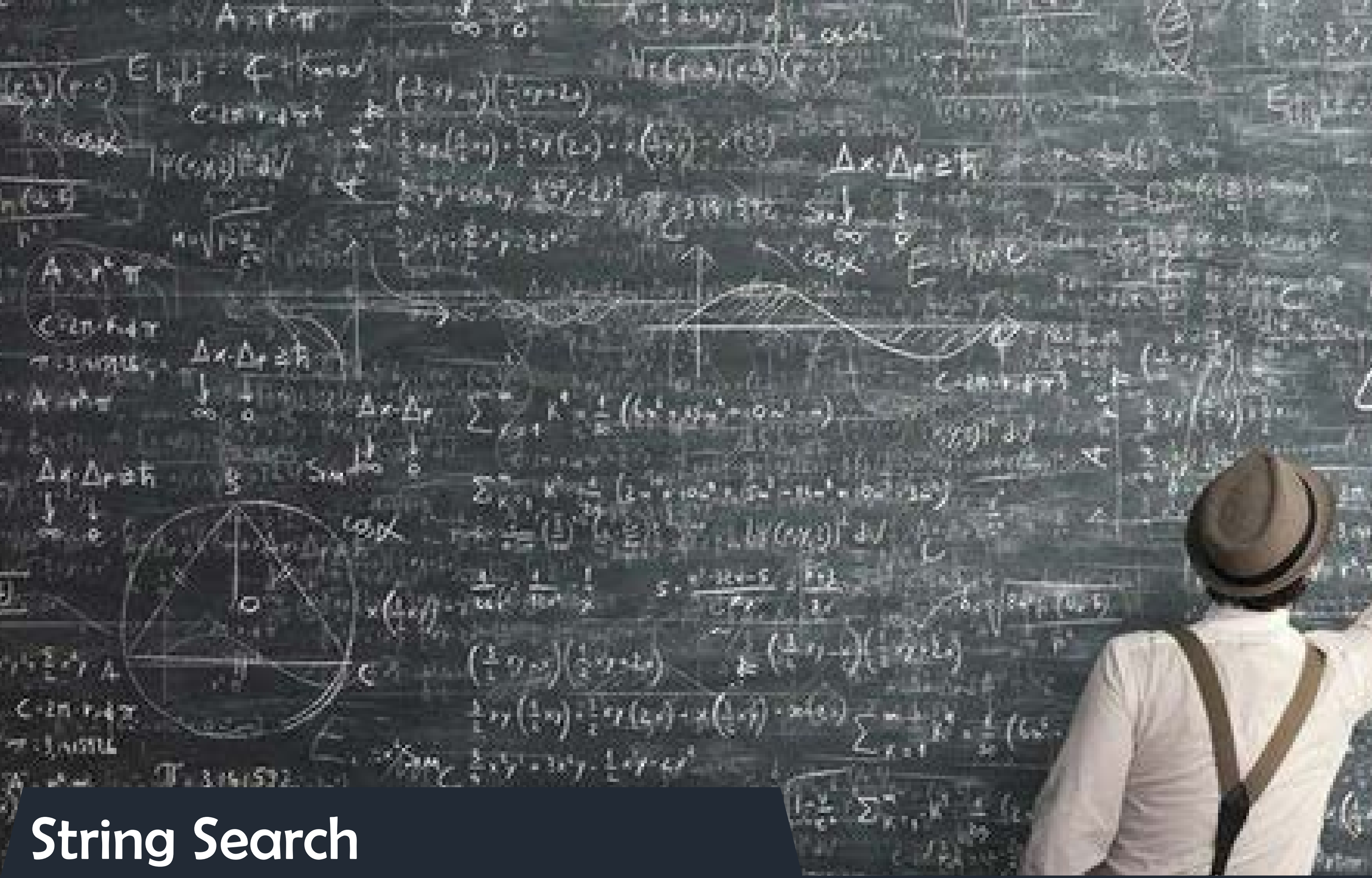
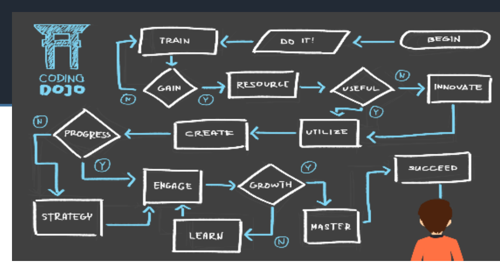


String Search

Ja-Hee Kim

Agenda



Introduction

Algorithms

Review

Introduction

String Matching Problem

- Given a text string *T* of length *n* and a pattern string *P* of length *m*, the exact string-matching problem is to find all occurrences of P in T.
- Example:
  - T="AAAAAACAAAA<u>AABA</u>BC"
  - P="AABA"
  - Return: index 11
- Applications:
  - Searching keywords in a file
  - Searching engines (like Google and Openfind)
  - Database searching (GenBank)

Terminologies

- S: String
  - Example: S="AGCTTGA"
- |S|: length of S
  - Example: |S|=7
- Substring: $S_{i,j}=S_i S_{i+1} \ldots S_j$
  - Example: $S_{2,4}$="GCT"
- Subsequence of S: deleting zero or more characters from S
  - "ACT" and "GCTT" are subsequences.
- Prefix of S: $S_{1,k}$
  - "AGCT" is a prefix of S.
- Suffix of S: $S_{h,|S|}$
  - "CTTGA" is a suffix of S.

