(1) Computing the transition function

- The given pattern is P[1...m] and need to computes the transition function δ From the pattern.
- A comparison is carried out between text and a pattern and if a character matches with the text then transition function $\delta(q, a)$ is updated and pointed to the next state $(q+1)$. {q = state}
- If a match is not found, then algorithm checks for the previous computed Prefix state and switches back to the state Prefix[q], thus retrieving the state $\delta(q, a)$

Algorithm : COMPUTE-TRANSITION-FUNCTION-UPDATED (P, Σ)

```
1:  #pattern = P.length
2:  for q=0 to #pattern do
3:      for each character a ∈ Σ do
4:          if a == P[q+1] AND q < #pattern
5:              δ(q,a) = q+1
6:          else
7:              δ(q,a) = δ(prefix[q], a)
8:  return δ
```

- Line 3 will be executed O(#pattern) times
- Line 4 will be executed O(|Σ|) times
- Therefore, the running time of the algorithm will be O(#pattern .* |Σ|)

n-m n #text

(2) text : | pattern doesn't match | = pattern [0...m-1] |

P: $0 \le m \le$ #pattern , notbegin (0, n-m-1)

$0 \le n \le$ #text , pattern [0..m-1] = text [n-m .. n-1]

1) Line 6 of algorithm match()

- text [n-m ... n-1] has the matched pattern [0...m-1] as per the given representation
- pattern [0...#pattern -1] = text [n - # pattern .. n-1] when m = #pattern
- That means pattern is found at text [n-#pattern..n-1]
- Implies that string is found at starting position n-m of the text.
- That is what line 6 of the match() algorithm returns when their is a character of pattern match.

2) Line 8 of algorithm match()

- Here n = text.length() means 'n' is the end of #text.
- We know that m ≤ #pattern, but when line 8 is executed then that proves that m < #pattern (value of m)
- At this point, the count of characters matched is less than the actual number of characters of the Pattern
- Implies no matched character is found in the ~~#text~~ Text and so the line 8 of the algorithm returns -1, which is unacceptable value for the location state.

3) Line 6 of Prefix() algorithm:

- When Prefix[i] == -1 that means prefix is not been computed and go further to store the prefixes for later use.

(2)

3) - But when Prefix[i] ≠ -1 that indicates prefix of i has already been computed {pre-computation} and so we can just retrieved the pre-computed prefixes.

4) Line 8 of Prefix() algorithm:

- If i==1 means we have a single character.
- Proper prefix of a single character is always an empty or null string
- Therefore, length of longest proper prefix of a character will be 0.
- Hence, it returns the length of zero.

③ In order to compute all values in the array we use extend()

Algorithm : Compute-Prefix-Function-Updated (String Pattern)

* Consider a new array Prefix[1 ... #pattern]

Prefix[1] = 0

For q = 2 to #pattern

Calculate Prefix[q] = extend(pattern, Prefix[q-1], pattern[q-1])

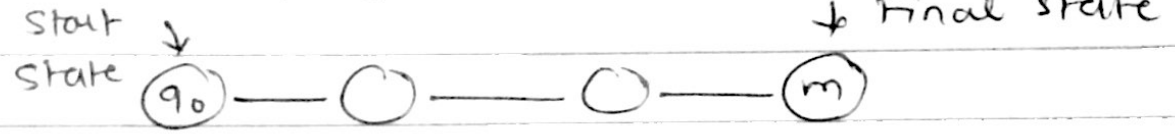
(3) return Prefix.

- In the above algorithm, Prefix [1] = 0 as we know proper prefix of a single character is an empty string.
- Prefix [i = 2 to #pattern]. is the computed array
- A ~~pre-c~~ pre-computed Prefix [i-1] and next character in the pattern helps invoke extend() to compute Prefix [i].
- As professor's notes terminates the algorithm, similarly this algorithm also terminates as it uses extend()
- Total time required to compute array values in Prefix

$$= 2 \times \#\text{pattern} + \text{Prefix}[1] - \text{Prefix}[\#\text{pattern}]$$

$$= O(\#\text{pattern})$$

(4) Yes, if we memoized the function extend (rather than prefix), then we are ~~constructing~~ constructing the FSA rather than shift function. This claim by professor is right if we consider set Φ consisting of $\{0, 1, \ldots m\}$ where q_0 is the start state and m is the final state - accepting state.



- The transition function would be

$$\delta(q, a) = \text{extend}(\text{pattern}, q, a)$$

q = any state

a = character.

④ - The memoized function will store matching pattern's substring and would point to next matching character if a match is found.

* Memoized function :-

```
private int extend (String pattern, int j,
                    char c)
{ if (Pattern[j] == -1)
else if (pattern.charAt(j) == c)
        Pattern[j] = j+1;
else if (j == 0)
        Pattern[j] = 0;

  Pattern[j] = extend (pattern, prefix(pattern, j),
                       c);
  return Pattern[j];
}
```