

Q01:Please introduce the process of Knuth-Morris-Pratt algorithm in substring search task from the point of view of code.

- prompt:You are an expert in the field of computer science, and you are teaching advanced algorithms. In the next conversation, I'm going to ask you a few questions about the course content. Please answer them in detail.
- model_name:ChatGPT
- model_version:ChatGPT 4o with canvas
- response:

The Knuth-Morris-Pratt (KMP) algorithm is an efficient method for substring searching, which improves upon the naive approach by avoiding unnecessary re-examinations of characters in the string. It preprocesses the pattern to create a partial match table (also known as the "prefix function" or "failure function") that tells you how much you can safely skip when a mismatch occurs. Here's a breakdown of the process in code:

i. Preprocessing Phase (Building the Prefix Function)

In this phase, we build an array called the **prefix function** (lps - longest proper prefix which is also a suffix). This array helps us to skip unnecessary comparisons when a mismatch occurs.

- For each position i in the pattern P , $\text{lps}[i]$ represents the length of the longest prefix of the substring $P[0..i]$ that is also a suffix of this substring.
- We use this array to determine how much we can shift the pattern relative to the text after a mismatch.

Code for Building the Prefix Function:

```
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m # LPS array to store the longest prefix-suffix lengths
    length = 0 # Length of the previous longest prefix suffix
    i = 1 # Start from the second character of the pattern

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1] # Use the previous longest prefix suffix
            else:
                lps[i] = 0
                i += 1
    return lps
```

ii. Searching Phase (Using the Preprocessed Prefix Function)

Once the **lps** array is computed, we can use it to perform the actual pattern matching on the text. When we compare the pattern against the text, we only need to use the information in the **lps** array to shift the pattern efficiently.

- If a mismatch occurs at index i of the pattern and index j of the text, instead of shifting the pattern by 1, we can use the value $\text{lps}[i-1]$ to shift the pattern ahead by $i - \text{lps}[i-1]$ positions.
- If a match is found, we continue by comparing the next character of the pattern with the next character of the text.

Code for Searching the Pattern in the Text:

```
def knuth_morris_pratt(text, pattern):
    n = len(text)
    m = len(pattern)

    # Step 1: Preprocess the pattern to create the lps array
    lps = compute_lps(pattern)

    i = 0 # Pointer for text
    j = 0 # Pointer for pattern

    while i < n:
        if pattern[j] == text[i]: # Match found, move both pointers
            i += 1
            j += 1

        if j == m: # Full pattern matched
            print(f"Pattern found at index {i - j}")
            j = lps[j - 1] # Use lps to shift the pattern

        elif i < n and pattern[j] != text[i]: # Mismatch
            if j != 0:
                j = lps[j - 1] # Use lps to skip ahead in the pattern
            else:
                i += 1 # If j == 0, just move to the next character in the text

    print("Search complete.")
```

Explanation:

i. Preprocessing Phase:

- We compute the lps array, which tells us the longest prefix that is also a suffix for each substring of the pattern.

ii. Searching Phase:

- We scan through the text, comparing the characters one by one.
- When a mismatch occurs, we use the lps array to skip ahead in the pattern instead of shifting by just one position.
- If a complete match is found, we print the position and use the lps array to continue checking for additional occurrences of the pattern in the text.

Time Complexity:

- **Preprocessing time:** $O(m)$, where m is the length of the pattern. This is the time it takes to compute the lps array.
- **Search time:** $O(n)$, where n is the length of the text. The algorithm processes each character of the text exactly once.
- **Total time complexity:** $O(n + m)$.

This is much faster than the naive approach, which has a time complexity of $O(n * m)$ in the worst case.

Example Usage:

```
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
knuth_morris_pratt(text, pattern)
```

This would output:


```
Pattern found at index 10
Search complete.
```

This shows that the pattern "ABABCABAB" is found at index 10 in the text.

- evaluation:

In general, the llm gives an accurate answer to this question. It introduces the principle of KMP algorithm accurately, and gives the accurate and detailed running process of KMP algorithm. At the same time, the analysis and calculation of time

complexity are also correct. Running the code generated by the LLM locally, it will run successfully and output the correct answer. In general, the output of the LLM for this problem is satisfying.



```
46
47 text = "ABABDABACDABABCABAB"
48 pattern = "ABABCABAB"
49 knuth_morris_pratt(text, pattern)
```

问题 输出 调试控制台 终端 端口

```
PS D:\python_file> & 'd:\conda\python.exe' 'c:
Pattern found at index 10
Search complete.
```

Q02:Please introduce the process of Finite State Automaton algorithm in substring searching task from the point of view of code.

