start index of the match
                                pattern_index = lps[pattern_index - 1] # reset
pattern_index using the LPS
```

```
                # mismatch
                else:
                        # if pattern_index is not 0 -> reset it using LPS to skip
comparisons
                        if pattern_index != 0:
                        pattern_index = lps[pattern_index - 1]
                        # advance if pattern index is 0
                        else:
                        text_index += 1

        print(f"Match indecies: {match_idx}")
        return match_idx

test_pattern = 'ababa'
kmp_string_search('abababbababa', test_pattern)
```

## Output

```
/usr/local/bin/python3.9
/Users/aaronfeinberg/Projects/classWork/CMPSC413/lab3/kmp_pattern_matching_algori
thm.py

---- KMP Search -----
Text=[abababbababa]
Patt=[ababa]

LPS=['a', 'b', 'a', 'b', 'a']
        ['0', '0', '1', '2', '3']

Match Indicies: [0, 7]

Process finished with exit code 0
```

# Conclusion

The naive approach is simple but suffers from some inefficiencies that the KMP Algorithm avoids. In the worst case, we would be making *m* comparison for each letter in the pattern over an array of (n-m+1) elements so we'd get at worst a time complexity of $O(nm)$ comparisons which is still $O(n)$ but will grow faster for large n's or m's.

The KMP Algorithm works by first traversing the pattern and finding the lengths of proper prefixes which are also proper suffixes in the pattern. This is then stored in an lps array. The second part of this algorithm traverses the actual text and uses the information of the lps array to help skip ahead in checks when a sub-pattern match fails. The LPS array is used to lookup the length of the previous match and move forward that amount, guaranteeing that we wont skip any potential matches along the while also not performing unnecessary checks.

This Algorithm takes $O(n + m)$ time since we have to create an lps table by traversing a pattern of m elements and then at worst, we'll compare $O(n)$ elements since we wont have to check the whole pattern during a comparison.