Pattern Matching
(Đối sánh mẫu)

table of contents

01.

Introduction

Introduction

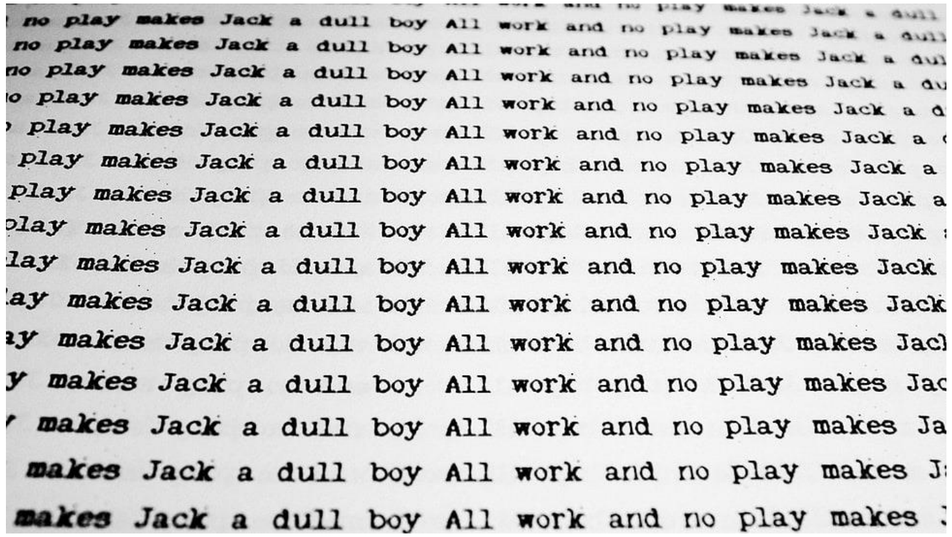
What is pattern matching? What is substring search?

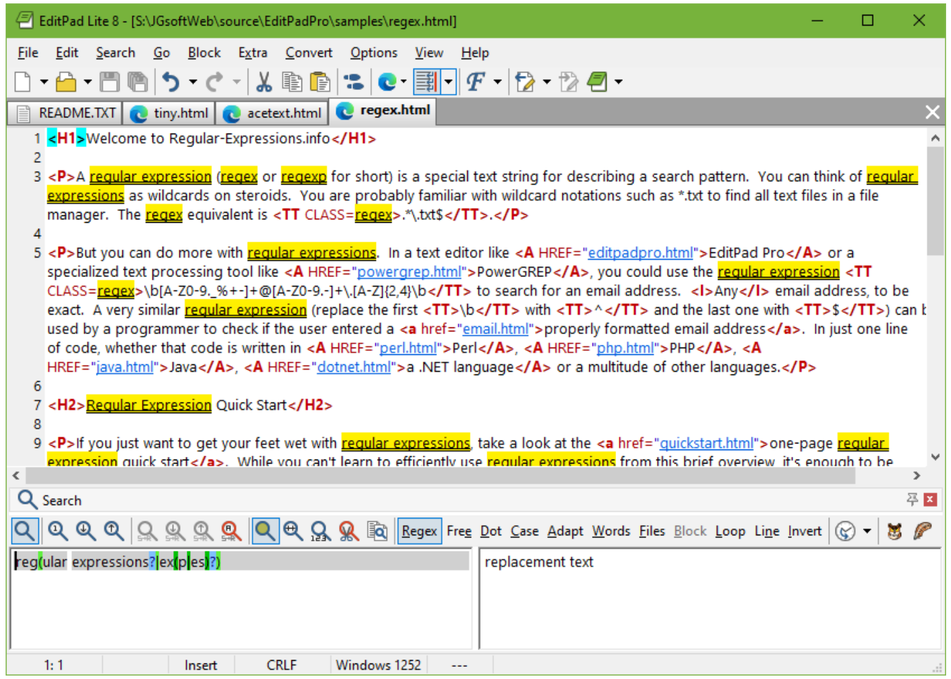
- *Pattern matching*: a technique used in computer science to check a given sequence of tokens for the presence of the constituents of some pattern.

- *Substring search*: given a text string of length N and a pattern string of length M, find an occurrence of the pattern within the text.

What is the difference between pattern matching and substring search?

Why we need to learn about “Pattern matching”?



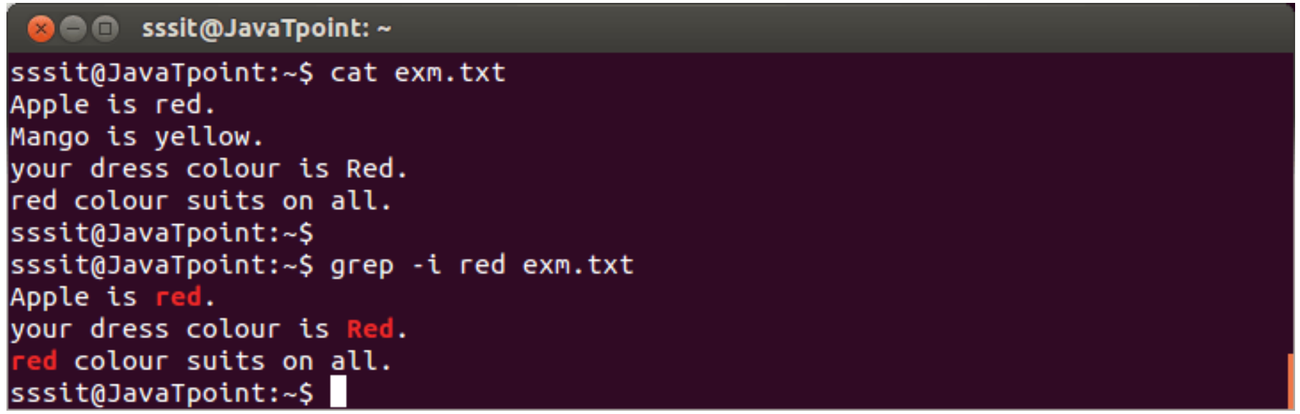



```
NAME
       grep, egrep, fgrep, rgrep - print lines matching a pattern

SYNOPSIS
       grep [OPTIONS] PATTERN [FILE...]
       grep [OPTIONS] [-e PATTERN]...  [-f FILE]...  [FILE...]

DESCRIPTION
       grep searches the named input FILEs for lines containing a
       match to the given PATTERN.  If no files are specified, or
       if  the  file  “-” is given, grep searches standard input.
       By default, grep prints the matching lines.

       In addition, the variant programs egrep, fgrep  and  rgrep
       are   the   same   as   grep -E,   grep -F,  and  grep -r,
       respectively.  These  variants  are  deprecated,  but  are
       provided for backward compatibility.
```



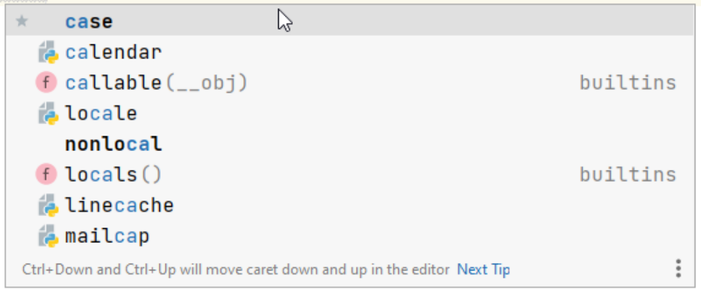
Why we need to learn about “Pattern matching”?

literals.py

```python
def get_color(color_code):
    match color_code:
        case '#FF0000':
            print("Red")
        case 3093151:
            print("Blue")
        case False:
            print("Not a color")
        case None:
            print("None")


get_color('#FF0000')
```

```python
def fib(arg):
    match arg:
        case n:
            return fib(n - 1) + fib(n - 2)
        ca
```



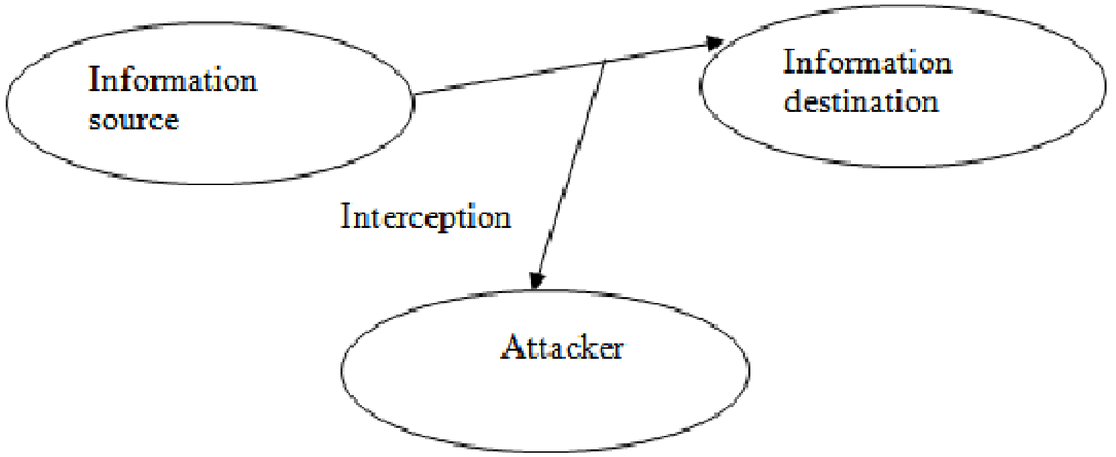
In the beginning God created the heaven and the earth.

And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.

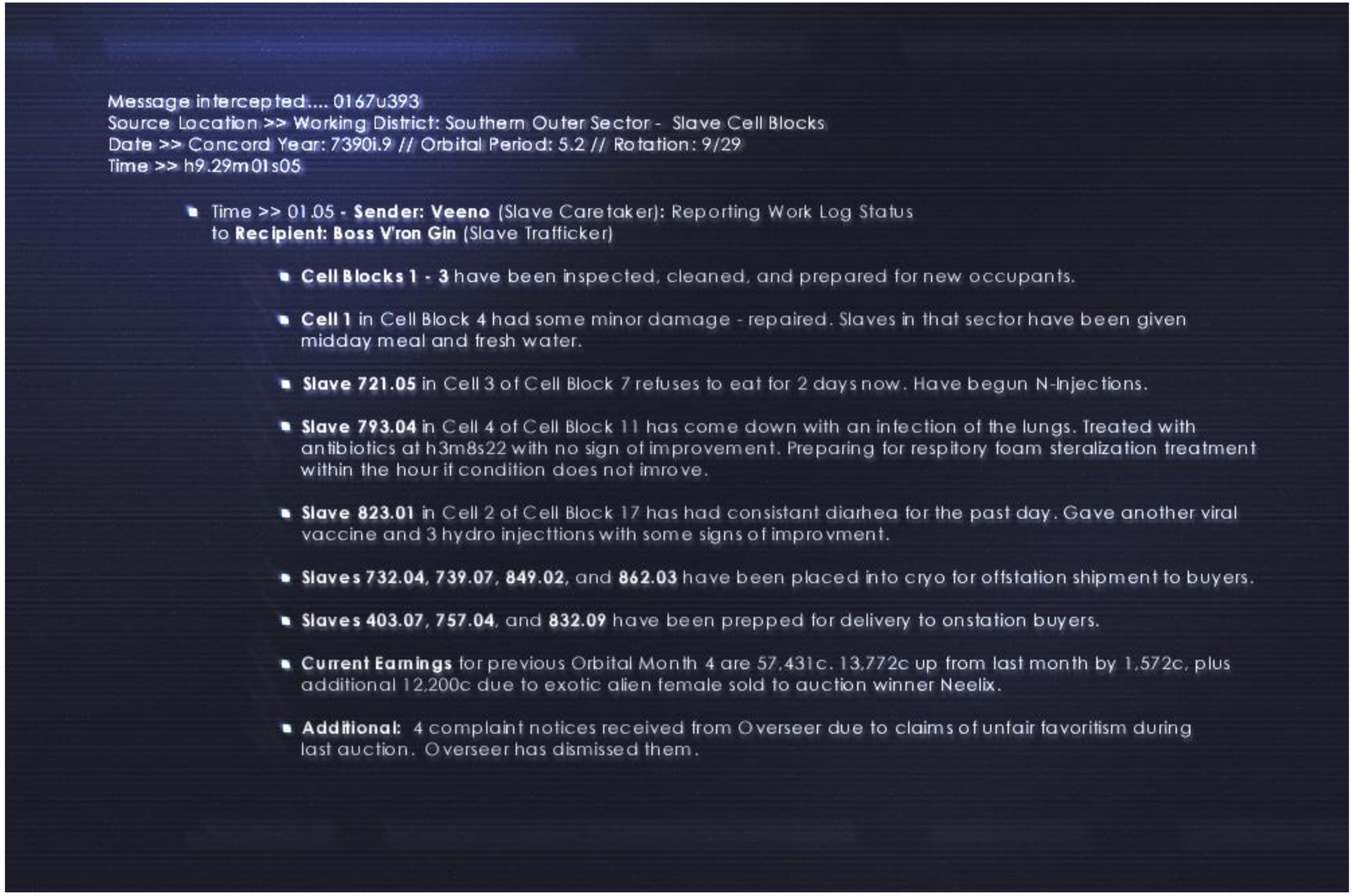
And God said, Let there be light: and there was light.

And God saw the light, that it was good: and God divided the light from the darkness.

Why we need to learn about “Pattern matching”?



Why we need to learn about “Pattern matching”?



Why we need to learn about “Pattern matching”?

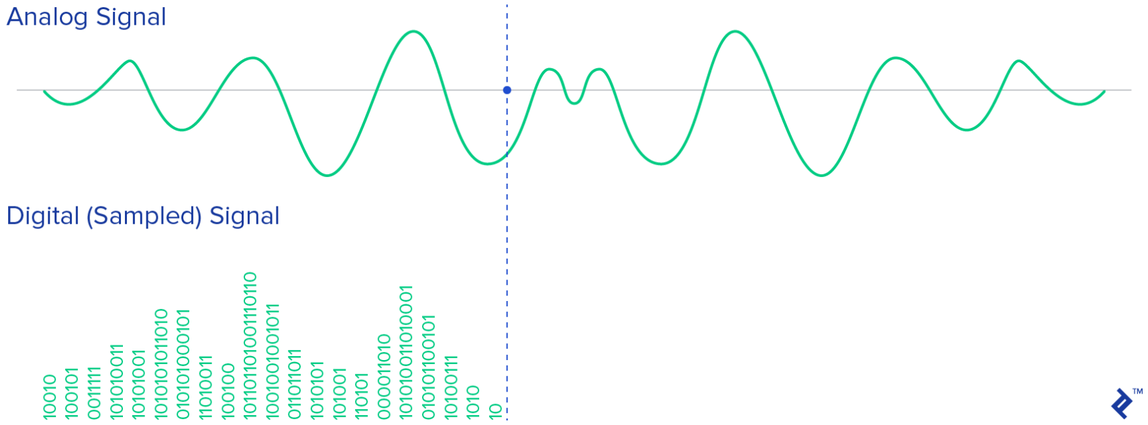
DANISH GROUP 4.5.45.

