Substring Search

Substring Search

- Goal : Find pattern of length *M* in a text of length *N*

Application

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

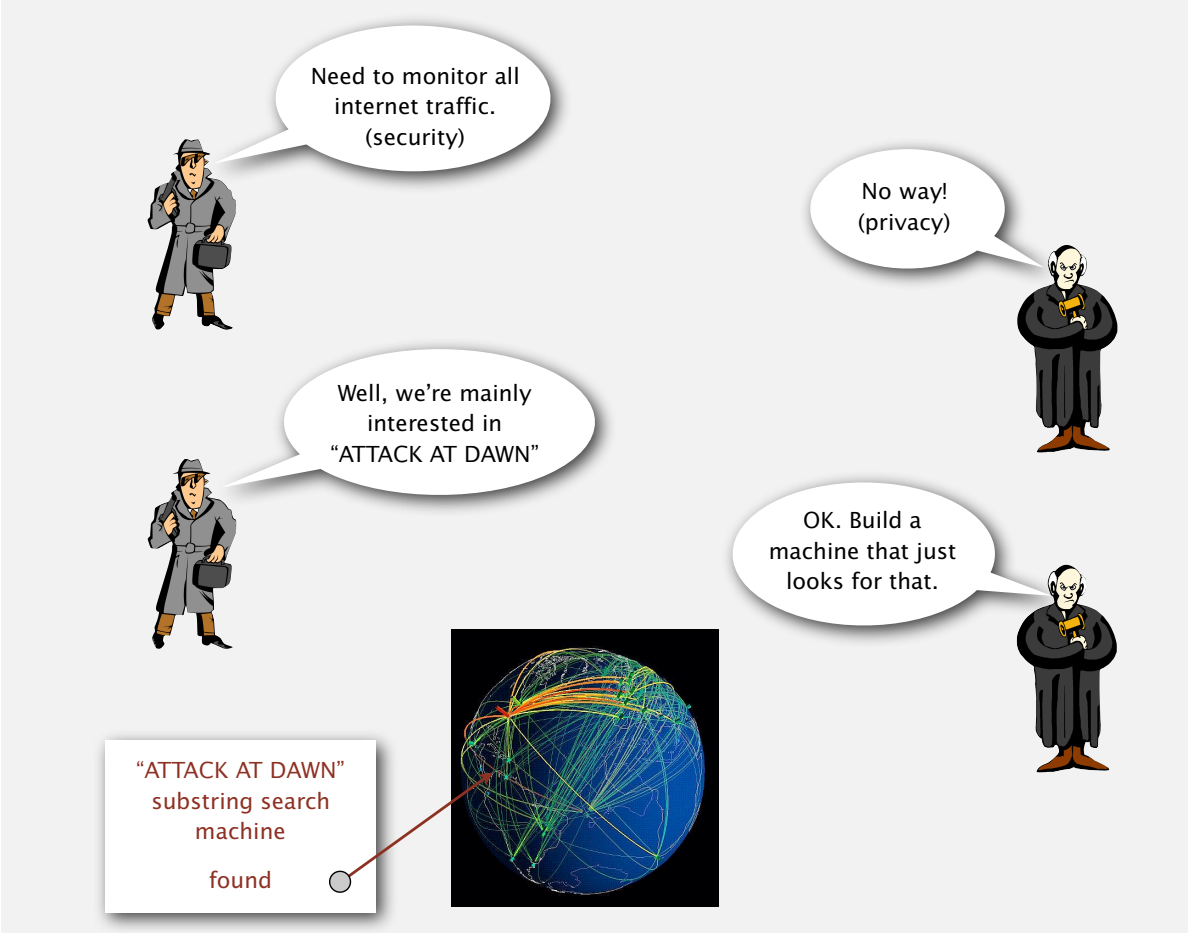
Application

- *Spam Filtering:* Identify patterns indicative of spam.

```
• There is no catch.
• L0W M0RTGAGE RATES
• This is a one-time mailing.
• This message is sent in compliance with
  spam regulations.
```

