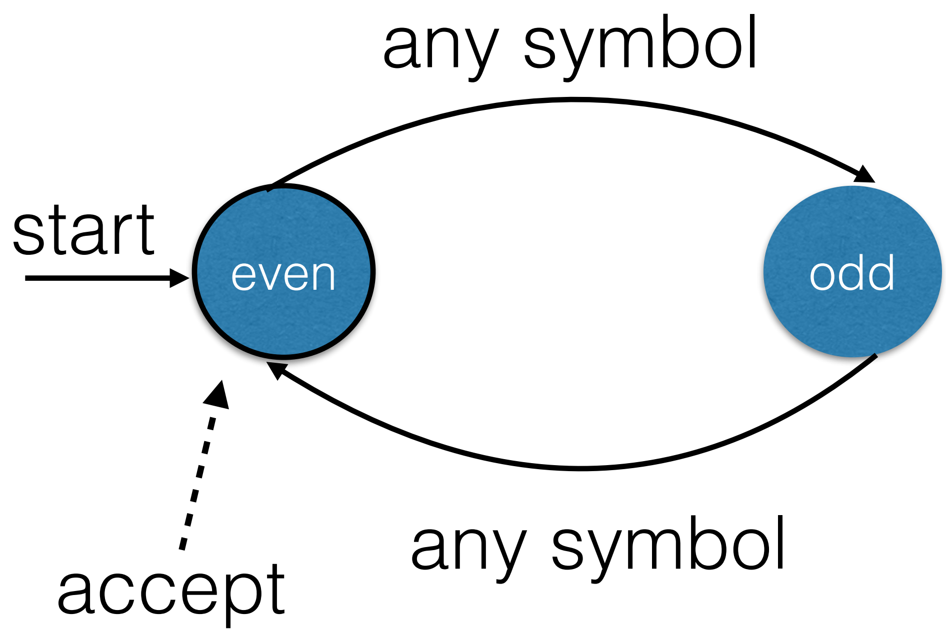
Knuth-Morrison-Pratt Algorithm

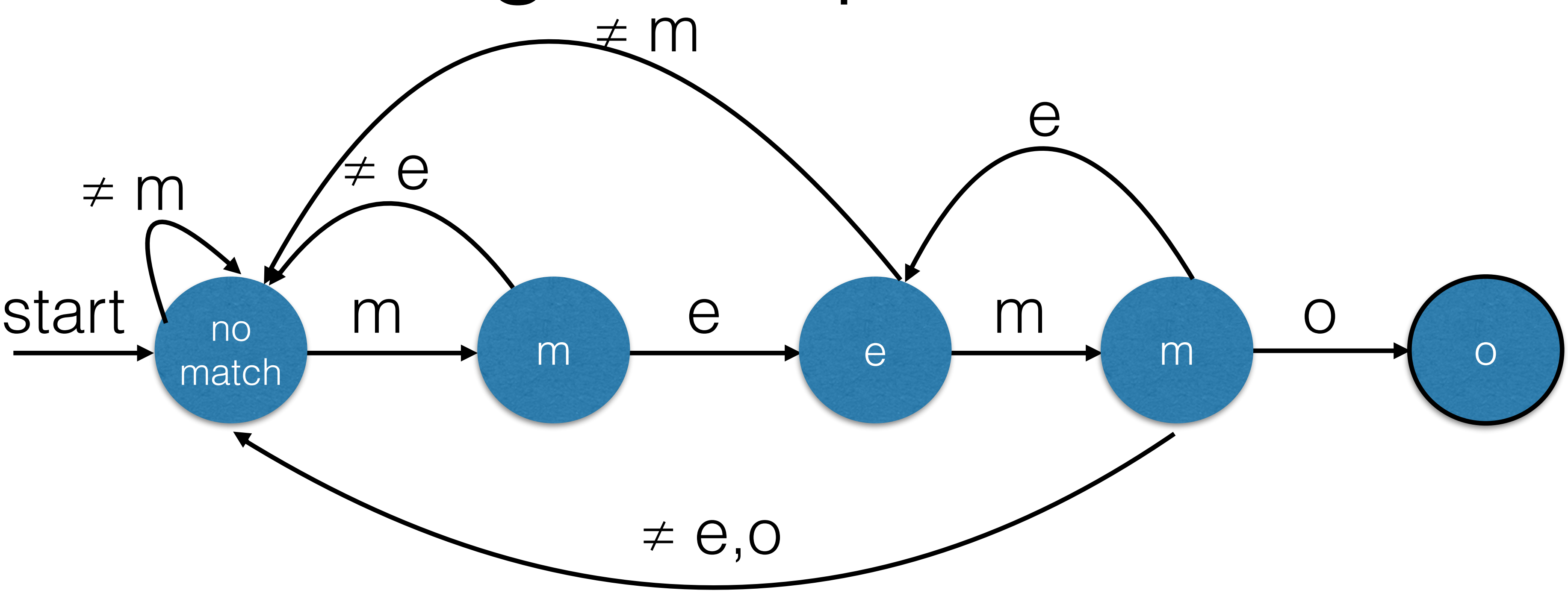
- String matching using finite automata (*string matching automata*)