Algorithms

- naïve algorithm
- Rabin-Karp Algorithm
- Finite state machine
- KMP algorithm
- Boyer-Moore algorithm

Naïve Algorithm

- Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

	0	1	2	3	4	5	6	7	8	9
Text:	A	A	A	C	A	A	B	A	B	C

Pattern: AABA

Naïve Algorithm

- Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

0 1 2 3 4 5 6 7 8 9

Text: A A A C A A B A B C

A A B A

Pattern found at 4

Pattern:

Naïve Algorithm

- Time complexity
  - Best case: O(n)
    - There is no first character of the pattern in the text
      - Text: AABCCAADDEE
      - Pattern: FAA
  - Worst case: O(m*(n-m+1))
    - When all characters of the text and pattern are same.(find all)
      - txt[] = "AAAAAAAAAAAAAAAAAA";
      - pat[] = "AAAAA";
    - Worst case also occurs when only the last character is different.(find one)
      - txt[] = "AAAAAAAAAAAAAAAAAB";
      - pat[] = "AAAAB";

Rabin-Karp Algorithm

- Slide the pattern over text one by one and matches the hash value of the pattern with the hash value of current substring of text
- Assume the alphabet uses *d* digit
  - $P = P[m-1]+d*P[m-2]+\ldots d^{m-1}P[0]$
  - $t_i = T[i+m-1]+d*T[i+m-2]+\ldots d^{m-1}T[i]$

    $= d*(t_i\text{-}1-d^{m-1}T[i-1])+T[i+m-1]$

Rabin-Karp Algorithm

- If we use decimal number, and A:1, B:2, C: 3.
- Pattern AABA is 1121

$\neq$

Hash value: 1113

Text: $\boxed{\text{A A A C}}$ A A B A B C

Rabin-Karp Algorithm

- If we use decimal number, and A:1, B:2, C: 3.
- Pattern AABA is 1121

≠

Hash value: 1113 Hash value: 1131

Text: A $\boxed{\text{AACA}}$ ABABC

Rabin-Karp Algorithm

- If we use decimal number, and A:1, B:2, C: 3.
- Pattern AABA is 1121

$\neq$

Hash value: 1311

Text: A A $\boxed{\text{A C A A}}$ B A B C

Rabin-Karp Algorithm

- If we use decimal number, and A:1, B:2, C: 3.
- Pattern AABA is 1121

$\neq$

Hash value: 3112

Text: A A A C A A B A B C

Rabin-Karp Algorithm

- If we use decimal number, and A:1, B:2, C: 3.
- Pattern AABA is 1121

Hash value: 1121

Text: A A A C $\boxed{\text{A A B A}}$ B C

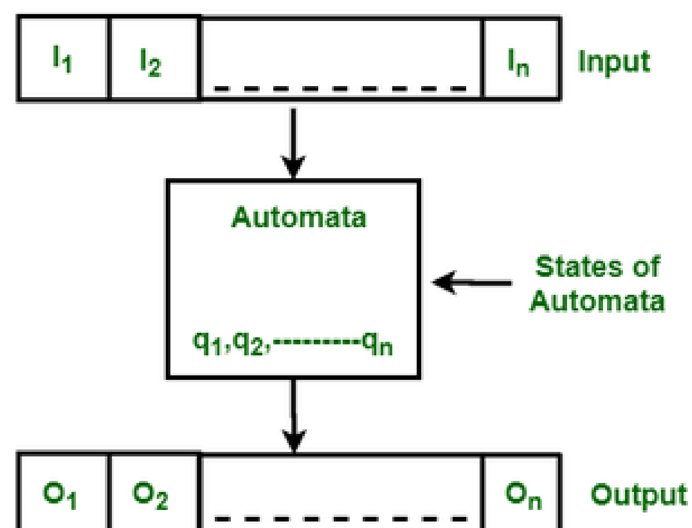
Pattern found at 4

Rabin-Karp Algorithm

- The average and best case running time of the Rabin-Karp algorithm is O(n+m)
  - Hash at the next shift must be O(1) operation.
  - $t_i = d*(t_i\text{-}1\text{-}d^{m-1}T[i\text{-}1])+T[i+m\text{-}1]$
- The worst case running time of the Rabin-Karp algorithm is O(nm)
- Problem of Rabin-Karp algorithm: overflow
  - If the set of alphabet and m is too big, the algorithm causes overflow.
  - Select a sufficiently large prime number 1
  - Use $a_i = t_i \bmod q$