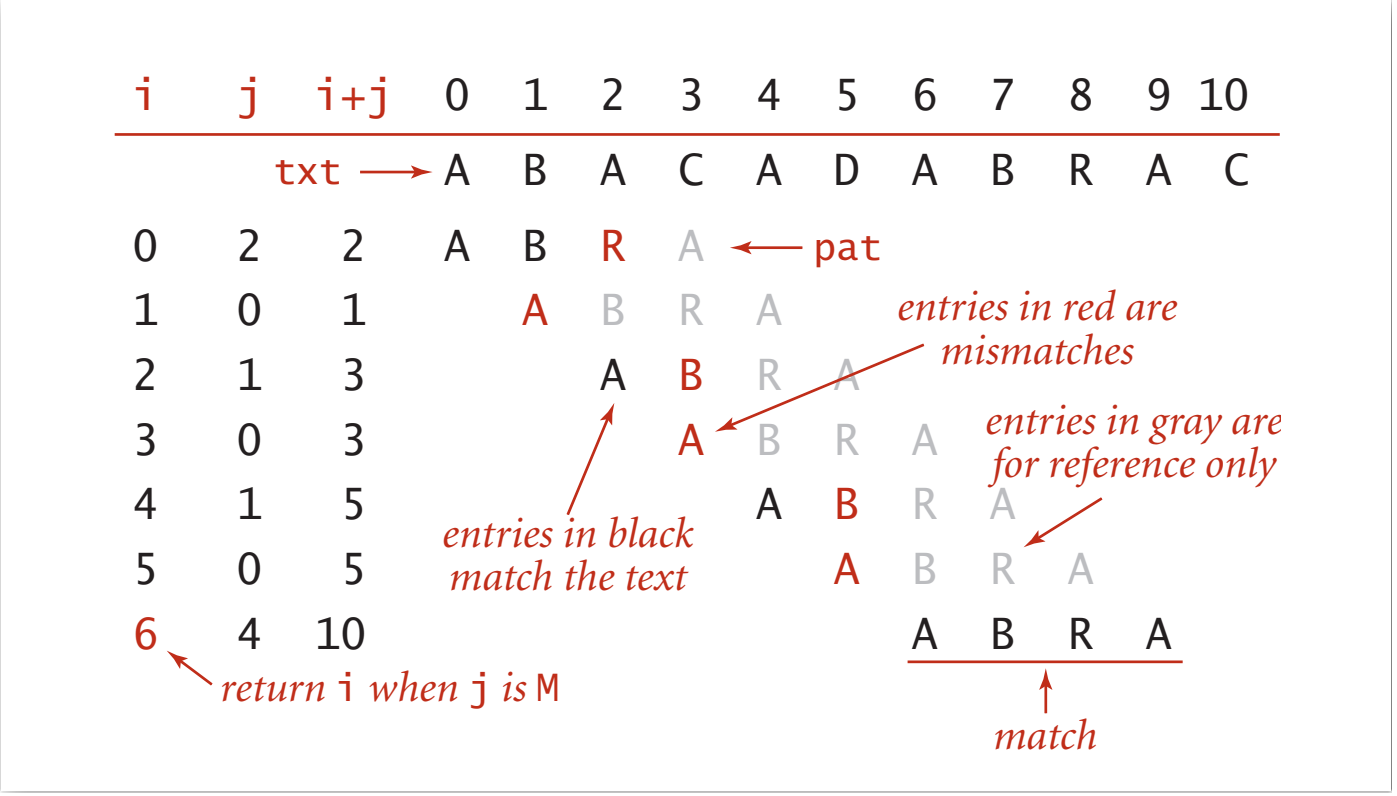
Application

- *Electronic Surveillance*



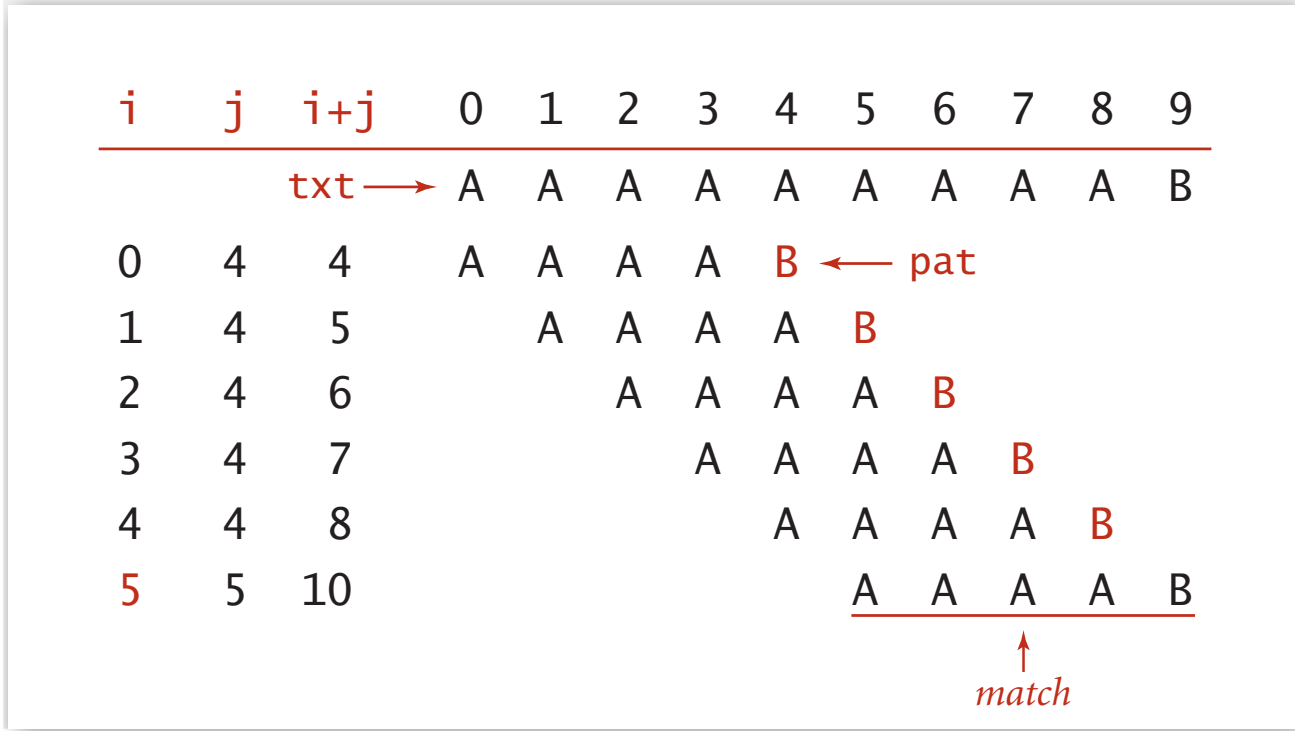
Brute-force substring search

- Check for pattern starting at each text position.



Brute-force substring search : worst case

- Brute-force algorithm can be slow if text and pattern are repetitive.
- Worst case happens when each time all m-1 characters match and the last one does not
- *character compare :*
  - *typical:* N
  - *worst case :* (N – M + 1) * M
  - *worst case :* N* M

