Vietnam National University – Ho Chi Minh city
University of Science
Faculty of Computer Science

COURSE PROJECT

DATA STRUCTURES AND ALGORITHMS

PROJECT of SEMESTER 1
2020 – 2021

STRING MATCHING

Class: 19CLC10
Instructor: Bùi Huy Thông

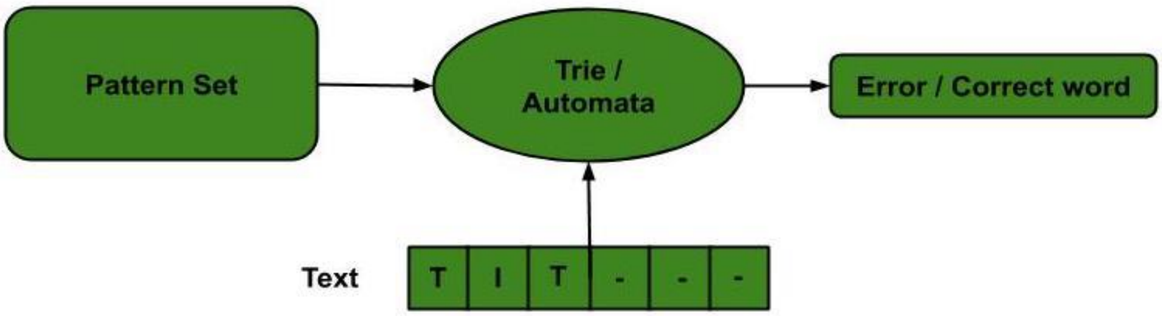
Full name	Students ID	Work	Accomplished
Trương Gia Đạt	19127017	_Research Knuth – Morris – Pratt algorithm. _Code Crossword game	100%
Nguyễn Văn Bình	19127104	_Research Rabin – Karp algorithm.	100%
Nguyễn Thành Hiệu	19127144	_Research the definition of string matching and outline two applications	100%
Lê Xuân Dĩnh	19127362	_Research Brute – Force algorithm.	100%

Ho Chi Minh , 2020

CONTENTS

INTRODUCTION

- **String matching algorithms** help in performing time-efficient tasks in multiple domains.
- The object of **string searching** is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).
- **String matching algorithms** are generally divided into two types of algorithms:
  - Exact String Matching Algorithms.
  - Approximate String Matching Algorithms.
- **Problem definition:** Given an alphabet A, a text T(an array of n characters in A) and a pattern P(another array of $m \leq n$ characters in A), we say that P occurs with shift s in T (or P occurs beginning at position $s + 1$ in T) if $0 \leq s \leq n - m$ and $T[s + j] = P[j]$ for $1 \leq j \leq m$. A shift is valid if P occurs with shift s in T and invalid otherwise. String-matching problem is the problem of finding all valid shifts for a given choice of P and T.
- **Notations:**
  - s, Strings will be denoted either by single letters or, occasionally, by boxed letters.
  - A, the set of all finite-length strings formed using character from the alphabet A.
- **Applications:**
  - **Spell Checkers**: We build a "trie" of pre-defined set of patterns. This trie is used for the string matching means if any such pattern occurs then it shows the occurrence by reaching to its final states.