Examples of a finite automaton

- two states (even, odd) and two transitions
- accepts when string read and in state even
- starts in state even
- per transition use one symbol at the time (from left to right)
- If input “finite” is entered it will end in state even and accept
- the automaton recognized the family of strings of even length



Examples of a finite automaton that recognizes pattern “memo”



advances in text symbol by symbol and in automaton according to transitions: for “amememorandummememo” the sequence of states is as follows: no match, no match, m, e, m, e, m, o

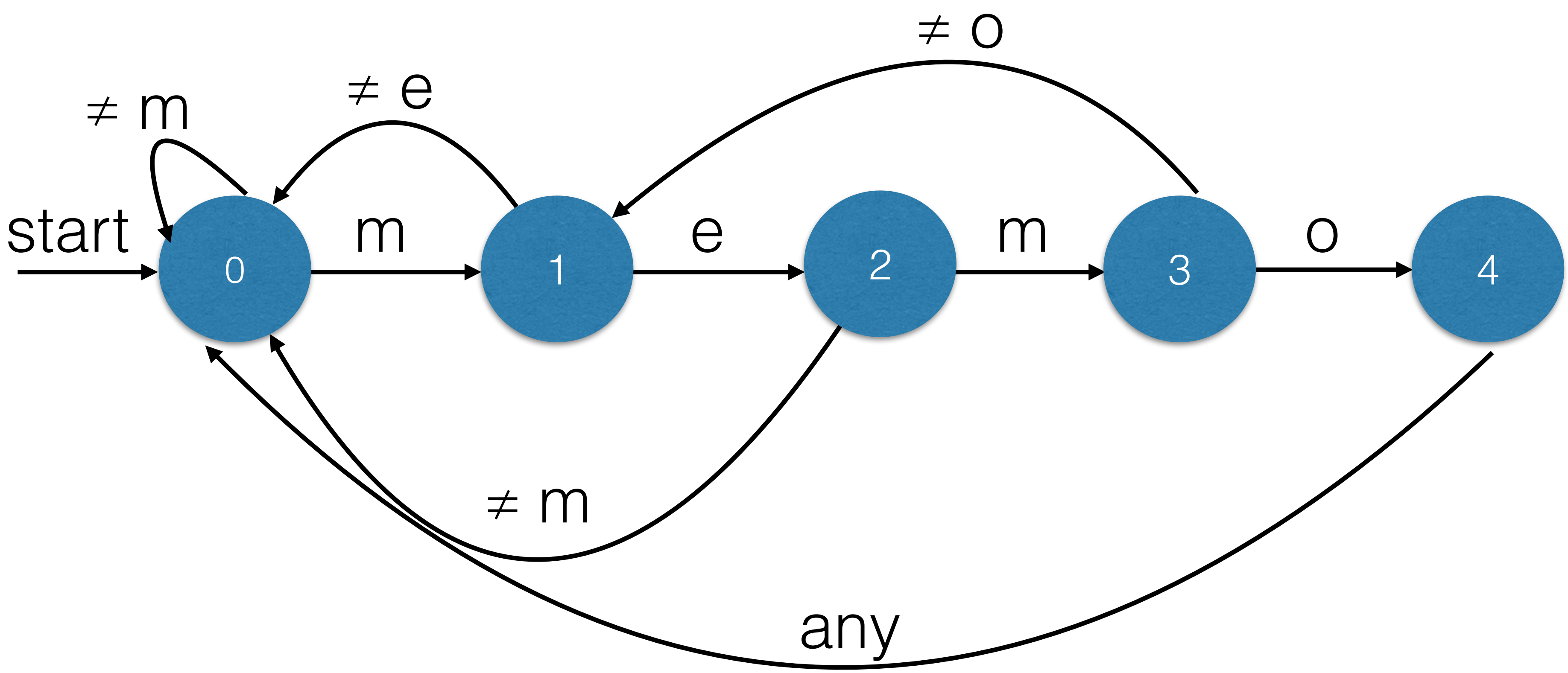
Problem

- automaton allows only to find the first occurrence of pattern in text
- Wanted: all occurrences in text

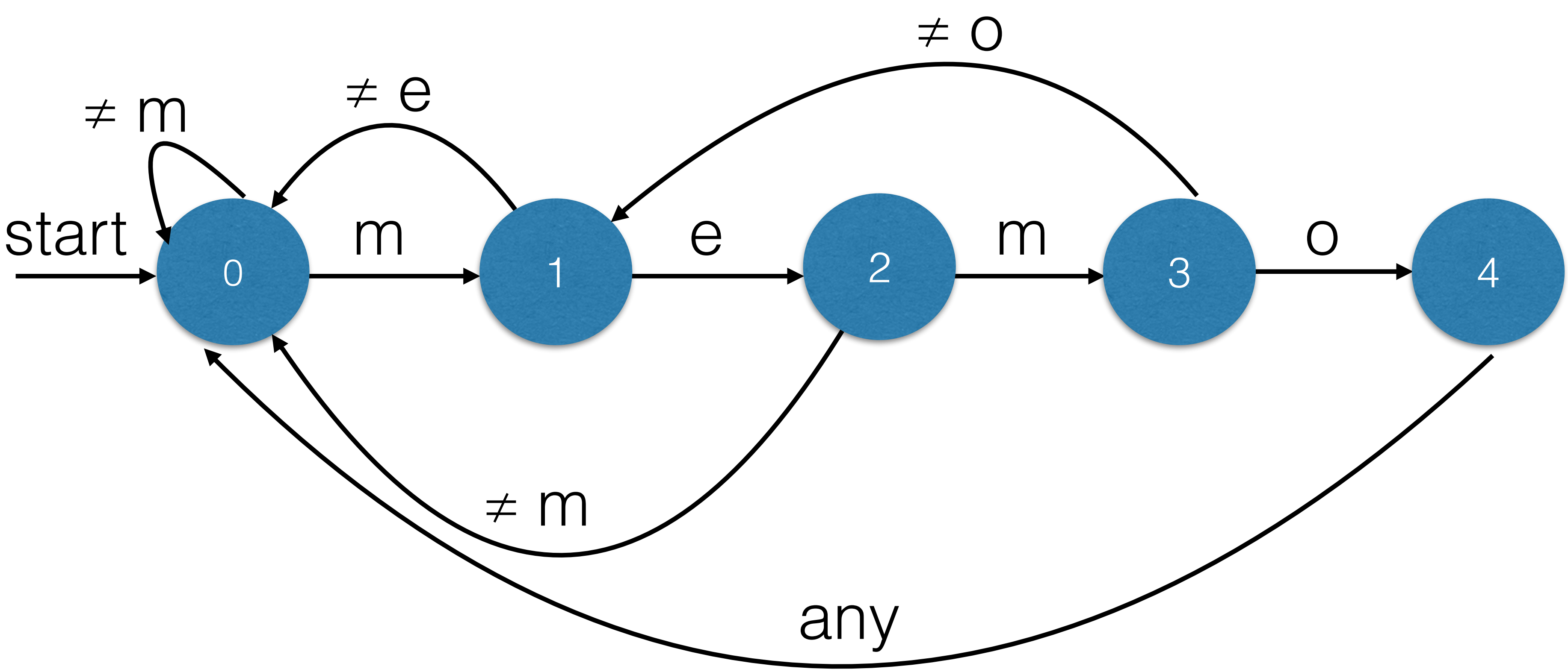
Finite automaton for the overlap of strings

