**CPSC 335 - Algorithm Engineering**  
**Project 3 Report**  
**Spring 2025**  
**Due: 04/11**

Wayne Muse  
88601211  
CSUF Email**:** [waynemuse@csu.fullerton.edu](mailto:waynemuse@csu.fullerton.edu)

Submission for Project 3

# Introduction

This project involves the implementation and analysis of three algorithms:

1. **The Spread of Fire in a Forest**
2. **Delivery Route Planning**

# **Algorithm 1: Spead of Fire**

## Problem Description

Imagine you are a forest ranger monitoring a forest that has recently experienced a wildfire. The forest is represented by a grid, where each cell can either be:

* 0: an empty area (no trees),
* 1: a healthy tree,
* 2: a burned tree (representing a tree affected by the wildfire).

The management needs to determine how many days it will take for all healthy trees to burn down, considering that every day, any healthy tree that is adjacent (up, down, left, or right) to a burned tree will also burn down. You need to calculate the minimum number of days it will take for all healthy trees to burn, or return -1, if it is impossible for some trees to burn (i.e., if there are healthy trees that are not adjacent to any burned trees).

Details:

1. A healthy tree (represented by 1) will burn down after one day, if it is adjacent to a burned tree (represented by 2).
2. An empty area (represented by 0) does not affect the burning process.
3. If there are no burned trees initially, it is impossible for any healthy trees to burn, so return -1.

## Pseudo Code

Function target\_terms(s, x):

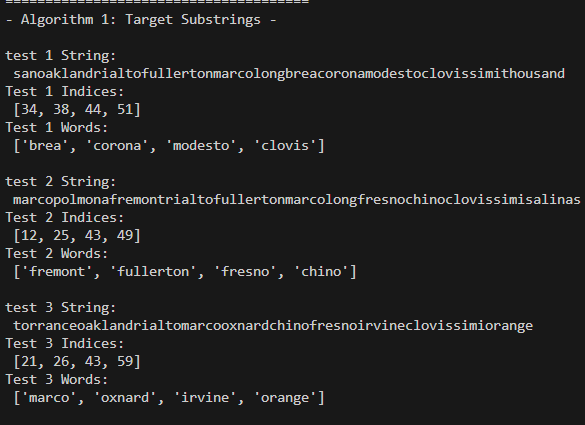
result = []  
 for each word in x:  
 index = s.find(word)  
 if index != -1:  
 result.append((index,word))  
  
 result.sort()  
 indices = [indexs in result]  
 words = [words in result]  
 return indices, words

## Time Complexity Analysis

Initialize result: **1 step**  
Loop through each target word (m words): **m steps**  
.find(word) scans S (length n): O(n) and O(m), thus = **m\*n**  
Appending to result: up to **m**  
Sorting the result ( assuming size ≤ m): **O(m log m)**  
Extracting index and word lists: **2m**  
Return :**1** step  
Thus, the total number of steps for this algorithm: 1 + m\*n + m log m + 2m + 1

Big-O Efficiency: O(m·n + m·log m)

## Example Execution

****

# **Algorithm 2: Run Encoding Problem**

## Problem Description

Given a string of lowercase letters and spaces, compress the string using run-length encoding. Consecutive repeated characters are replaced by the count followed by the character (e.g., "aaabb" becomes "3a2b"). Single characters remain unchanged.

The algorithm iterates over the input string, counting how many times each character repeats consecutively. It then appends either the count and character or just the character to the result list. At the end, it joins and returns the result.

## Pseudo Code

Function run\_encod\_prob(s):

encode = []  
count = 1

For i from 1 to length(s) - 1:

If s[i] is equal to last element(s[i-1]):

count += 1

Else:

If count > 1:

encode.append(count + s[i-1])

Else:

encode.append(s[i-1])

count = 1

If count > 1:

encode.append(count + s[-1])

Else:

encode.append(s[-1])

Return ''.join(encode)

## Time Complexity Analysis

Init Variables: 2

Loop from i to n-1: n-1 steps

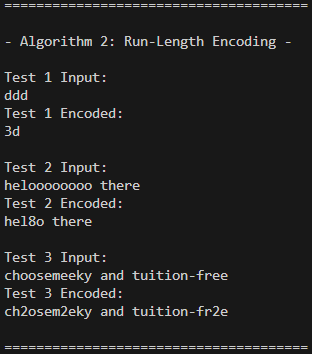
encode append: 2 steps

join encode: n

Total Steps: 13n-4

Big O efficiency: O(n)

## Example Execution



# **Algorithm 3: Merging Arrays**

## Problem Description

Given multiple sorted integer arrays, merge them into a single sorted array. The goal is to do this efficiently using a min-heap so that the result preserves the sorted order without performing a full sort on the final array.

This algorithm uses a min-heap to track the smallest elements across all arrays. Initially, the first element from each list is pushed into the heap. As elements are popped from the heap, their successors in the respective lists are pushed back in.

## Pseudo Code

Import heapq module

Function merge\_sorted\_arrays(arrayofarrays):

result = []

min\_heap = []

For i, array in arrays:

If array is not empty:

heapq.push((array[0], i, 0))

While min\_heap is not empty:

val, list\_idx, elem\_idx = heapq.pop()

result.append(val)

If elem\_idx + 1 < len(arrays[list\_idx]):

next\_val = arrays[list\_idx][elem\_idx + 1]

heapq.push((next\_val, list\_idx, elem\_idx + 1))

Return result

## Time Complexity Analysis

Initialize result and heap: 2

heap push over ‘k’ loop: k \* log k

While loop over n total elements:

heap pop: O(log k)

append: O(1)

conditional heap push: O(log k)

Total steps: 2n log k + k log k + n  
Overall time complexity: O(n log k)

## Example Execution

# 

# Project Components

The project submission includes the following files:

* **README.md** – Contains instructions for running the project.
* **main.py** – The main driver program to execute the algorithms.
* **alg1.py** – Implementation of the Target Terms
* **alg2.py** – Implementation of the Encoding String
* **alg3.py** - Implementation of the