- prompt:You are an expert in the field of computer science, and you are teaching advanced algorithms. In the next conversation, I'm going to ask you a few questions about the course content. Please answer them in detail.
- model_name:KIMI
- model_version:KIMI探索版
- response:

Finite State Automaton (FSA) algorithm for substring searching is an efficient method for finding occurrences of a pattern within a text. Here's how the process works from a coding perspective:

 - i. **Understanding the Automaton:**
 - A FSA is defined by a 5-tuple $(Q, \Sigma, q_0, F, \delta)$, where:
 - Q is a finite set of states.
 - Σ is the set of input symbols (the alphabet).
 - q_0 is the initial state.
 - F is the set of final or accepting states.
 - δ is the transition function, which maps the current state and input symbol to the next state.
 - ii. **Building the Automaton:**
 - The number of states in the FSA is $M+1$, where M is the length of the pattern.
 - The main challenge is to determine the next state from the current state for every possible character. This is done by considering the string "pat[0..k-1]x", where "pat" is the pattern and "x" is the next character from the text. The goal is to find the longest prefix of the pattern that is also a suffix of "pat[0..k-1]x", which gives the next state.
 - iii. **Algorithm Implementation:**
 - The computeTF() function constructs the FSA. It initializes a transition table TF, where TF[state][x] represents the next state when in state state and reading character x.
 - The getNextState() function is used to determine the next state. It first checks if the current state and character match the pattern. If not, it checks for a matching suffix and prefix to find the next state.
 - The search() function uses the FSA to scan the text. It maintains a current state and updates it based on the transition table when a character from the text is read. If the current state reaches M (the length of the pattern), a match is found.
 - iv. **Code Example (Python):**

```
NO_OF_CHARS = 256
```

```
def getNextState(pat, M, state, x):
    if state < M and x == ord(pat[state]):
        return state + 1
    i = 0
    for ns in range(state, 0, -1):
        if ord(pat[ns - 1]) == x:
            while i < ns - 1:
                if pat[i] != pat[state - ns + 1 + i]:
                    break
                i += 1
            if i == ns - 1:
                return ns
    return 0

def computeTF(pat, M):
    TF = [[0 for i in range(NO_OF_CHARS)] for _ in range(M + 1)]
    for state in range(M + 1):
        for x in range(NO_OF_CHARS):
            z = getNextState(pat, M, state, x)
            TF[state][x] = z
    return TF

def search(pat, txt):
    M = len(pat)
    N = len(txt)
    TF = computeTF(pat, M)
    state = 0
    for i in range(N):
        state = TF[state][ord(txt[i])]
        if state == M:
            print("Pattern found at index: {}".format(i - M + 1))

def main():
    txt = "abababacaba"
    pat = "ababaca"
    search(pat, txt)

if __name__ == '__main__':
    main()
```

This code snippet demonstrates how to implement the FSA algorithm for substring searching in Python. It includes functions to compute the transition table, get the next state, and search for the pattern within the text using the built FSA.

By following these steps and using the provided code structure, you can implement a FSA for substring searching in any programming language. This approach is efficient because it examines each character of the text exactly once, making the matching time linear with respect to the length of the text. However, the preprocessing time to build the FSA can be significant if the size of the alphabet Σ is large.

- evaluation:

ChatGPT's answer to this question seems problematic. In its generation, the relevant functions of computeTF are not implemented, resulting in the code provided by it to output wrong results, and its time complexity is also incorrectly calculated (incorrectly calculated as $O(m * \Sigma)$). After several prompts, ChatGPT couldn't get the answer right. Therefore, this question uses the answer provided by KIMI AI. Compared to ChatGPT, it has a detailed and correct explanation of FSA concepts, a correct introduction to the FSA algorithm process, and provides code that can run and output correct answers.

```
34
35 def main():
36     txt = "abababacaba"
37     pat = "ababaca"
38     search(pat, txt)

问题 输出 调试控制台 终端 端口

PS D:\python_file> & 'd:\conda\python
Pattern found at index: 2
```

Q03:How to calculate the time complexity of Finite State Automaton algorithm in substring searching task?