- *Overlap* of two strings x and y: longest substring that is a (proper) suffix of x and a (proper) prefix of y.

Pattern automaton for “memo” set up to support the finding of all occurrences



Each state (but last) has one forward and one backward transition, last state has only a backward transition

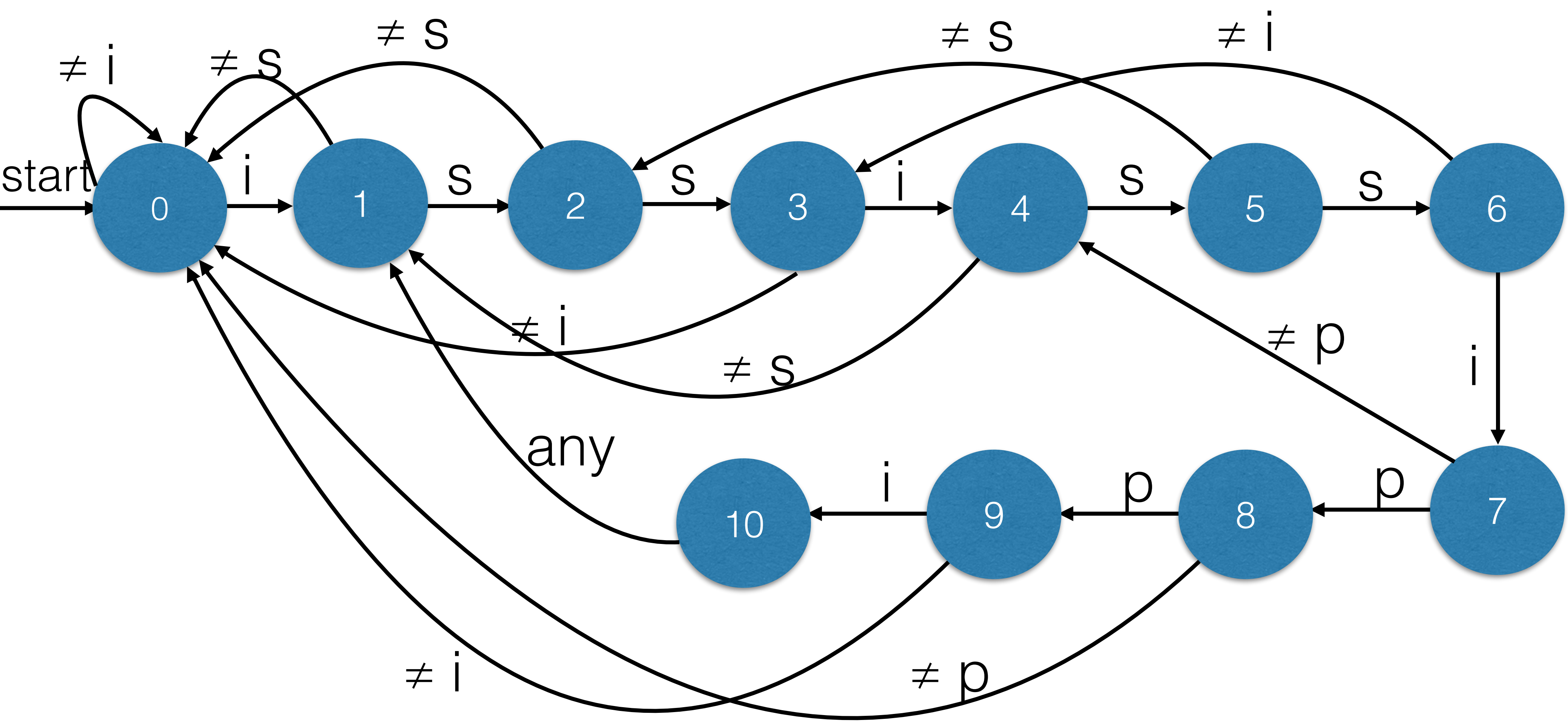


Each state (but 4, the last one) has one forward and one backward transition, last state has only a backward transition

Forward: match expanded, advance in pattern

Backward: reset to state that represents the current longest match. Current symbols in both pattern and text remain the same; the start of the match in the text is updated according to the current longest match

Pattern automaton for “ississippi”



Computing the overlap

- Recall that the *overlap* of two strings x and y is a longest substring that is a (proper) suffix of x and a (proper) prefix of y

- Note: if string w is a suffix of x and a prefix of y then it is also a suffix of the overlap of x and y

Algorithm ComputeOverlap(P)

overlap[1] ← 0

for k from 1 to m-1 **do** \\current sub-pattern is $P[1.. k+1]$

current ← $P[k+1]$

pre ← overlap[k]

while P[pre+1] ≠ current & pre ≠ 0 **do** \\find the precomputed largest fitting overlap

pre ← overlap[pre]

if P[pre + 1] = current **then** \\extend precomputed overlap if possible

overlap[k +1] = pre + 1

else overlap[k + 1] ← 0

return overlap

Running time of ComputeOverlap

- for loop: O(m)
- How often is the while loop statement $\mathbf{pre} \leftarrow \mathbf{overlap[pre]}$ executed in total?
