GUJARAT TECHNOLOGICAL UNIVERSITY

BE-5 SEMESTER – OLD PAPER – S22 TO W25 – QUESTION BANK ANSWER

Subject Name & Code:

ANALYSIS AND DESIGN OF ALGORITHMS (3150703)

Unit 8: String Matching Algorithms

Q1 – Explain naive string matching algorithm with example and give its time complexity. (4 marks)

Ans:

Naive String Matching Algorithm – Also called brute-force pattern matching. It slides the pattern over the text and compares characters one by one.

Algorithm:

```
NaiveMatch(T, P):
   n = length(T), m = length(P)
   for i = 0 to n-m:
      j = 0
      while j < m and T[i+j] == P[j]:
         j = j+1
      if j == m:
         print "Pattern found at index i"
```

Example:

Text T = "ABABABC", Pattern P = "ABA"

- i=0: compare T[0..2]="ABA" with P → match at 0
- i=1: "BAB" vs "ABA" → mismatch at first char
- i=2: "ABA" vs "ABA" → match at 2
- i=3: "BAB" vs "ABA" → mismatch
- i=4: "ABC" vs "ABA" → mismatch

Time complexity:

- Worst-case: O((n-m+1) × m) ≈ O(n×m)
  Example: T = "AAAAAA", P = "AAA" → many comparisons
- Best-case: O(n) (first character mismatch each shift)

Real-world application: Small-scale text search (e.g., Ctrl+F in a short document).

Q2 – Explain spurious hits in Rabin-Karp algorithm with example. (3 marks)

Ans:

Spurious hit – In Rabin-Karp algorithm, when the hash value of a text window matches the pattern's hash value, but the actual strings are **not equal**. This is a false positive.

Cause: Hash collisions – different strings produce the same hash value (modulo q).

Example:

Text T = "235902", Pattern P = "902".

Assume hash function: numeric value modulo 11.

Pattern hash = 902 mod 11 = 0.

Windows:

- "235" → 235 mod 11 = 4 ≠ 0 → no spurious
- "359" → 359 mod 11 = 7
- "590" → 590 mod 11 = 7
- "902" → 902 mod 11 = 0 → hash match, actual match (not spurious)

Now change pattern to "803". Pattern hash = 803 mod 11 = 0.
Window "902" also gives hash 0 → hash match but "902" ≠ "803" → **spurious hit**.
Then algorithm checks actual string, finds mismatch, continues.
Impact: Increases running time but does not affect correctness (final check eliminates spurious hits).
Real-world application: Used in plagiarism detection tools where hashing speeds up search but occasional false positives are verified.

Q3 – Explain KMP string matching algorithm and prefix function. (7 marks)
Ans:
KMP (Knuth-Morris-Pratt) Algorithm – An efficient string matching algorithm that avoids re-examining characters by using information from previous comparisons.
Preprocesses the pattern to compute a **prefix function** (also called failure function or LPS – longest proper prefix which is also suffix).
Prefix function π(q): For each prefix of the pattern of length q, π[q] is the length of the **longest proper prefix** of that prefix which is also a **suffix**. It indicates how many characters to skip after a mismatch.
Example – Pattern P = "ABABAC"
Compute π for each prefix:

Length q	Prefix	Longest proper prefix which is also suffix	π[q]
1	A	none (empty)	0
2	AB	none	0
3	ABA	"A" (prefix) = "A" (suffix)	1
4	ABAB	"AB" (prefix) = "AB" (suffix)	2
5	ABABA	"ABA" (prefix) = "ABA" (suffix)	3
6	ABABAC	none (suffix cannot be prefix)	0

Thus π = [0, 0, 1, 2, 3, 0] (1-based indexing often used).
KMP algorithm:

```
KMP(T, P):
   n = len(T), m = len(P)
   π = computePrefix(P)
   q = 0  // number of chars matched
   for i = 0 to n-1:
      while q > 0 and P[q] != T[i]:
         q = π[q-1]   // fallback using prefix function
      if P[q] == T[i]:
```

```
        q = q+1
    if q == m:
        print "Pattern found at", i-m+1
        q = π[q-1]   // continue searching
```

Example trace: T = "ABABABAC", P = "ABABAC"
At i=5 (T[5]=B), q=5 (matched "ABABA"), but P[5]=C ≠ B → fallback q=π[4]=3. Then compare P[3]=A? Actually next step continues.
Time complexity: O(n + m) – linear.
Real-world application: Searching in DNA sequences, text editors (find/replace), file indexing.

