**String Matching**

Pattern matching is the process of checking a perceived sequence of string for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form sequences of pattern matching include outputting the locations of a pattern within a string sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other string sequence (i.e., search and replace). Pattern matching concept is used in many applications Following figure shows the different applications.

**Algorithms used for pattern searching**

1. Naive Pattern Searching
2. Rabin Karp String Search Algorithm
3. Knuth–Morris–Pratt algorithm
4. Boyer–Moore string search algorithm

##### **Naive Pattern Searching**

A naive string matching algorithm compares the given pattern against all positions in the given text. Each comparison takes time proportional to the length of the pattern, and the number of positions is proportional to the length of the text. Therefore, the worst-case time for such a method is proportional to the product of the two lengths. In many practical cases, this time can be significantly reduced by cutting short the comparison at each position as soon as a mismatch is found, but this idea cannot guarantee any speedup.

**Technique** : Each character of the pattern is compared to a substring of the text which is the length of the pattern, until there is a mismatch or a match.

##### **Code / Algorithm**

##### 

**Time Complexity**

The **best case** occurs when the first character of the pattern is not present in text at all.

txt[] = "AABCCAADDEE";

pat[] = “FAA”;

The number of comparisons in best case is O(n).

The **worst case** of Naive Pattern Searching occurs in when all characters of the text and pattern are same or hen only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAAA";

pat[] = “AAAAA";

txt[] = "AAAAAAAAAAAAAAAAAB";

pat[] = "AAAAB";

The number of comparisons in the worst case is O(m\*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts).

**Space Complexity**

This is an in-place algorithm. So O(1) auxiliary space is needed.

##### **Rabin–Karp string search algorithm**

Rabin-Karp algorithm slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

**Technique** : Hashing! So Rabin Karp algorithm needs to calculate hash values for following strings.

1. Pattern itself.
2. All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say hash(txt[s+1 .. s+m]) must be efficiently computable from hash(txt[s .. s+m-1]) and txt[s+m] i.e., hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])) and rehash must be O(1) operation.

To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value.

Rehashing is done using the following formula.

hash( txt[s+1 .. s+m] ) = ( d ( hash( txt[s .. s+m-1]) – txt[s]\*h ) + txt[s + m] ) mod q

hash( txt[s .. s+m-1] ) : Hash value at shift s.

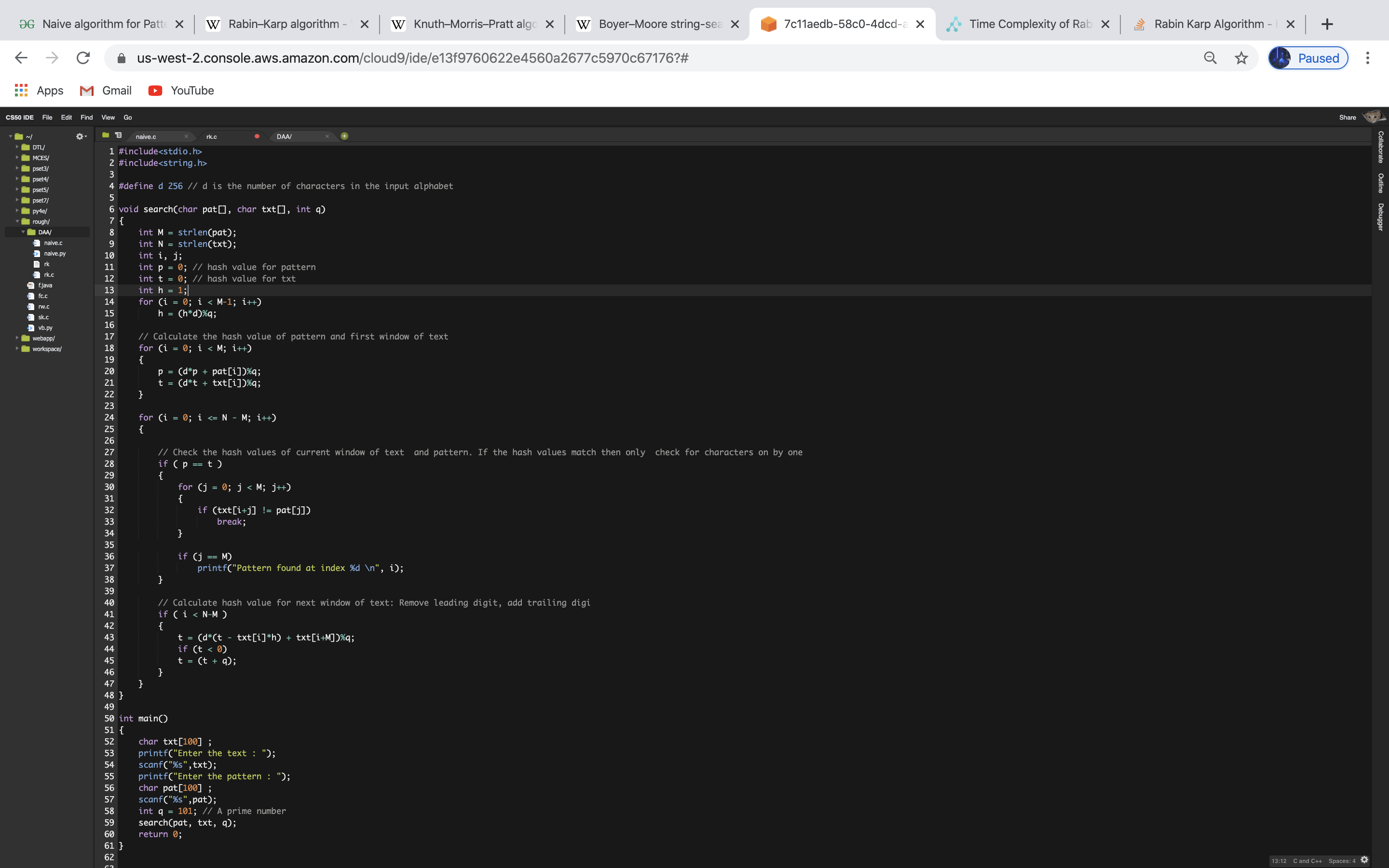
hash( txt[s+1 .. s+m] ) : Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

h: d^(m-1)

##### **Code / Algorithm**



**Time Complexity**

The **average** case running time of the Rabin-Karp algorithm is O(n+m),

but its **worst-case** time is O((n-m+1) m).