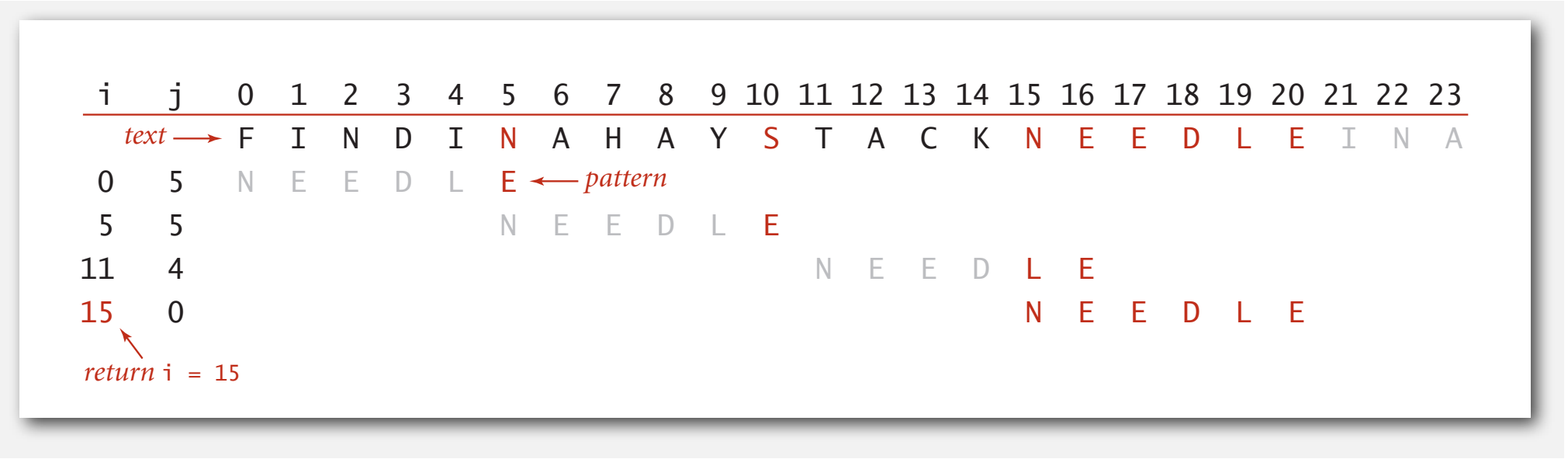


Brute-force substring search

```
int search( char PAT[], char TEXT[])
{
    int i, j, N, M;
    N = strlen(TEXT);
    M = strlen(PAT);
    for( i = 0;  i <= N-M ; i++)
    {
                    j = 0;
                    while(( j < M ) && (PAT[ j ] == TEXT[ i+j ] ))
                                    j++;
                     if(j==M)
                                    return i;  // index in text where pattern starts
     }
     return -1;    // not found
}
```