Finite state machine

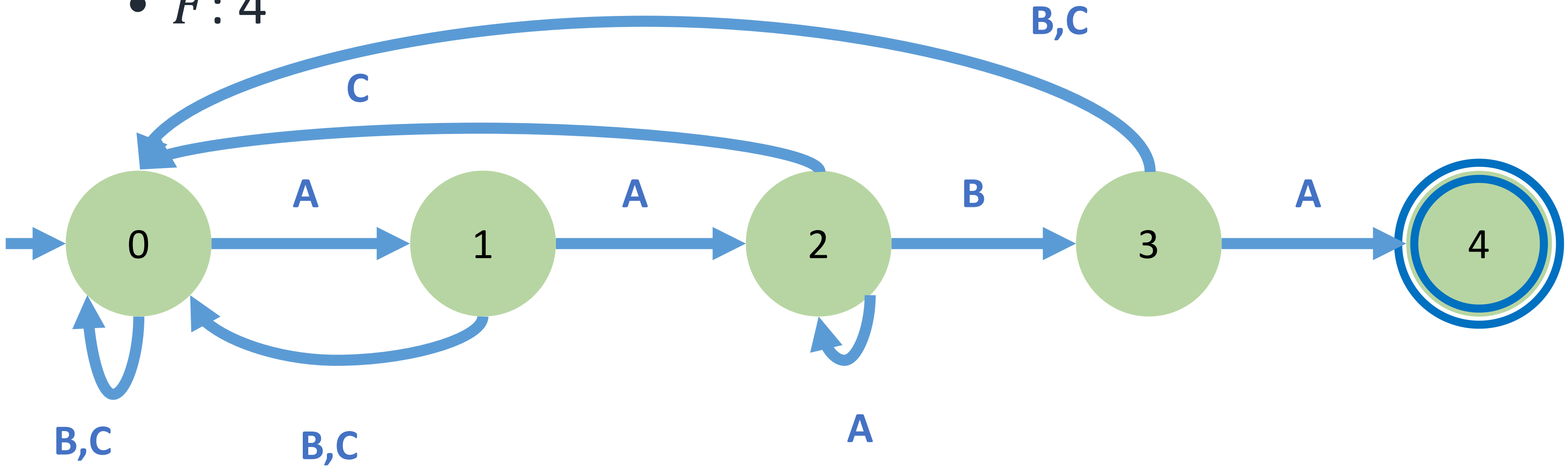
- Build finite state machine
- Finite state machine
  - accepts or rejects strings of symbols and only produces a unique computation.
  - 5-tuple $(Q, \Sigma, \delta, q_0, F)$
    - a finite set of states Q
    - a finite set of input symbols called the alphabet Σ
    - a transition function $\delta: Q \times \Sigma \longrightarrow Q$
    - an initial or start $q_0 \in Q$
    - a set of accept states $F \in Q$
- Time complexity: O(n)



Finite state machine

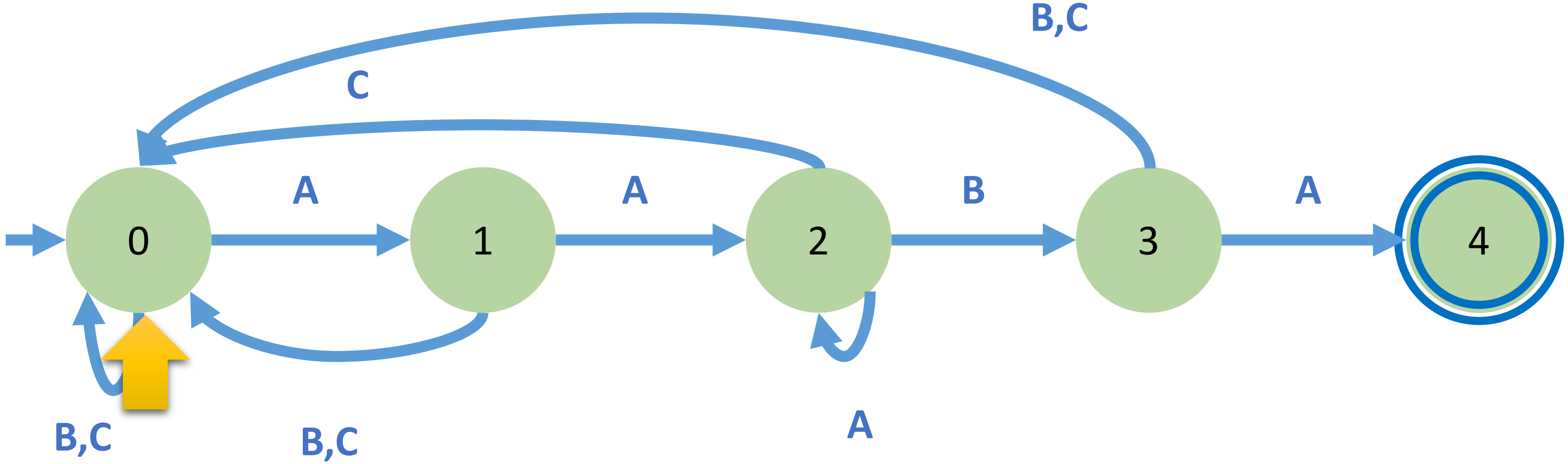
- Text: A A A C A A B A B C
- Pattern: AABA
- 4-tuple $(Q, \Sigma, \delta, q_0, F)$
  - *Q: 0*
  - Σ={A, B, C}
  - q_0:0
  - F: 4

