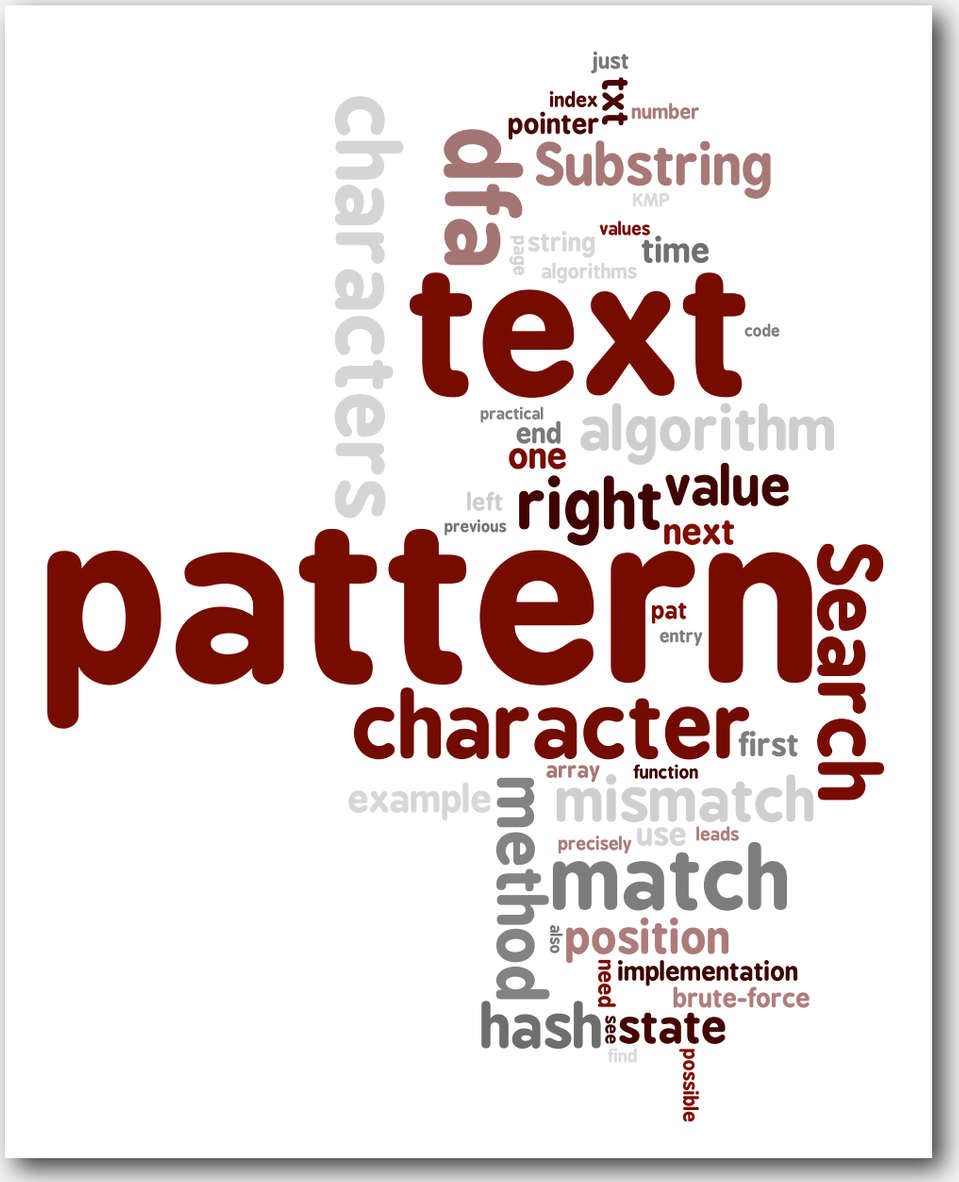
5.3 Substring Search



- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Substring search

Goal. Find pattern of length M in a text of length N.

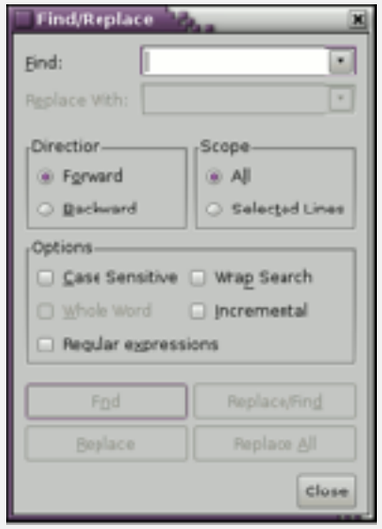
typically N >> M

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.

http://citp.princeton.edu/memory

Applications

- Parsers.
- Spam filters.
- Digital libraries.
- Screen scrapers.
- Word processors.
- Web search engines.
- Electronic surveillance.
- Natural language processing.
- Computational molecular biology.
- FBIs Digital Collection System 3000.
- Feature detection in digitized images.
- ...



Application: spam filtering

Application: spam filtering

Identify patterns indicative of spam.

Application: spam filtering

Identify patterns indicative of spam.

- **`PROFITS`**

Application: spam filtering

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`

Application: spam filtering

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`
- `herbal Viagra`

Application: spam filtering

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`
- `herbal Viagra`
- `There is no catch.`

Application: spam filtering

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`
- `herbal Viagra`
- `There is no catch.`
- `L0W M0RTGAGE RATES`

Application: spam filtering

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`
- `herbal Viagra`
- `There is no catch.`
- `L0W M0RTGAGE RATES`
- `This is a one-time mailing.`

Application: spam filtering

Identify patterns indicative of spam.

- **PROFITS**
- **L0SE WE1GHT**
- **herbal Viagra**
- **There is no catch.**
- **L0W M0RTGAGE RATES**
- **This is a one-time mailing.**
- **This message is sent in compliance with spam regulations.**