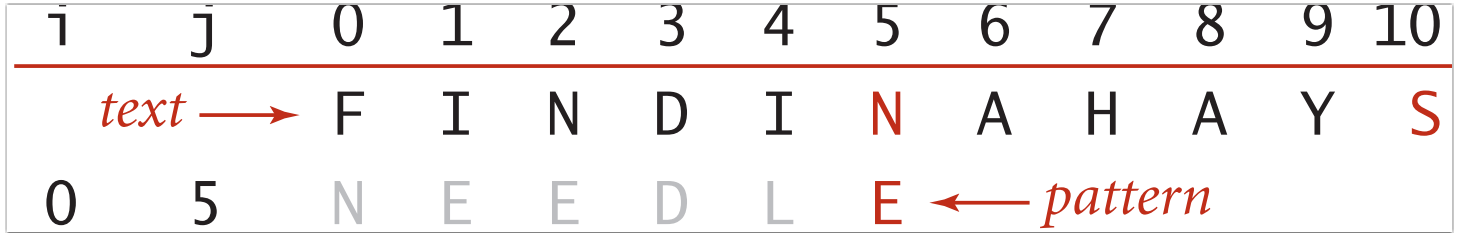
Sublinear search algorithms

- move right to left in pattern ←
- shift left to right in text →
- Search time proportional to N/M for typical problems



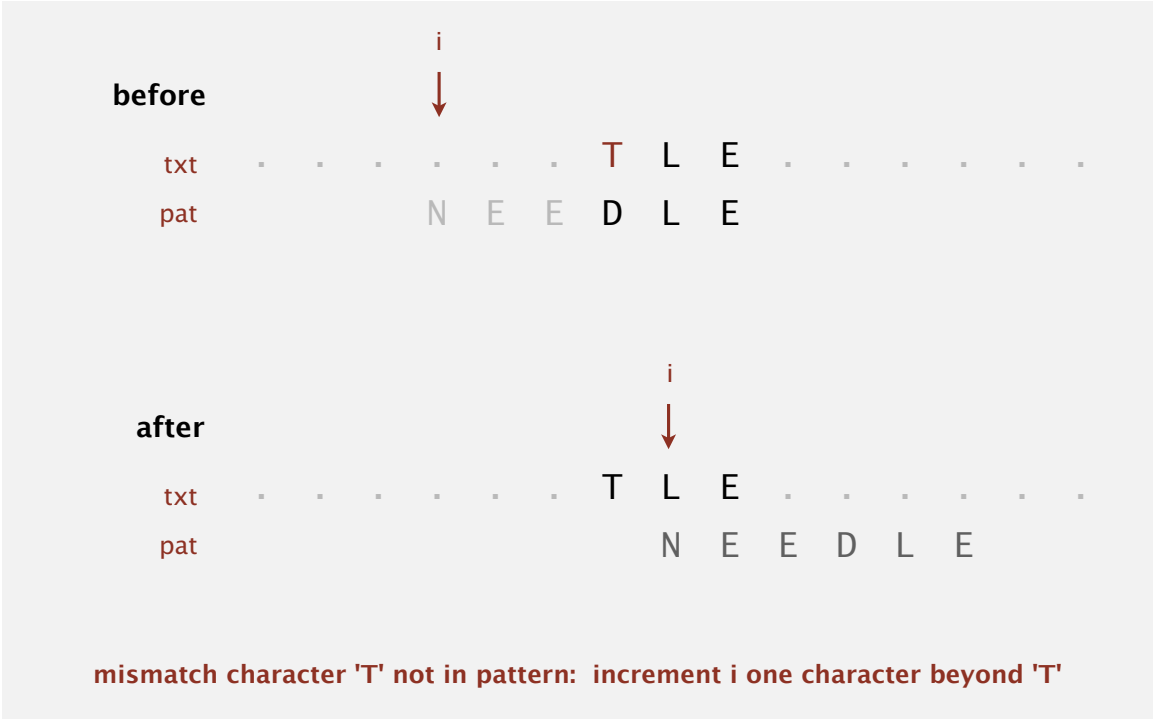
Boyer-Moore Algorithm

```
.......
i= 0;
J = M – 1 ;
........
while ( ( j >= 0 ) && ( p[j] == t[i+j] ) )    //scan text from right to left
{
        j--;
}
.......
```



Mismatch character - How much to skip ?

- *Case 1 :* Mismatch character not in pattern
  i= i + (j + 1)



