Pattern Matching Algorithms

Amit Kr Mandal, PhD

Brute-force search

- The aim: we would like to construct an algorithm thats capable of finding a pattern in a text
- Brute-force search is the naive approach ~ intuitive !!!
- Keep iterating through the text and if there is a mismatch we shift the pattern one step to the right
- Not so efficient especially when there are lots of matching prefixes
- For example: pattern is **DDDDE** and the text is **DDDDDDDDDDDE**
- Main problem: needs backup for every mismatch… if there is a mismatch we jump back to the next character (!!!)
- Lots of compares: ~ **N*M** where **N** is the length of text, **M** is the length of the pattern we are looking for
- Linear time guarantee would be better !!!

Brute-force search

T	H	I	S		I	S		A		T	E	S	T

Brute-force search

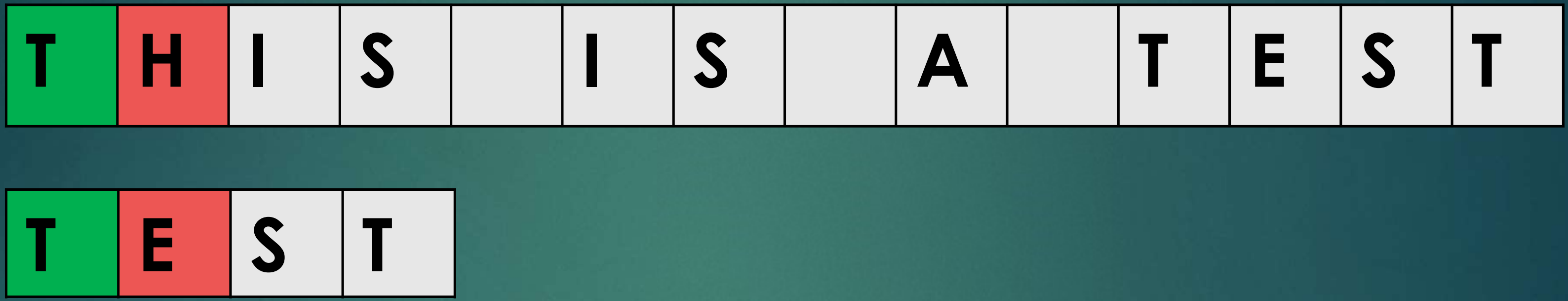
