Team Members:

Caleb Heydon

Luke Pepin

Jordan Howard

Mitchell Worthington

Selected Project:

Rabin-Karp String Search Algorithm

**Milestone Check**

The milestones outlined in the proposal are showcased more in detail down below with completion markers (Done ✔, Started ⬔, Not Started ✖ ). We are currently on track to complete the project with the current project plan.

Project Plan:

Week 4-5: Team Formation and Algorithm Selection

* **✔** Form teams of 3-4 members.
* **✔** Choose an algorithm Rabin-Karp String Search Algorithm, not covered in class.
* **✔** Submit Group Project proposal on HuskyCT

Week 6: Research

* **✔** Start research and gain a deep understanding of the topic
* **✔** Create a document with the highlights of the research
* **✔** Focus on creating notes in a format that easily goes into report writing

**Research Progress Key Findings:**

The Rabin-Karp string search algorithm is a pattern matching algorithm used to find occurrences of a substring (or pattern) within a larger text or string. It was developed by Michael O. Rabin and Richard M. Karp in 1987. The algorithm is known for its efficiency in searching for patterns in text, particularly in situations where multiple patterns need to be searched for simultaneously.

The Rabin-Karp algorithm employs a technique called hashing to compare substrings of the text with the pattern. Here's a high-level overview of how it works:

1. Preprocessing:

- Calculate the hash value of the pattern.

- Calculate the hash value of the first substring of the text, which has the same length as the pattern.

2. Search Process:

- Compare the hash value of the current substring in the text with the hash value of the pattern. If they match, it means there is a potential match between the pattern and the current substring.

- If the hash values match, perform a character-by-character comparison of the pattern and the current substring to confirm the match.

- If the hash values don't match, move the sliding window one character to the right in the text and calculate the hash value for the new substring. Repeat the hash comparison and character comparison until you find a match or reach the end of the text.

3. Handling Hash Collisions:

- Hash collisions can occur when different substrings have the same hash value. To handle this, the algorithm uses a rolling hash function that efficiently updates the hash value as the window slides.

The key advantages of the Rabin-Karp algorithm include its ability to search for multiple patterns in a single pass and its average-case time complexity of (theta)(N+M), where N is the length of the text and M is the length of the pattern. However, the worst-case time complexity can be O(N\*M) if hash collisions are not handled properly.

The Rabin-Karp algorithm is particularly useful when searching for multiple patterns in the same text or when you need to perform approximate string matching by allowing a certain level of mismatch tolerance. Additionally, it has applications in plagiarism detection, file comparison, and DNA sequence matching, among others.

Knuth-Morris-Pratt (KMP) Algorithm: The KMP algorithm efficiently searches for a pattern in a text by preprocessing the pattern to create a partial match table, which is used to skip unnecessary comparisons. It has linear time complexity and is known for its robustness in string searching.

Boyer-Moore Algorithm: The Boyer-Moore algorithm is another widely used string searching technique. It uses a heuristic approach, focusing on character comparisons from right to left, and can significantly reduce the number of character comparisons in practice.

Aho-Corasick Algorithm: This algorithm is designed for multiple pattern matching. It constructs a finite automaton that can search for multiple patterns simultaneously in linear time.

Brute-Force Algorithm: Brute-force string searching is the simplest method where you compare the pattern to the text character by character. While it's not as efficient as the other algorithms mentioned, it serves as a baseline for performance comparison.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Week 7-8: Algorithm Implementation

* **✔** Start developing the implementation of the selected algorithm in python.
* **✔** Ensure the code is well-documented and follows coding standards.
* **✔** Begin drafting the implementation details section of your report.