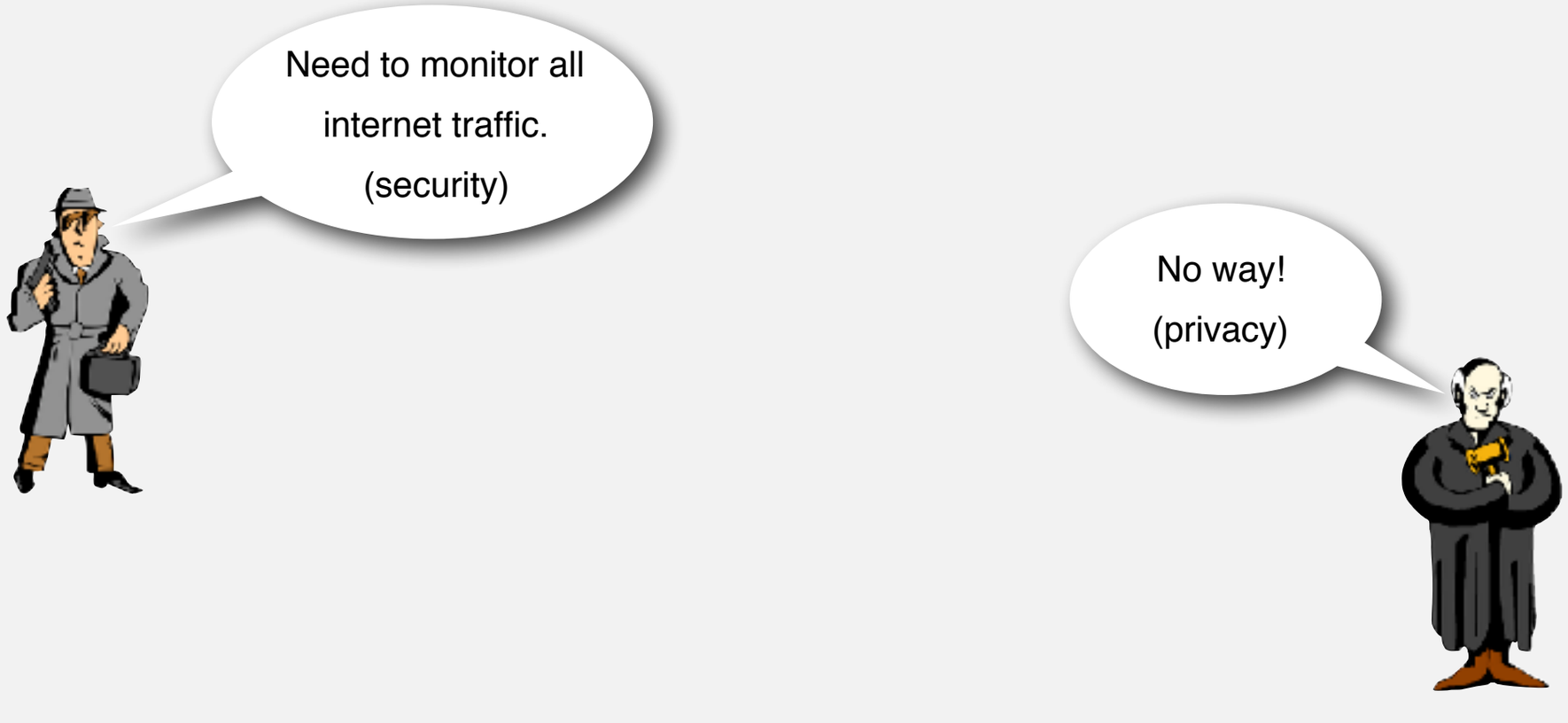
Application: spam filtering

Identify patterns indicative of spam.

- PROFITS
- L0SE WE1GHT
- herbal Viagra
- There is no catch.
- L0W M0RTGAGE RATES
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.

Application: electronic surveillance

Application: electronic surveillance



Application: electronic surveillance

Need to monitor all internet traffic. (security)

Well, we're mainly interested in "ATTACK AT DAWN"

No way! (privacy)

Application: electronic surveillance

Need to monitor all internet traffic. (security)

No way! (privacy)

Well, we're mainly interested in "ATTACK AT DAWN"

OK. Build a machine that just looks for that.

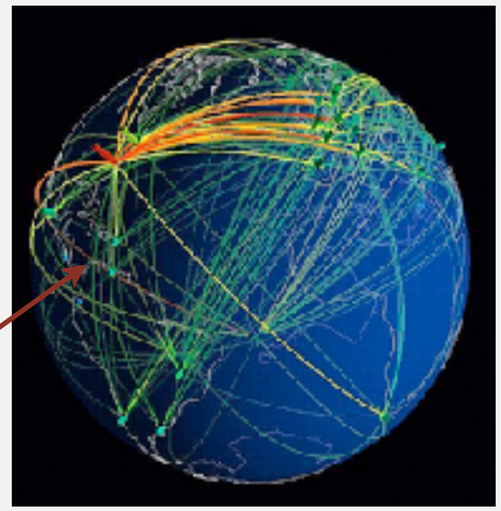
Application: electronic surveillance

Need to monitor all internet traffic. (security)

Well, we're mainly interested in "ATTACK AT DAWN"

No way! (privacy)

OK. Build a machine that just looks for that.



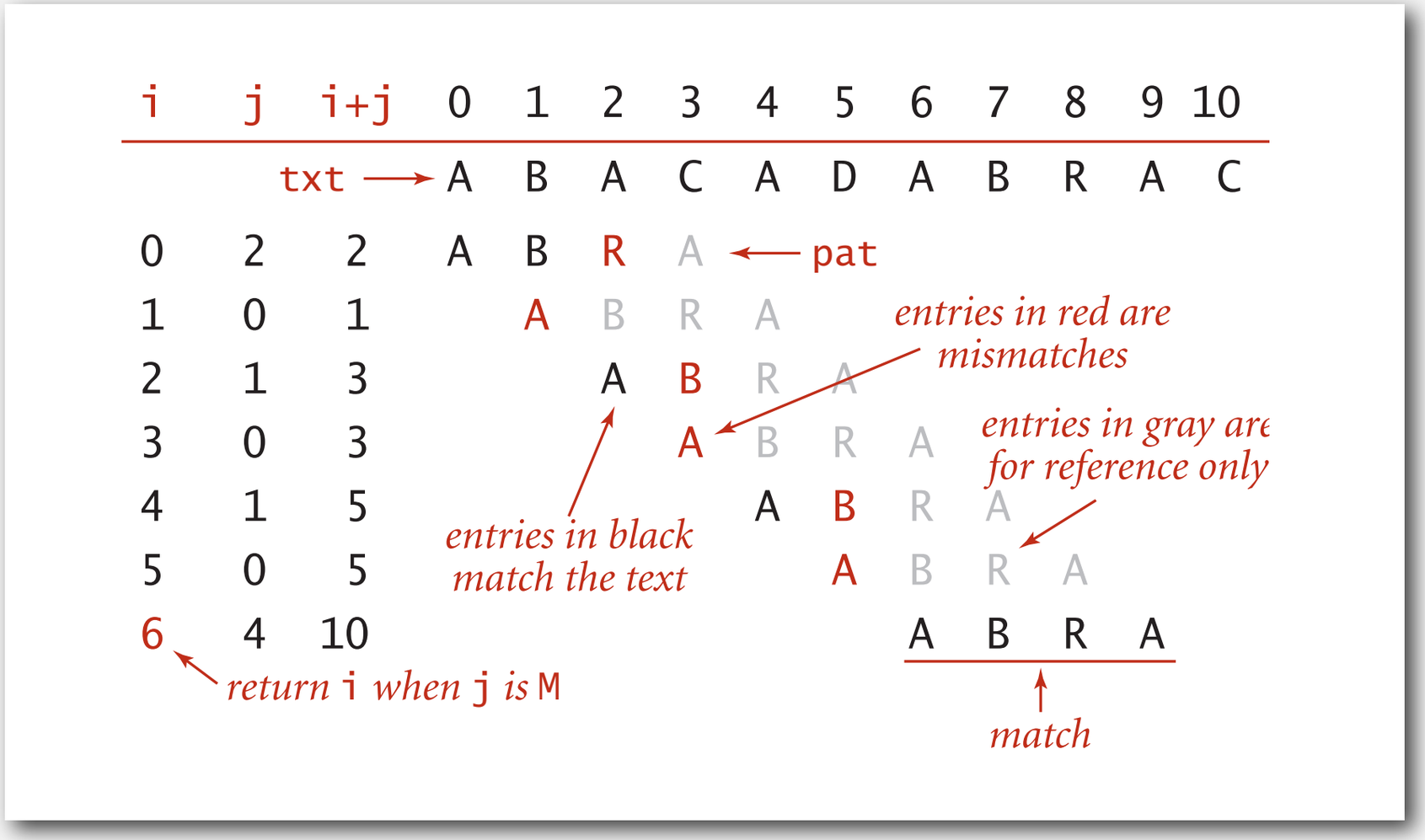
"ATTACK AT DAWN" substring search machine

found

- **brute force**
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Brute-force substring search

Check for pattern starting at each text position.