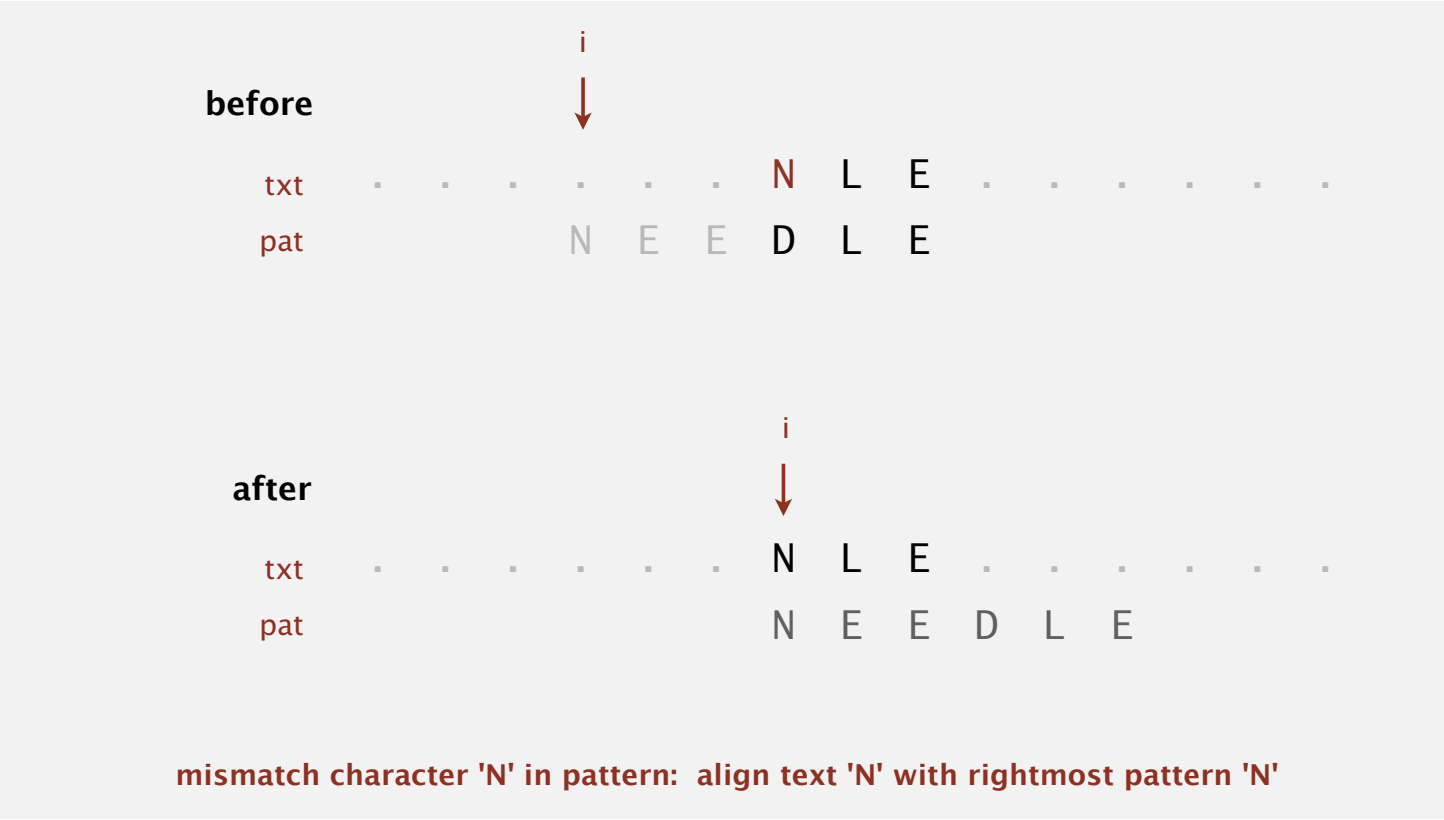
5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I
N	E	E	D	L	E											
						N	E	E	D	L	E					

before : i = 8, j = 3
after : i = 8 + 3 + 1 = 12

Mismatch character - How much to skip ?

- *Case 2-a :* Mismatch character in the pattern

  i = i + (j – rightMostOccurenceInPATTERN)



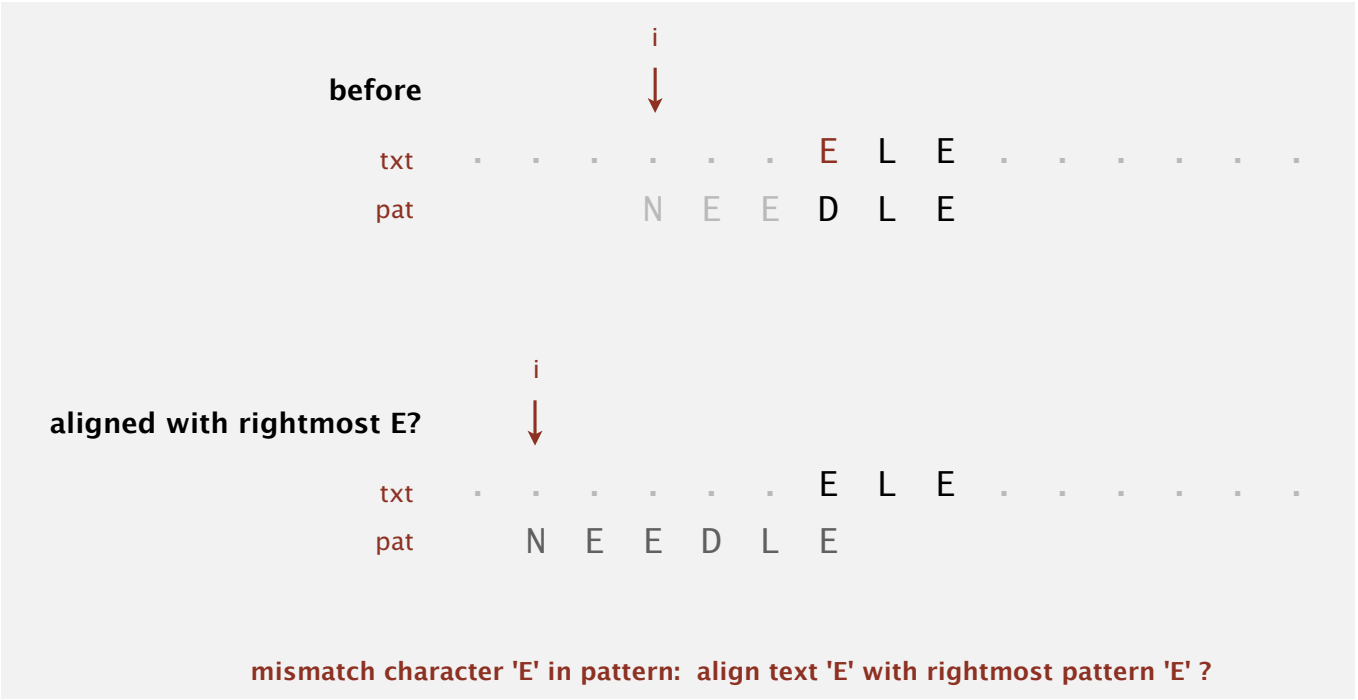
0	1	2	3	4	5	6	7	8	9	10
F	I	N	D	I	N	A	H	A	Y	S
N	E	E	D	L	E	← *pattern*				
					N	E	E	D	L	E