  - only when $\mathbf{pre}$ is decreased
  - Since $\mathbf{pre}$ is increased by at most one per for-loop iteration, it is increased by at most m-1 in total
  - Therefore, $\mathbf{pre} \leftarrow \mathbf{overlap[pre]}$ can be executed at most m-1 times in total
- Total running time of ComputeOverlap is O(m)

KMP Algorithm

- Preprocess pattern P: Execute ComputeOverlap(P)
- Execute KMP(T,P)

Algorithm KMP(T, P)

$i \leftarrow 0; j \leftarrow 1$

while $i < n\text{-}m$ **do** \\scan through the text for occurrences of P

\\Invariant: pattern $P[1..(j\text{-}1)]$ occurs in T: $T[(i+1)..(i+j\text{-}1)] = P[1..(j\text{-}1)]$

if $j = m$ & $T[i + j] = P[j]$ **then** \\output occurrence; advance in text and adjust pattern according to overlap

output i

$i \leftarrow i + j$ - overlap[j]

$j \leftarrow$ overlap[j] + 1

else if $j < m$ & $T[i + j] = P[j]$ **then** j++ \\advance the match

else if j-1 = 0 **then** i++ \\no match so far, advance in text

else \\ pattern match aborted; advance in text and adjust pattern according to overlap

$j \leftarrow$ overlap[j] + 1

$i \leftarrow i + j$ - overlap[j] - 1

Running time

- number of while loop iterations?
- idea: count the number of symbol comparisons
  $T[i+j] = P[j]$ performed
    - For each symbol in T (that is each $T[k]$):
        - once a symbol in T was successfully matched with a symbol in P, the symbol will not be reused (the overlap takes care of this)
        - symbols in a mismatch comparison either match or we advance in text (2nd else if)
    - No symbol in T is used for more than two comparisons
- O(n)