- prompt:You are an expert in the field of computer science, and you are teaching advanced algorithms. In the next conversation, I'm going to ask you a few questions about the course content. Please answer them in detail.
- model_name:KIMI
- model_version:KIMI探索版
- response:

To calculate the time complexity of the Finite State Automaton (FSA) algorithm in substring searching tasks, we need to consider two main aspects: the preprocessing time to build the FSA and the matching time to search for the pattern within the text.

i. Preprocessing Time (Building the FSA):

- The time complexity for building the FSA is $O(M^3|\Sigma|)$, where M is the length of the pattern and $|\Sigma|$ is the size of the alphabet (the total number of possible characters in the pattern and text).
- This complexity arises because, for each state and each possible input symbol, we may need to check all possible prefixes that could also be a suffix of the pattern concatenated with the input symbol. This results in a cubic time complexity.

ii. Matching Time:

- Once the FSA is constructed, the time complexity for scanning the text string T to find all occurrences of the pattern P is $O(n)$, where n is the length of the text string.
- This is because each character of the text is examined exactly once, with each examination being a constant-time operation.

In summary, the overall time complexity of the FSA algorithm for substring searching is determined by both the preprocessing time and the matching time. The preprocessing time is the overhead of building the FSA, while the matching time directly affects the algorithm's real-time performance. In practical applications, if the pattern P is fixed or changes infrequently, the preprocessing time can be considered a one-time cost, and the matching time directly impacts the algorithm's efficiency. Therefore, the FSA algorithm is particularly suitable for searching for the same or similar patterns multiple times within a large text, as once the FSA is built, repeated search operations are very fast.

- evaluation:
IIm is basically accurate in the calculation of time complexity, but the explanation of the calculation process is too brief and lacks the necessary mathematical formulas to assist the explanation. In the end, the IIm expands on scenarios suitable for using FSA algorithms, which is very good.

Q04:Why can't string S=abcab build a suffix tree?

- prompt:You are an expert in the field of computer science, and you are teaching advanced algorithms. In the next conversation, I'm going to ask you a few questions about the course content. Please answer them in detail.
- model_name:KIMI
- model_version:KIMI探索版
- response:

The reason why the string S = "abcab" cannot build a suffix tree is due to a fundamental property of suffix trees. According to the definition of a suffix tree, each path from the root to a leaf node must represent a unique suffix of the input string.

However, in the case of the string "abca**b**", the suffix "ab" (which is $S[5..6]$) matches a prefix of another suffix "abca**b**" (which is $S[1..6]$). This creates a conflict because if the suffix "ab" ends at a leaf, then both "ab" and "abca**b**" would start from the root, violating the property that no two paths from the root to leaf nodes should have overlapping labels unless they are exactly the same path.

To resolve this issue, a special termination character that does not appear in the original string is appended to the string. This character, often denoted by '\$', ensures that no suffix of the string is a prefix of another suffix, thus allowing the construction of a valid suffix tree. In the case of "abca**b**", by appending '\$' to make it "abca**b**\$", all suffixes end at unique leaf nodes, satisfying the requirement for constructing a suffix tree.

- evaluation:

If you ask the suffix tree of the llm string "abca**b**" directly, llm will output the wrong answer. But if you ask why the string can't build a suffix tree, llm can give you the exact reason and the corresponding solution. However, the llm does not output suffix tree results under this solution, which needs to be improved.

Q05: In addition to Brute force algorithm, Knuth-Morris-Pratt Algorithm, Finite State Automaton algorithm, Suffix Tree algorithm, what are the advanced algorithms for Substring searching task?

- prompt: You are an expert in the field of computer science, and you are teaching advanced algorithms. In the next conversation, I'm going to ask you a few questions about the course content. Please answer them in detail.

- model_name: ChatGPT

- model_version: ChatGPT 4o with canvas

- response:

In addition to the algorithms you've mentioned (Brute Force, Knuth-Morris-Pratt (KMP), Finite State Automaton (FSA), and Suffix Tree), there are several other advanced algorithms for the substring searching task. Below are some of the prominent ones:

- i. **Boyer-Moore Algorithm**

- **Overview:** Boyer-Moore is one of the most efficient substring search algorithms, especially when the alphabet is large. It preprocesses the pattern to build two heuristic tables:
 - **Bad character heuristic:** This tells how far we can skip ahead based on a mismatch.
 - **Good suffix heuristic:** This allows skipping parts of the text that have already been matched.
 - **Time Complexity:** In the best case, it can perform in $O(n/m)$, where **n** is the length of the text and **m** is the length of the pattern, making it very efficient in practice for many real-world cases.
 - **Worst-Case Time Complexity:** $O(n * m)$ in the worst case, but this is rare.
 - **Advantages:** Very efficient for long patterns and texts, and it often performs better than KMP in practice.

- ii. **Rabin-Karp Algorithm**

- **Overview:** This algorithm uses hashing to detect substring matches. It computes a hash value for the pattern and for each substring of the text with the same length as the pattern. If the hash values match, the substrings are compared character by character to confirm the match.
 - **Time Complexity:**
 - Average case: $O(n + m)$, where **n** is the length of the text and **m** is the length of the pattern.
 - Worst case: $O(n * m)$, which occurs when hash collisions happen frequently.
 - **Advantages:** Particularly useful when searching for multiple patterns simultaneously, as it allows searching for all patterns in one pass over the text.