State	A	B	C
0	1	0	0
1	2	0	0
2	2	3	0
3	4	0	0



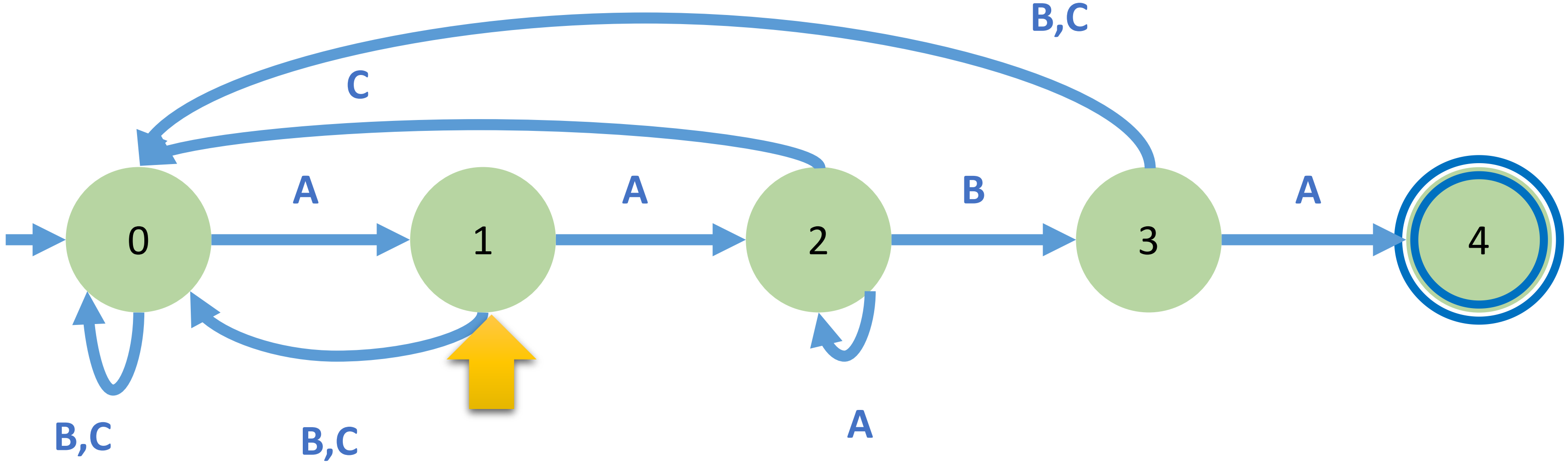
Finite state machine

- Text: A A A C A A B A B C



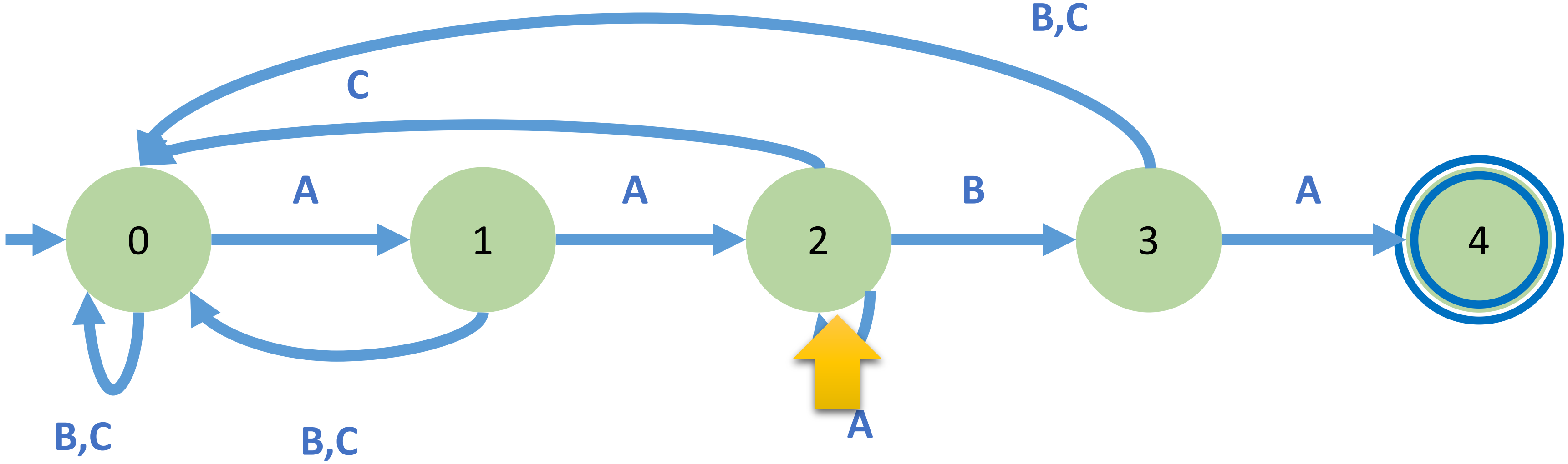
Finite state machine

- Text: A A A C A A B A B C



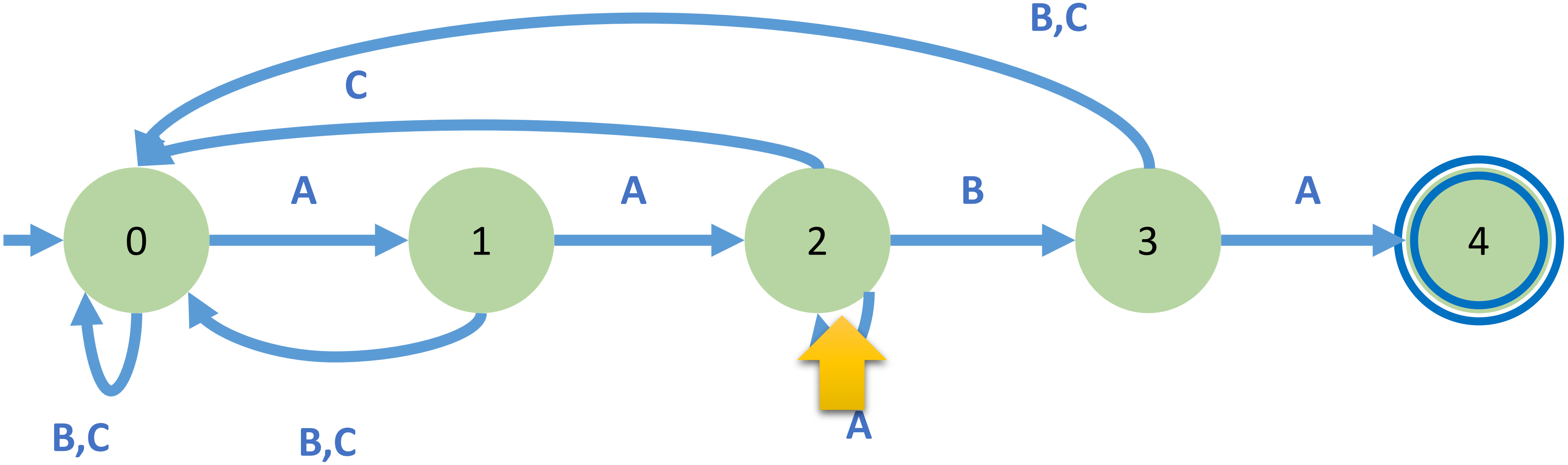
Finite state machine

- Text: A A A C A A B A B C



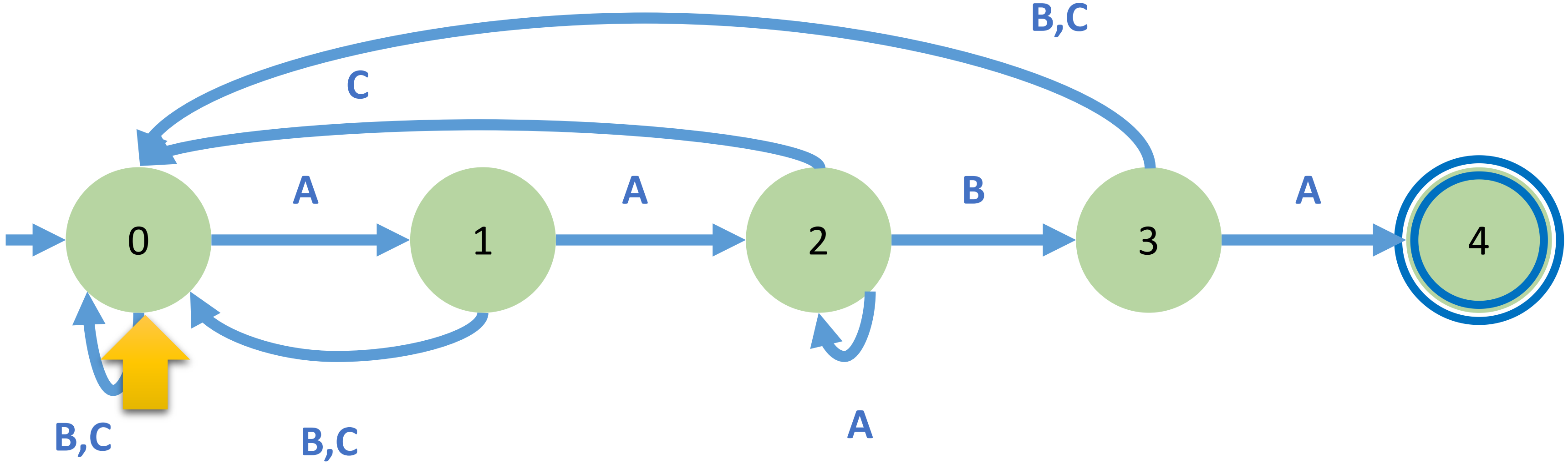
Finite state machine

- Text: A A A C A A B A B C



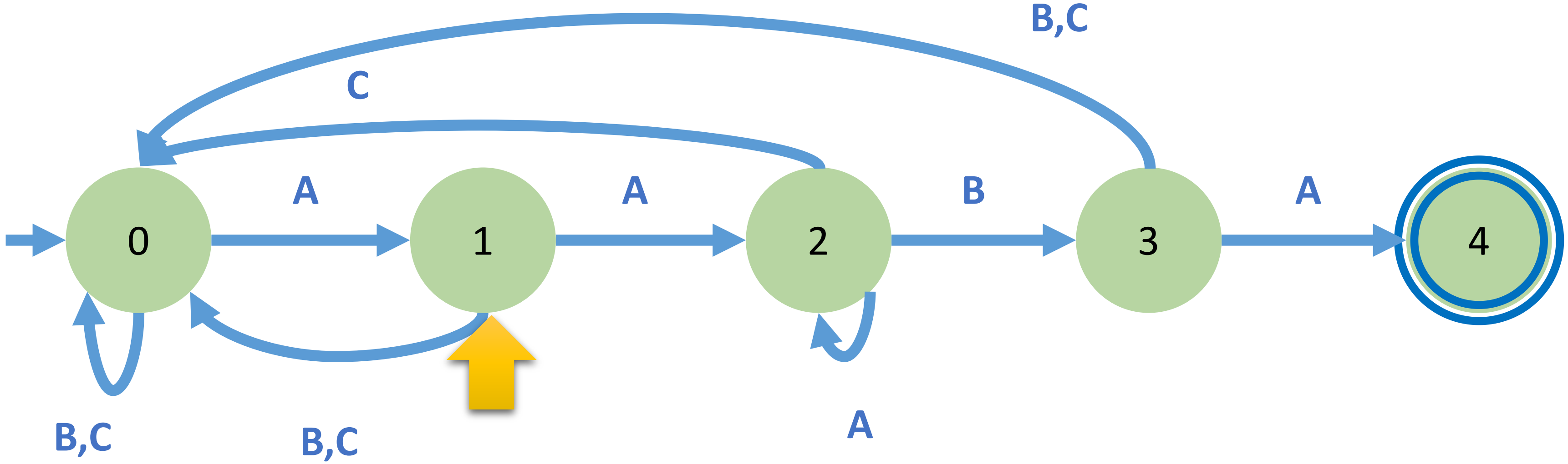
Finite state machine

- Text: A A A C A A B A B C



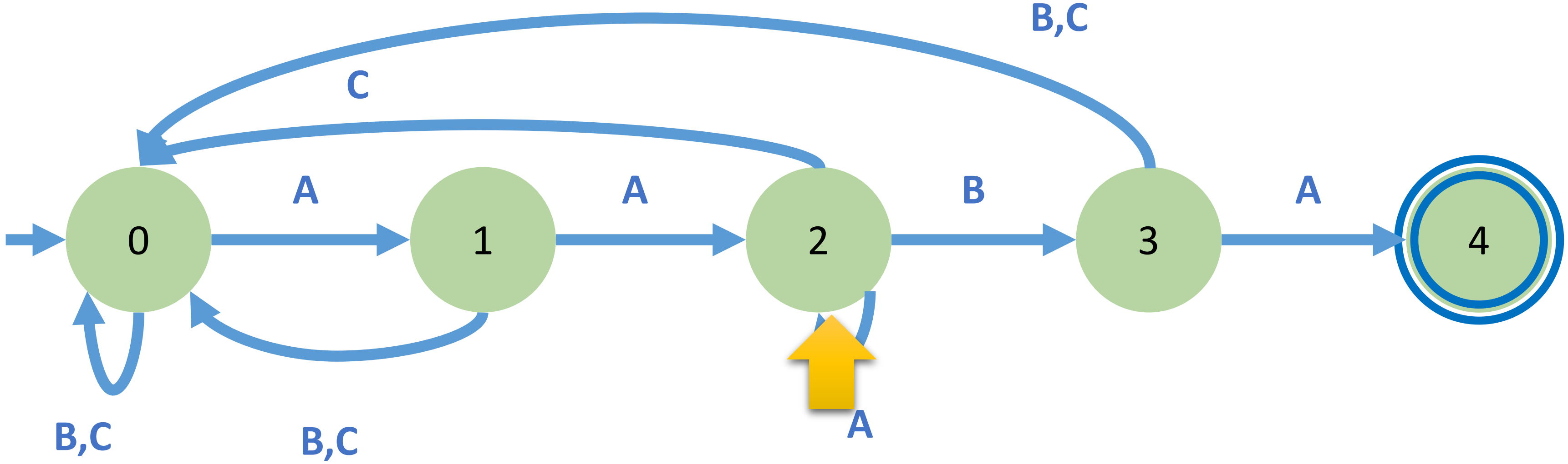
Finite state machine

- Text: AAACAABABC



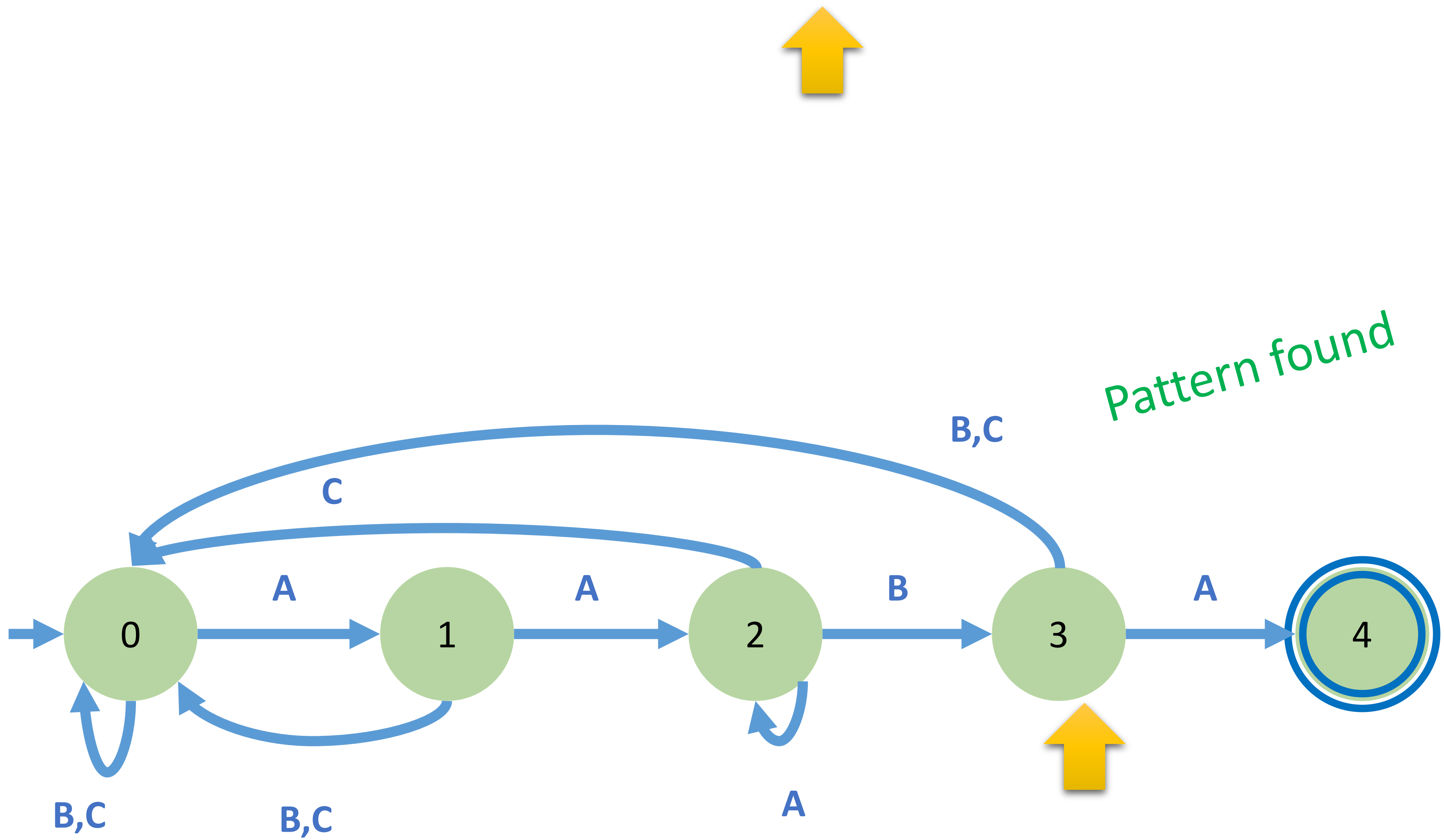
Finite state machine

- Text: A A A C A A B A B C



Finite state machine

- Text: A A A C A A B A B C



KMP algorithm

- KMP (Knuth Morris Pratt) Pattern Searching
- whenever we detect a mismatch (after some matches), 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.
- Example:
