Knuth Morris Pratt Algorithm

String Searching

Background Concepts Needed

- Core
  - Basic Control Structures
    - while, for, if
  - Arrays
  - String Manipulation
    - substring
- Auxiliary
  - ArrayList
  - Methods

Basis - Basic String Search

- Knuth Morris Pratt is an optimized string search
- The most basic form of string search follows this algorithm
    - Let String A be the string being searched through
    - Let String B be the string being searched for, must be shorter than String A
    - For every character in String A, the ith character
        - Check if String B exists at that index
        - Do this by checking for every character in String B, the nth character
            - Look for a match for the nth index of String B and the i + nth character of String A
            - If the full String B is found, record the answer and move to the next ith character
            - If there is a mismatch, move to the next ith character
- Complexity of O(n^2)
    - Might check every character of String A the same number of times as the number of characters in String B, not efficient

Basic String Search, Inefficiency Example

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next
index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Mismatch, Reset

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next
index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Mismatch, Reset

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next
index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
Complete match, add answer, reset

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	"A"	"A"	"A"	"A"	"A"	"B"
String B Index	0	1	2	3		
String B Value	"A"	"A"	"A"	"B"		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
Mismatch, Reset

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
Match, check if next index matches

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
Mismatch, Reset

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
Mismatch, Reset

Basic String Search, Inefficiency Example

Legend (String A)

RED - Not Processed

YELLOW - Being Processed

GREEN - Processed

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Answer Indices:
{2}

Legend (String B)

WHITE - Current Pointer

Result:
All characters
searched, finish

Search Optimizations in Knuth Morris Pratt

- Knuth Morris Pratt is an optimized string search
- Will only ever iterate over every index once
- Concept: Never go backwards, if there is a mismatch, continue iterating forward
  - Problem: you can accidentally skip a solution if you do this

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		

Search Optimizations in Knuth Morris Pratt

- Concept: It is only possible to miss a solution while only iterating forward if String B contains duplicate characters
- Knuth Morris Pratt creates a specialized table (array) based on String B to detect these duplicates
- The table allows the search algorithm to only iterate forward but still not miss any solutions by changing the targeted character in String B in an intelligent way

String B Value	“A”	“A”	“A”	“B”
String B Table	-1	-1	-1	2

Table Generation in Knuth Morris Pratt, uses String B

- In the table, 0 or -1 indicate to go back to the beginning for a mismatch
- Will only use other values if there are duplicates of the first character
- If there are no duplicates of the first character, the table is full of 0 except for the first value
  - The first value is always initialized to -1 to represent the first character
  - This essentially empty table leads to the search algorithm operating in the way explored earlier, only ever moving forward
- Table generation uses two pointers, position and candidate
  - Position continually moves forward
  - Candidate stays at 0 until a duplicate is detected
  - Both represent a character currently being analyzed

Table Generation in Knuth Morris Pratt Continued

- Position is initialized to 1
- Candidate is initialized to 0
- Table[0] = -1
- Operates by comparing the characters at index position and index candidate
    - If there is not a match, set the position table value to candidate (will be the last duplicate of the previous character, 0 if there is no duplicate of the previous character), increase position by 1, and set candidate to 0
    - If there is a match, set the position table value to the table value at candidate, increase position by 1, and increase candidate by 1
    - If the end of String B is reached, the table is finished
- Finished table will be used by the search algorithm to locate potential instances of String B inside other instances of String B

Search Algorithm in Knuth Morris Pratt

- Let String A be the string being searched through
- Let String B be the string being searched for, must be shorter than String A
- Let InputPointer refer to a character in String A
- Let TargetPointer refer to a character in String B
- Let Table be a table generated using the KMP table algorithm off of String B
- Total Complexity of O(n)

Search Algorithm in Knuth Morris Pratt Continued

- If the character at inputPointer and the character at targetPointer equal
    - Increase both inputPointer and targetPointer by 1
    - If targetPointer is the length of String B and therefore an instance of String B has been found
        - Add the inputPointer - targetPointer to the list of correct indices
        - Set targetPointer to the table value at index targetPointer
- If not
    - Set targetPointer to the table value at index targetPointer
    - If the targetPointer is now negative and therefore there is no possibility for there to be an instance of String B here
        - Increase inputPointer by 1
        - Set targetPointer to 0
- Repeat until the end of String A is reached