Brute-force substring search: Java implementation

Check for pattern starting at each text position.

i = 4, j = 3

```
  0 1 2 3 4 5 6 7 8 9 10
  A B A C A D A B R A C
        A D A C R
```

```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;   <-- index in text where pattern starts
   }
   return N;   <-- not found
}
```

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

i	j	i+j		0	1	2	3	4	5	6	7	8	9
			txt →	A	A	A	A	A	A	A	A	A	B
0	4	4		A	A	A	A	B	← pat				
1	4	5			A	A	A	A	B				
2	4	6				A	A	A	A	B			
3	4	7					A	A	A	A	B		
4	4	8						A	A	A	A	B	
5	5	10							A	A	A	A	B

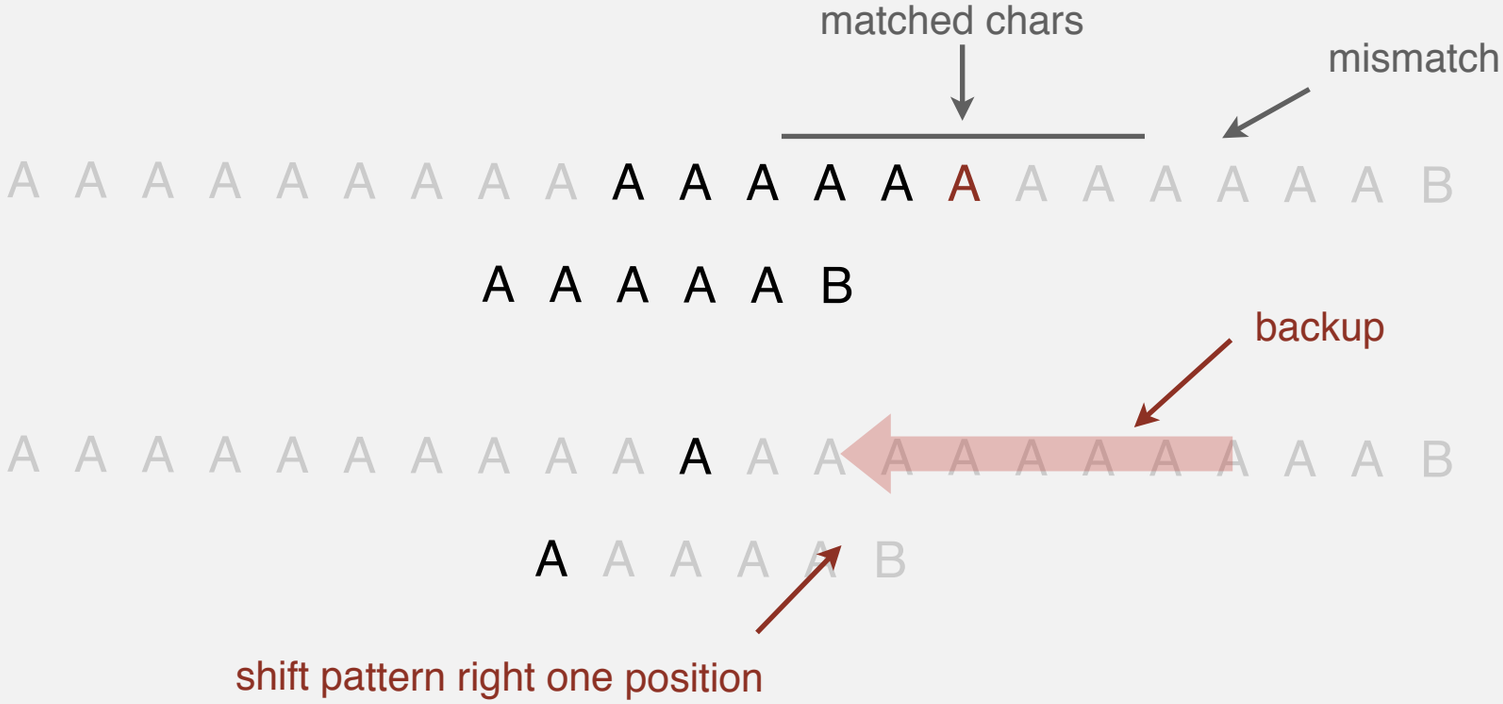
Worst case. $\sim M N$ char compares.

Backup

In typical applications, we want to avoid backup in text stream.

- Treat input as stream of data.
- Abstract model: standard input.

Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of size M (build backup into standard input).

Approach 2. Stay tuned.

Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.

- `i` points to end of sequence of already-matched chars in text.
- `j` stores number of already-matched chars (end of sequence in pattern).

i = 6, j = 3

0	1	2	3	4	5	6	7	8	9	10
A	B	A	C	A	D	A	B	R	A	C
			A	D	A	C	R			

```
public static int search(String pat, String txt)
{
   int i, N = txt.length();
   int j, M = pat.length();
   for (i = 0, j = 0; i < N && j < M; i++)
   {
      if (txt.charAt(i) == pat.charAt(j)) j++;
      else { i -= j; j = 0;  }
   }
   if (j == M) return i - M;
   else        return N;
}
```

backup

Algorithmic challenges in substring search

Brute-force is often not good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

```
Now is the time for all people to come to the aid of their party. Now is the time for all good people to
come to the aid of their party. Now is the time for many good people to come to the aid of their party.
Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good
people to come to the aid of their party. Now is the time for all of the good people to come to the aid of
their party. Now is the time for all good people to come to the aid of their party. Now is the time for
each good person to come to the aid of their party. Now is the time for all good people to come to the aid
of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the
time for all good people to come to the aid of their party. Now is the time for many or all good people to
come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now
is the time for all good Democrats to come to the aid of their party. Now is the time for all people to
come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now
is the time for many good people to come to the aid of their party. Now is the time for all good people to
come to the aid of their party. Now is the time for a lot of good people to come to the aid of their
party. Now is the time for all of the good people to come to the aid of their party. Now is the time for
all good people to come to the aid of their attack at dawn party. Now is the time for each person to come
to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is
the time for all good Republicans to come to the aid of their party. Now is the time for all good people
to come to the aid of their party. Now is the time for many or all good people to come to the aid of their
party. Now is the time for all good people to come to the aid of their party. Now is the time for all good
Democrats to come to the aid of their party.
```