- iii. **Aho-Corasick Algorithm**

- **Overview:** The Aho-Corasick algorithm is used for searching multiple patterns in a text. It builds a **trie** (prefix tree) for the set of patterns and then constructs a **failure function** that allows for efficient searching. The algorithm essentially automates the process of matching multiple patterns in a text in **linear time** relative to the size of the text and the total size of the patterns.
 - **Time Complexity:** $O(n + m + z)$, where **n** is the length of the text, **m** is the total length of all patterns, and **z** is the number of matches found.

- **Advantages:** It is optimal for scenarios where you need to search for a large set of patterns simultaneously, such as in dictionary matching or intrusion detection systems.

iv. Suffix Array (with LCP Array)

- **Overview:** A **Suffix Array** is an array of integers that represents the starting indices of all suffixes of a string in lexicographical order. When combined with an **LCP (Longest Common Prefix) Array**, it allows efficient substring searching, especially when performing multiple searches over the same text.
- **Time Complexity:** Constructing the suffix array takes $O(n \log n)$ time, and searching for a substring takes $O(m \log n)$ time using binary search.
- **Advantages:** Extremely efficient for repeated substring searches, and the suffix array can be built once, after which multiple queries can be answered quickly.

v. Trie-based Search (with Suffix Links)

- **Overview:** A **Trie** is a tree data structure that represents a set of strings, where each node represents a character in a pattern. When combined with **suffix links** (links to the longest suffix that is also a prefix of another pattern), this allows efficient substring searching for multiple patterns.
- **Time Complexity:** Building the trie takes $O(n)$ time, and searching for each pattern takes $O(m)$ time where m is the length of the pattern.
- **Advantages:** Efficient for large-scale pattern matching, particularly when multiple patterns are involved.

vi. Ukkonen's Algorithm (for Suffix Tree Construction)

- **Overview:** Ukkonen's algorithm is an efficient method for constructing a **suffix tree** in **linear time**. A suffix tree is a compressed trie of all suffixes of a string and is widely used in tasks like substring search, pattern matching, and text indexing.
- **Time Complexity:** $O(n)$ for constructing the suffix tree, and substring search can be done in $O(m)$ time, where m is the length of the pattern.
- **Advantages:** Once constructed, the suffix tree allows fast substring search and is also useful for many other text processing tasks (e.g., finding longest common substrings, pattern matching, and compressed storage of strings).

vii. Shift-Or Algorithm

- **Overview:** This is a bitwise algorithm that is used for searching a pattern in a text, often when the pattern is small. It works by representing the pattern as a bitmask and performing bitwise operations over the text.
- **Time Complexity:** $O(n/m)$, where n is the length of the text and m is the length of the pattern.
- **Advantages:** The shift-or algorithm can be very efficient for searching short patterns, particularly when the alphabet is small.

viii. Fibonacci Search

- **Overview:** While not as commonly used for substring searching as other algorithms, Fibonacci search is a variation of binary search that uses Fibonacci numbers to determine the search points. It is mainly used in situations where the data set is sorted and continuous.
- **Time Complexity:** $O(\log n)$ for searching in a sorted list.
- **Advantages:** Useful in contexts where the data is sorted or there is a requirement for searching in a specific range of values.

Summary of Key Advanced Algorithms:

- **Boyer-Moore:** Efficient for large alphabets and long patterns, with good heuristics for skipping unnecessary comparisons.
- **Rabin-Karp:** Efficient for multiple pattern searches using hashing.
- **Aho-Corasick:** Optimized for searching multiple patterns simultaneously.
- **Suffix Array:** Excellent for repeated substring queries, often used with LCP array for optimal performance.
- **Suffix Tree (Ukkonen's Algorithm):** Optimal for multiple substring queries and text processing tasks.
- **Shift-Or:** Bitwise approach for small pattern matching, highly efficient when the alphabet is small.

Each of these algorithms has its use cases depending on the size of the input text, the pattern size, the alphabet, and whether you're searching for multiple patterns or a single one.

• evaluation:

Ilm introduces a variety of advanced algorithms for Substring searching task, which have different advantages in different scenarios. The Ilm has a brief introduction to the principles of these algorithms, which seem feasible in principle for the

Substring searching task. However, because I am not familiar with these advanced algorithms, the algorithm principle of llm output and the accuracy of time complexity calculation are not clear.