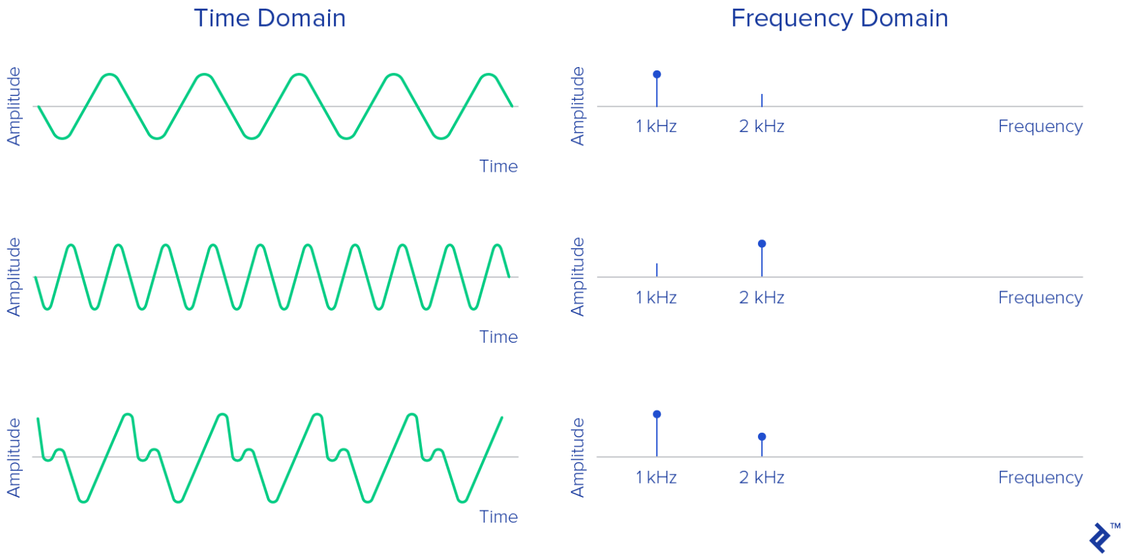
3261 0727 Control sends CQ "QRX 1430".
0742 Control sends a 2-Fig. message to 63 and 62 (WENN KRIEG AUS NACH DORT KOMME) and then sends greetings from FRIEDR. K.
0756 Control sends a 2-Fig.message to 62 from ROH to ENG (NACH ENDE KOMME NACH DORT ALBIN). ENG at 62 replies in 2-Fig. to ROH at Bussard. (HALT DIE OHREN STEIF JUNG(or EUER --E).
0801 63 replies in 2-Fig. (-U--AUSE)
0805 63 sends another 2-Fig.message (SSOENLKP-EN)
- - - -
1230 Control announces a CQ message.
1251 Control sends message 1 (TOO 1000) CQ from 'YHL'.
62 promises to phone it to 53 and 63.
1258 Control sends CQ "AUF WIEDERSEHEN - ES ENDE - DEUTSCHEN - NW QRX NIL FUER TOX (65) ABU (63) 4ZH (53) 4HX (54), 3X9(67) - QRX NIL - 9N - ALLES GUTE".
1302 67 appears and asks "MEINE CIGARETTEN-RAUCHEN". Control replies "ER CIG NIL".
1305 62 takes "QRX 0930".
1309 61 sends message 2 (TOO 1030 KR) to Control for 'YHL' and 'YUL' adding that the answer is urgent. Contrpl replies that 'YHL' is not there but promises to send the answer tomorrow, when he will be there. 61 takes "QRX "
1327 Control sends to ? "- EN3 KLUECK LEBE WOHL" followed by greetings to GOEHLER (GOEHLER was at Auster in February) from RICHT (Richter).

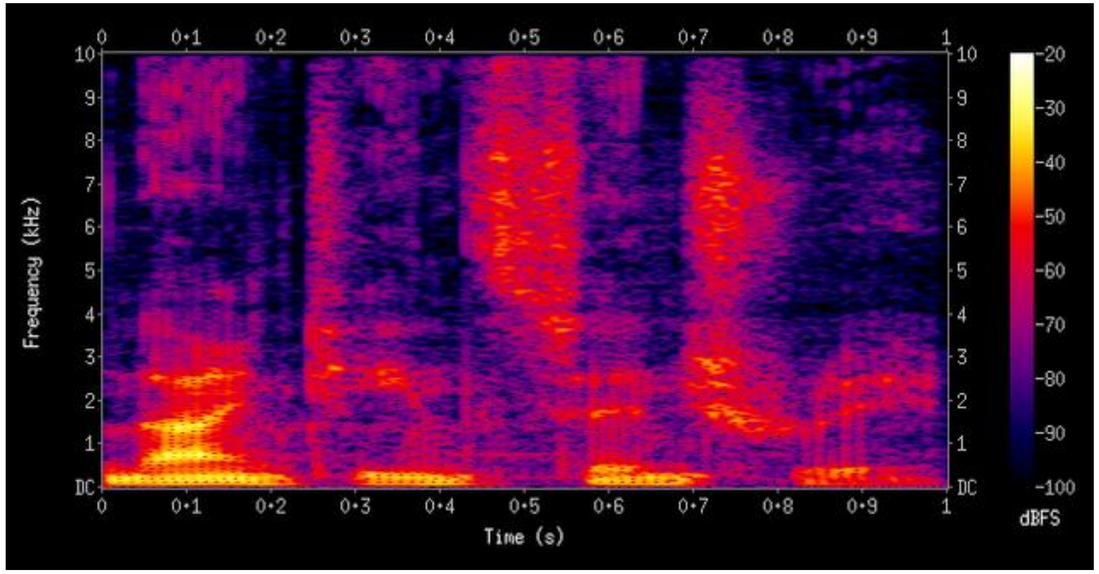
BALTIC GROUP

3404 1504 Bussard calls 51 and 52 without success.

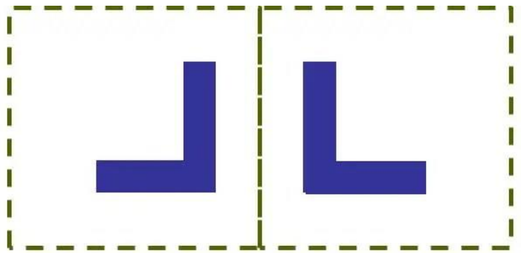
Why we need to learn about “Pattern matching”?



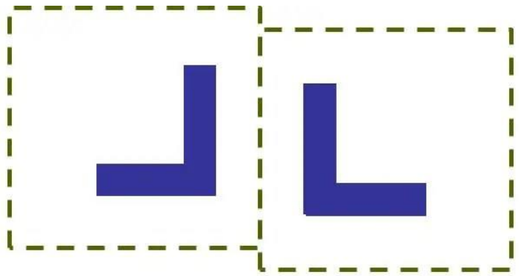




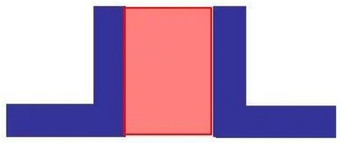
Why we need to learn about “Pattern matching”?



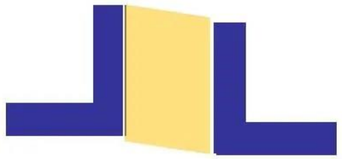
Pattern generates DRC error, but it is waived



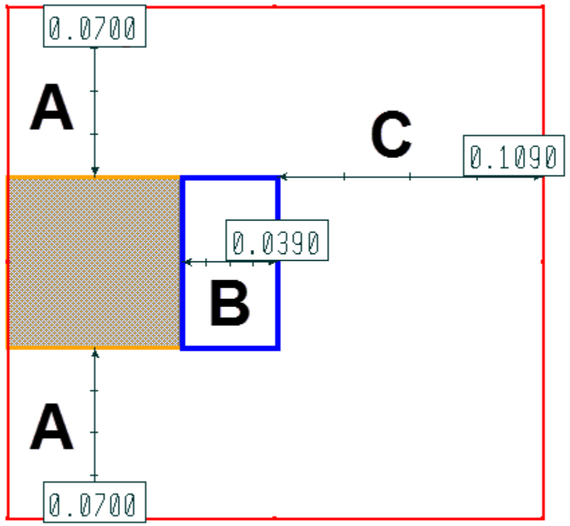
Pattern represents a real DRC error

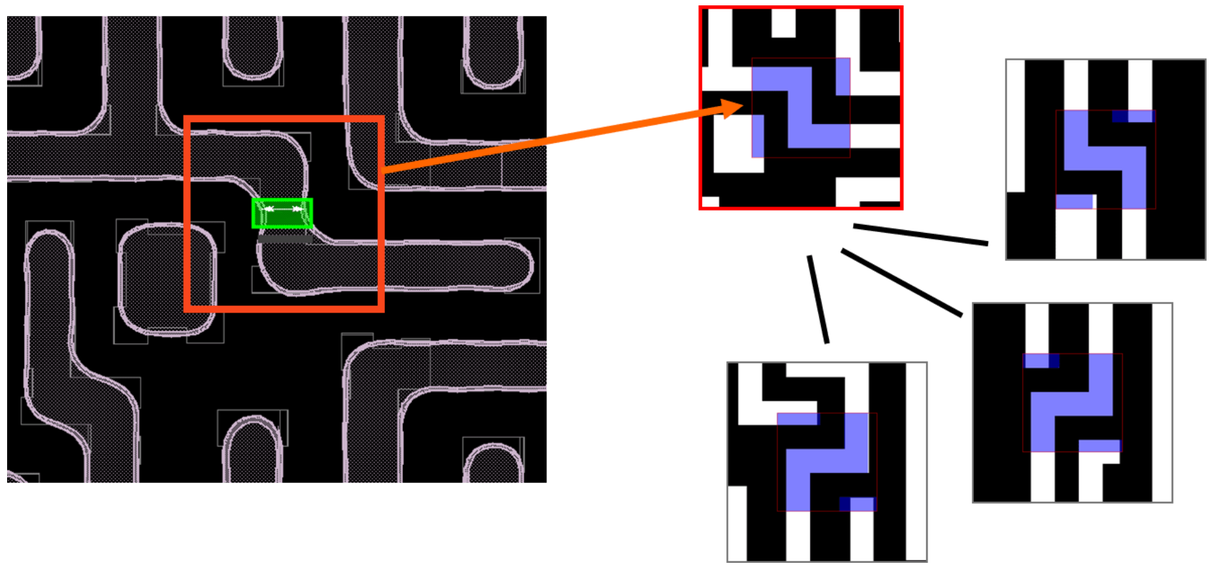


Blue = Pattern to match
Red = Auto-generated waiver shape



Doesn’t match the generated waiver shape





A Short History

- Brute Force (Naive) algorithm – early 20th century
- Knuth-Morris-Pratt algorithm – publish in 1976
- Boyer-Moore algorithm – 1977
- Aho-Corasick algorithm (1975)
- Rabin-Karp fingerprint search algorithm – 1980
- Wu-Manber algorithm (1992)

02.

Substring Search Algorithms

2.1 Brute – Force

2.2 Knuth – Morris – Pratt

2.3 Boyer – Moore

2.4 Rabin Karp

2.1

Brute – Force

2.1 Brute – Force

The Brute-Force algorithm is a naive string-matching algorithm that involves comparing each character of a given pattern to each character of a text one-by-one.

- ***Idea***:

A pointer (i) into the text(length = N) and a pointer (j) into the pattern (length = M)
For each i, resets j to 0 and increments j until finding a mismatch or the end of the pattern (j == M).
(j == M) => return i => the pattern occur in the text
If i reach the end of the text (i == N-M+1) and then there is no match.
=>Return N to indicate a mismatch => the pattern does not occur in the text

Example:

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

Pattern	A	B	R	A

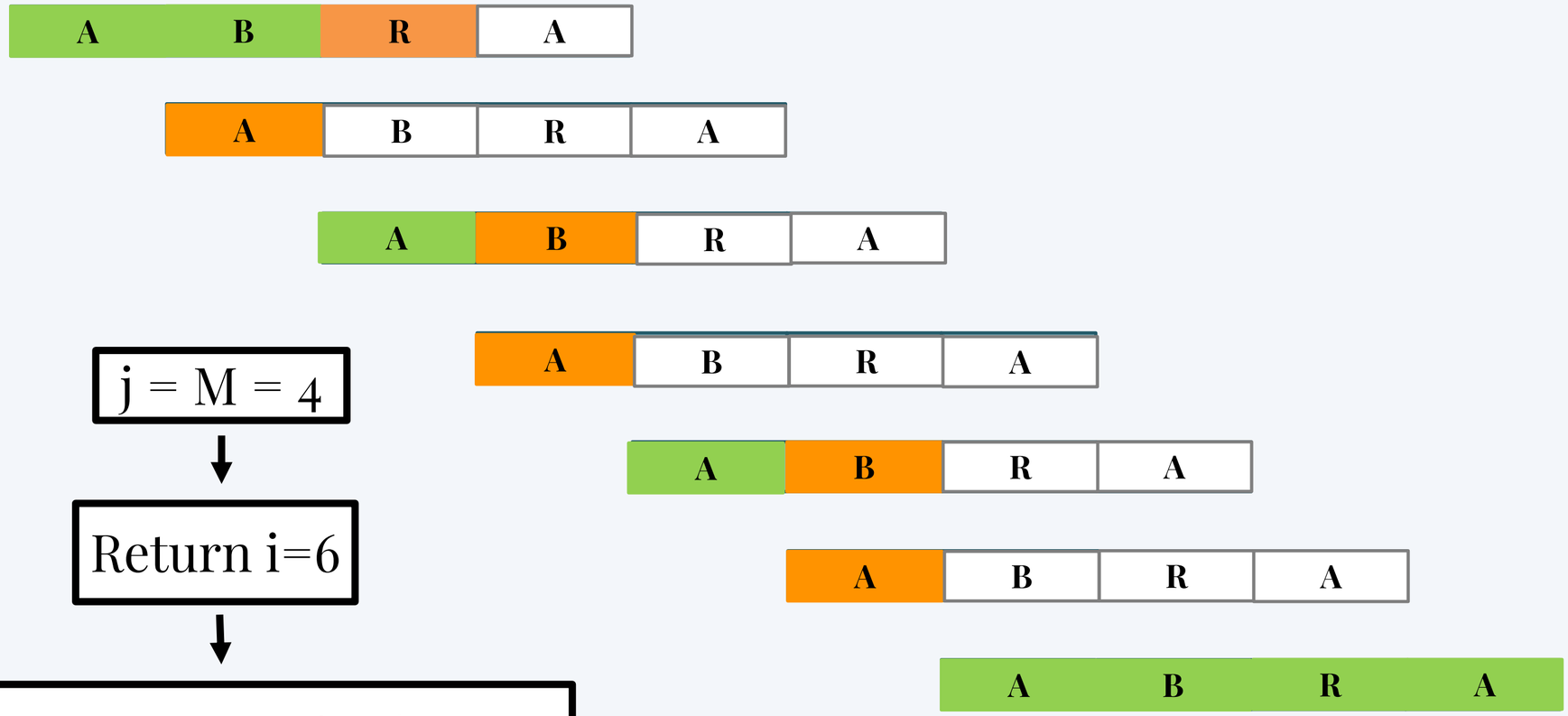
2.1 Brute – Force Visualization

N	M	N-M+1
10	4	7

i	j	i+j
0	2	2
1	0	1
2	1	3
3	0	3
4	1	5
5	0	5
6	4	10

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

Pattern



j = M = 4

Return i=6

the pattern occur in the text

2.1 Brute – Force Visualization

N	M	N-M+1
10	4	7

i	j	i+j
0	2	2
1	0	1
2	1	3
3	0	3
4	1	5
5	0	5
6	3	9
7		

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

Pattern: A B R A

A B R A

A B R A

A B R A

i=(N-M+1)
j !=(M = 4)