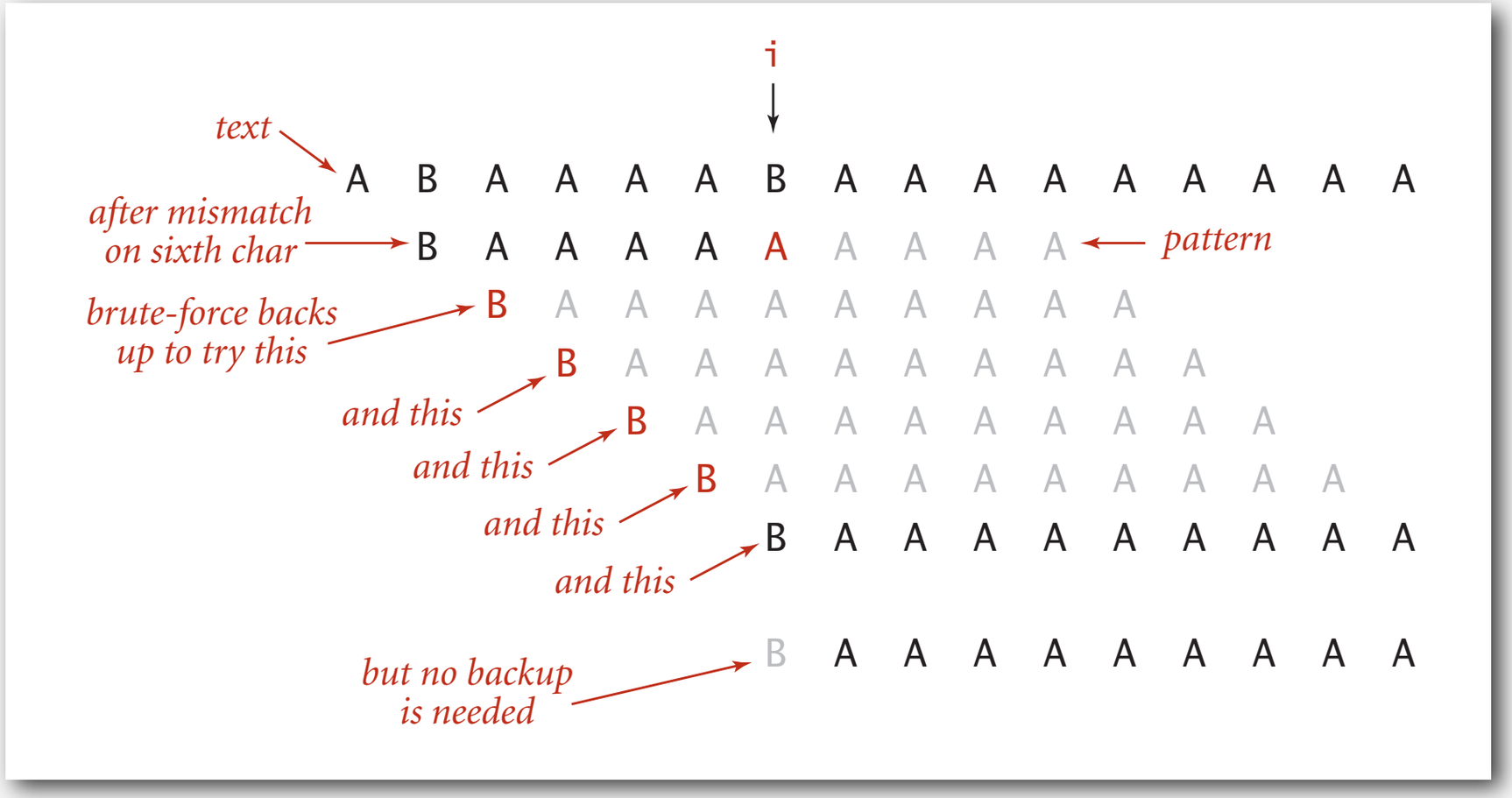
- brute force
- **Knuth-Morris-Pratt**
- Boyer-Moore
- Rabin-Karp

Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern **BAAAAAAAAA**.

- Suppose we match 5 chars in pattern, with mismatch on 6^{th} char.
- We know previous 6 chars in text are **BAAAAB**.
- Don't need to back up text pointer!

assuming { A, B } alphabet



Knuth-Moris-Pratt algorithm. Clever method to always avoid backup. (!)

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

internal representation

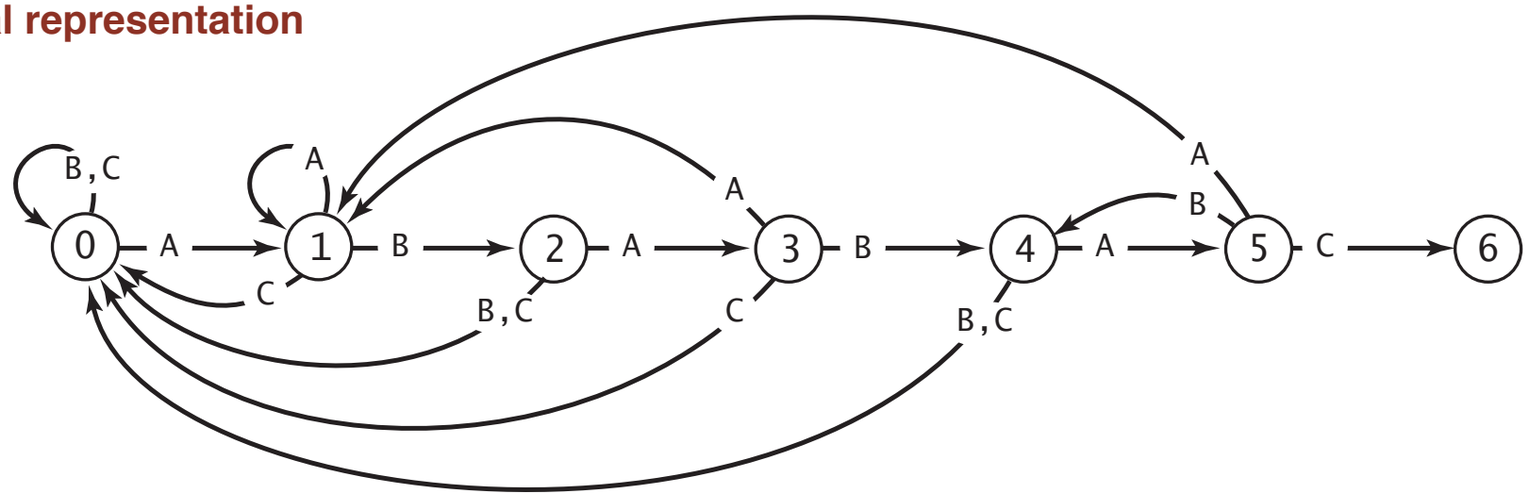
	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

If in state `j` reading char `c`:

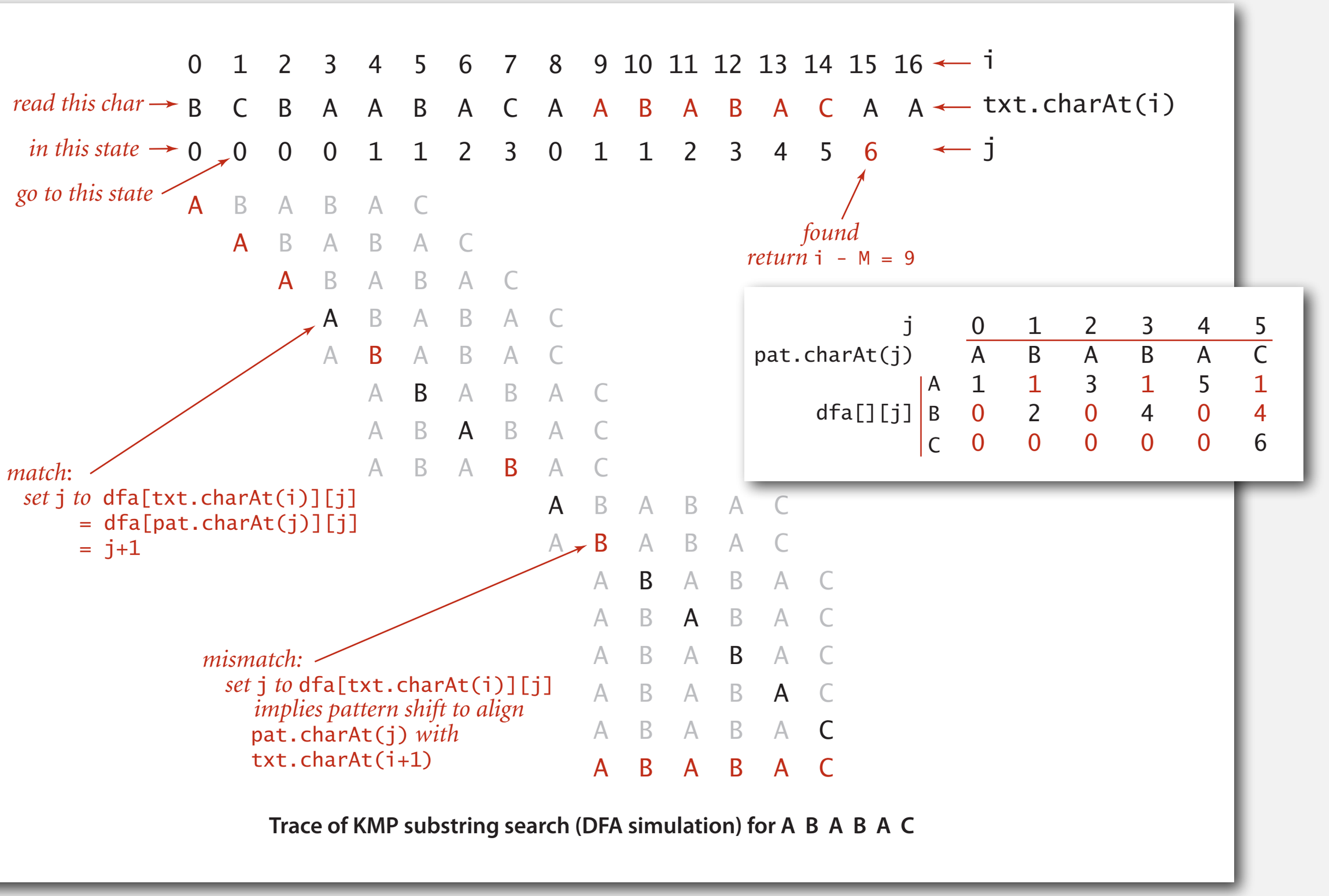
if `j` is `6` halt and accept

else move to state `dfa[c][j]`

graphical representation



KMP substring search: trace



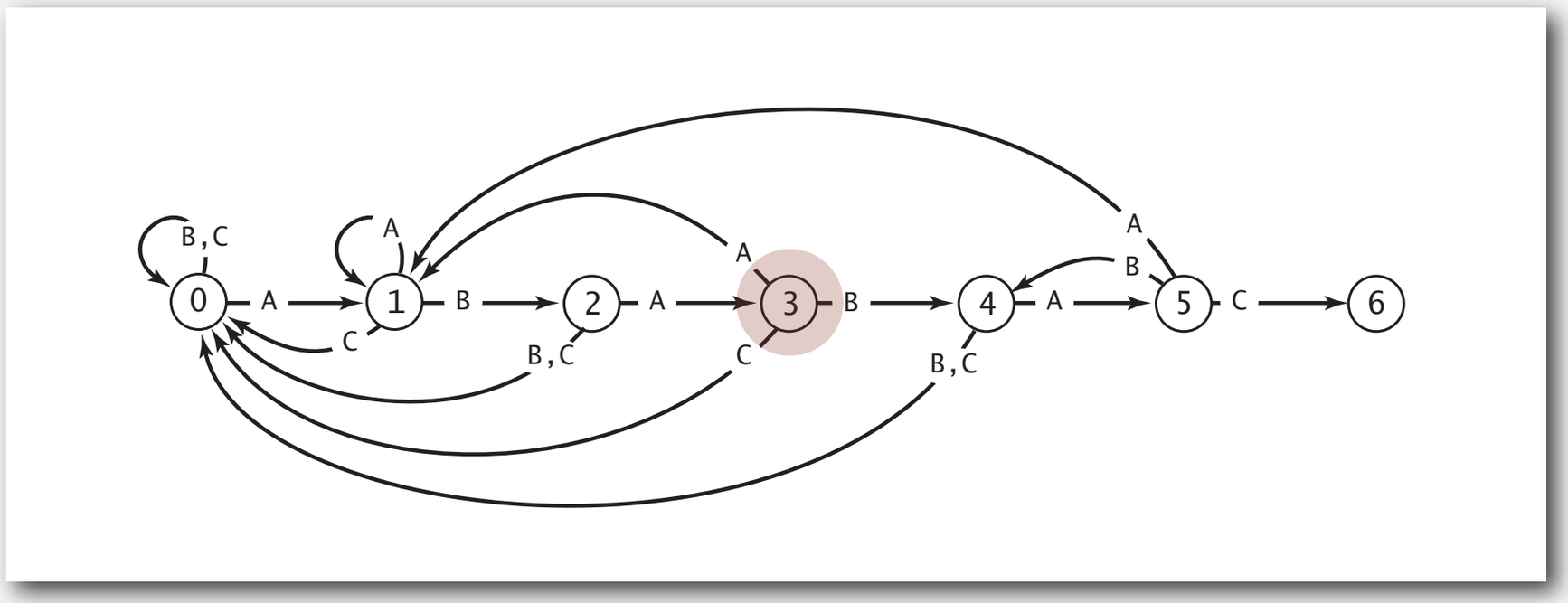
Trace of KMP substring search (DFA simulation) for A B A B A C

Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in `txt[i]`?

A. State = number of characters in pattern that have been matched.
(length of longest prefix of `pat[]` that is a suffix of `txt[0..i]`)

Ex. DFA is in state 3 after reading in character `txt[6]`.



KMP search: Java implementation

Key differences from brute-force implementation.

- Text pointer `i` never decrements.
- Need to precompute `dfa[][]` from pattern.

```
public int search(String txt)
{
   int i, j, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
      j = dfa[txt.charAt(i)][j];
   if (j == M) return i - M;
   else        return N;
}
```

no backup

Running time.

- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

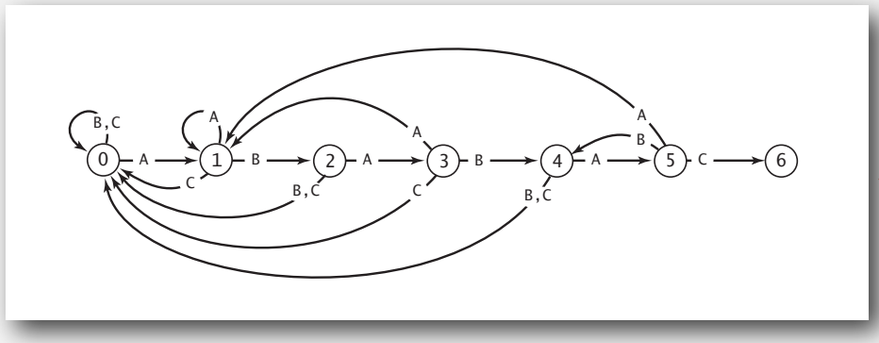
KMP search: Java implementation

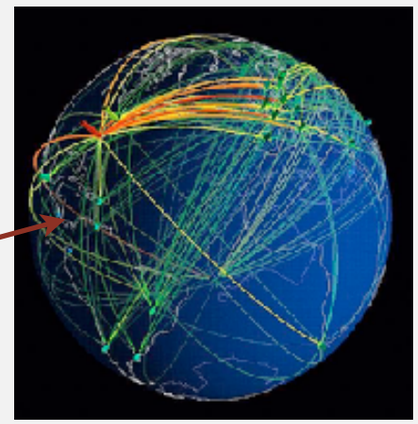
Key differences from brute-force implementation.