  - Text: AAAAABAAABA
  - Pattern = AAAA
- Preprocessing is necessary.

KMP algorithm

- Preprocessing for constructing an auxiliary array
  - Same size of pattern
  - used to skip characters while matching
  - length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].
  - Algorithm
    - i=1, len=0, lpr[0]=0
    - while(i<m)
      - If(len==0 or P[i]==P[len]): len++, lpr[i]=len, i++
      - else: len = lpr[len-1]

KMP algorithm

- Algorithm
  - i=1, len=0, lpr[0]=0
  - while(i<m)
    - If(P[i]==P[len]): len++, lpr[i]=len, i++
    - else if (len == 0) lpr[i]=len, i++
    - else: len = lpr[len-1]
- Example
  - 0 1 2 3 4 5 6 7 8
    A B C D A B C W Z
  - Pattern: AABA
  - lpr:

0	1	2	3	4	5	6	7	8

  - i:
  - len :

KMP algorithm

```
int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }

    if (j == M) {
        printf("Found pattern at index %d ", i - j);
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}
```

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

KMP algorithm

- Example
  - Text: A A A C A A B A B C
  - Pattern: AABA

0	1	2	3
0	1	0	1

```
int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }

    if (j == M) {
        printf("Found pattern at index %d ", i - j);
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}
```

0	1	2	3	4	5	6	7	8	9
A	A	C	A	A	A	B	A	B	C

0	1	2	3
A	A	B	A

KMP algorithm

- Example
  - Text: A A A C A A B A B C
  - Pattern: AABA

0	1	2	3
0	1	0	1

```
int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }

    if (j == M) {
        printf("Found pattern at index %d ", i - j);
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}
```

0	1	2	3	4	5	6	7	8	9
A	A	C	A	A	A	B	A	B	C
0	1	2	3						
A	A	B	A						

KMP algorithm

- Example
  - Text: A A A C A A B A B C
  - Pattern: AABA

0	1	2	3
0	1	0	1

```
int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }

    if (j == M) {
        printf("Found pattern at index %d ", i - j);
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}
```

0	1	2	3	4	5	6	7	8	9
A	A	C	A	A	A	B	A	B	C

0	1	2	3
A	A	B	A

KMP algorithm

- Example
  - Text: A A A C A A B A B C
  - Pattern: AABA

0	1	2	3
0	1	0	1

```
int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }

    if (j == M) {
        printf("Found pattern at index %d ", i - j);
        j = lps[j - 1];
    }

    // mismatch after j matches
    else if (i < N && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}
```

0	1	2	3	4	5	6	7	8	9
A	A	C	A	A	A	B	A	B	C

0	1	2	3
A	A	B	A

KMP algorithm

- Time complexity
  - Preprocessing: O (m)
  - Matching: O(n)
- Overall time complexity: O(m+n)
- Space time complexity: O(m)

Boyer-Moore algorithm

- It requires a preprocessing
- Heuristics
  - Bad character matching
  - Good suffix
- It starts matching from the **last** character of the pattern.