A B R A

A B R A

A B R A

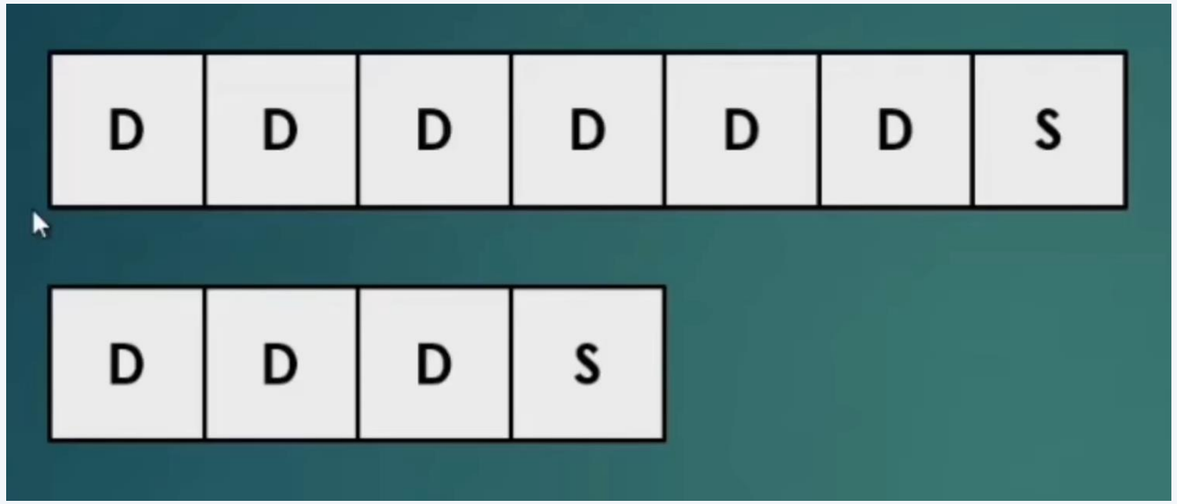
Return N=10

A B R A

the pattern does not occur in the text

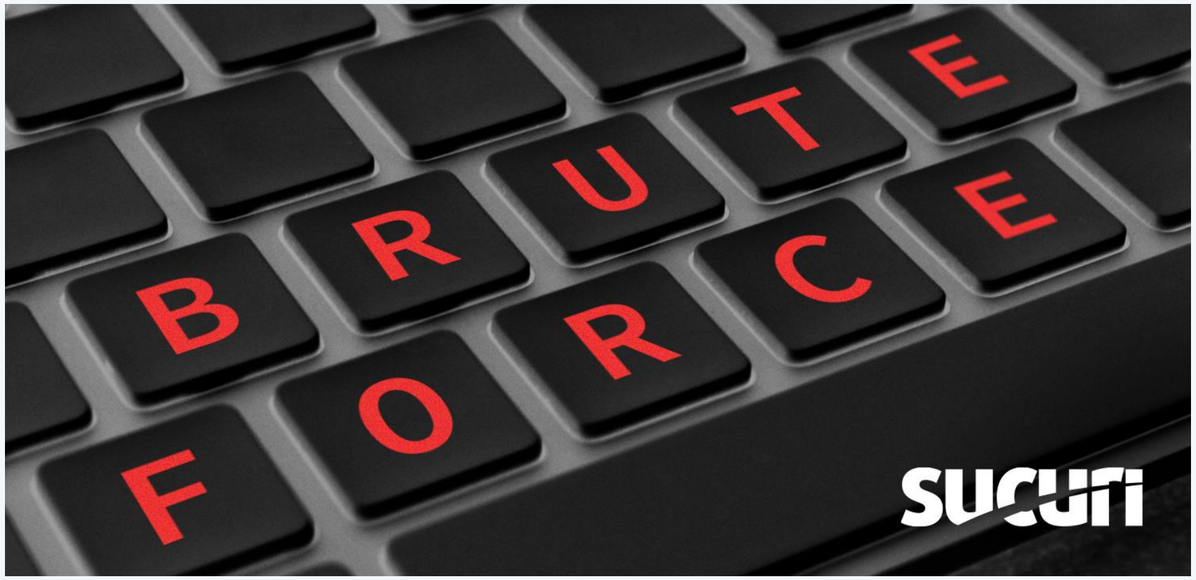
2.1 Brute – Force | Drawback

- Main problem: needs back up for every mismatch.
- Not so efficient especially when there are lots of matching prefixes.
  For example: pattern is DDDS and the text is DDDDDDS



- Lots of compares: ~N*M where N is length of text, M is length of the pattern.
  Linear time guarantee would be better.

2.1 Brute – Force



3gA8bjqI	m5MoKRJJ	0WZCBfwI	zm9jqUiN	glYqI53X	eokUQTMr	LDzoPv9e	qXUCvXrT	Enju6m2B
oiKq1eah	9WtsbkvW	W3gyedbV	12j9P5xy	8SxkUE9w	nXXADMm5	GiB1Z6l0	jio5wIbp	gozIsTIm
lzOWmgPt	i9XnarBa	yo0acST0	Q1acFGJq	IZkFcv0u	5ugNFvlw	w2Gl7xIN	w2bejXwY	RvYxtMjU
nRlO6HW5	cHRSQrAb	DoqQQrgd	RTUevfn2	faFLIMRf	Mm1nJoZE	JilIRGJ5	crEHftjc	eDmS5WJp
RfVugGsl	RgsGp0oU	tHd41yuV	L0x6wZkG	IdqZQtxK	TIBeHAVa	XYVA4UQI	VXoncM5C	uk3Eaauk
dB90qMas	DJrU178v	nCZrMpk8	Xaup9lw5	miufIFTB	Z9p5K9Ut	suAjQYep	x5CosAtw	bHTdQPkj
A1gr9Utx	Pik7Il04	vaYlGHjc	BtJ8ktAk	au0greB1	P9FTnI7c	a6UErOcr	iEp0z3tC	Hzg7iYWZ
6FFcHAoe	Yfa3SY5I	351sV8w5	J3PxPYNz	WGjLGuBW	c2503M6c	pDfsu2Q4	cdP5cwB5	9vFjEHQu
2MxkJa3i	B4bLGWH4	UIJcx0ns	IMT1fNa3	PASSWORD	mfviEj5x	EsKPneug	GKJUOutG	FK92JFQ3
rP1owpJr	Yr30oFJ5	GHcDJqvx	A3QA5Ye3	YbtwXwnn	NGJLCNL8	2vJsptvH	zCinx0EC	UN3j3pXC
vmjRD4iO	Q1kh5j6Y	5i6TSEaT	lId407YG	deYv9OSn	2nczWHh6	vFXjiFRI	4sDHxCZm	Qpe5zL30
4eggPjtZ	KRfuFRnU	VtQhz1v9	XV9DkP4x	S9mMEd5S	bXyfJTGK	NQxNSTOH	qfSCnY1M	WjJz8X2c
9rpYjpuU	ZS69eKWL	7iMwKrlo	mtCQSeYd	mmam9dn9	5ha4ddzy	o9KYUF5Y	fJAzwIdn	zzHoKGY1
DDGTFjZL	Yt7Fm3lV	zqJ8pdW1	7YcJfnB9	5oywq9fK	3sA0ewmA	zHJyTRNe	UupQOTkY	XJpvqp2B
q3LWOtcJ	4Pm6aoip	iE9NzJgc	ntSxFnxl	qW7eAqKE	1VmD6lf0	5uzTxti6	2gRsURBz	yxseYGgg
beCDYzD2	dcrV4A54	jaT6KoQH	MtEulhJ3	xt67N62g	zKQIfxdi	KbBFpDhY	Qd5PGAPW	csfwIRrf
UAidOMw8	ZDqQ2xZq	QIJfG6Se	prhomHT2	scLAHNAS	NmIQ0UFQ	7TqPhSZP	kp7MUZMk	WSKd1TOU

2.2

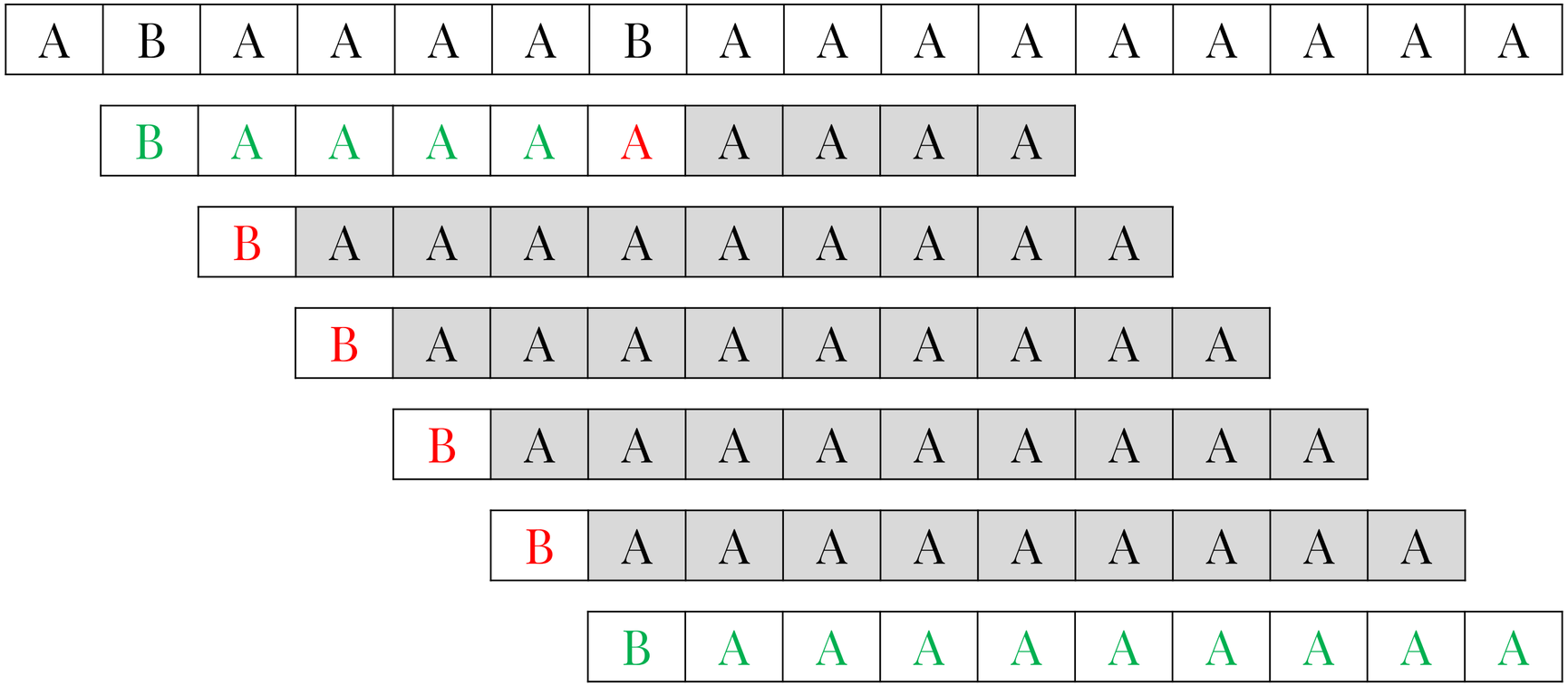
Knuth-Morris-Pratt

Dr. Donald Ervin Knuth

Dr. Vaughan Pratt

Dr. James Hiram Morris

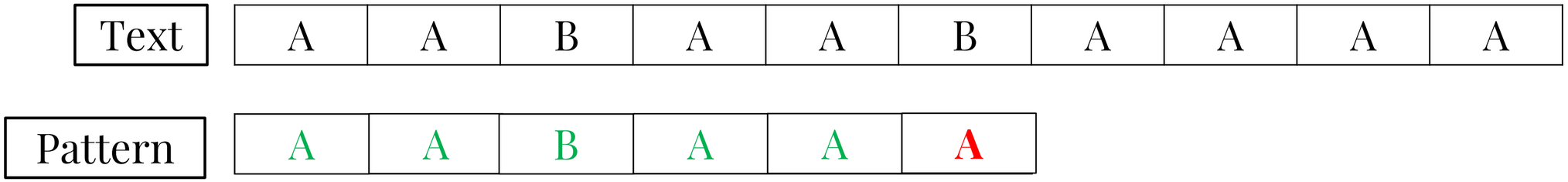
The KMP Algorithm



Idea:

- Whenever we detect a mismatch, we already know some of the characters in the text
- We can take advantage of this information to avoid backing up the text pointer over all those known characters.

The KMP Algorithm

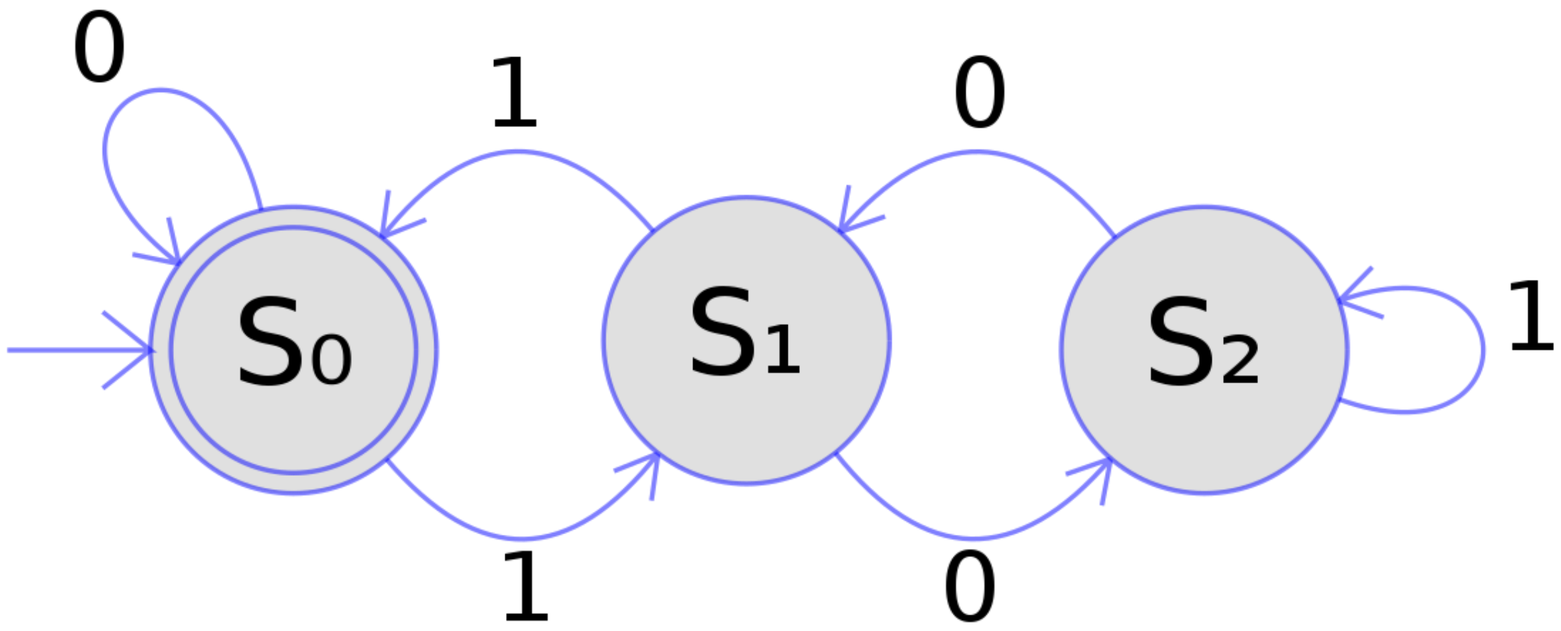


Not found

→ Fully skipping past all the matched characters when detecting a mismatch will not work when the pattern could match itself at any position overlapping the point of the mismatch.

The KMP Algorithm

- We can use a DFA (deterministic finite-state automaton) to simulate the process.



- So, what exactly does the DFA do?