T	H	I	S		I	S		A		T	E	S	T
T	E	S	T										

Brute-force Search

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

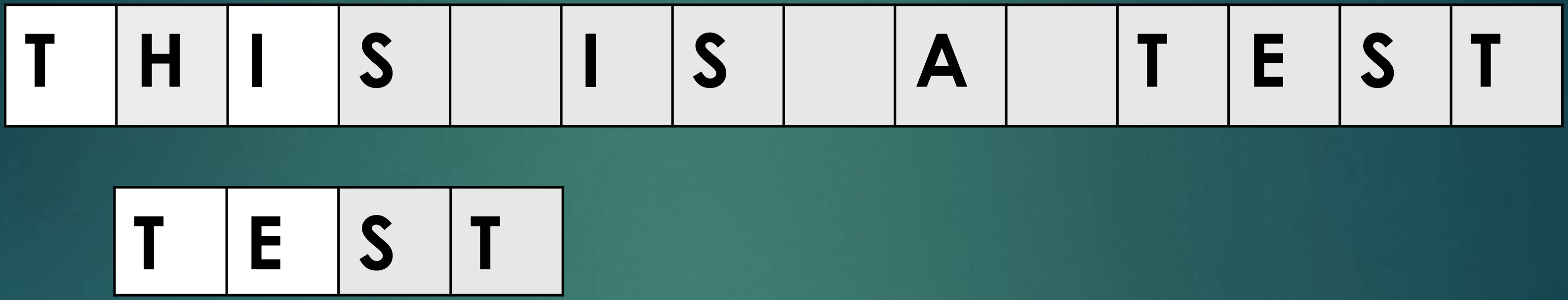
Brute-force Search



T	H	I	S		I	S		A		T	E	S	T

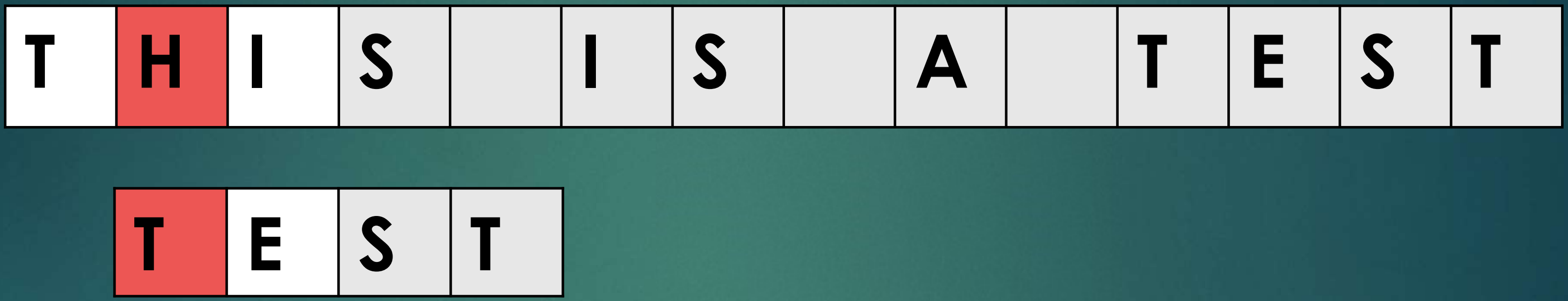
T	E	S	T

Brute-force Search



T	H	I	S		I	S		A		T	E	S	T
	T	E	S	T									

Brute-force Search

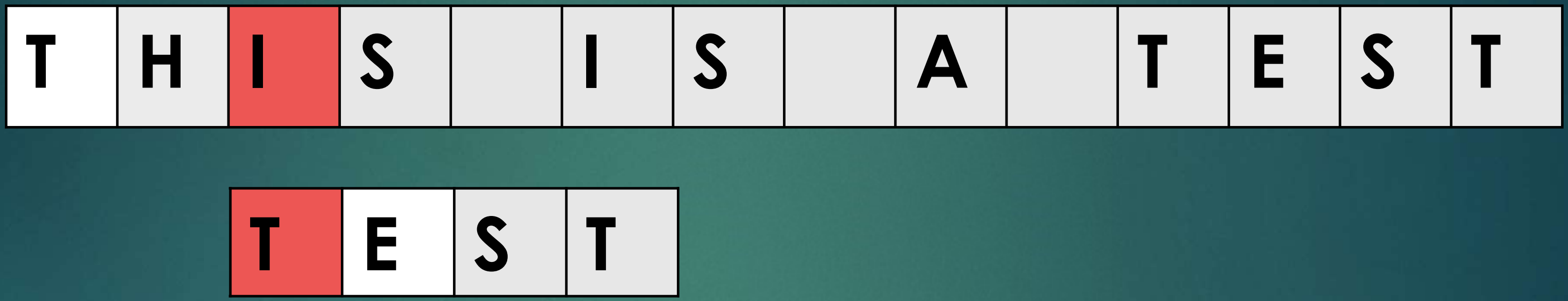


T	H	I	S		I	S		A		T	E	S	T