- Text pointer `i` never decrements.
- Need to precompute `dfa[][]` from pattern.
- Could use input stream.

```
public int search(In in)
{
   int i, j;
   for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
      j = dfa[in.readChar()][j];
   if (j == M) return i - M;
   else        return NOT_FOUND;
}
```

no backup



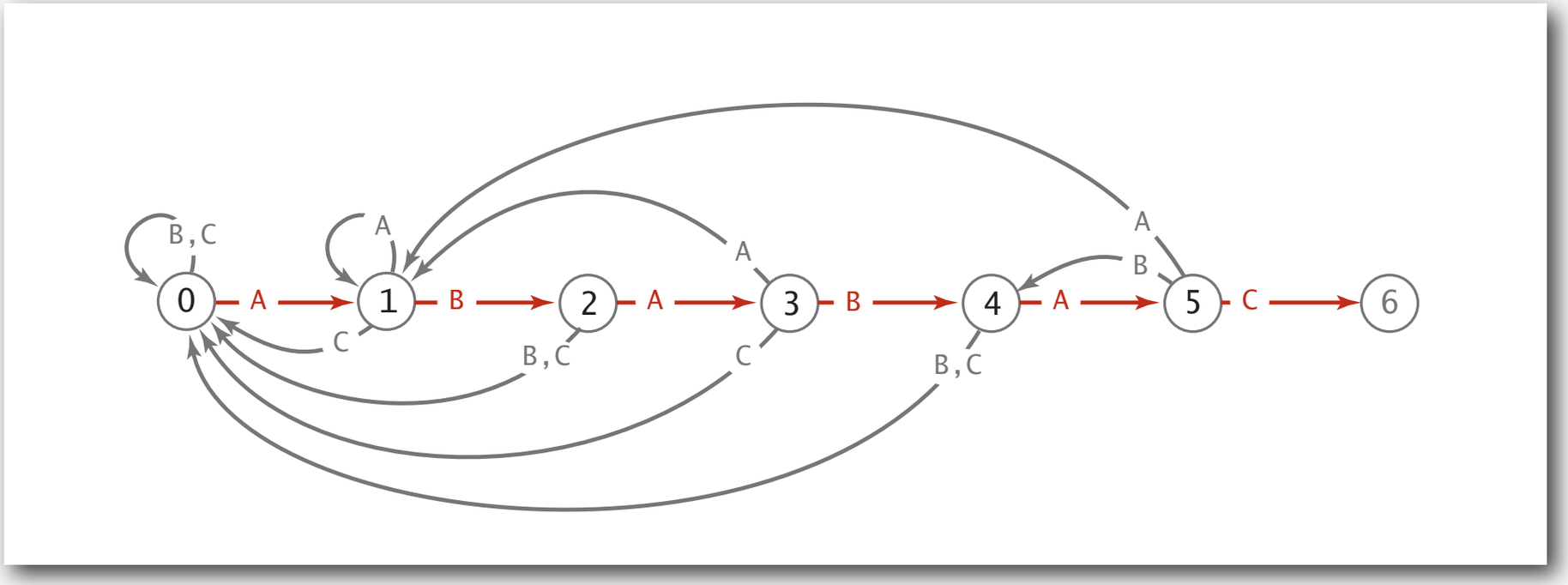


How to build DFA from pattern?

Match transition. If in state `j` and next char `c == pat.charAt(j)`, then go to state `j+1`.

- `j+1`: now first j+1 characters of pattern have been matched
- `j`: first j characters of pattern have already been matched
- `c == pat.charAt(j)`: next char matches

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



How to build DFA from pattern?

Mismatch transition. If in state `j` and next char `c != pat.charAt(j)`, then the last `j` characters of input are `pat[1..j-1]`, followed by `c`.

To compute `dfa[c][j]`: Simulate `pat[1..j-1]` on DFA and take transition `c`.

Running time. Seems to require j steps.

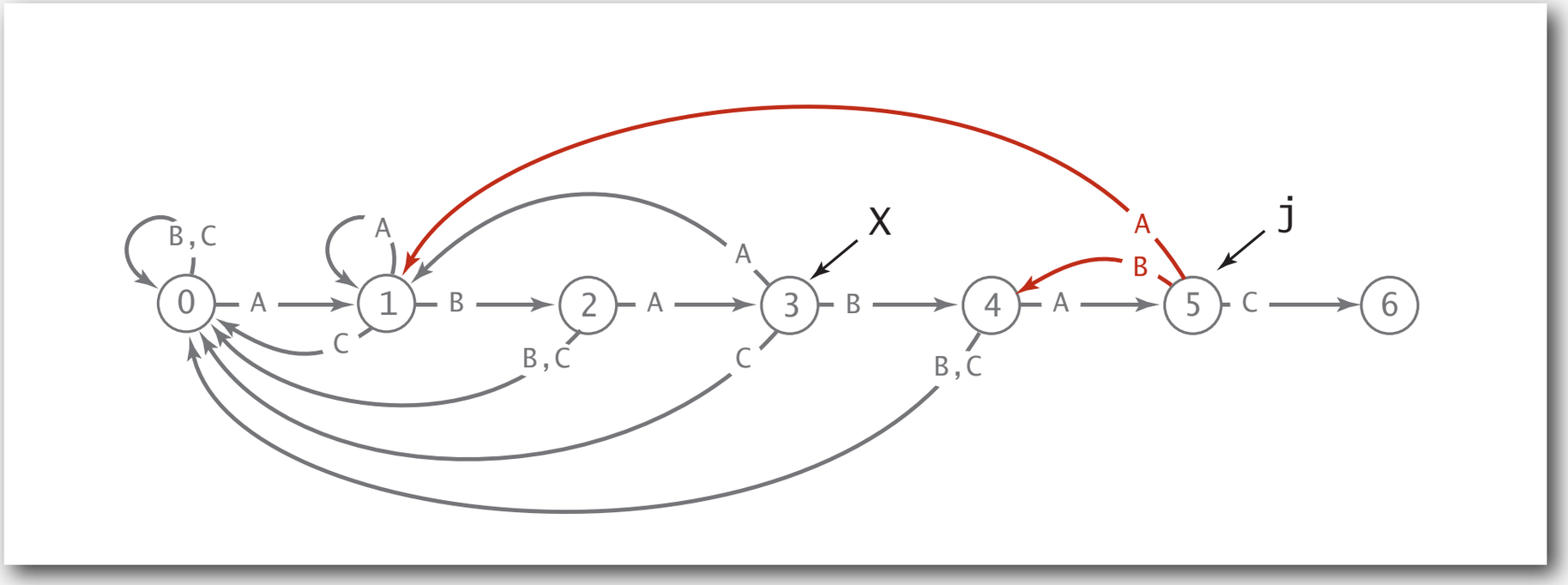
still under construction (!)

Ex. `dfa['A'][5] = 1;` `dfa['B'][5] = 4`

simulate BABA (state X); take transition 'A'

simulate BABA (state X); take transition 'B'

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



How to build DFA from pattern?

Mismatch transition. If in state `j` and next char `c != pat.charAt(j)`, then the last `j` characters of input are `pat[1..j-1]`, followed by `c`.

To compute `dfa[c][j]`: Simulate `pat[1..j-1]` on DFA and take transition `c`.