The KMP Algorithm

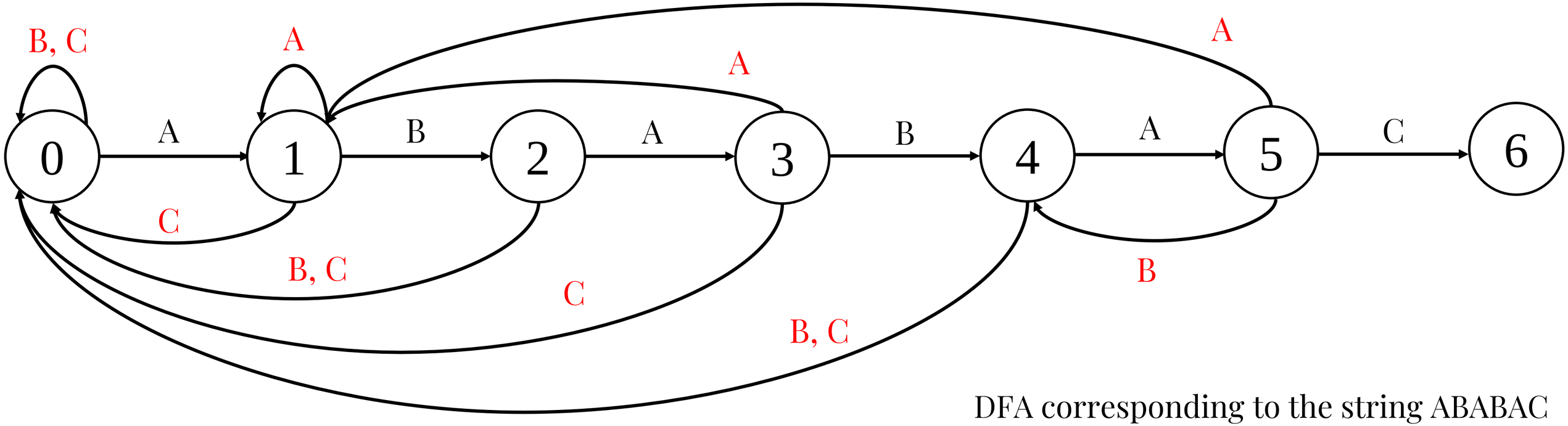
➢ DFA simulation:

Internal representation

j	0	1	2	3	4	5
Pat.char(j)	A	B	A	B	A	C

dfa[][j]							
	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Graphical representation



DFA corresponding to the string ABABAC

The KMP Algorithm

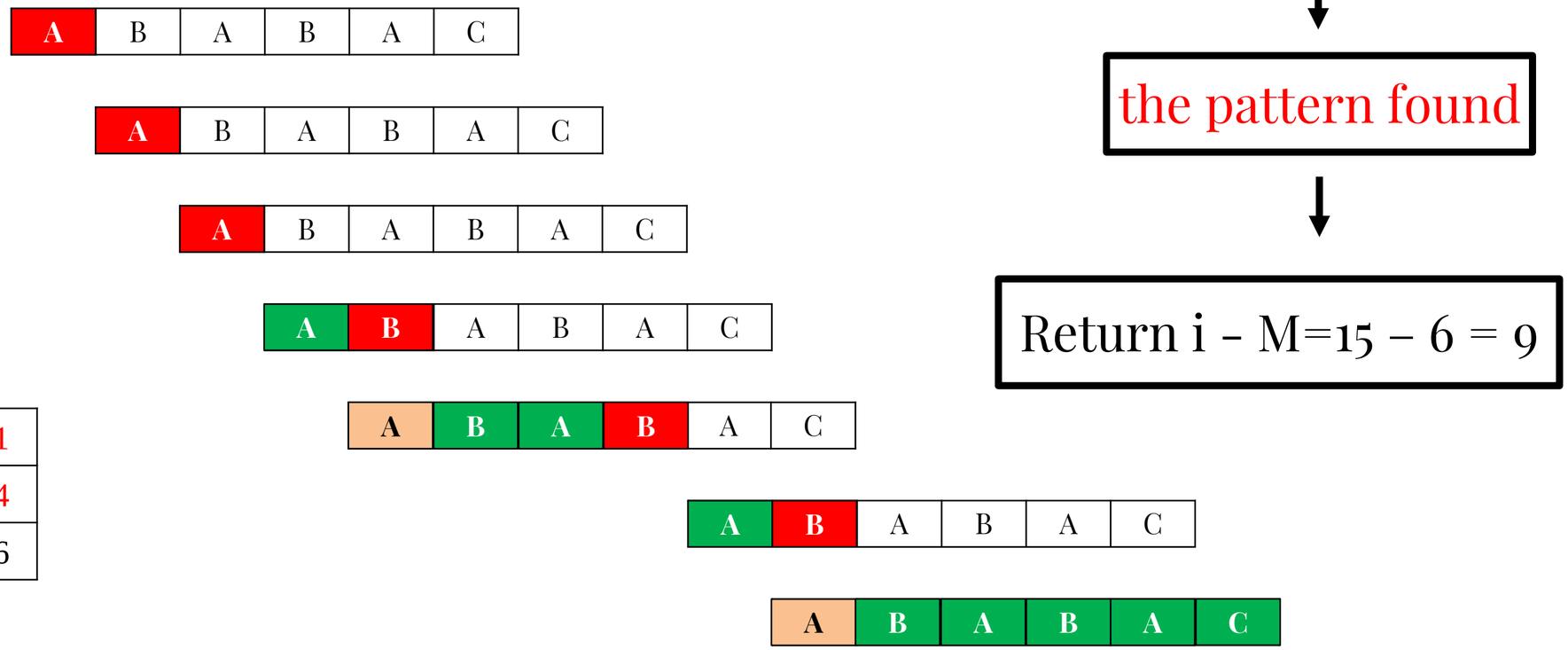
➢ DFA simulation:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Txt.charAt(i)	B	C	B	A	A	B	A	C	A	A	B	A	B	A	C	A	A
	0	0	0	0	1	1	2	3	0	1	1	2	3	4	5	6	

Mismatch:
Set j to dfa[txt.CharAt(i)] [j]
implies pattern shift to align
pat.CharAt(j) with txt.charAt(i+1)

Match:
Set j to dfa[txt.charAt(i)][j]
=dfa[pat.charAt(i)][j]
= j+1

dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



the pattern found

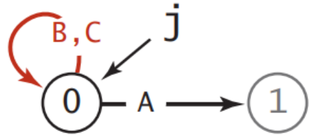
Return i - M=15 – 6 = 9

Trace of KMP substring search (DFA simulation) for ABABAC

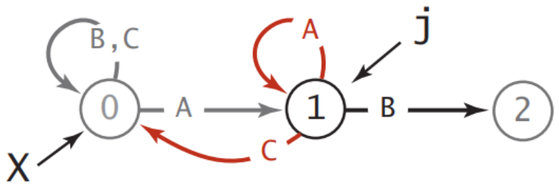
The KMP Algorithm

- How to construct the DFA?

		j = 0
pat.charAt(j)		A
dfa[][j]	A	1
	B	0
	C	0



		X	j = 1
pat.charAt(j)		A	B
dfa[][j]	A	1	1
	B	0	2
	C	0	0



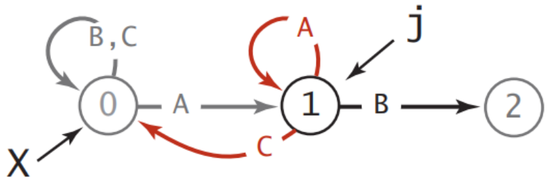
copy `dfa[][X]` *to* `dfa[][j]`

```
dfa[pat.charAt(j)][j] = j+1;
X = dfa[pat.charAt(j)][X];
```

The KMP Algorithm

- How to construct the DFA?

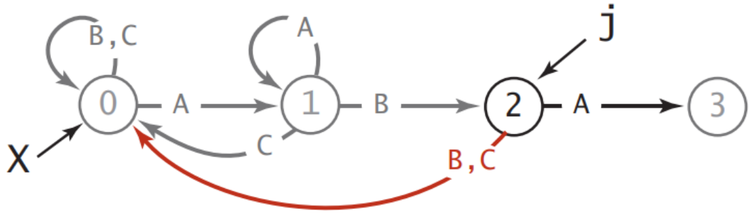
		X	
	j	0	1
pat.charAt(j)		A	B
dfa[][j]	A	1	1
	B	0	2
	C	0	0



copy `dfa[][X]` *to* `dfa[][j]`

```
dfa[pat.charAt(j)][j] = j+1;
X = dfa[pat.charAt(j)][X];
```

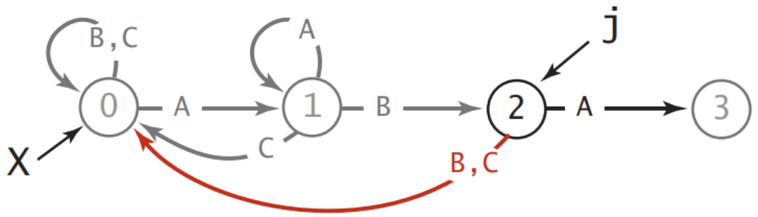
		X		
	j	0	1	2
pat.charAt(j)		A	B	A
dfa[][j]	A	1	1	3
	B	0	2	0
	C	0	0	0



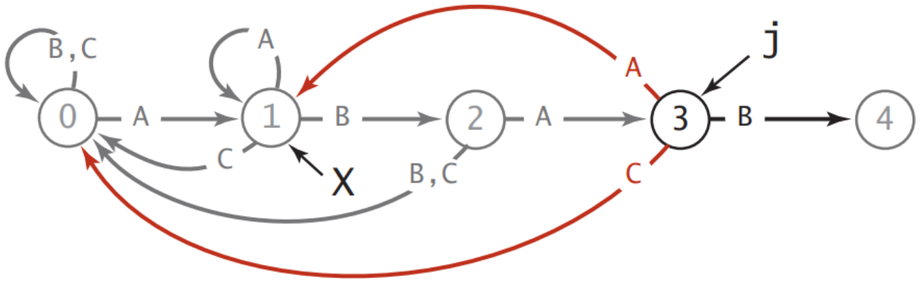
The KMP Algorithm

- How to construct the DFA?

		X		
	j	0	1	2
pat.charAt(j)		A	B	A
dfa[][j]	A	1	1	3
	B	0	2	0
	C	0	0	0



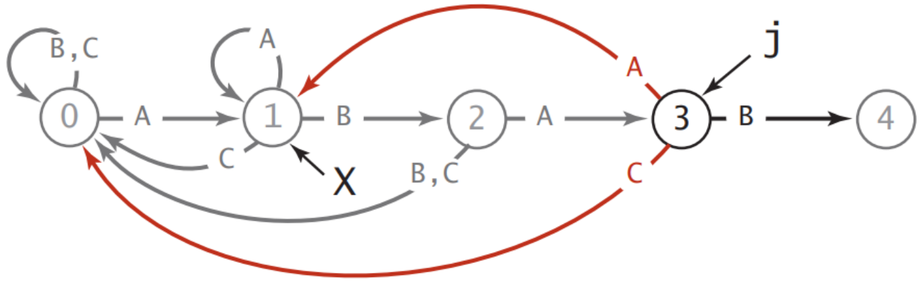
			X		
	j	0	1	2	3
pat.charAt(j)		A	B	A	B
dfa[][j]	A	1	1	3	1
	B	0	2	0	4
	C	0	0	0	0



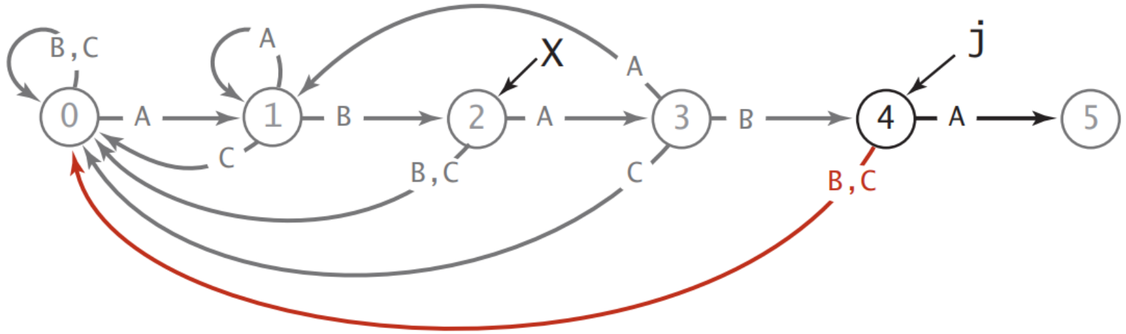
The KMP Algorithm

- How to construct the DFA?

		X			
	j	0	1	2	3
pat.charAt(j)		A	B	A	B
dfa[][j]	A	1	1	3	1
	B	0	2	0	4
	C	0	0	0	0



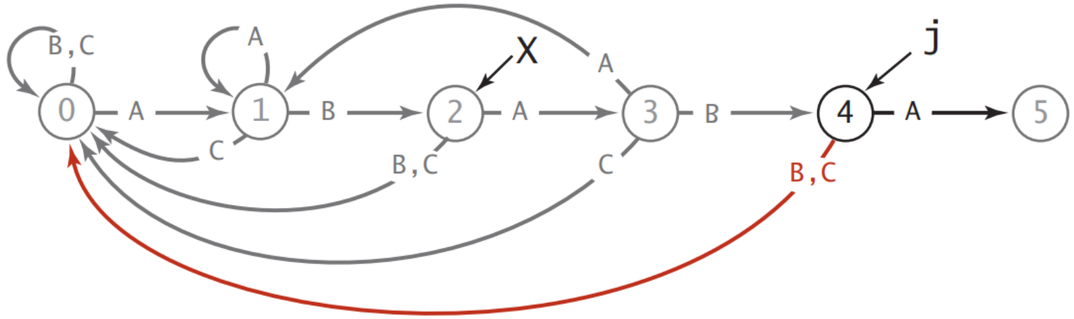
			X			
	j	0	1	2	3	4
pat.charAt(j)		A	B	A	B	A
dfa[][j]	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	0



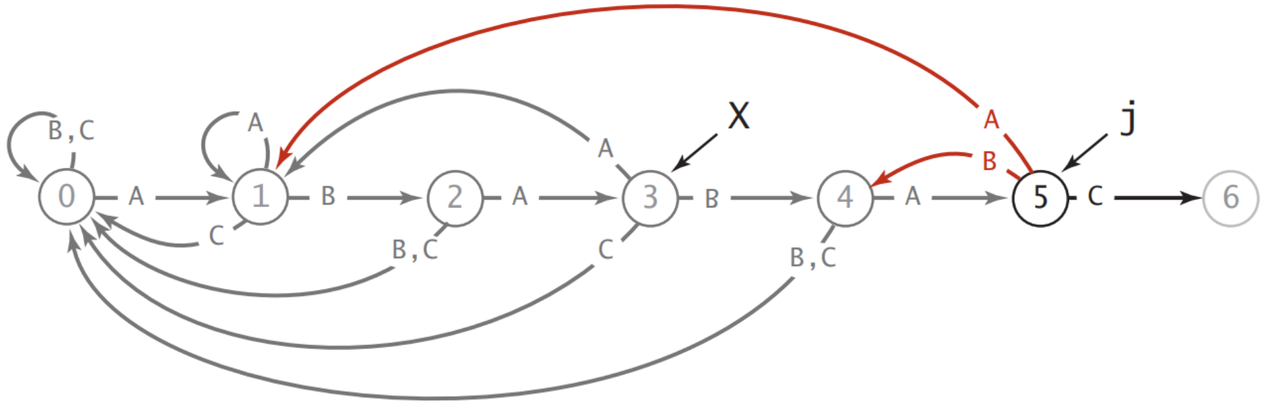
The KMP Algorithm

- How to construct the DFA?

	j	0	1	2	3	4
	pat.charAt(j)	A	B	A	B	A
dfa[][j]	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	0



	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



The KMP Algorithm

Pros

- It is guaranteed worst-case efficient, the preprocessing time is O(m) and the searching time is O(n).
- The algorithm never needs to move backwards in the input text.

Cons

- Time and space for pre-processing stage (use extra space)
- Complexity
- Limited use

2.3

Boyer-Moore

Dr. Robert Stephen Boyer (left) & Dr. J Strother Moore (right)

The Boyer-Moore Algorithm

The Boyer-Moore's pattern matching algorithm is based on two heuristics

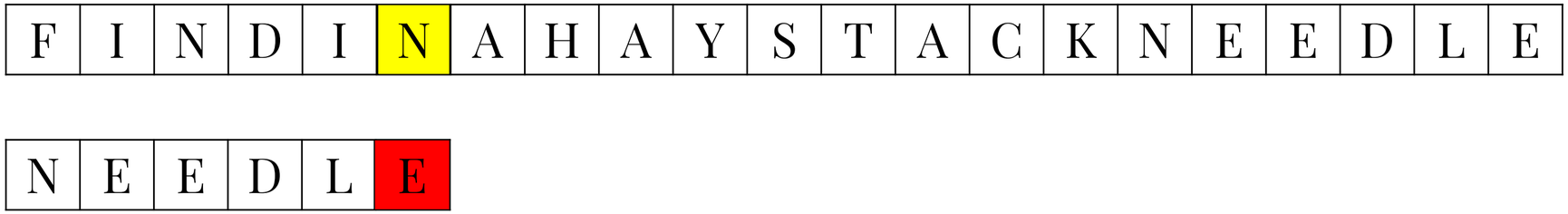
- Looking-glass heuristic: Scanning the pattern from right to left.
- Character-jump heuristic: Skipping as many characters as possible by using information from the pattern itself.