Q4 – Write modified naive string matching algorithm if all characters in pattern are different. (4 marks)
Ans:
Observation: If all characters in the pattern are distinct, when a mismatch occurs at position j of the pattern (0-based), the text character T[i+j] cannot match any earlier pattern character (since all are distinct). Therefore we can shift the pattern by **j+1** positions instead of 1.
Modified algorithm:

```
ModifiedNaiveMatch(T, P):
  n = len(T), m = len(P)
  i = 0
  while i <= n-m:
    j = 0
    while j < m and T[i+j] == P[j]:
      j = j+1
    if j == m:
      print "Pattern found at index i"
      i = i + 1   // or shift by 1 to find overlapping matches
    else:
      // Mismatch at position j
      // Since all pattern chars are distinct, T[i+j] cannot match any P[0..j-1]
      // Shift so that T[i+j] aligns with P[0] (or just skip ahead)
      i = i + j + 1
```

Example: T = "ABCDEABF", P = "ABF" (all distinct: A,B,F)

- i=0: compare T[0]=A, T[1]=B, T[2]=C vs A,B,F → mismatch at j=2 (C vs F)
- Shift by j+1 = 3 → i = 0+3 = 3 (skip to index 3)
- i=3: compare T[3]=D, T[4]=E, T[5]=A → mismatch at j=0 → shift by 1? Actually j=0 so shift 1. Continue.

Time complexity: Worst-case O(n) because each text character is compared at most once.
Note: This is a simplified version of the Boyer-Moore algorithm (bad character heuristic).

Q5 – What is finite automata? Explain use in string matching with example. (4 marks)
Ans:
Finite automaton (FA) – A mathematical model of computation consisting of:

- Finite set of states (including a start state and accepting states)
- Alphabet Σ
- Transition function δ(state, character) → next state

Use in string matching: Build a **pattern-matching automaton** for a given pattern P. The automaton processes the text character by character. After each character, the current state

represents the length of the longest prefix of P that is a suffix of the text scanned so far. When the state reaches m (pattern length), a match is found.

Construction:

For pattern P of length m, create m+1 states (0 to m). State 0 is start, state m is accepting. For each state q ($0 \le q \le m$) and each character c in Σ, compute δ(q, c) = length of longest prefix of P which is a suffix of (P[0..q-1] + c). This can be precomputed using prefix function.

Example – Pattern P = "ABAB"

Alphabet Σ = {A, B}.

δ(0, A) = 1 (prefix "A"), δ(0, B) = 0

δ(1, A) = 1? Actually P[0]='A', prefix "A" + 'A' = "AA" – longest suffix that is prefix of P: "A" (length1) → δ=1. δ(1, B) = 2 ("AB")

δ(2, A) = 3 ("ABA"), δ(2, B) = 0? Check "ABB" – longest prefix suffix? "AB" not suffix, "B" not prefix, so 0.

δ(3, A) = 4 (full match), δ(3, B) = 2? "ABAB"+"B" = "ABABB" – suffix "AB" is prefix length2.

Processing text T = "ABABAB"

Start state 0:

T[0]=A → state 1

T[1]=B → state 2

T[2]=A → state 3

T[3]=B → state 4 → **match found at index 3-4+1=0?** Actually state 4 at index 3 means pattern ends at position 3 (0-based).

Continue: T[4]=A → from state 4, δ(4,A) = ? For full pattern, typically δ(m, c) = δ(π[m], c). For "ABAB", π[4]=2? Actually prefix of length4 "ABAB" – LPS length? "AB" is prefix and suffix → π=2. Then δ(4,A)=δ(2,A)=3. So next state 3. T[5]=B → state 4 → match at index 5-3=2? Yes second match.