before : i=6, j = 3
after : i = 6 +3 – 0 = 9

Mismatch character - How much to skip ?

- *Case 2-b :* Mismatch character in pattern, but skip is negative ??

i = i +(j – rightMostOccurenceInPATTERN)

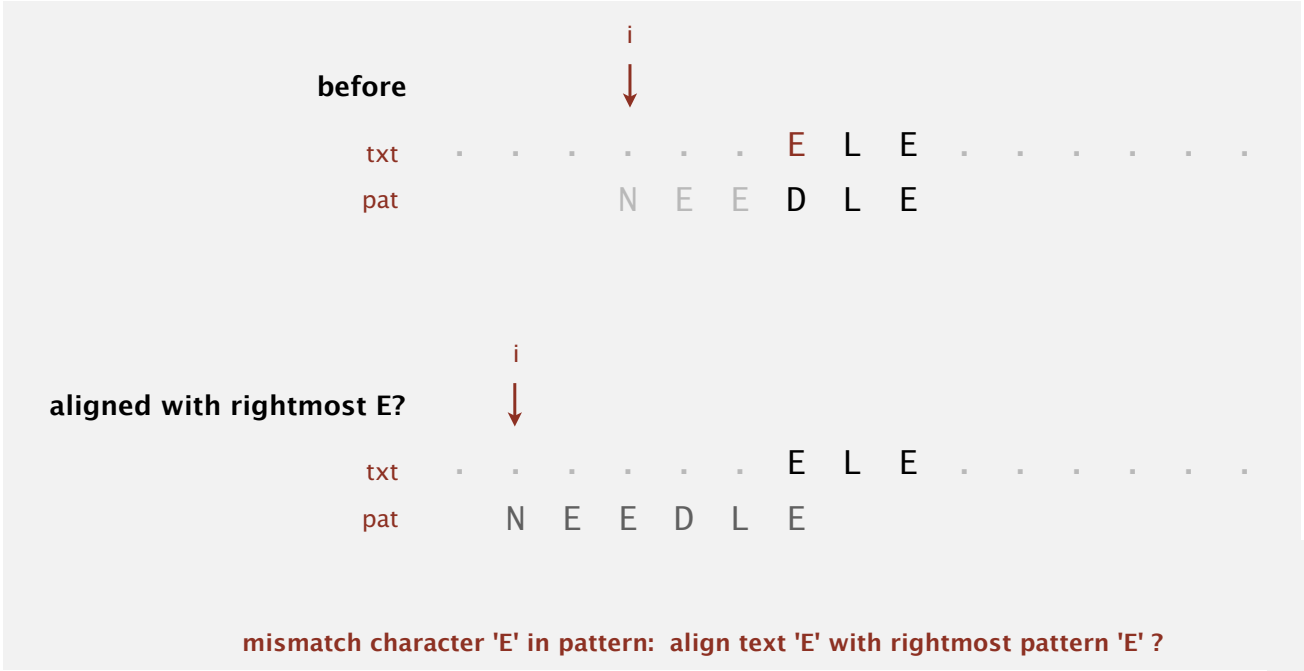


before : i= 16, j= 3

after : i = 16 + (3 – 5) ?