Example:

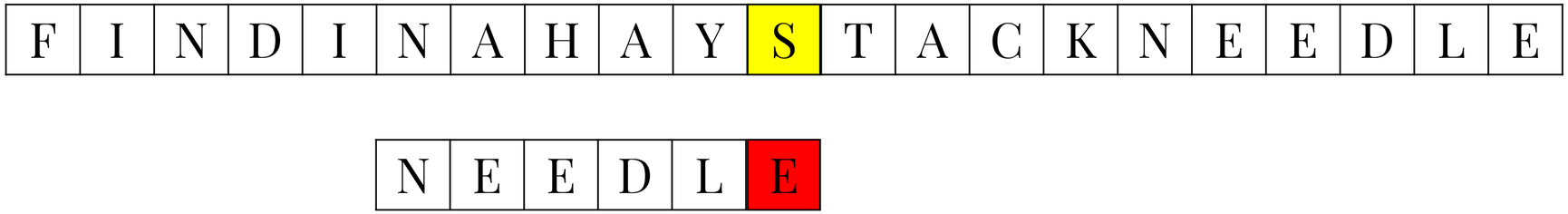
Text:	F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E

Pattern:	N	E	E	D	L	E

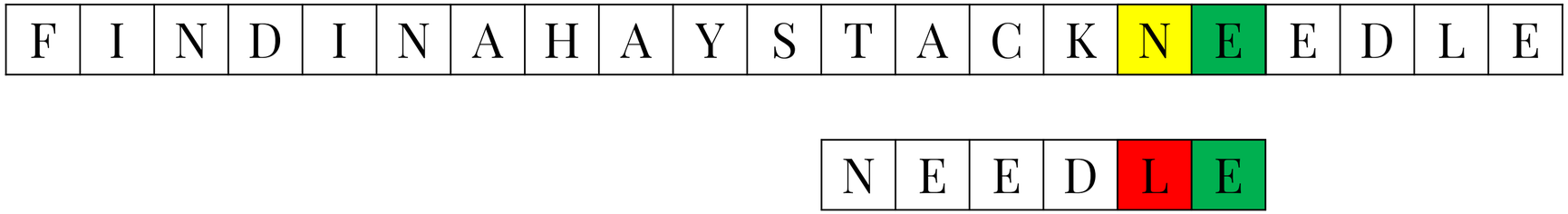
The Boyer-Moore Algorithm



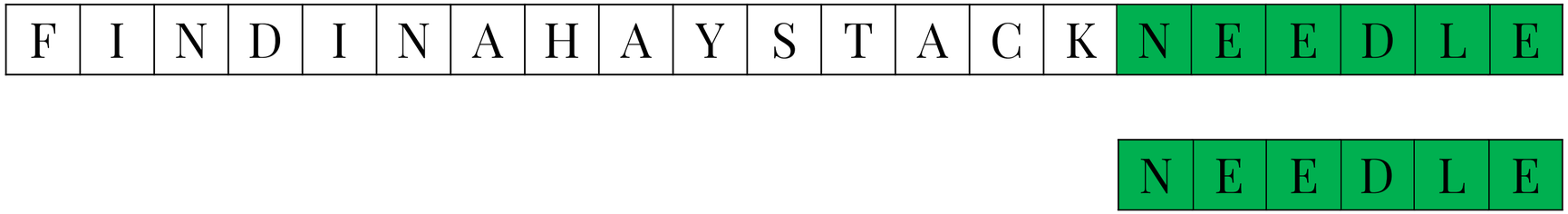
The Boyer-Moore Algorithm



The Boyer-Moore Algorithm



The Boyer-Moore Algorithm



The Boyer-Moore Algorithm

- Boyer-Moore skip table computation

		N	E	E	D	L	E

c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
…								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
…								-1

The Boyer-Moore Algorithm

- *Substring search (in Boyer-Moore algorithm)*:
  +) If txt.charAt(i + j) is equal to pat.charAt(j) for all j from M – 1 down to o, then there is a match.
  +) If there is a mismatch:
    - If the character causing the mismatch is not found in the pattern, slide the pattern j + 1 positions to the right
    - If the character c causing the mismatch is found in the pattern, use right[] array to line up the pattern with the text, so that the character will match its rightmost occurrence in the pattern (i += (j – right[c]))
    - If this computation would not increase i, we just increment i instead.

The Boyer-Moore Algorithm

Pros

+) Efficient: works particularly well for long search patterns, and it can be sub-linear

+) Considered the most efficient string-matching algorithm in usual application, such as, in text editors and command substitutions.

Cons

+) Not as good for binary strings or for very short patterns.

+) Worst-case time complexity: can be O(M*N) in certain cases.

2.4

Rabin-Karp

Dr. Michael Oser Rabin

Dr. Richard Manning Karp

The Rabin-Karp Algorithm

- The method is developed by M.O.Rabin and R.A.Karp:
- Approach: Based on hashing
- How it works?:
  - If we find a text substring with the same hash value as the pattern → check for a match
  - Do a search for each substring of the text till finding → If not found, we can lead to the conclusion that the pattern does not exist in the text

Example:

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

The Rabin-Karp Algorithm

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

% 997 = 613

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

0	3	1	4	1	5

% 997 = 508

The Rabin-Karp Algorithm

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

% 997 = 613

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

1	1	4	1	5	9

% 997 = 201

The Rabin-Karp Algorithm

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

% 997 = 613

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

2					
	4	1	5	9	2

% 997 = 715

The Rabin-Karp Algorithm

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

% 997 = 613

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

3		1	5	9	2	6

% 997 = 971

The Rabin-Karp Algorithm

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

% 997 = 613

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

4

5	9	2	6	5

% 997 = 442

The Rabin-Karp Algorithm

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

% 997 = 613

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

5					
	9	2	6	5	3

% 997 = 929

The Rabin-Karp Algorithm

pat.charAt(j)					
j	0	1	2	3	4
	2	6	5	3	5

% 997 = 613

txt.charAt(j)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

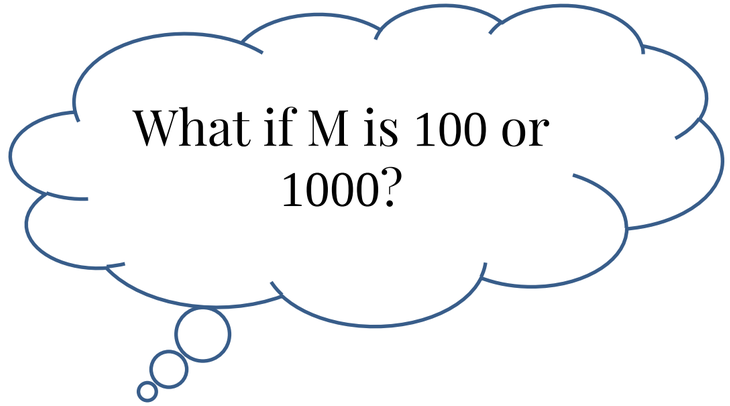
6

2	6	5	3	5

% 997 = 613 → Pattern found!!!

The Rabin-Karp Algorithm

- *Notation*:
  +) A string of length M corresponds to an M-digit base-R number
  +) Use a hash table of size Q. In practice, we use a random prime Q.



Use Horner's method, which computes the hash function for an M-digit base-R number represented as a char array in time proportional to M.

The Rabin-Karp Algorithm

pat.charAt(j)

i	0	1	2	3	4
	2	6	5	3	5

0	2	% 997 = 2				
1	2	6	% 997 = (2 * 10 + 6) % 997 = 26			
2	2	6	5	% 997 = (26 * 10 + 5) % 997 = 265		
3	2	6	5	3	% 997 = (265 * 10 + 3) % 997 = 659	
4	2	6	5	3	5	% 997 = (265 * 10 + 3) % 997 = 659

Computing the hash value for the pattern with Horner’s method

The Rabin-Karp Algorithm

- *Key idea*:
- The Rabin-Karp method is based on efficiently computing the hash function for position i+1 in the text, given its value for position i:

  +) Using the notation t_i for txt.charAt(i), the number corresponding to the M-character substring of txt that starts at position i is

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0$$

  +) Shifting 1 position right in the text corresponds to replacing x_i by

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

The Rabin-Karp Algorithm

- *A trick Monte-Carlo correctness*:
  +) Do not do the test for a true match but rather make the hash table “size” Q as large as possible.
  +) If a long value greater than 10^{20}, making the probability of collision less than 10^{-20}

→ Guarantees completion time but has a small probability of outputting an incorrect answer

- *Las Vegas version*:
  +) Add a check to check that text matches the pattern or not

→ always output a correct answer but may have variable (slower) running time.

The Rabin-Karp Algorithm

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

0	3	% 997 = 3

1	3	1	% 997 = (3 * 10 + 1) % 997 = 31

2	3	1	4	% 997 = (31 * 10 + 4) % 997 = 314

3	3	1	4	1	% 997 = (314 * 10 + 1) % 997 = 150

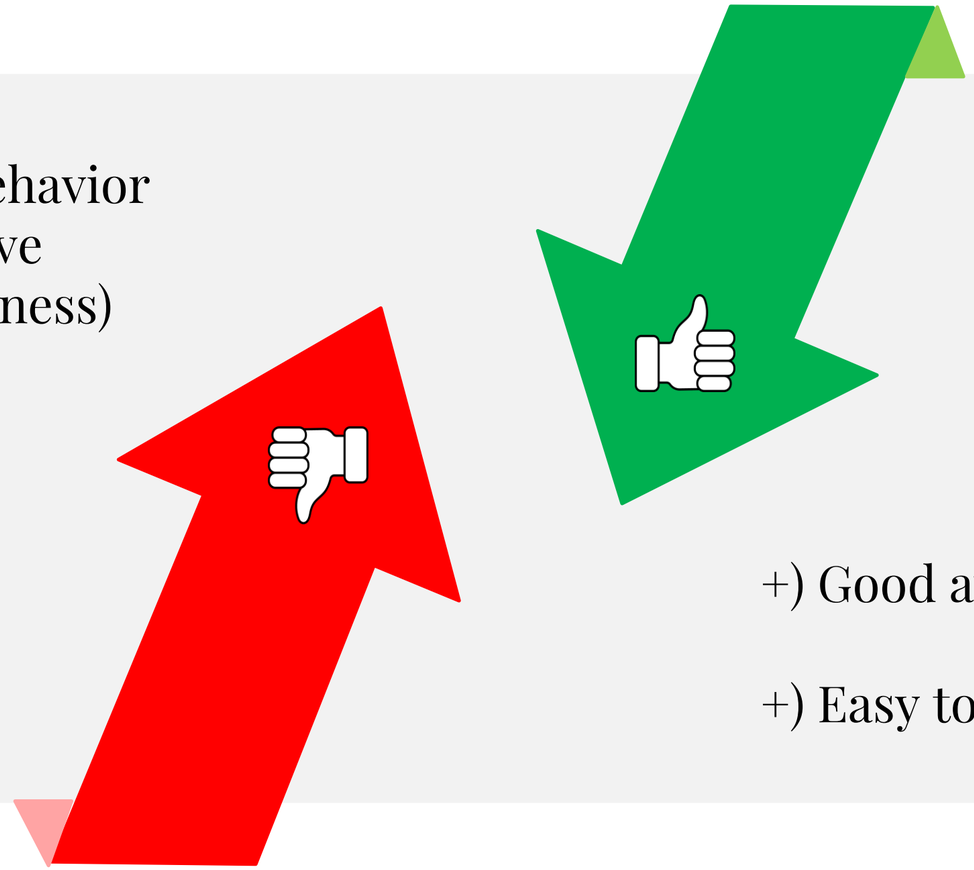
4	3	1	4	1	5	% 997 = (3141 * 10 + 5) % 997 = 508

The Rabin-Karp Algorithm

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

5	1	4	1	5	9	% 997 = ((508 + 3 * (997 – 30)) * 10 + 9) % 997 = 201	
6	4	1	5	9	2	% 997 = ((201 + 1 * (997 – 30)) * 10 + 2) % 997 = 715	
7	1	5	9	2	6	% 997 = ((715 + 4 * (997 – 30)) * 10 + 6) % 997 = 971	
8	5	9	2	6	5	% 997 = ((971 + 1 * (997 – 30)) * 10 + 5) % 997 = 442	
9	9	2	6	5	3	% 997 = ((442 + 5 * (997 – 30)) * 10 + 3) % 997 = 929	
10	2	6	5	3	5	% 997 = ((929 + 9 * (997 – 30)) * 10 + 5) % 997 = **613**	

The Rabin-Karp Algorithm



Pros

+) Good average-case performance: O(M + N)

+) Easy to implement

Cons

+) Slow worst-case behavior

+) Can be false-positive (Monte-Carlo correctness)

Cost summary for substring search implementations

Algorithm	Version	Operation count		Backup in input?	Correct?	Extra space
		Guarantee	Typical			
Brute Force	_ _	MN	1.1N	yes	yes	1
Knuth-Morris-Pratt	full DFA	2N	1.1N	no	yes	MR
	mismatch transitions only	3N	1.1N	no	yes	M
Boyer-Moore	full algorithm	3N	N/M	yes	yes	R
	mismatched char heuristic only	MN	N/M	yes	yes	R
Rabin-Karp	Monte Carlo	7N	7N	no	yes (*)	1
	Las Vegas	7N(*)	7N	yes	yes	1

*: probabilistic guarantee, with uniform and independent hash function

Comparison

Algorithm	Preprocessing Time	Matching Time	Best Case Time	Worst Case Time	Space Required
Brute-Force	O(1)	O(mn)	O(n)	O(nm)	O(1)
Knuth-Morris-Pratt (KMP)	O(m)	O(n)	O(m+n)	O(m+n)	O(m)
Boyer-Moore	O(m + k)	O(n)	O(n/m)	O(mn)	O(m + k)
Rabin-Karp	O(m)	O(n)	O(n+m)	O(nm)	O(1)

Regular Expression (Optional)

Describing pattern with Regex

Definition. Each RE represents a set of strings, defined as follows:

The RE	Representation
Empty RE	*empty* set of strings, with 0 elements
A character	set of strings with one element, itself.
RE enclosed in parentheses	the same set of strings as the RE without the parentheses
RE consisting of two *concatenated* REs	*cross product* of the sets of strings represented by the individual components
RE consisting of the *or* of two Res (|)	the *union* of the sets represented by the individual components
RE consisting of the *closure* of an RE (*)	(the empty string) or the union of the sets represented by the concatenation of any number of copies of the RE

Describing pattern with Regex

Name or meaning	Notation	Example
Concatenation		A B
Or	|	A|B
Closure	*	A*
Parentheses	Enclosed in ()	(A|B)*C
Wildcard	.	A.B
Specified set	Enclosed []	[ARIOU]*
Range	Enclosed in [] separated by -	[A-Z] [0-9]
Complement	Enclosed in [] preceded by ^	[^AEIOU]*
At least 1	+	(AB)+
0 or 1	?	(AB)?
Specific number of copies	Count or range in {}	(AB){3} (AB){1-2}

Describing pattern with Regex

Basic Rules:

Concatenation.
A B => specifying the language { A B } that has one two-character string, formed by concatenating A and B.