Time complexity: O(n) after automaton is built (construction O(m|Σ|)).

Real-world application: Regular expression engines, lexical analysis in compilers (lex/flex).

Q6 – What is string matching problem? Define valid shift and invalid shift. (3 marks)

Appeared in: S22 (Q5a, 03 marks), W24 (Q5a, 03 marks)

Ans:

String matching problem – Given a text string T of length n and a pattern string P of length m (with $m \le n$), find all occurrences of P in T. An occurrence is an index s (shift) such that T[s … s+m-1] = P[0 … m-1].

Valid shift – A shift s where the pattern P matches exactly with the substring of T starting at s. That is, T[s + j] = P[j] for all j = 0 … m-1.

Invalid shift – A shift s where the pattern does **not** match; at least one character position j has T[s+j] ≠ P[j].

Example:

T = "ABABABC", P = "ABA"

- Shift s = 0: T[0..2] = "ABA" → valid shift
- Shift s = 1: T[1..3] = "BAB" ≠ "ABA" → invalid shift

Real-world application: Text editors (find function), search engines, DNA sequence alignment.

Q7– Explain Rabin-Karp algorithm with modulo q=13, find spurious hits for given T and P. (7 marks)

Appeared in: S24 (Q5c, 04 marks), W25 (Q5c, 07 marks) – highest 7

⚠️ **Note:** The CQB does **not** provide the actual text T and pattern P. Below is a **complete explanation** of the algorithm with a **generic example** using q=13. For your specific exam data, replace the values accordingly.

Ans:

Rabin-Karp algorithm – A string matching algorithm that uses hashing to compare pattern with text substrings. Instead of comparing characters directly, it compares hash values. Only when hashes match does it verify the actual strings (to avoid spurious hits).

Key steps:

1. Compute hash of pattern P (length m) and hash of first text window T[0..m-1].
2. Slide the window across the text; for each new window, compute hash in O(1) using rolling hash.
3. If hash matches, compare actual strings to confirm match (or identify spurious hit).
4. If hash does not match, skip to next window.

Hash function (modulo q):

For a string S of length m over alphabet of digits (or mapped to integers),
hash(S) = (S[0]*d^(m-1) + S[1]*d^(m-2) + … + S[m-1]*d^0) mod q
where d is the number of characters in alphabet (typically 10 for digits, 256 for ASCII).

Rolling hash computation:

If current window hash is h, next window hash is:
h = ((h - T[i]*d^(m-1)) * d + T[i+m]) mod q

Example with modulo q=13 (generic):

Assume digits only (d=10). Let T = "235902", P = "902", m=3, d=10, q=13.
Compute pattern hash: P = "902" → 9*10^2 + 0*10 + 2 = 900+0+2 = 902.
902 mod 13 = 902 ÷ 13 = 69*13=897, remainder 5. So hash(P) = 5.
Compute first window T[0..2] = "235" → 2*100 + 3*10 + 5 = 235.
235 mod 13 = 235 ÷ 13 = 18*13=234, remainder 1. Hash = 1 ≠ 5 → no match.
Slide window:

- Window "359": 359 mod 13 = 359÷13=27*13=351, remainder 8 → hash=8 ≠5
- Window "590": 590 mod 13 = 590÷13=45*13=585, remainder 5 → hash=5 matches pattern hash.
  → Compare actual strings: "590" vs "902" → **not equal → spurious hit**.
- Window "902": 902 mod 13 = 5 → hash matches. Compare actual: "902" = "902" → **valid match**.

Spurious hits in this example: Occurred at window "590" (hash=5 but string differs). The algorithm then verifies and moves on.

Time complexity:

- Average case: O(n+m)
- Worst case (many spurious hits): O(n×m) but less than naive due to modulo arithmetic.

Real-world application: Detecting plagiarism, searching for patterns in large texts (e.g., virus signature scanning).

To solve your specific exam question: Substitute the given T, P, and q=13 into the steps above. Compute hashes, identify windows where hash matches pattern hash but strings differ – those are spurious hits.

****