	T	E	S	T									

Brute-force Search

T	H	I	S		I	S		A		T	E	S	T
		T	E	S	T								

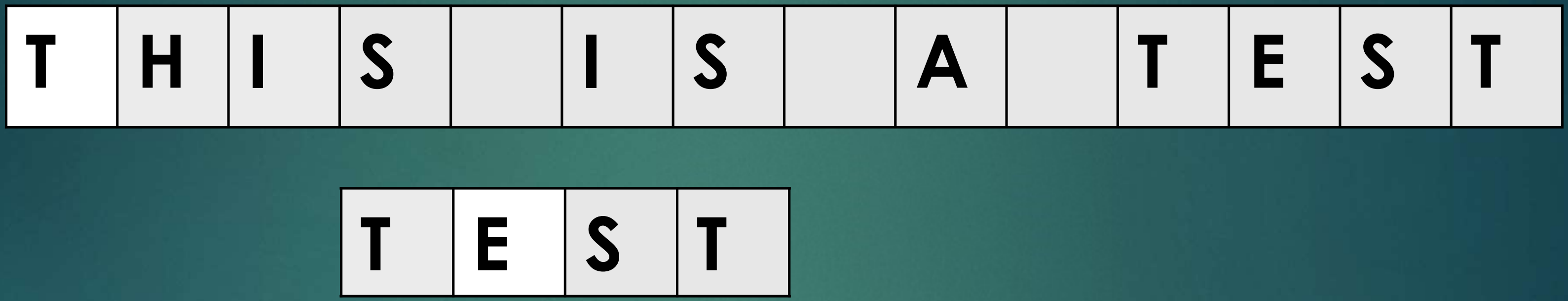
Brute-force Search



T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

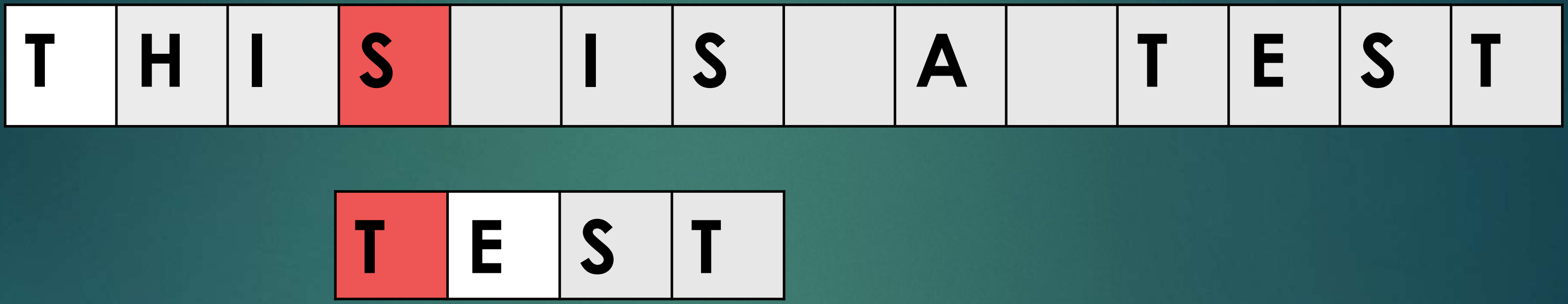
Brute-force Search



T H I S   I S   A   T E S T

T E S T

Brute-force Search

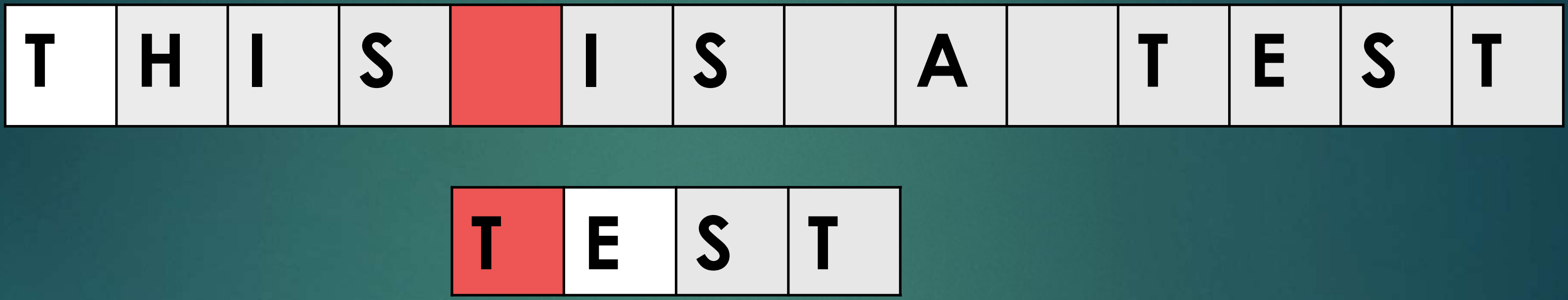


T	H	I	S		I	S		A		T	E	S	T
			T	E	S	T							

Brute-force Search