Or
an or (|) between two alternatives, then both are in the language
For example: A | B specifies the language {A,B}
***Concatenation has higher precedence than *or*
=> A B | B C D specifies the language { A B , B C D }

Closure
allows parts of the pattern to be repeated arbitrarily, formed by concatenating the pattern with itself any number of times (including zero).
For example: A* specifies the language {∈, A , AA , AAA , ...}
A*B specifies the language {B, AB ,AAB ,AAAB, ...}
AB* specifies the language {A, AB ,ABB ,ABBB, ...}

Parentheses.
use parentheses to override the default precedence rules.
For example: C(AC|B)D specifies the language { CACD, CBD}

Word	Reference
language specifically	a set of strings
pattern	language specification
∈	empty string

Example:

RE	Matchs	Does not match
(A|B)(C|D)	AC AD BC BD	Every other string
A(B|C)*D	AD ABD ACD ABCCBD	BCD, ADD, ABCBC
A*|(A*BA*BA*) *	AAA BBAABB BABAAA	ABA, BBB, BABBAAA

Describing pattern with Regex

Shortcuts :

Set-of-characters descriptors:
directly specify sets of characters

Notations		Representation
dot character (.) is a wildcard		any single character
Sequence of characters within	square brackets	any of those characters
	preceded by a ^ , square brackets	any character but one of those characters

Name	Notation	Example
wildcard	.	A.B
Specified set	enclosed in [ ]	[AEIOU]*
range	enclosed in [ ] separated by -	[A-Z] [0-9]
complement	enclosed in [ ] preceded by ^	[^AEIOU]*

Escape sequences.
Some characters, such as \, ., |, *, (,), are metacharacters used to form regular expressions.
Escape sequences begin with a backslash character \ separating metacharacters from characters in the alphabet.
An escape sequence may be a \ followed by a single metacharacter (which represents that character).
For example, \\ represents \.
Other escape sequences represent special characters and whitespace.
For example: \t => a tab character, \n => a newline, and \s => any whitespace character.

Describing pattern with Regex

Shortcuts :

Closure shortcuts

Closure operator specifies anynumber of copies of its operand.

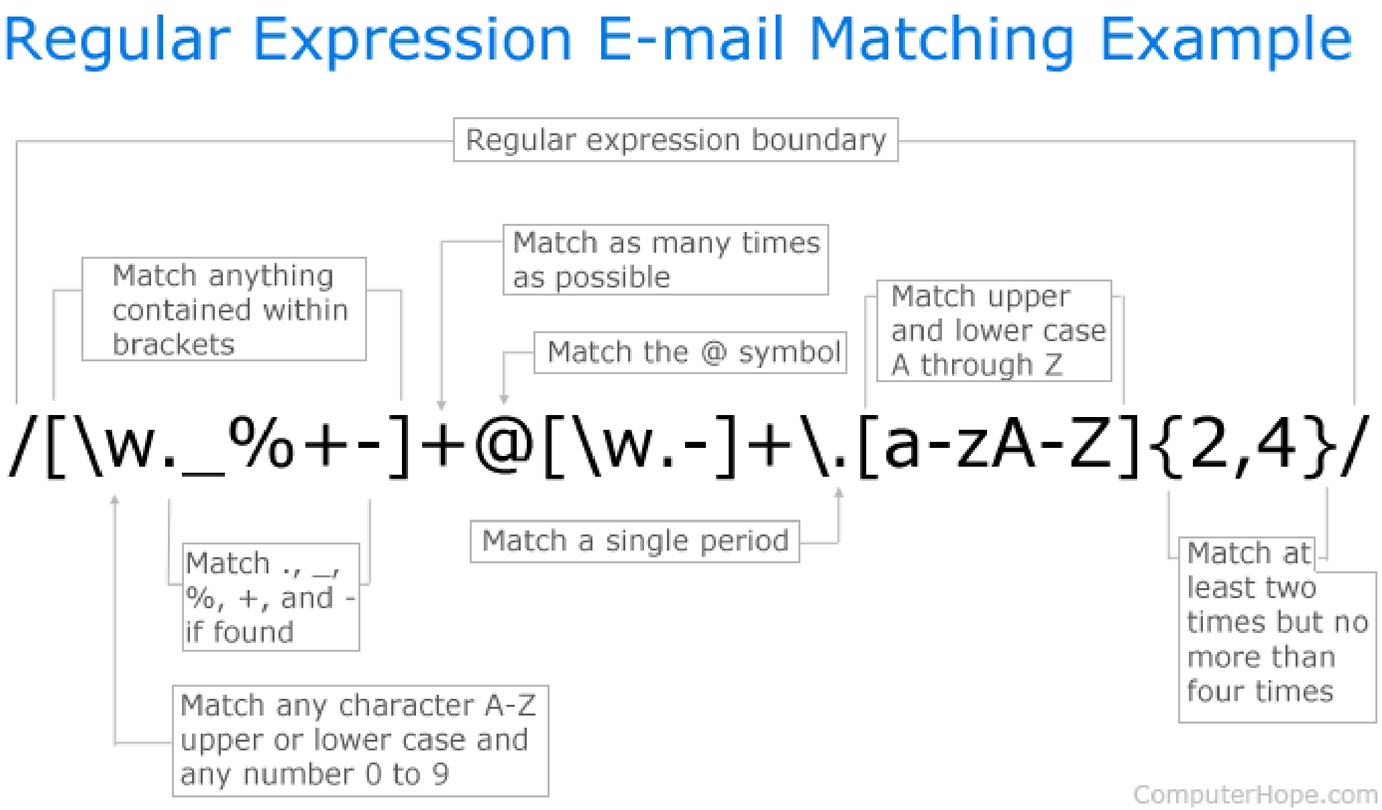
Notations	Representation
plus sign (+)	at least one copy
question mark (?)	zero or one copy
a count or a range within braces ({})	a given number of copies

Example:

Option	Notation	Example	Shortcut for	In language	Not in language
At least 1	+	(AB)+	(AB)(AB)*	AB ABABAB	∈ BBBAAA
0 or 1	?	(AB)?	∈ | AB	∈ AB	any other string
specific	count in { }	(AB){3}	(AB)(AB)(AB)	ABABAB	any other string
range	range in { }	(AB){1-2)	(AB) | (AB)(AB)	AB ABAB	any other string

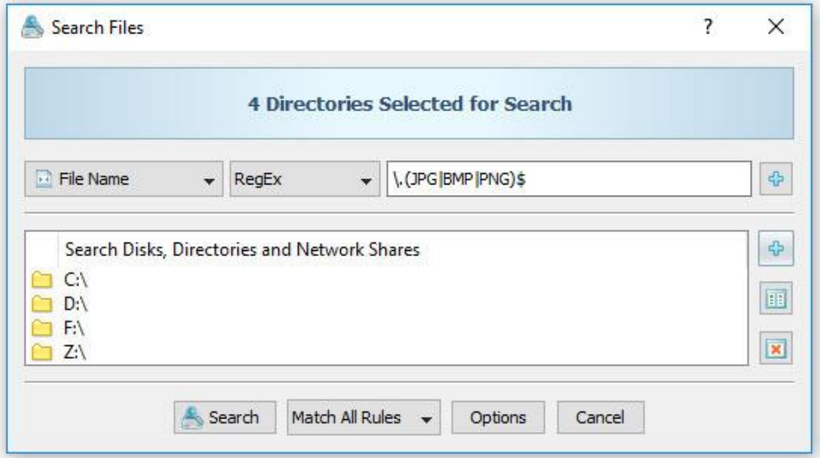
REs in applications

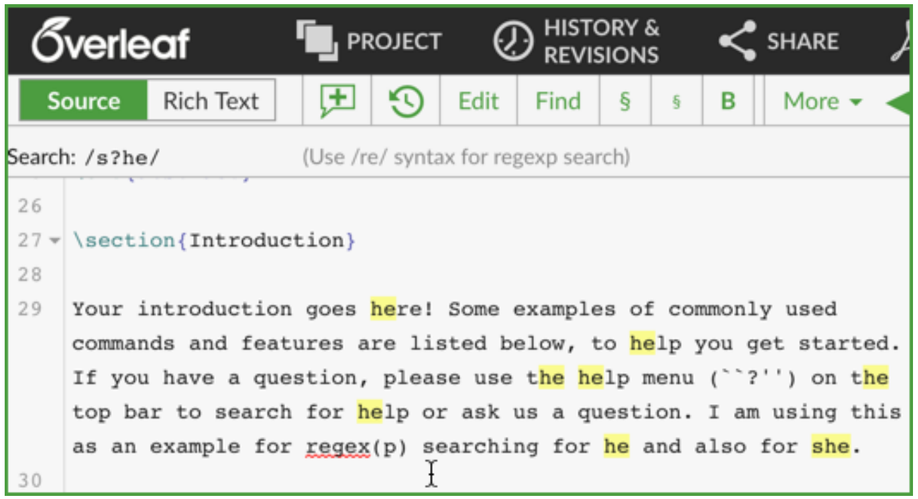
- Search

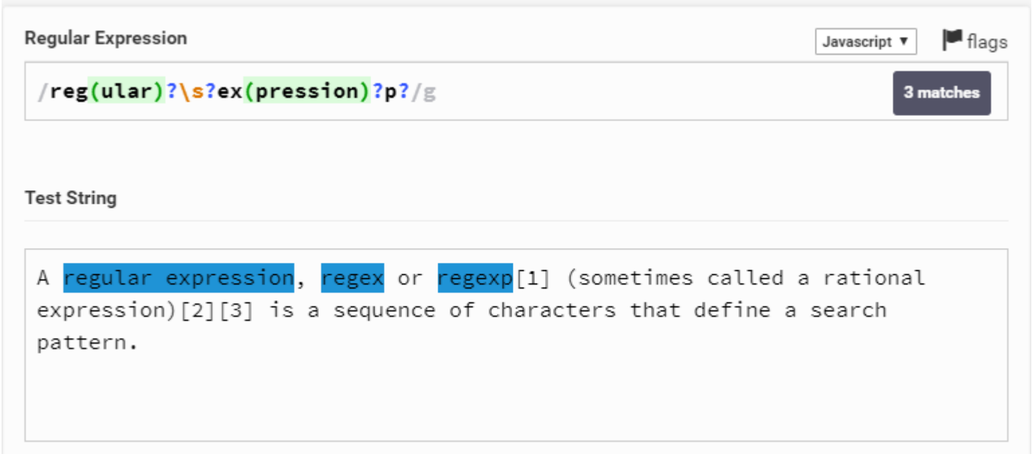


REs in applications

- Search







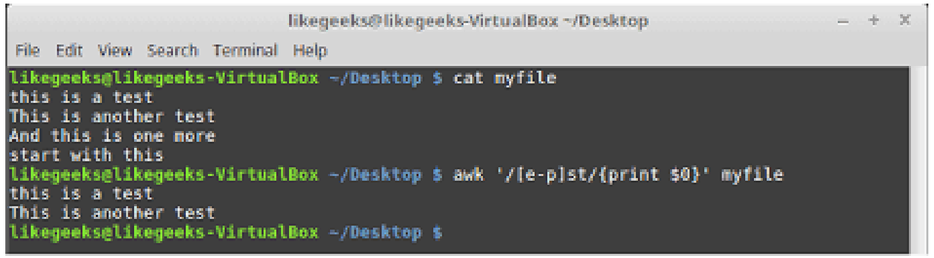
REs in applications

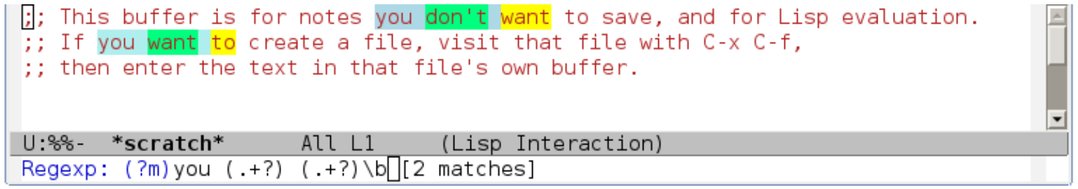
- Validity checking

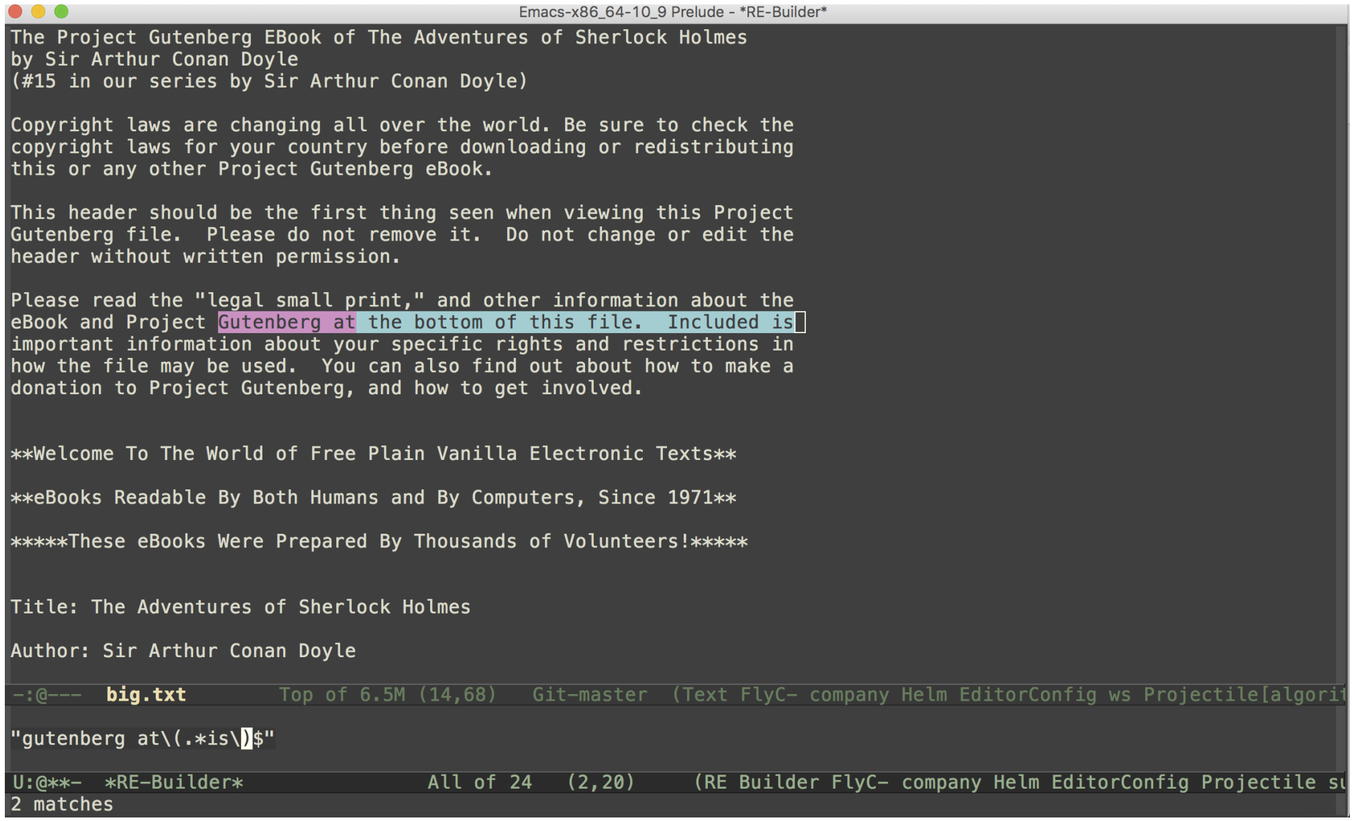
Context	Regular expression	Matches
Substring search	.*NEEDLE.*	A HAYSTACK NEEDLE IN
Phone number	\([0-9]{3}\)\ [0-9]{3}-[0-9]{4}	(800) 867-5309
Java identifier	[$_A-Za-z][$_A-Za-z0-9]*	Pattern_matching
Genome marker	gcg(cgg|agg)*ctg	gcgaggaggcggcggctg
Email address	[a-z]+@([a-z]+\.) + (edu|com)	rs@cs.princeton.edu
Typical regular expression in applications (simplified versions)		