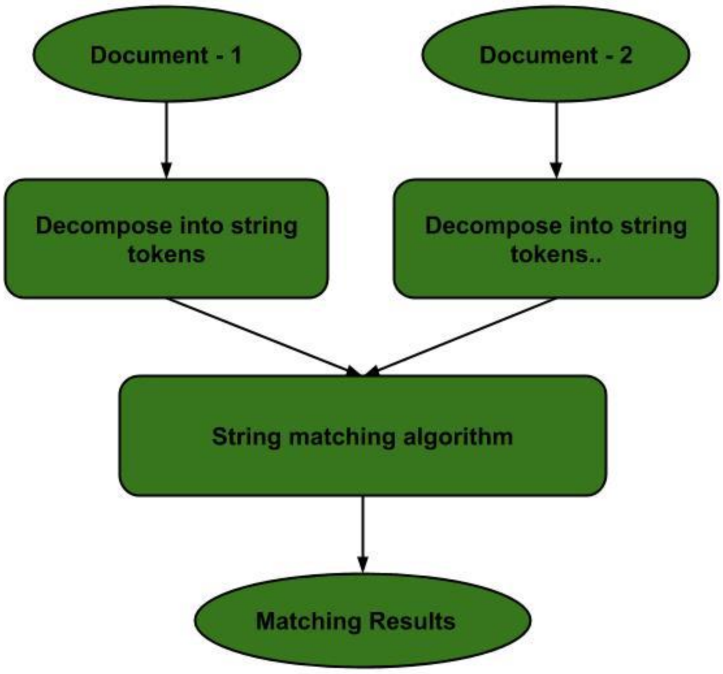


Input: text = algorimth.

trie = algorithm.

Output: Error, change "algorimth" to "algorithm".

- **Plagiarism Detection:** We use **String searching** algorithm to compare texts and find the similarities between them. From that we can conclude whether the original text is plagiarism or not.



Input:

Document – 1: Start with a well-written resume that has appropriate keywords for your occupation.

Document – 2: Begin by writing a good resume with appropriate keywords for your occupation.

Ouput: PLAGIARISM.

RESEARCH

1. **Brute – Force (Naïve String Matching)**

_The Brute – Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found.

_The algorithm can be designed to stop on either the first occurrence of the pattern or upon reaching the end of the text.

_ Explanations: Assume text as T with n size and pattern as P with m size. ($m \leq n$)

+Step 1: Iterating through the main string (text). j always start at index 0.

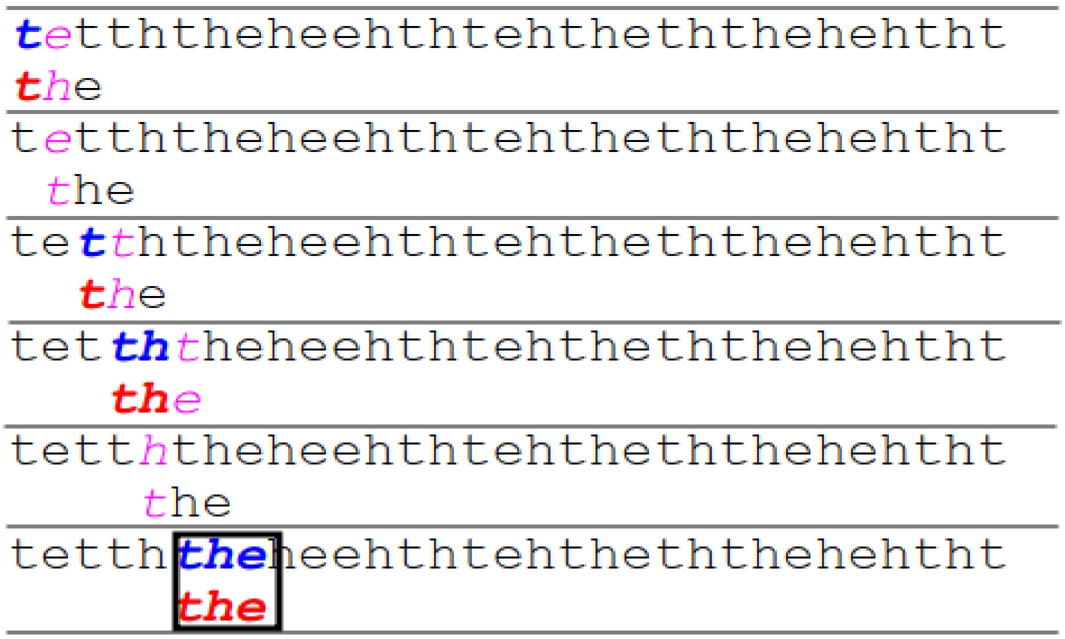
If T[i] != P[j], then i increases by 1. (Moving to the next character)

If T[i] = P[j], j increases by 1, apply for loop from j to m – 1 and T[i + j] vs P[j].

+Step 2: If j = m so we print out the starting index of the first substring of T matching P.