Previous Example Optimized using KMP

Legend (String B)

RED - Pos Pointer GREEN - Cnd Pointer

String B Index	0	1	2	3
String B Value	“A”	“A”	“A”	“B”
Table Index	0	1	2	3
Table Value	-1	-1		

Result:
Match

Cnd = 1

Previous Example Optimized using KMP

Legend (String B)

RED - Pos Pointer

GREEN - Cnd Pointer

String B Index	0	1	2	3
String B Value	“A”	“A”	“A”	“B”
Table Index	0	1	2	3
Table Value	-1	-1	-1	

Result:
Match

Cnd = 2

Previous Example Optimized using KMP

Legend (String B)

RED - Pos Pointer GREEN - Cnd Pointer

String B Index	0	1	2	3
String B Value	“A”	“A”	“A”	“B”
Table Index	0	1	2	3
Table Value	-1	-1	-1	2

Result:
Mismatch

Cnd = 0

Previous Example Optimized using KMP

Legend

RED - String A Pointer GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Result:
Match

Previous Example Optimized using KMP

Legend

RED - String A Pointer GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Result:
Match

Previous Example Optimized using KMP

Legend

RED - String A Pointer GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Result:
Match

Previous Example Optimized using KMP

Legend

RED - String A Pointer GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Result:
MisMatch

Previous Example Optimized using KMP

Legend

RED - String A Pointer GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Result:
Match

Previous Example Optimized using KMP

Legend

RED - String A Pointer   GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Result:
Mismatch

Previous Example Optimized using KMP

Legend

RED - String A Pointer GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Result:
Match

Previous Example Optimized using KMP

Legend

RED - String A Pointer GREEN - String B Pointer

String A Index	0	1	2	3	4	5
String A Value	“A”	“A”	“A”	“A”	“A”	“B”
String B Index	0	1	2	3		
String B Value	“A”	“A”	“A”	“B”		
Table Index	0	1	2	3		
Table Value	-1	-1	-1	2		

Answers:
{2}

Result:
Match, solution found, end of String A reached

Comparison

- Steps needed in basic search: 19
- Steps needed in KMP: 11
- KMP 8 steps faster
    - Far more distinct advantage in longer string comparisons
- Conclusion: KMP is better

Application

- Lot of data querying systems
    - Spell check
    - Search engine
    - Plagiarism detection
- Imagine you have a large database to search from such as the pokedex
    - You want to search for pokemon with some string in their name
    - Surrounding data structure for such a system may be rather complex
    - On the small scale, the Knuth-Morris-Pratt algorithm can be used for individual comparisons while looking through the pokedex for said pokemon
    - Can save much time across many comparisons compared to naive search

Sample Problem

- Your principle has tasked you with creating a program to find just the first instance of a given String B in a given String A. You are required to use an algorithm of at least time complexity O(n) in order to be successful, as the program will be used as a module in a larger system and must not hold up operations. Create such a program using the following specifications
- Input
    - String A, maximum 10^6 characters
    - String B, smaller or equal to String A
- Output
    - Just the index of the first occurrence of String B
    - -1 if there are no occurrences of String B

Sample Problem

- Online Judge
- Solution

Quiz Questions

- What is the primary difference between KMP and basic string searching?
  a. KMP is a greedy algorithm, making it less accurate and leading to potential miscalculations
  b. KMP uses less memory space than basic string searching
  c. Basic string searching iterates over each character of String A more times than KMP
  d. There is no significant difference between KMP and basic string searching
- What are the time complexities of KMP and basic string searching
  a. KMP: O(n), Basic: O(n)
  b. KMP: O(n^2), Basic: O(n)
  c. KMP: O(n), Basic: O(2^n)
  d. KMP: O(n), Basic: O(n^2)
- Solutions separately