REs in applications

- Programmer's toolbox

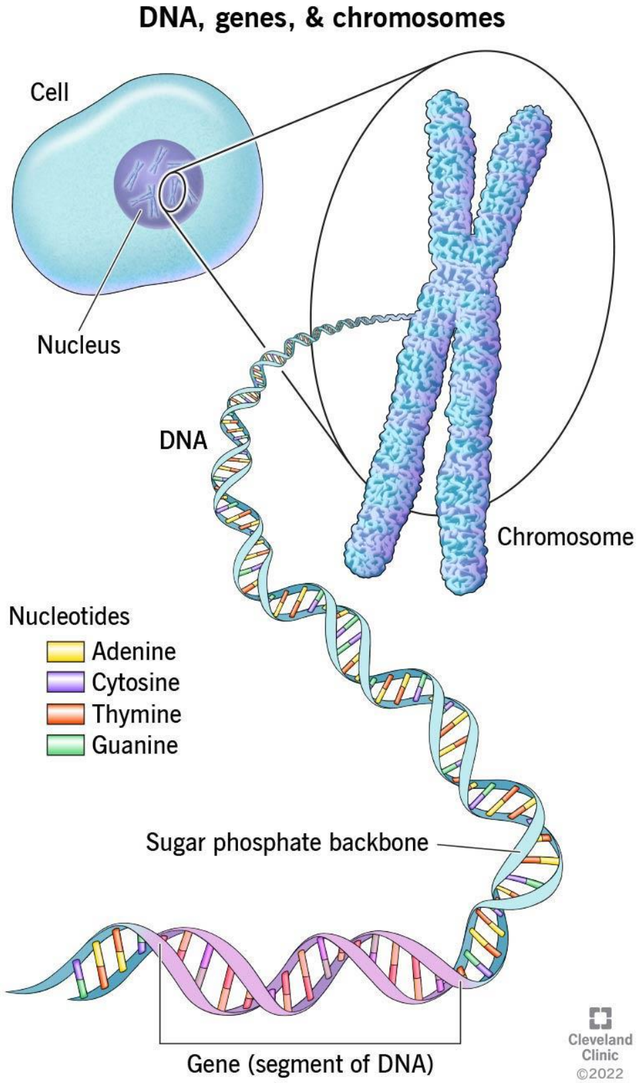


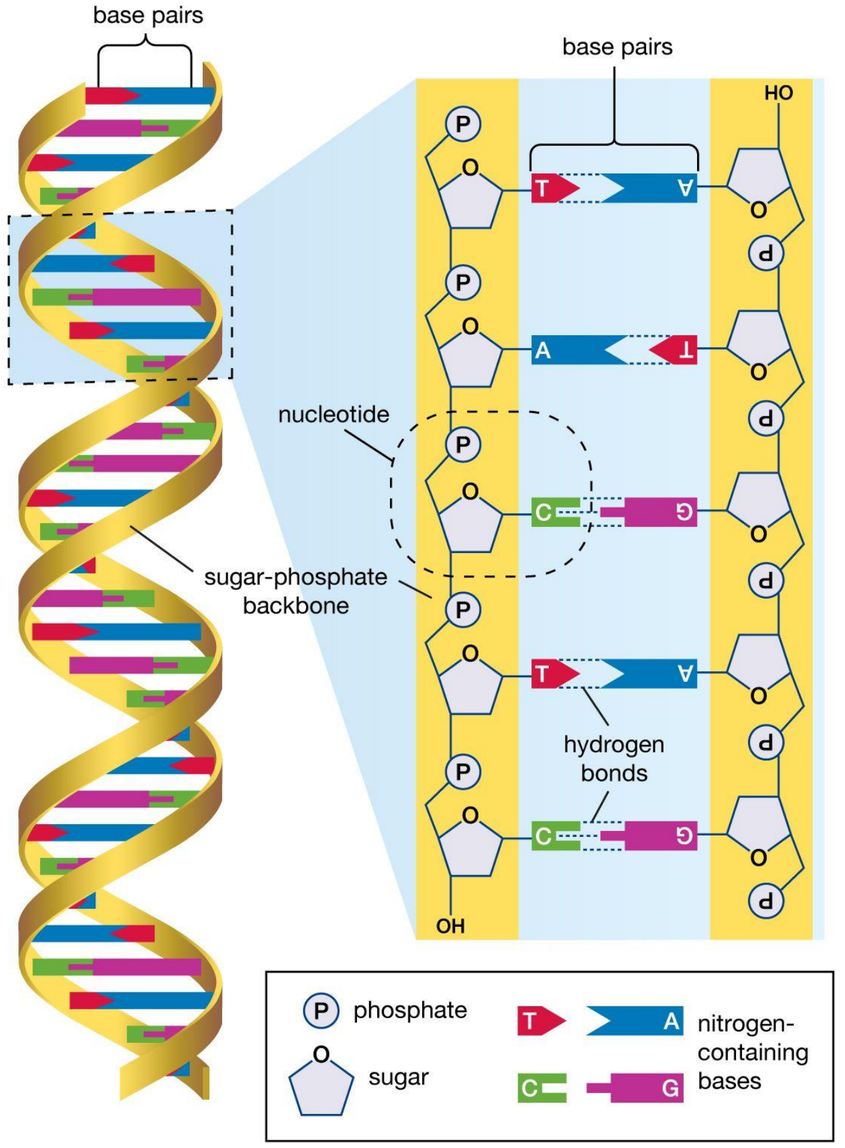




REs in applications

- Genomics



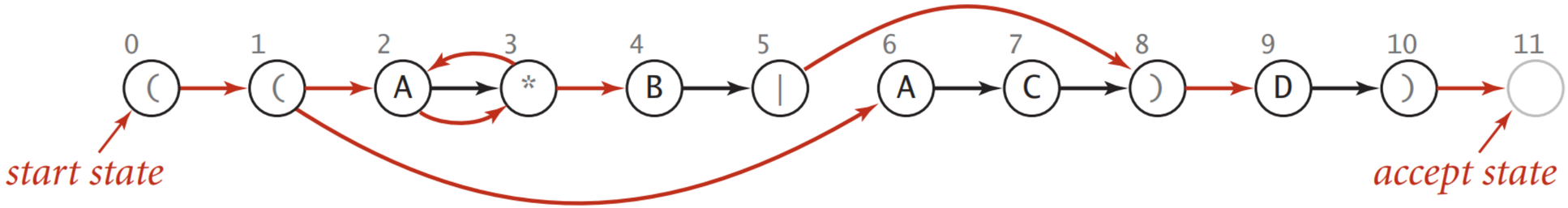


© Encyclopædia Britannica, Inc.

Non-deterministic finite-state automata

Kleene's Theorem, a fundamental result of theoretical computer science, asserts that there is an NFA corresponding to any given RE (and vice versa).

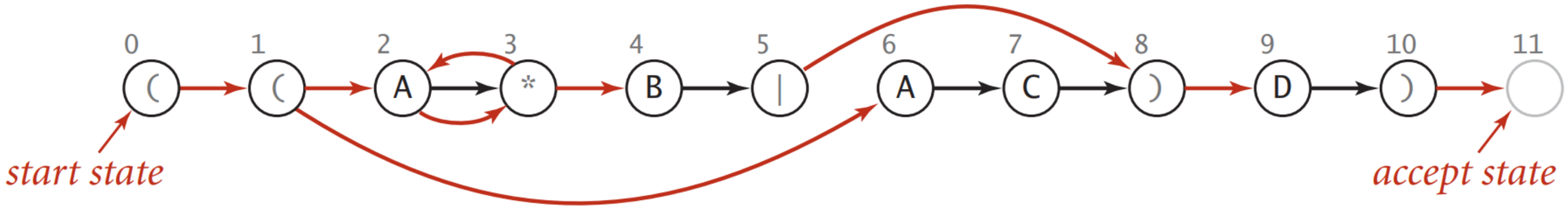
- Consider the figure below, which shows an NFA that determines whether a text string is in the language described by the RE ((A * B | A C) D) .



NFA corresponding to the pattern ((A * B | A C) D)

Non-deterministic finite-state automata

- The NFA corresponding to an RE of length M has exactly one state per pattern character, starts at state 0, and has a (virtual) accept state M.
- States corresponding to a character from the alphabet have an outgoing edge that goes to the state corresponding to the next character in the pattern (black edges in the diagram).
- States corresponding to the metacharacters (,), |, and * have at least one outgoing edge (red edges in the diagram), which may go to any other state.
- Some states have multiple outgoing edges, but no state has more than one outgoing black edge.

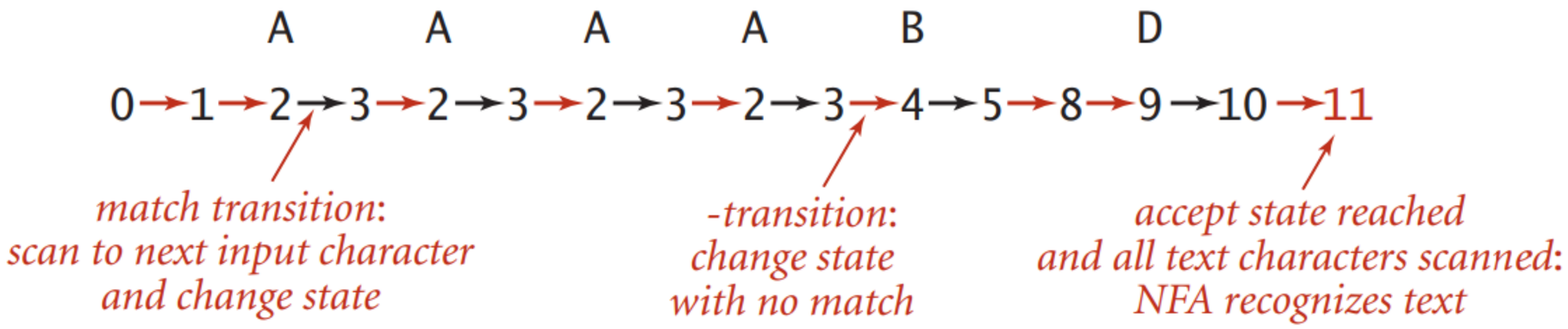


NFA corresponding to the pattern ((A * B | A C) D)

Non-deterministic finite-state automata

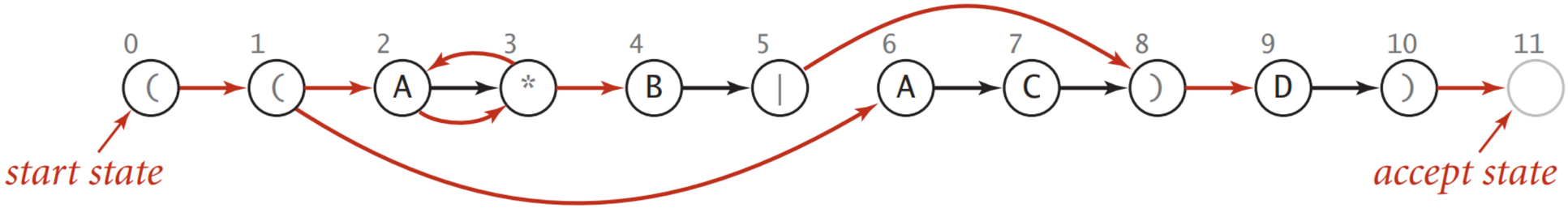
- The rules for moving from one state to another are also different than for DFAs—an NFA can do so in one of two ways:

- If the current state corresponds to a character in the alphabet and the current character in the text string matches the character, the automaton can take the (black) transition to the next state - a match transition.
- The automaton can follow any red edge to another state without scanning any text character. We refer to such a transition as an ε-transition, referring to the idea that it corresponds to "matching" the empty string ε.

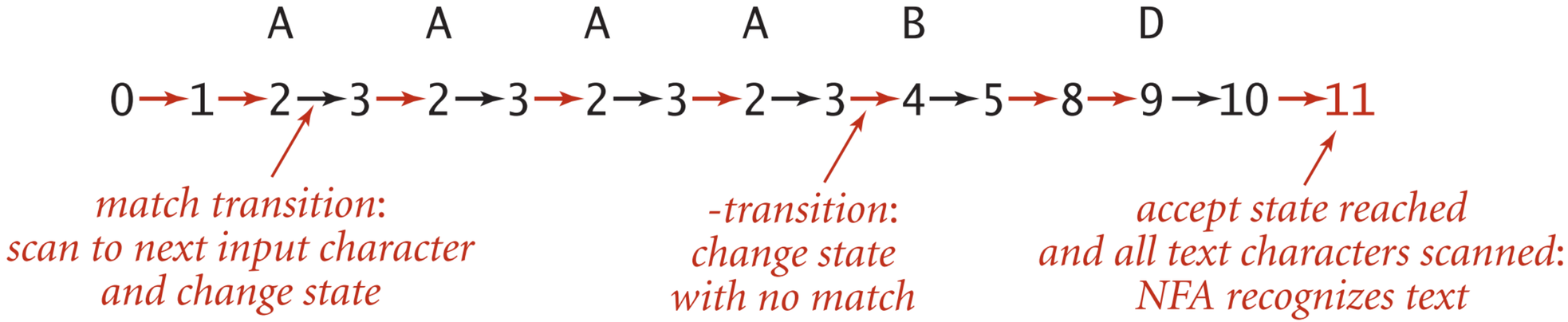


Finding a pattern with ((A * B | A C) D) NFA

Non-deterministic finite-state automata



NFA corresponding to the pattern ((A * B | A C) D)



Finding a pattern with ((A * B | A C) D) NFA

Non-deterministic finite-state automata

Simulating the **NFA**

- *Idea:* To simulate an NFA, we keep track of the set of states that could possibly be encountered while the automaton is examining the current input character.
- Steps:
  - For each such state, we check whether a match transition for the first input character is possible. This check gives us the set of possible states for the NFA just after matching the first input character.
  - To this set, we add all states that could be reached via ε –transitions from one of the states in the set.

→ Given the set of possible states for the NFA just after matching the first character in the input, the solution to the multiple-source reachability problem in the transition digraph gives the set of states that could lead to match transitions for the second character in the input.

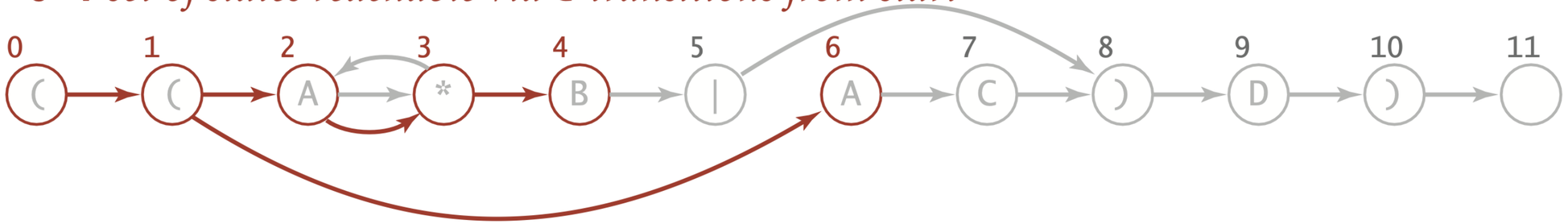
→ Iterating the process until all text characters are exhausted leads to one of two outcomes:

+) The set of possible states contains the accept states → *report success*

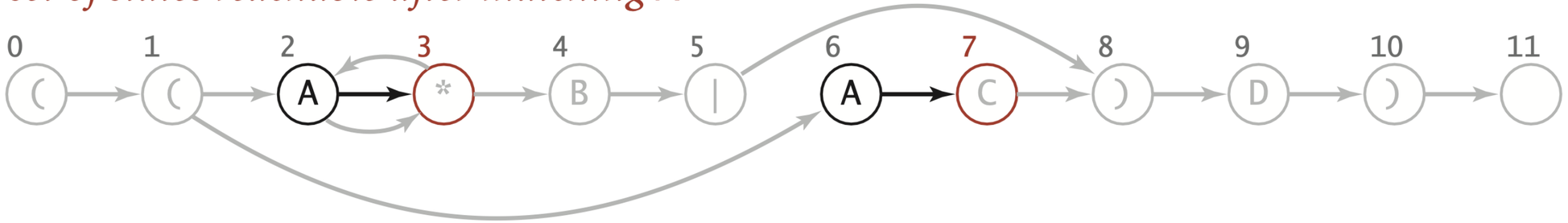
+) The set of possible states does not contain the accept state → *report failure*