state X

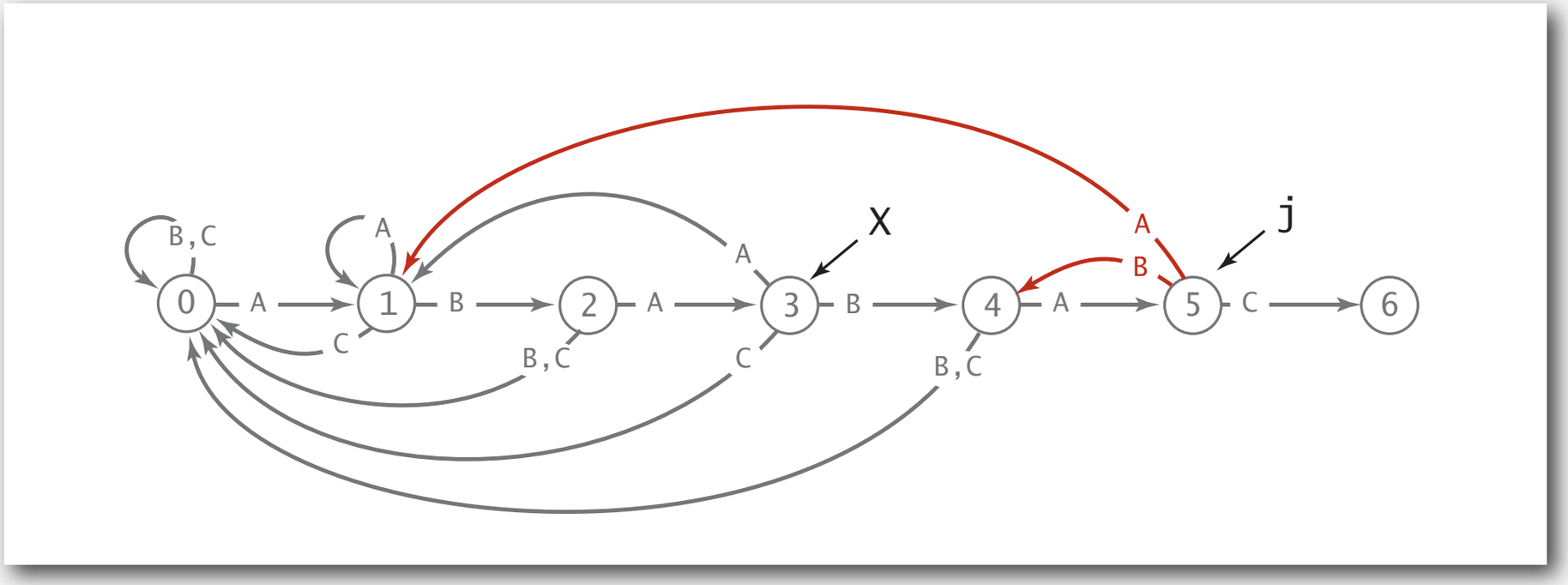
Running time. Takes only constant time if we know state X. (!)

Ex. `dfa['A'][5] = 1;` `dfa['B'][5] = 4;` `X'= 0`

from state X, take transition 'A' = dfa['A'][X]

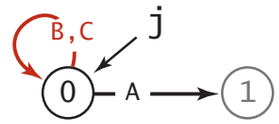
from state X, take transition 'B' = dfa['B'][X]

from state X, take transition 'C' = dfa['C'][X]

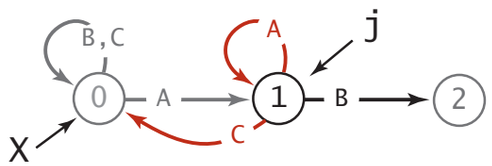


Constructing the DFA for KMP substring search: example

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



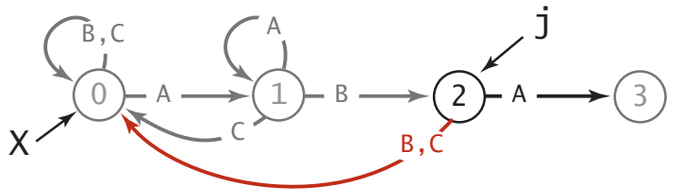
j	0	1
pat.charAt(j)	A	B
dfa[][j] A	1	1
dfa[][j] B	0	2
dfa[][j] C	0	0



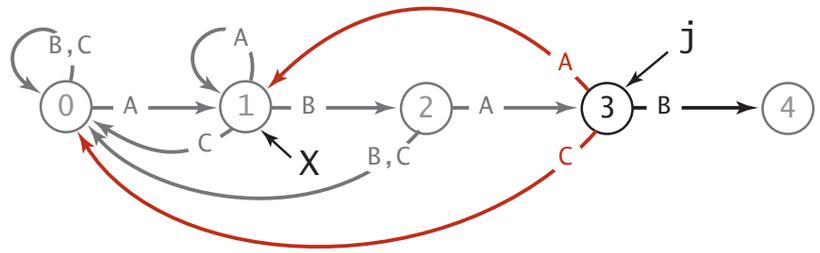
copy `dfa[][X]` *to* `dfa[][j]`

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

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

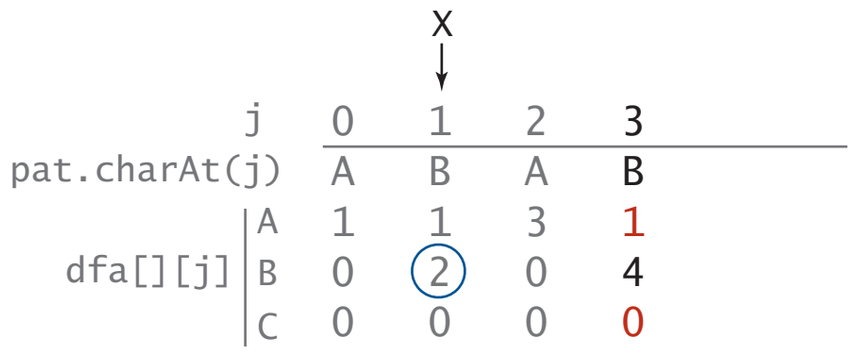


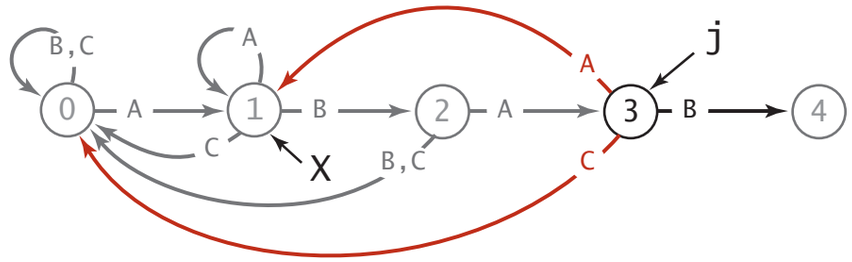
j	0	1	2	3
pat.charAt(j)	A	B	A	B
dfa[][j] A	1	1	3	1
dfa[][j] B	0	2	0	4
dfa[][j] C	0	0	0	0

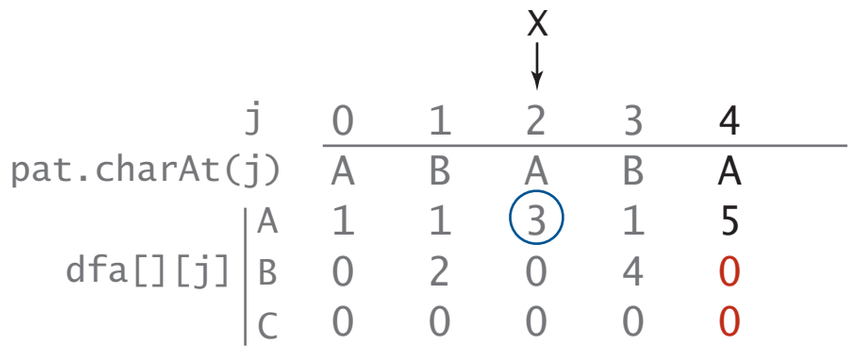


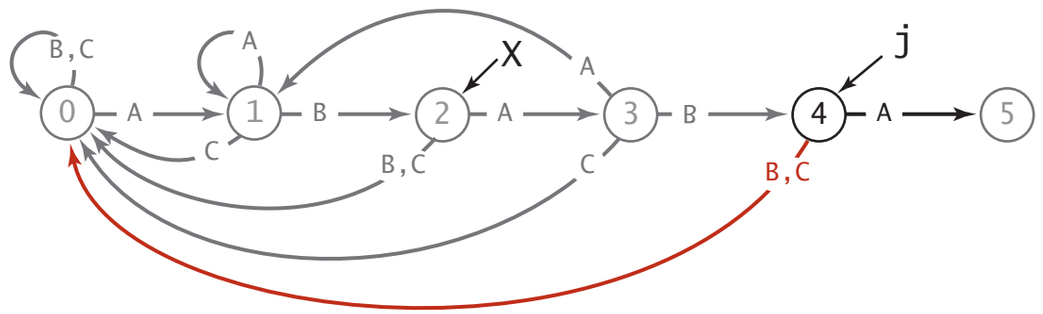
Constructing the DFA for KMP substring search for A B A B A C