+Step 3: If j != m continue step 1 until i reaches n.

_Example:



_Total number of comparisons: $\boldsymbol{m(n-m+1)}$.

_Searching phase : $\boldsymbol{O(nm)}$ time complexity.

RESEARCH

2. Rabin – Karp

_ The Rabin – Karp algorithm instead achieves its speedup by using a hash function to quickly perform an approximate check for each position, and then only performing an exact comparison at the positions that pass this approximate check**.**

_ A hash function is a function which converts every string into a numeric value, called its hash value. It exploits the fact that if 2 strings are equal, their hash values are also equal.

_Formular:

+Rehashing: Hash (text[s + 1 … s + m]) = (d (hash (text[s ... s + m - 1]) – text[s] * h)+ text[s + m]) mod q.

+Hash (text[s … s + m - 1]): Hash value at shift s.

+Hash (text[s +1 ... s + m]): Hash value at next shift (or shift s + 1)

+Hash at the next shift must be efficiently computable $O(1)$ from the current hash value and next character in text.

_Notations:

+d: number of characters in the alphabet .

+q: a prime number (11, 13, 101, …). If we take a large prime number, the collision will be less.

+h: d^{m-1} mod q.

_Explanations:

+Step 1: Firstly, we calculate the hash value of pattern and first window of text. Start from first element to (m - 1)th element in 2 strings by using **loop** with index "i". Beside the value of p & t is assigned by 0.

+Step 2: After that, we will compare the hash value of pattern to that of first window. The first window does not same as pattern ➔ Slide the window ahead by one character.

+Step 3: If hash value of both window and pattern matched, we compare characters within them. If matched, return the starting index of the first substring of T matching P.

+Step 4: Continue step 1.

_Example: T[] = "CTDLGT" . n = 6
P[] = "LG". m = 2

q = 11 (a prime number.)

d = 256 (the number of characters in the alphabet.)

h: d^{m-1} % q = 256 % 11 = 3.

p: the hash value for pattern. **p = (d * p + P[i]) % q**

t: the hash value for text.

Hast value for pattern → p = 2. (Throughout the example).

T = "CTDLGT".
P = "LG".
t = (d * t + T[i]) % q = 10 → t != p so slide the window ahead by one character.

T = "CTDLGT".
P = "LG".
t = (d * (t – (text[i] * h) + text[i + M]) % q = -10. Because of the negative value of t so we add q to it. → t = -10 + 11 = 1 != p so slide the window ahead by one character.

T = "CTDLGT".
P = "LG".
t = (256*(1 – 84*3) + 76) % 11 = -6 → t = -6 + 11 = 5 != p so slide the window ahead by one character.

T = "CTDLGT".
P = "LG".
t = (256*(5 – 68*3) + 71) % 11 = -9 → t = -9 + 11 = 2 = 9. Now t equals to p. We compare this window to pattern, character by character → it comes out to be same. Therefore, the pattern exists at the index 3.

T = "CTDLGT".
P = "LG".
t = (256*(2 – 76*3) + 84) % 11 = 0.
Since the hash value of this window and pattern not equal so move on the next window but the next window is not possible. Therefore, the loop ends here.

_Preprocessing phase : $\boldsymbol{O}(\boldsymbol{m})$ time complexity.
_Searching phase : $\boldsymbol{O}(\boldsymbol{nm})$ time complexity (independent from the alphabet size).

RESEARCH

3 Knuth – Morris – Pratt

_The KMP string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.

_A preprocessing array (**lps**[]) is computed that indicates how many characters to be skipped and has a same size as pattern.

_Name **lps** indicates longest proper prefix which is also suffix.

_lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

_Example:

j	0	1	2	3	4	5
pat[]	a	b	a	b	a	c
lps[]	0	0	1	2	3	0

_Explanations: Assume text as T with n size and pattern as P with m size. ($m \leq n$)

+Step 1: Create a lps[] array from a given pattern.

+Step 2: Iterating through the main string (text), one of three things happens.

if T[i] = P[j], then i increases by 1, as does j.

if T[i] != P[j] and j > 0, then i does not change and j changes to j = lps[j – 1].

if T[i] != P[j] and j = 0, then i increases by 1 and j remains the same.