T	H	I	S		I	S		A		T	E	S	T
				T	E	S	T						

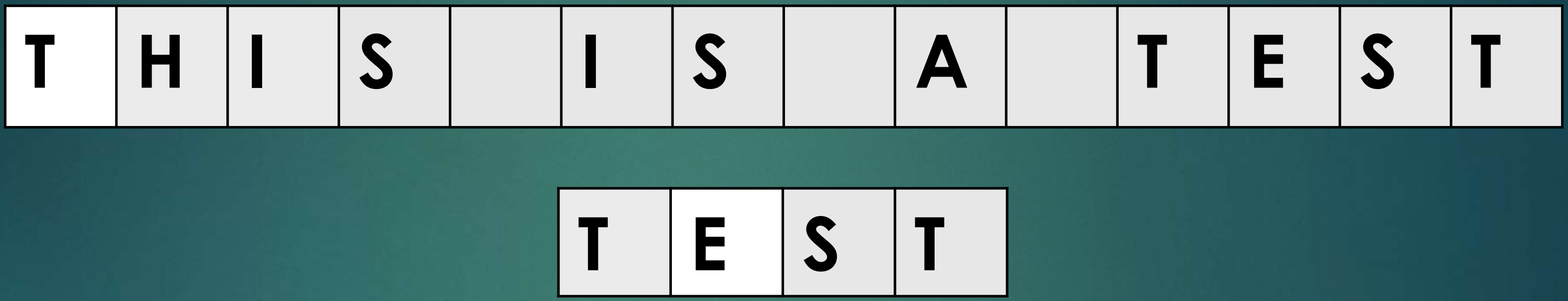
Brute-force Search



T	H	I	S		I	S		A		T	E	S	T

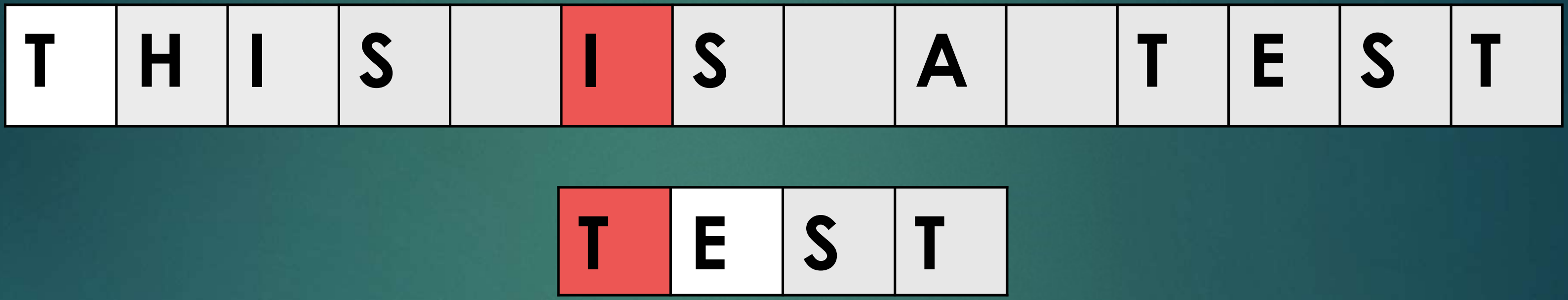
				T	E	S	T

Brute-force Search



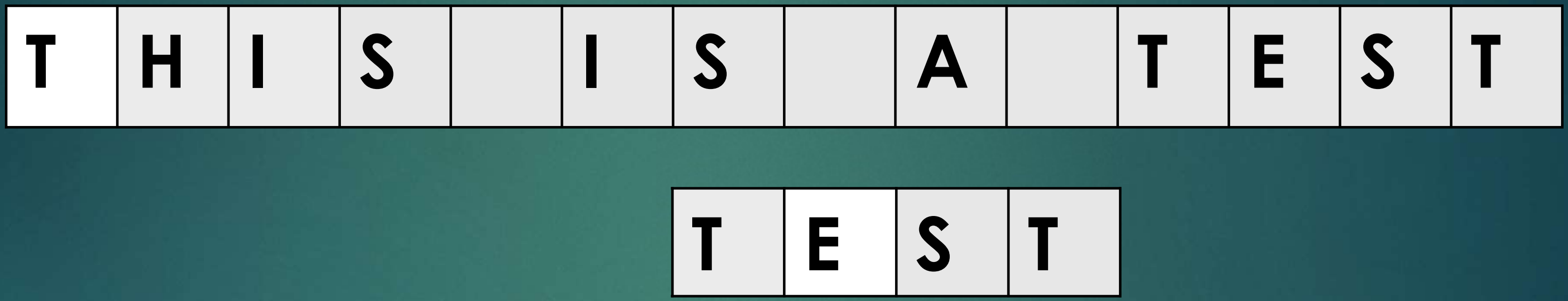
T	H	I	S		I	S		A		T	E	S	T
					T	E	S	T					

Brute-force Search



T	H	I	S		I	S		A		T	E	S	T
					T	E	S	T					

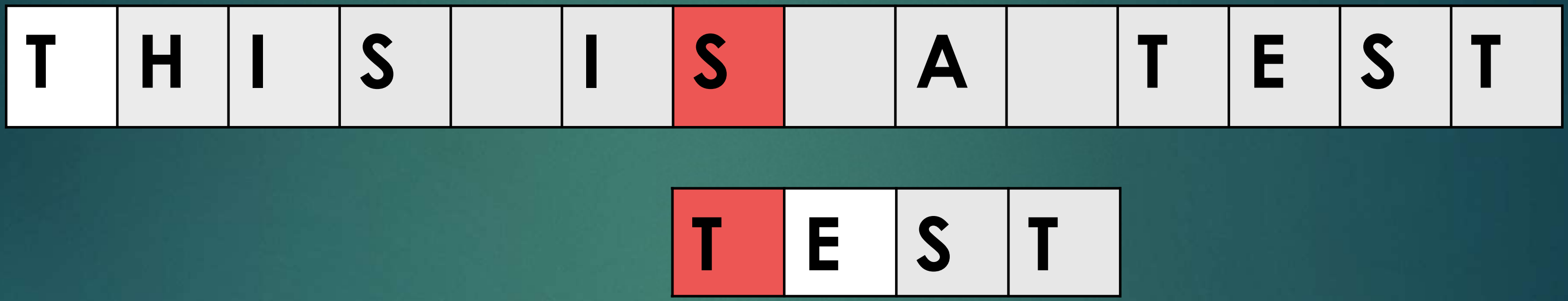