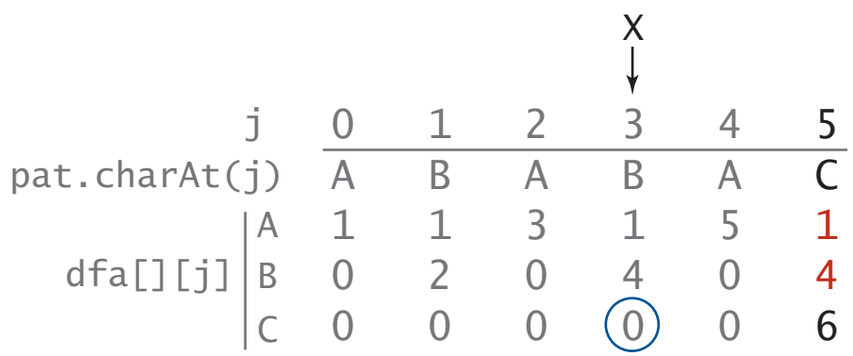
Constructing the DFA for KMP substring search: example

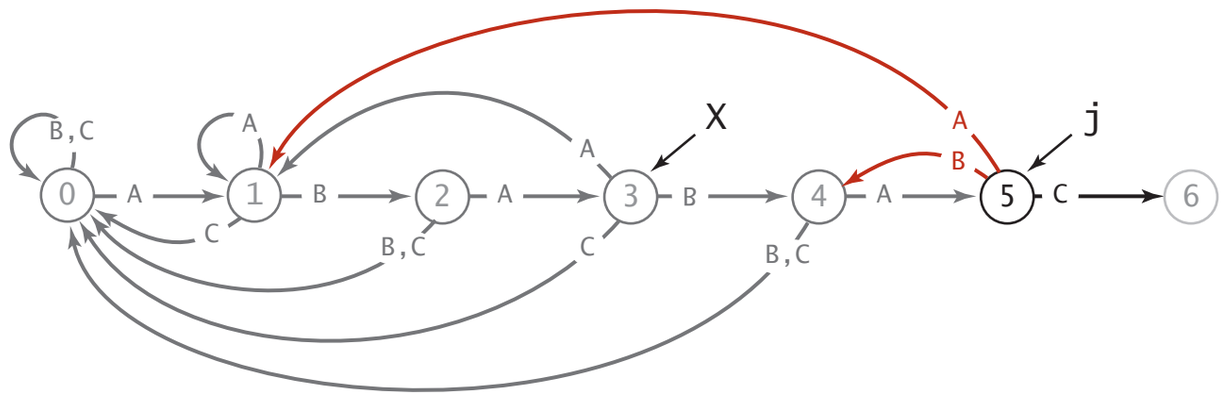












Constructing the DFA for KMP substring search for A B A B A C

- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- **Rabin-Karp**

Michael Rabin, Turing Award '76

and Dick Karp, Turing Award '85

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of pattern characters 0 to $M - 1$.
- For each i, compute a hash of text characters i to $M + i - 1$.
- If pattern hash = text substring hash, check for a match.

```
          pat.charAt(i)
 i        0  1  2  3  4
          2  6  5  3  5   % 997 = 613


                          txt.charAt(i)
 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
 1           1  4  1  5  9   % 997 = 201
 2              4  1  5  9  2   % 997 = 715
 3                 1  5  9  2  6   % 997 = 971
 4                    5  9  2  6  5   % 997 = 442
 5                       9  2  6  5  3   % 997 = 929
 6 <-- return i = 6            2  6  5  3  5   % 997 = 613
```

match

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

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

Intuition. M-digit, base-R integer, modulo Q.

Horner's method. Linear-time method to evaluate degree-M polynomial.

	pat.charAt()				
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 = (659*10 + 5) % 997 = 613

R Q

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

```
// Compute hash for M-digit key
private int hash(String key, int M)
{
   int h = 0;
   for (int j = 0; j < M; j++)
      h = (R * h + key.charAt(j)) % Q;
   return h;
}
```

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i.

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

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

Key property. Can update hash function in constant time!

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

shift left	subtract leftmost digit	add new rightmost digit

i	...	2	3	4	5	6	7	...	
current value	1	4	1	5	9	2	6	5	*text*
new value		4	1	5	9	2	6	5	

	4	1	5	9	2	*current value*
-	4	0	0	0	0	
		1	5	9	2	*subtract leading digit*
			*	1	0	*multiply by radix*
	1	5	9	2	0	
				+	6	*add new trailing digit*
	1	5	9	2	6	*new value*

Rabin-Karp substring search example

```
 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                                  Q
                                                    ↓
 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 = (150*10 + 5) % 997 = 508
                                          RM    R
                                          ↓     ↓
 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
                                                                              match
                                                                                ↓
10 ← return i-M+1 = 6          2  6  5  3  5  % 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613
```

Rabin-Karp: Java implementation

```
public class RabinKarp
{
   private int patHash;      // pattern hash value
   private int M;            // pattern length
   private int Q;            // modulus
   private int R;            // radix
   private int RM;           // R^(M-1) % Q

   public RabinKarp(String pat) {
      M = pat.length();
      R = 256;
      Q = largeRandomPrime();

      RM = 1;
      for (int i = 1; i <= M-1; i++)
         RM = (R * RM) % Q;
      patHash = hash(pat, M);
   }

   private int hash(String key, int M)
   {  /* as before */  }

   public int search(String txt)
   {  /* see next slide */  }
}
```

a large prime (but not so large to cause int overflow)

precompute $R^M - 1 \pmod{Q}$

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; j++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check for hash collision using rolling hash function

Las Vegas version. Check for substring match if hash match; continue search if false collision.

Rabin-Karp analysis

Theory. If Q is a sufficiently large random prime (about $M N^2$), then the probability of a false collision is about $1 / N$.

Practice. Choose Q to be a large prime (but not so large as to cause overflow). Under reasonable assumptions, probability of a collision is about $1 / Q$.

Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $M N$).

Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Poor worst-case guarantee.
- Requires backup.

Q. How would you extend Rabin-Karp to efficiently search for any one of P possible patterns in a text of length N?

Substring search cost summary

Cost of searching for an M-character pattern in an N-character text.

algorithm	version	operation count: guarantee	operation count: typical	backup in input?	correct?	extra space
brute force	—	MN	1.1 N	*yes*	*yes*	1
Knuth-Morris-Pratt	*full DFA* (*Algorithm 5.6*)	$2N$	1.1 N	*no*	*yes*	MR
	mismatch transitions only	$3N$	1.1 N	*no*	*yes*	M
Boyer-Moore	*full algorithm*	$3N$	N / M	*yes*	*yes*	R
	mismatched char heuristic only (*Algorithm 5.7*)	MN	N / M	*yes*	*yes*	R
Rabin-Karp†	*Monte Carlo* (*Algorithm 5.8*)	$7\ N$	$7\ N$	*no*	*yes* †	1
	Las Vegas	$7\ N$ †	$7\ N$	*yes*	*yes*	1

† *probabilisitic guarantee, with uniform hash function*