+Step 3: If j = m so we print out the starting index of the first substring of T matching P.

+Step 4: If j != m, continues step 2.

_Example: T[] = "aabcaaabc". n = 9

P[] = "aabc". m = 4 → lps[] = {0, 1, 0, 0};

i = 0, j = 0.

T = "aabcaaabc"

P = "aabc"

T[i] = P[j] So i increases by 1, as does j. -> i = 1, j = 1.

T = "aabcaaabc"

P = "aabc"

T[i] = P[j] So i increases by 1, as does j. -> i = 2, j = 2.

T = "aabcaaabc"

P = "aabc"

T[i] = P[j] So i increases by 1, as does j. -> i = 3, j = 3.

T = "aabcaaabc"

P = "aabc"

T[i] = P[j] So i increases by 1, as does j. -> i = 4, j = 4.

j = m = 4 so we print out the starting index of the first substring of T which is 0 and then reset j to 0.

i = 4, j = 0
T = “aabcaaabc”
P = “aabc”
T[i] = P[j] So i increases by 1, as does j. -> i = 5, j = 1.

T = “aabcaaabc”
P = “aabc”
T[i] = P[j] So i increases by 1, as does j. -> i = 6, j = 2.

T = “aabcaaabc”
P = “aabc”
T[i] != P[j] So i does not change and j = lps[j – 1] = lps[1] = 1.

T = “aabcaaabc”
P = “aabc”
T[i] = P[j] So i increases by 1, as does j. -> i = 7, j = 2.

T = “aabcaaabc”
P = “aabc”
T[i] = P[j] So i increases by 1, as does j. -> i = 8, j = 3.

T = “aabcaaabc”
P = “aabc”
T[i] = P[j] So i increases by 1, as does j. -> i = 9, j = 4.

j = m = 4 so we print out the starting index of the first substring of T which is 5 and then reset j to 0. However, we stop the iteration because i equals to n.

_Preprocessing phase : $\boldsymbol{O(m)}$ space and time complexity.
_Searching phase : $\boldsymbol{O(n+m)}$ time complexity (independent from the alphabet size).

RESEARCH

Table of Algorithms Time Complexity

Algorithm	Preprocessing	Time Complexity (Worst Case)	Search type	Approach
Brute – Force	N/A	$O(nm)$	Prefix	Linear Searching
Rabin – Karp	$O(m)$	$O(nm)$	Prefix	Hashing based
Knuth – Morris – Pratt	$O(m)$	$O(n+m)$	Prefix	Heuristics based

PROGRAMMING

Project of Data Structures and Algorithms

Crossword game is built in C++ Programming Language.
Description: Given a 2D table of characters with size W x H see **Figure 1**. Build the ideal program to get meaningful words of table based on a file. A string of word is considered **FOUND** if its represented word appears either in a row from left to right or in a column from up to down. Point out the position and direction of a word when it is **FOUND**.

Algorithms:

- Brute – Force (Naïve String – matching).
- Basic algorithms and other techniques.

Additional Techniques:

- Struct data type technique.
- 2 – Dimensional Dynamic Array technique.
- Pointers technique.
- Vector technique.
- File Handling technique.

PROGRAMMING

	1	2	3	4	5	6	7	8	9	10
1	U	C	M	A	R	V	E	L	O	L
2	S	F	D	Q	U	E	U	E	G	I
3	T	R	A	V	E	R	S	A	L	N
4	E	F	G	S	T	A	X	F	N	K
5	K	D	P	U	Z	U	V	U	C	E
6	I	S	I	B	L	I	N	G	S	D
7	E	N	N	T	S	L	Y	C	L	L
8	D	E	G	R	E	E	A	B	I	I
9	G	X	Z	E	M	O	Q	R	F	S
10	E	R	T	E	B	G	R	O	O	T

- 2019/CS1.png

Figure 1: Table of characters

Input	Output
9 10 U C ... L ... Figure 1 E R ... T MARVEL LIST XXX #	2 MARVEL (1,3) LR LIST (7,10) TD XXX (0, 0) NF