Brute-force Search



T H I S   I S   A   T E S T

T E S T

Brute-force Search

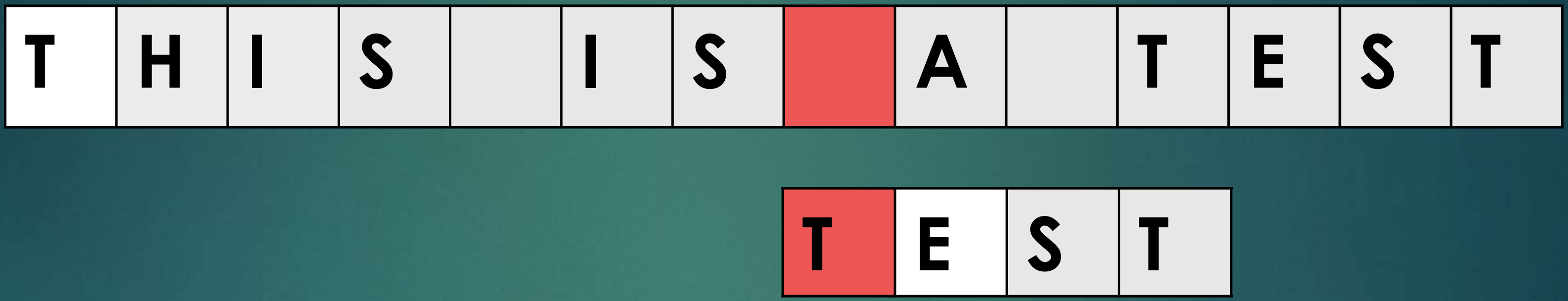


T	H	I	S		I	S		A		T	E	S	T
						T	E	S	T				

Brute-force Search

T	H	I	S		I	S		A		T	E	S	T
							T	E	S	T			

Brute-force Search



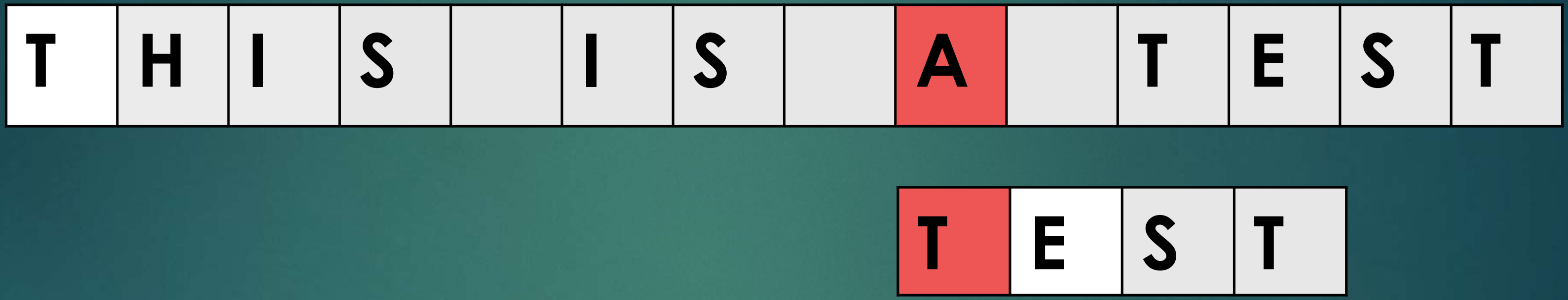
T	H	I	S		I	S	🟥	A		T	E	S	T

							🟥 T	E	S	T			

Brute-force Search

T	H	I	S		I	S		A		T	E	S	T
								T	E	S	T		

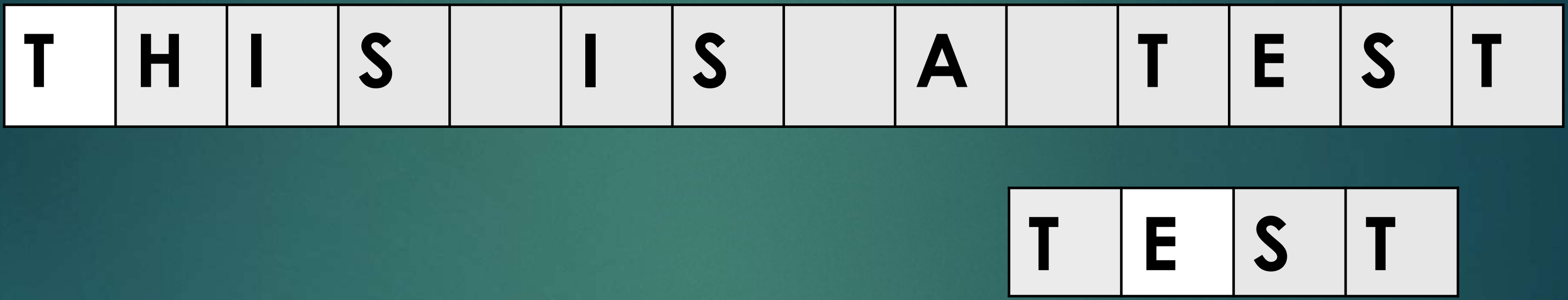