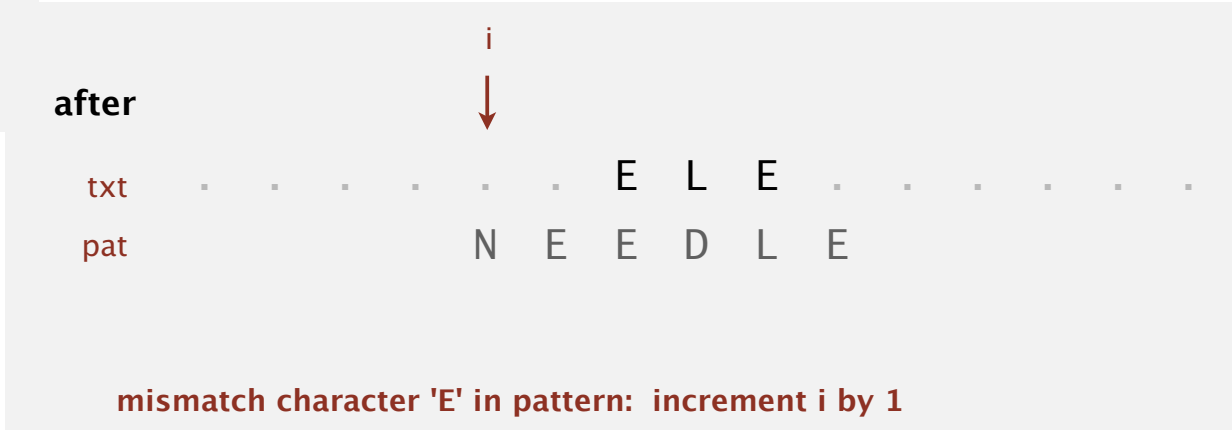
Boyer-Moore Algorithm - How much to skip ?

- *Case 2-b :* Mismatch character in pattern, but shift would not increase i



just increment i by 1

i = i +1



Boyer-Moore Algorithm- How much to skip ?

- Do not search every time for an occurrence of a character in the pattern
- Precompute index of rightmost occurrence of character **c** in pattern

NEEDLE →

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
			3	5							4		0												

Skip Table Computation Algorithm

```
void shiftTable(int PAT[], int right[], int offset, int size)
{
      int i, M = strlen(PAT);
      for(i=0; i <size ; i++)
                  right[i] = -1;     // -1 if character not in pattern
      for(i=0; i<M; i++)
                  right[P[i]-offset] = i ;  // offset is  'A' for this example
}
```

NEEDLE →

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
			3	5							4		0												

Skip Table Computation Algorithm

		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

Boyer-Moore skip table computation

Boyer Moore Algorithm

```
int search( char PAT[], char TEXT[])
{
    int i, j, skip=0,  N = strlen(TEXT), M = strlen(PAT);
    for( i = 0;  i <= N-M ; i += skip)
    {
            j = M – 1;
            while(( j  >= 0 ) && (PAT[ j ] == TEXT[ i+j ] ))
                    j--;
             if(j >= 0)
            {
                    skip = j – right [ TEXT[ I + j ]];
                     if( skip <  0)
                                skip = 1;
            }
                else
                    return i;  // index in text where pattern starts
      }
     return -1;     // not found
}
```

Example

```
if(j >= 0)
 {
        skip = j – right [ TEXT[ I + j ]];
        if( skip <  0)
                skip = 1;
 }
```

		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

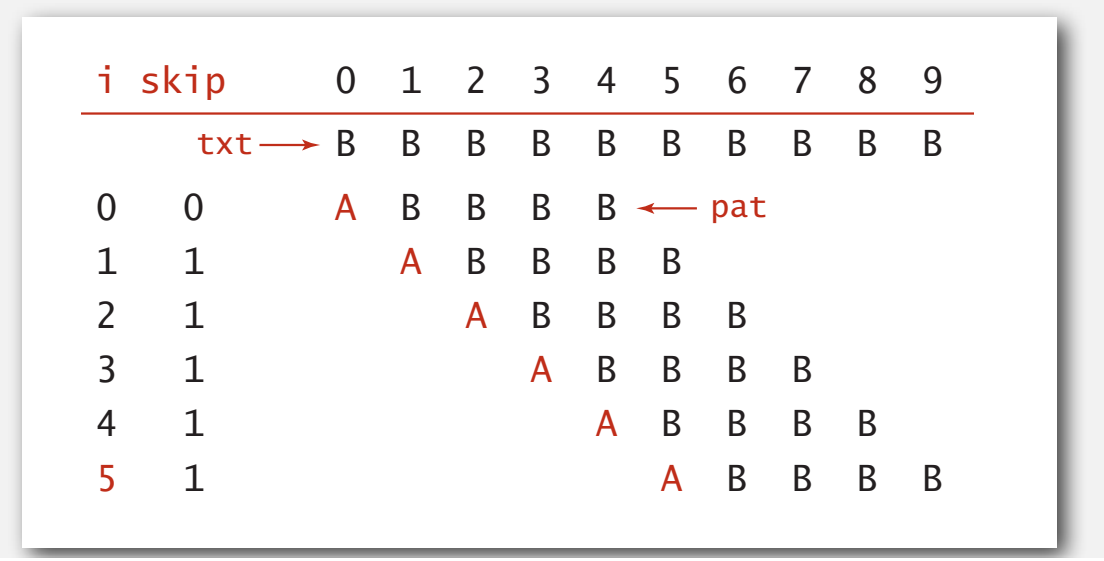
Boyer-Moore skip table computation

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	text →	F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E	← *pattern*																	
5	5						N	E	E	D	L	E													
11	4												N	E	E	D	L	E							
15	0																N	E	E	D	L	E			