_Declare two global variables for Width and Height.

```
static int W, H;
```

_Create struct data type for Word.

```
struct Word
{
	int _X, _Y; //Coordinates
	string _Name, _Dir; //Name and Direction
};
```

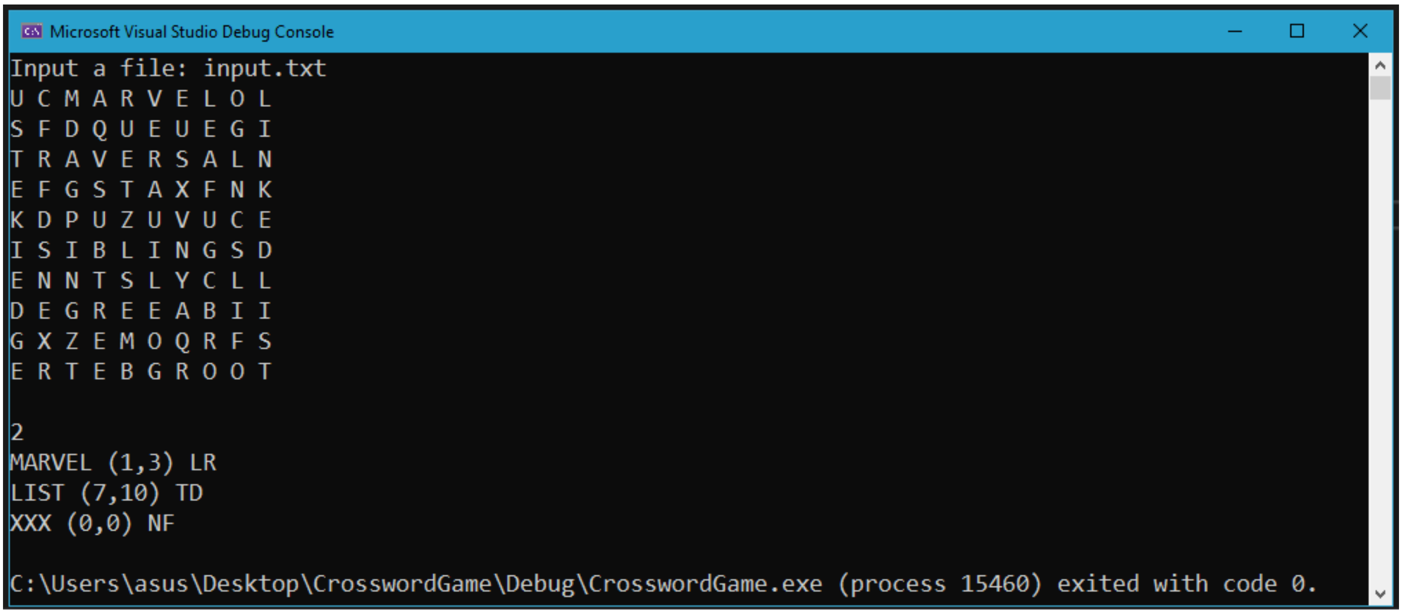
```
char** createCrosswordTable (const string& file, vector<string>& wordsList) {...}
```

➔ This function reads the information in a file and returns 2 – dimensional dynamic char array.

```
bool Search2D(char** table, const int& row, const int& col, const string& word, int& direction) {...}
void patternSearch(char** table, const vector<string>& words) {...}
```

➔ These two are the most crucial functions in this program. `patternSearch` provides us with all information that we required such as the meaningful word, co-ordinates and direction. Whereas, `Search2D` is a function within `patternSearch` which is used to check whether the word exists or not.

_Console:



!!!NOTE: RETURN (0, 0) FOR NON-EXISTING WORD

REFERENCES

1. https://edutechlearners.com/download/Introduction_to_algorithms-3rd Edition.pdf. (Chapter VII, Unit 32, page 985)
2. GeeksforGeeks Tutorial: https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/
3. https://examples.yourdictionary.com/examples-of-plagiarism.html