**Best** case is O(m).

One of the Worst case of Rabin-Karp algorithm O(mn) occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[].

For example: pat[] = “AAA” and txt[] = “AAAAAAA”.

**Space Complexity**

Space Complexity of Rabin-Karp algorithm is O(m)

**Knuth–Morris–Pratt algorithm**

KMP string searching algorithm searches for occurrences of a "word" W within a main "text string" T by employing the observation that when a mismatch occurs, the word itself contains sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. This algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP’s algorithm is: whenever we detect a mismatch we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

**Technique** : We start comparison of pat[j] with j = 0 with characters of current window of text.

* We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
* When we see a **mismatch**
  + We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
  + We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.
  + From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match.

**Code / Algorithm :**

**algorithm** *kmp\_search*:

**input**:

an array of characters, S (the text to be searched)

an array of characters, W (the word sought)

**output**:

an array of integers, P (positions in S at which W is found)

an integer, nP (number of positions)

**define variables**:

an integer, j ← 0 (the position of the current character in S)

an integer, k ← 0 (the position of the current character in W)

an array of integers, T (the table, computed elsewhere)

**let** nP ← 0

**while** j < length(S) **do**

**if** W[k] = S[j] **then**

**let** j ← j + 1

**let** k ← k + 1

**if** k = length(W) **then**

(occurrence found, if only first occurrence is needed, m ← j - k may be returned here)

**let** P[nP] ← j - k, nP ← nP + 1

**let** k ← T[k] (T[length(W)] can't be -1)

**else**

**let** k ← T[k]

**if** k < 0 **then**

**let** j ← j + 1

**let** k ← k + 1

**Time Complexity**

Preprocessing Time : O(m)

Matching time/Searching Phase/Running Time : O(m+n)

**Space Complexity**

Its Space complexity is also O(m).

##### **Boyer–Moore string search algorithm**

It is a particularly efficient string searching algorithm, The algorithm preprocesses the target string (key) that is being searched for, but not the string being searched in. Generally the algorithm gets faster as the key being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text it is searching. The Boyer Moore algorithm does preprocessing and processes the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by the max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step.

Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.

**Technique** : the algorithm begins at alignment , so the start of *P* is aligned with the start of *T*. Characters in *P* and *T* are then compared starting at index *n* in *P* and *k* in *T*, moving backward. The strings are matched from the end of *P* to the start of *P*. The comparisons continue until either the beginning of *P* is reached (which means there is a match) or a mismatch occurs upon which the alignment is shifted forward (to the right) according to the maximum value permitted by a number of rules. The comparisons are performed again at the new alignment, and the process repeats until the alignment is shifted past the end of *T*, which means no further matches will be found.

The shift rules are implemented as constant-time table lookups, using tables generated during the preprocessing of *P.* A shift is calculated by applying two rules: the bad character rule and the good suffix rule. The actual shifting offset is the maximum of the shifts calculated by these rules.

**The bad character rule :** The bad-character rule considers the character in *T* at which the comparison process failed (assuming such a failure occurred). The next occurrence of that character to the left in *P* is found, and a shift which brings that occurrence in line with the mismatched occurrence in *T* is proposed. If the mismatched character does not occur to the left in *P*, a shift is proposed that moves the entirety of *P* past the point of mismatch.

**The good suffix rule :** The good suffix rule is markedly more complex in both concept and implementation than the bad character rule. It is the reason comparisons begin at the end of the pattern rather than the start, and is formally stated thus.

##### **Code / Algorithm :**

##### 

**Time Complexity**

Preprocessing Time : O(m + |Σ|)

Matching time/Searching Phase/Running Time : Ω(n/m), O(n)

**best** : Ω(n/m)

**worst** : O(mn)

**Space Complexity**

Its Space complexity is also O(m + |Σ|).

**Comparison Tables for Algorithms**

*Different Techniques used by Different Algorithms*

| **Algorithm** | **Techniques** |
| --- | --- |
| Naive string search algorithm | Each character of the pattern is compared to a substring of the text which is the length of the pattern, until there is a mismatch or a match. |
| Rabin–Karp string search algorithm | Hashing |
| Knuth–Morris–Pratt algorithm | Two indices l and r into text string t |
| Boyer–Moore string search algorithm | Use both good suffix shift and bad character shift |

##### *Comparisons of Worst case Complexity of different algorithms*

| **Algorithm** | **Preprocessing** | **time** | **Matching time/Searching Phase/Running Time** |
| --- | --- | --- | --- |
|  | **Time complexity** | **Space complexity** |  |
| Naive string search algorithm | O(n) | O(1) | O((n-m+1) m) |
| Rabin–Karp string search algorithm | O(m) | O(m) | Average O(n+m), worst O((n-m+1) m) |
| Knuth–Morris–Pratt algorithm | O(m) | O(m) | O(m+n) |
| Boyer–Moore string search algorithm | O(m + |Σ|) | O(m + |Σ|) | Ω(n/m), O(n) |