Rabin\_Karp.py

'''Rabin-Karp Algorithm: Finds a text pattern in a body of text'''

def rabin\_karp\_search(text, pattern):

'''Calculates the hash of the pattern we'd like to find'''

def calculate\_hash(string, length):

hash\_value = 0

for i in range(length):

hash\_value = (hash\_value \* 256 + ord(string[i])) % prime

return hash\_value

'''Function to handle hash collisions'''

def rehash(old\_hash, old\_char, new\_char, length):

new\_hash = (old\_hash - ord(old\_char) \* (256\*\*(length-1)) % prime) \* 256 + ord(new\_char)

return new\_hash % prime

prime = 101 # A prime number for the hash function (can be any prime)

text\_length = len(text)

pattern\_length = len(pattern)

pattern\_hash = calculate\_hash(pattern, pattern\_length) # calculate initial hash of the pattern

text\_hash = calculate\_hash(text, pattern\_length) # calculate initial hash of the text

# Iterate through the text from start to end of where the pattern could be found

for i in range(text\_length - pattern\_length + 1):

# Check if the current hash of the pattern and the hash of the current substring match

# Also compare the substring with the pattern to confirm if its a match

if pattern\_hash == text\_hash and text[i:i+pattern\_length] == pattern:

print(f'Starting at index {i}, "{pattern}" is in "{text}"')

# if there are more characters in the text to process, update text\_hash for the next iteration

if i < text\_length - pattern\_length:

text\_hash = rehash(text\_hash, text[i], text[i+pattern\_length], pattern\_length)

# Example usages

text = "ABABDABACDABABCABAB"

pattern = "ABABCAB"

rabin\_karp\_search(text, pattern)

text2 = "Q&%P7^@t4)Rf\*a9N|vW#sCzB+dLhM~wK6$Y2xT1U8jI=eG3ZyXoV5nDqO!A\_~{gH`[]?l"

pattern2 = "ZyXoV"

rabin\_karp\_search(text2, pattern2)

text3 = "The crimson leaves rustled in the autumn breeze as the sun dipped below the horizon."

pattern3 = "sun"

rabin\_karp\_search(text3, pattern3)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Week 9-14: Report Writing

* ⬔Write the report, under the standards the syllabus outlined including:

Report link: <https://docs.google.com/document/d/1NTdbnJ4mJtVQhg7l2dT7UD_TXAT0jQv2Kavjytipphs/edit?usp=sharing>

* + ⬔ Introduction
  + ⬔ Comparison with other Algorithms
  + ✖ Justification of its superiority
  + ✖ Detailed explanation of the algorithm
  + ⬔ Implementation details
  + ✖ Results
  + ✖ Analysis of time complexity with proofs
  + ✖ List of references
  + ✖ Complete the report and ensure little errors are made.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Implementation Status**

We are mostly complete with implementing the algorithm. It was challenging at first to understand how the algorithm worked with the hashing and hash collisions. Solving this problem required extensive research. There were many videos and images online explaining the algorithm, which greatly improved our understanding of the algorithm. As for coding the hash, it took a lot of trial and error figuring out what to add, multiply, and take the remainder of, but we figured it out in time. Otherwise, it was straightforward to code how the algorithm iterates through the list. One thing to improve upon in the algorithm is it currently stops after finding one instance of the pattern in the string. In the future we will implement it to find all instances of the pattern in the string.

**QA Testing**

During development we got lots of “index out of range” errors. Lots of our tests involved trial-and-error figuring out the correct ranges to use. However, we eventually got it to work.

**Evaluation Report**

Our team has effectively coordinated tasks and communicated by setting up a group text loop as well as sharing members’ discord accounts. Members have met through a combination of virtual meetings on discord as well as in person.

Distribution of Tasks:

Task List - Luke, Jordan

Implementation - Luke (coding), Mitchell (comments/examples)

Research - Mitchell, Caleb

Project Update - Mitchell, Caleb

| Team Member | Rating (of assigned tasks) |
| --- | --- |
| Caleb Heydon | 100 % |
| Luke Pepin | 100 % |
| Jordan Howard | 100 % |
| Mitchel Worthington | 100 % |