Non-deterministic finite-state automata

0 1 2 3 4 6 : *set of states reachable via ϵ-transitions from start*



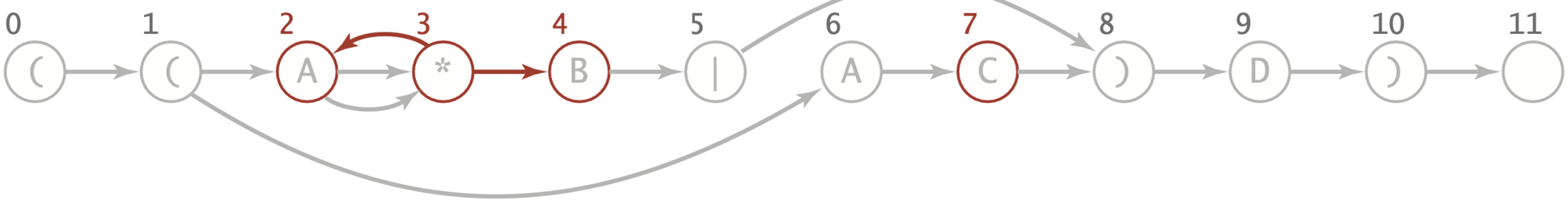
3 7 : *set of states reachable after matching A*



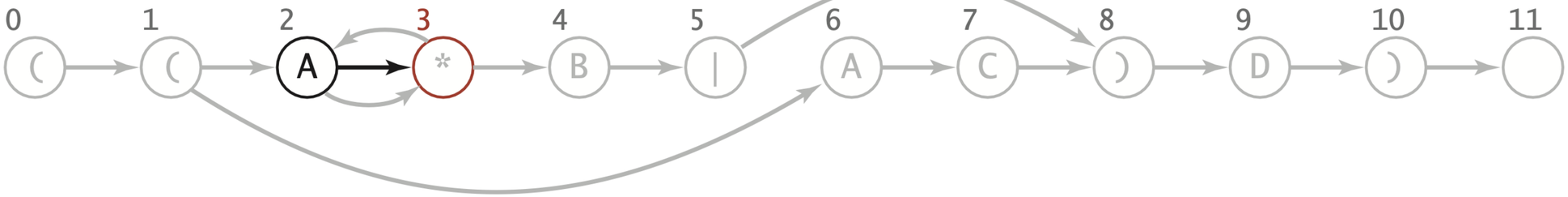
Simulation of **((A * B | A C) D)** NFA for input **A A B D**

Non-deterministic finite-state automata

2 3 4 7 : *set of states reachable via ϵ-transitions after matching* A



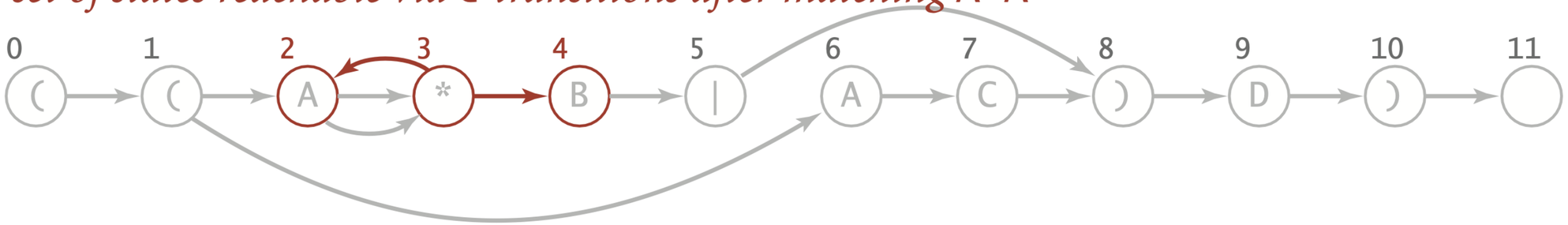
3 : *set of states reachable after matching* A A



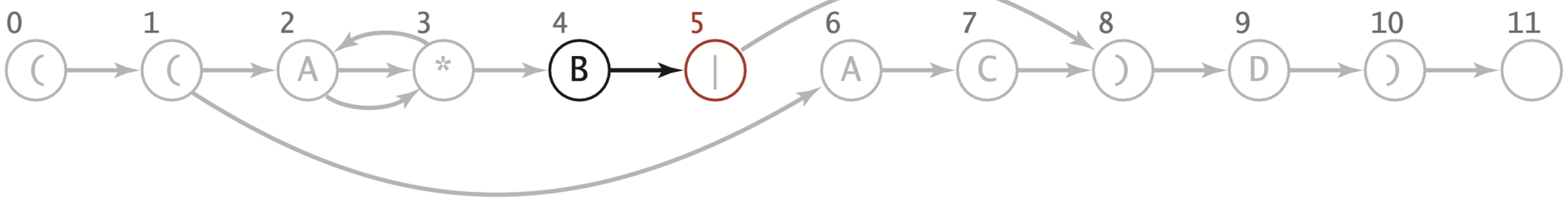
Simulation of **((A * B | A C) D)** NFA for input **A A B D**

Non-deterministic finite-state automata

2 3 4 : *set of states reachable via ϵ-transitions after matching* A A



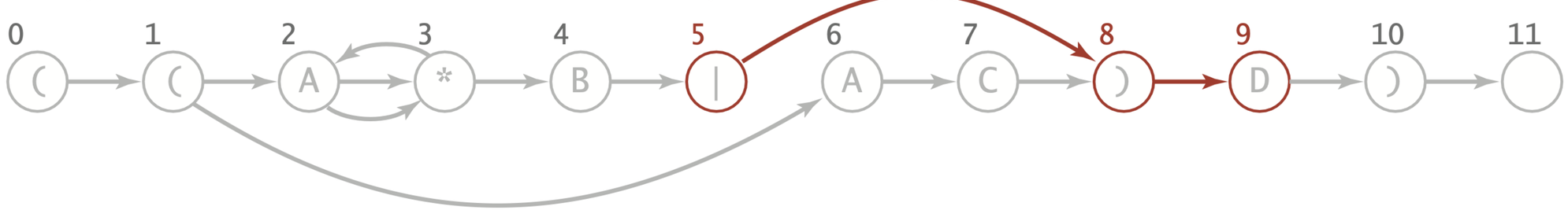
5 : *set of states reachable after matching* A A B



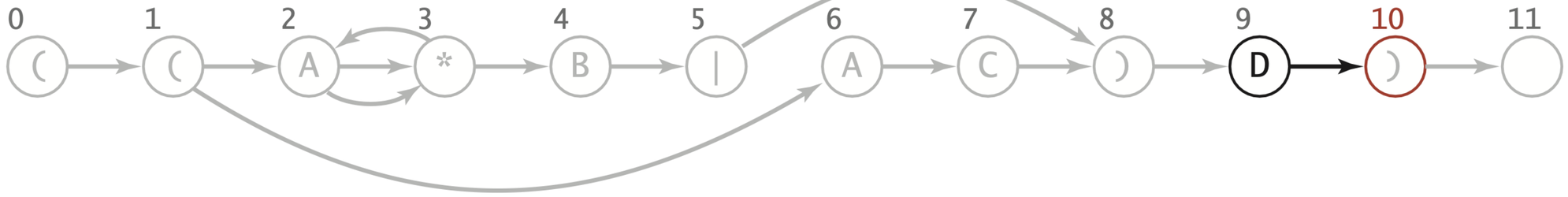
Simulation of **((A * B | A C) D)** NFA for input **A A B D**

Non-deterministic finite-state automata

5 8 9 : *set of states reachable via ϵ-transitions after matching* A A B



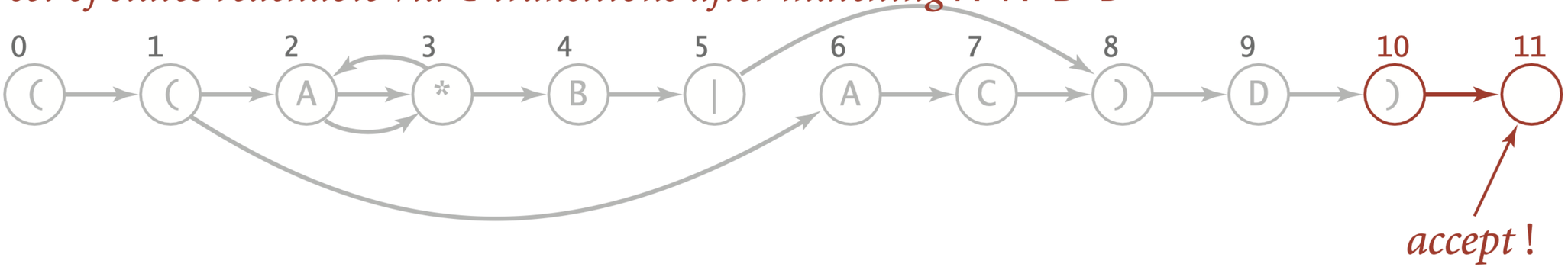
10 : *set of states reachable after matching* A A B D



Simulation of **((A * B | A C) D)** NFA for input **A A B D**

Non-deterministic finite-state automata

10 11 : *set of states reachable via ϵ-transitions after matching* A A B D

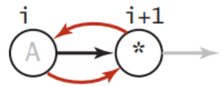


Simulation of **((A * B | A C) D)** NFA for input **A A B D**

Non-deterministic finite-state automata

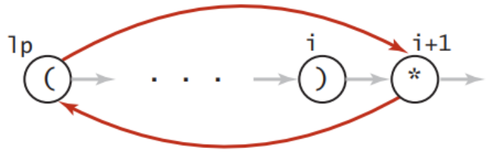
Building an **NFA** corresponding to an **RE**

single-character closure



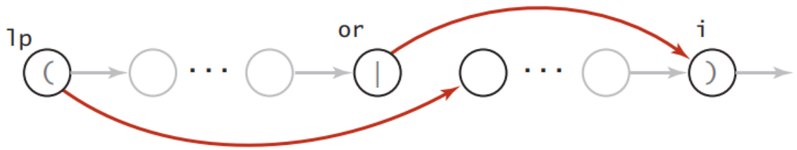

```
G.addEdge(i, i+1);
G.addEdge(i+1, i);
```

closure expression




```
G.addEdge(lp, i+1);
G.addEdge(i+1, lp);
```

or expression



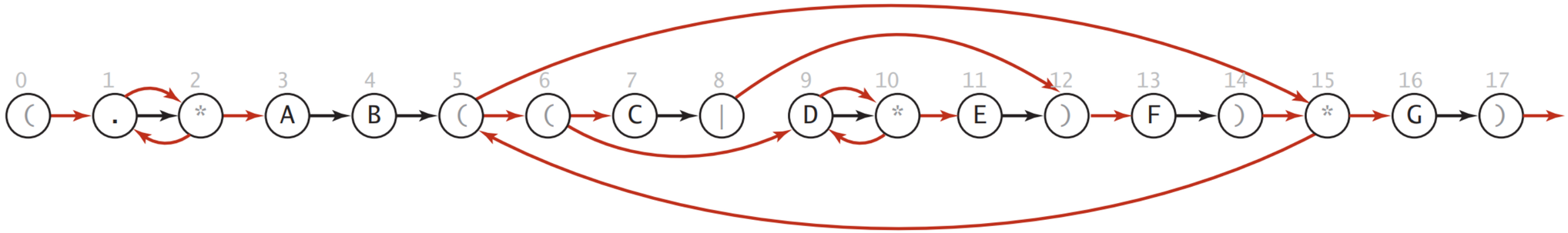

```
G.addEdge(lp, or+1);
G.addEdge(or, i);
```

NFA construction rules

Non-deterministic finite-state automata

Building an **NFA** corresponding to an **RE**

- Another example of an NFA corresponding to the given pattern.

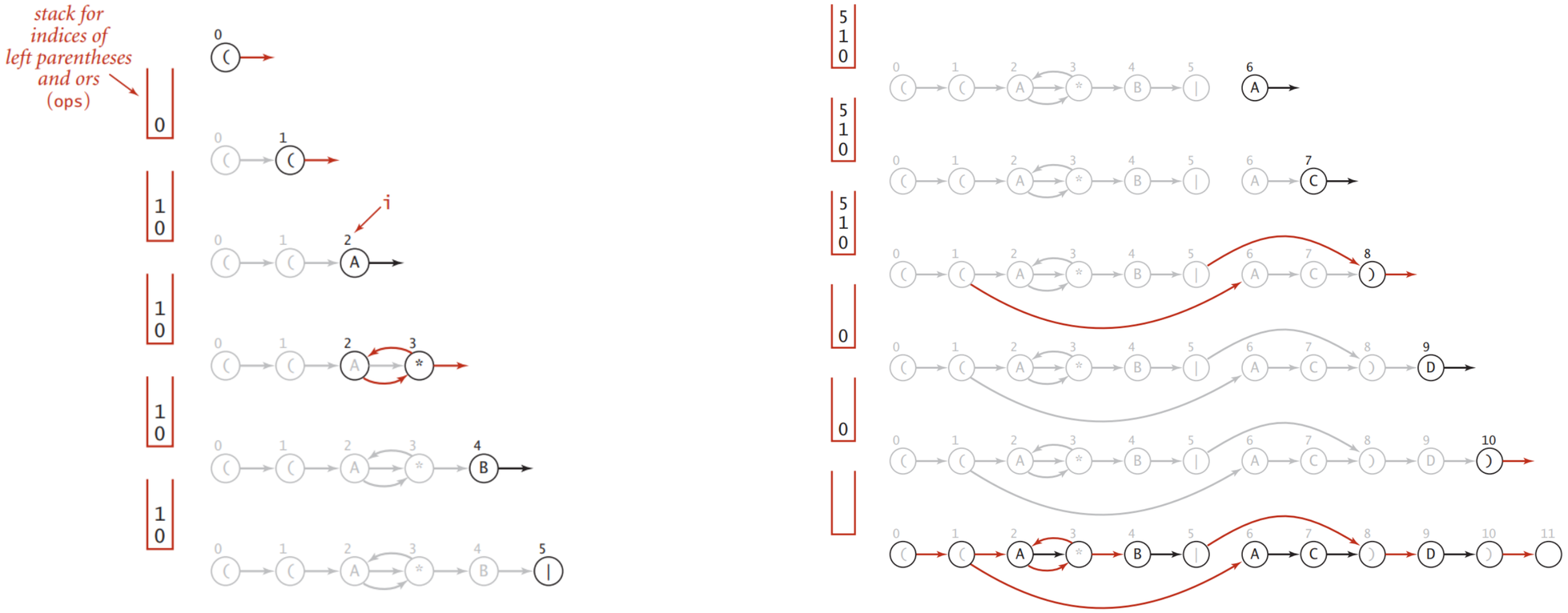


NFA corresponding to the pattern (. * A B ((C | D * E) F) * G)

Non-deterministic finite-state automata

Building an **NFA** corresponding to an **RE**

- Taking a cue from Dijkstra's algorithm, we will use a stack to keep track of the positions of left parentheses and or operators.



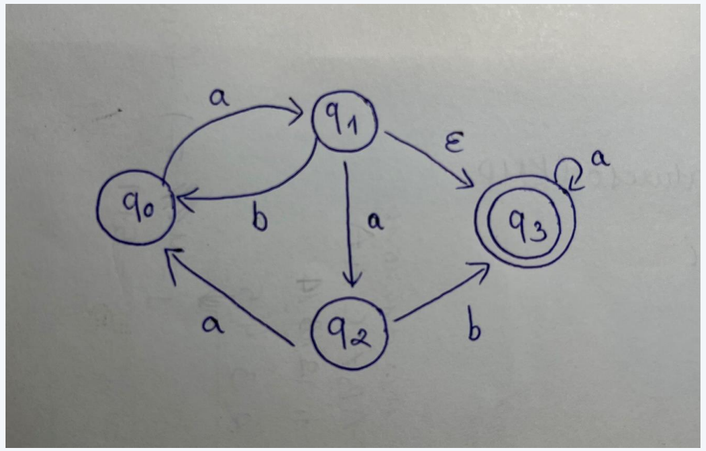
Building the **NFA** corresponding to **((A * B | A C) D)**

Homework

1. Construct the KMP DFA table and the LPS array for the following pattern string:
ABAABCABAABCB

2. Compute the right[] array for the pattern: **ABRACADABRA**.

3. Convert the following NFA graph to the corresponding RE.



thank you!