return i = 15

Boyer - Moore Analysis

- Substring search with the Boyer-Moore mismatched character heuristic takes about ~ *N / M* character compares to search for a pattern of length *M* in a text of length *N*.
- worst case : N * M



Rabin – Karp Fingerprint Algorithm

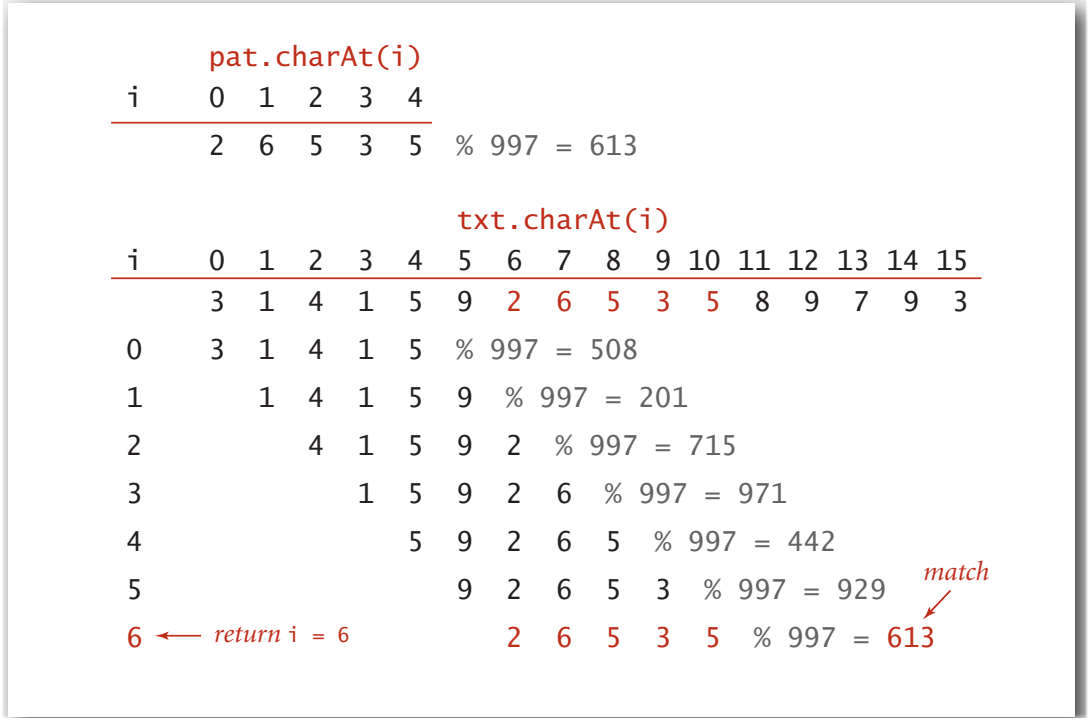
- We want to convert an M-digit base-R number to an int value between 0 to Q-1.
- Modular Hashing :

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

t : text M : length of text R : constant integer

Rabin – Karp Fingerprint Algorithm

- 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



Compute hash for M digit key

```
int hash(char PAT[], int R, int Q, int offset)
{
        int i, h = 0, M = strlen(PAT);
        for(i=0; i<M; i++)
                h = ( R * h + (PAT[i] – offset)) ;
        h = h % Q ;
        return h;
}
```

Efficiently computing the hash function

- How to efficiently compute *x[i+*1] given that we know *x[i]*.

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

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

Efficiently computing the hash function

- 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 idea : Can update hash function in constant time!

Efficiently computing the hash function

- Key idea : Can update hash function in constant time!

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

can precompute

current value — subtract leading digit — multiply by radix — add new trailing 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
                                                                    match
 9                   9  2  6  5  3  % 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929
10 ← return i-M+1 = 6    2  6  5  3  5  % 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613
```

Rabin – Karp Fingerprint Algorithm 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*.

Rabin – Karp Fingerprint Algorithm Analysis

- Monte Carlo version : Return match if hash match.
  - Always runs in linear time N
  - Extremely likely to return correct answer (but not always!).
- Las Vegas version : Check for substring match if hash match; continue search if false collision
  - Always returns correct answer.
  - Extremely likely to run in linear time N (but worst case is *M*N*).

Summary

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1\,N$	*yes*	*yes*	1
Boyer-Moore	*full algorithm*	$3\,N$	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