Boyer-Moore algorithm

- Case 1: pattern move past the mismatch character

A A C A A A B A B C

≠ =

A A B A

Pattern has no C

Boyer-Moore algorithm

- Case 2: mismatch become match

A A C A A A B A B C

A A B A

B is matched

Boyer-Moore algorithm

- Case 2: mismatch become match

A A C A A A B A B C

A A B A

- Time complexity
  - Worst case - O(mn): all characters are the same
  - Best case- O(n/m): all the characters are the different

Regular Expression

Regular Expression

- A regular expression is a kind of pattern that can be applied to text (`StriXng`s, in Java)
- A regular expression either matches the text (or part of the text), or it fails to match
- Beginning with Java 1.4, Java has a regular expression package, `java.util.regex`

Process

- First, you must *compile* the pattern
- Next, you must create a *matcher*

StringMatch.java

```java
// Pattern and Matcher are both in java.util.regex
import java.util.regex.*;

public class StringMatch {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("[a-z]+");
        String str = "Now is the time";
        Matcher m = p.matcher(str);
        System.out.println(m.find());
        System.out.println(str.substring(m.start(), m.end()));
        System.out.println(m.lookingAt());

    }
}

```

Console | Problems | Debug Shell

<terminated> StringMatch [Java Application] C:₩Program Files₩Java₩jdk-15.0.1₩bin₩javaw

```
true
ow
false
```

Methods

- m.find()
  - true if the pattern exists in the text string
  - false otherwise
- m.lookingAt()
  - true if the pattern matches at the beginning of the text string
  - false otherwise

```java
        Pattern p = Pattern.compile("[a-z]+");
        String str = "Now is the time";
        Matcher m = p.matcher(str);
        System.out.println(m.find());
        System.out.println(str.substring(m.start(), m.end()));
        System.out.println(m.lookingAt());

    }
}
```

Console ⌧ Problems Debug Shell

<terminated> StringMatch [Java Application] C:₩Program Files₩Java₩jdk-15.0.1₩bin₩ja

```
true
ow
false
```

Methods

- *After a successful match*
  - m.start() will return the index of the first character matched
  - m.end() will return the index of the last character matched, *plus one*
  - If no match was attempted, or if the match was unsuccessful, m.start() and m.end() will throw an IllegalStateException
    - This is a RuntimeException, so you don't have to catch it

```java
        Pattern p = Pattern.compile("[a-z]+");
        String str = "Now is the time";
        Matcher m = p.matcher(str);
        System.out.println(m.find());
        System.out.println(str.substring(m.start(), m.end()));
        System.out.println(m.lookingAt());

    }
}
```

Console ⌧ Problems Debug Shell

<terminated> StringMatch [Java Application] C:₩Program Files₩Java₩jdk-15.0.1₩bin₩ja

```
true
ow
false
```

*StringMatch.java

```java
// Pattern and Matcher are both in java.util.regex
import java.util.regex.*;

public class StringMatch {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("[a-z]+");
        String str = "Now is the time";
        Matcher m = p.matcher(str);
        while (m.find())
            System.out.print(str.substring(m.start(), m.end()) + "*");
    }
}
```

Console Problems Debug Shell

```
<terminated> StringMatch [Java Application] C:₩Program Files₩Java₩jdk-15.0.1₩bin₩javaw.exe (202
ow*is*the*time*
```

Regular Expression

abc	exactly this sequence of three letters
[abc]	any *one* of the letters a, b, or c
[^abc]	any character *except* one of the letters a, b, or c (immediately within an open bracket, ^ means "not," but anywhere else it just means the character ^)
[a-z]	any *one* character from a through z, inclusive
[a-zA-Z0-9]	any *one* letter or digit
|	is used to separate alternatives

Special character in Regular Expression

^	the beginning of a line
$	the end of a line
\b	a word boundary
\B	not a word boundary
\A	the beginning of the input (can be multiple lines)
\Z	the end of the input except for the final terminator, if any
\z	the end of the input
\G	the end of the previous match

Backslash in Regular Expression

.	any one character except a line terminator
\d	a digit: [0-9]
\D	a non-digit: [^0-9]
\s	a whitespace character: [\t\n\x0B\f\r]
\S	a non-whitespace character: [^\s]
\w	a word character: [a-zA-Z_0-9]
\W	a non-word character: [^\w]

Cardinality

Assume *X* represents some pattern

X?	optional, *X* occurs once or not at all
*X**	*X* occurs zero or more times
X+	*X* occurs one or more times
X{*n*}	*X* occurs exactly *n* times
X{*n*,}	*X* occurs *n* or more times
X{*n*,*m*}	*X* occurs at least *n* but not more than *m* times

Example

StringMatch.java

```java
// Pattern and Matcher are both in java.util.regex
import java.util.regex.*;

public class StringMatch {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("[rR]evolution[s]?");
        System.out.println(p.matcher("Revolutions").matches());
        System.out.println(p.matcher("revolutions").matches());
        System.out.println(p.matcher("Revolution").matches());
        System.out.println(p.matcher("Revolutionss").matches());
    }
}
```

Console Problems Debug Shell

<terminated> StringMatch [Java Application] C:₩Program Files₩Java₩jdk-15.0.1₩bin₩javaw.e

```
true
true
true
false
```

Thanks