Brute-force Search



T H I S I S A T E S T

T E S T

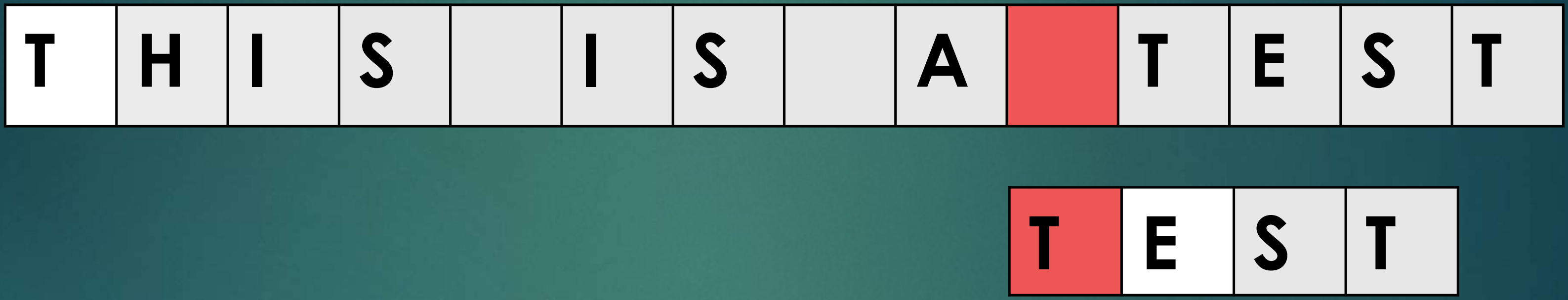
Brute-force Search



T H I S I S A T E S T

T E S T

Brute-force Search



T	H	I	S		I	S		A		T	E	S	T
									T	E	S	T	

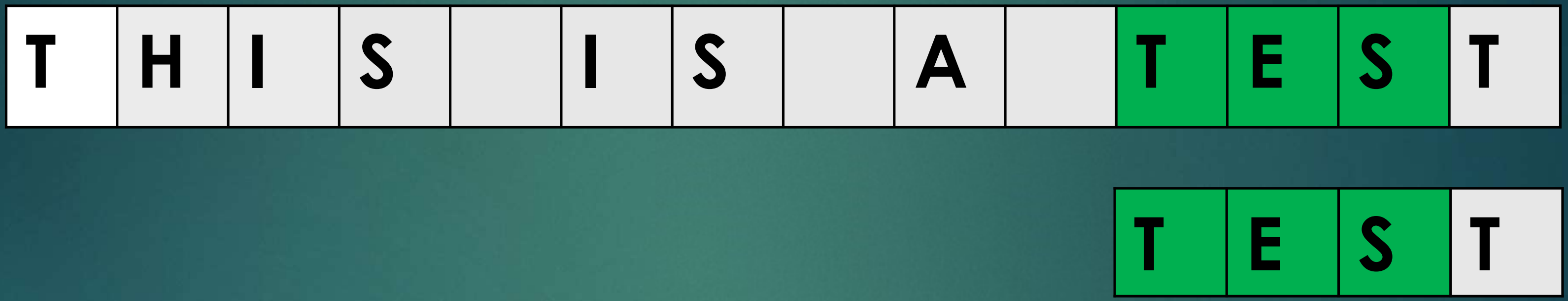
Brute-force Search

T	H	I	S		I	S		A		T	E	S	T
										T	E	S	T

Brute-force Search

T	H	I	S		I	S		A		T	E	S	T
										T	E	S	T

Brute-force Search



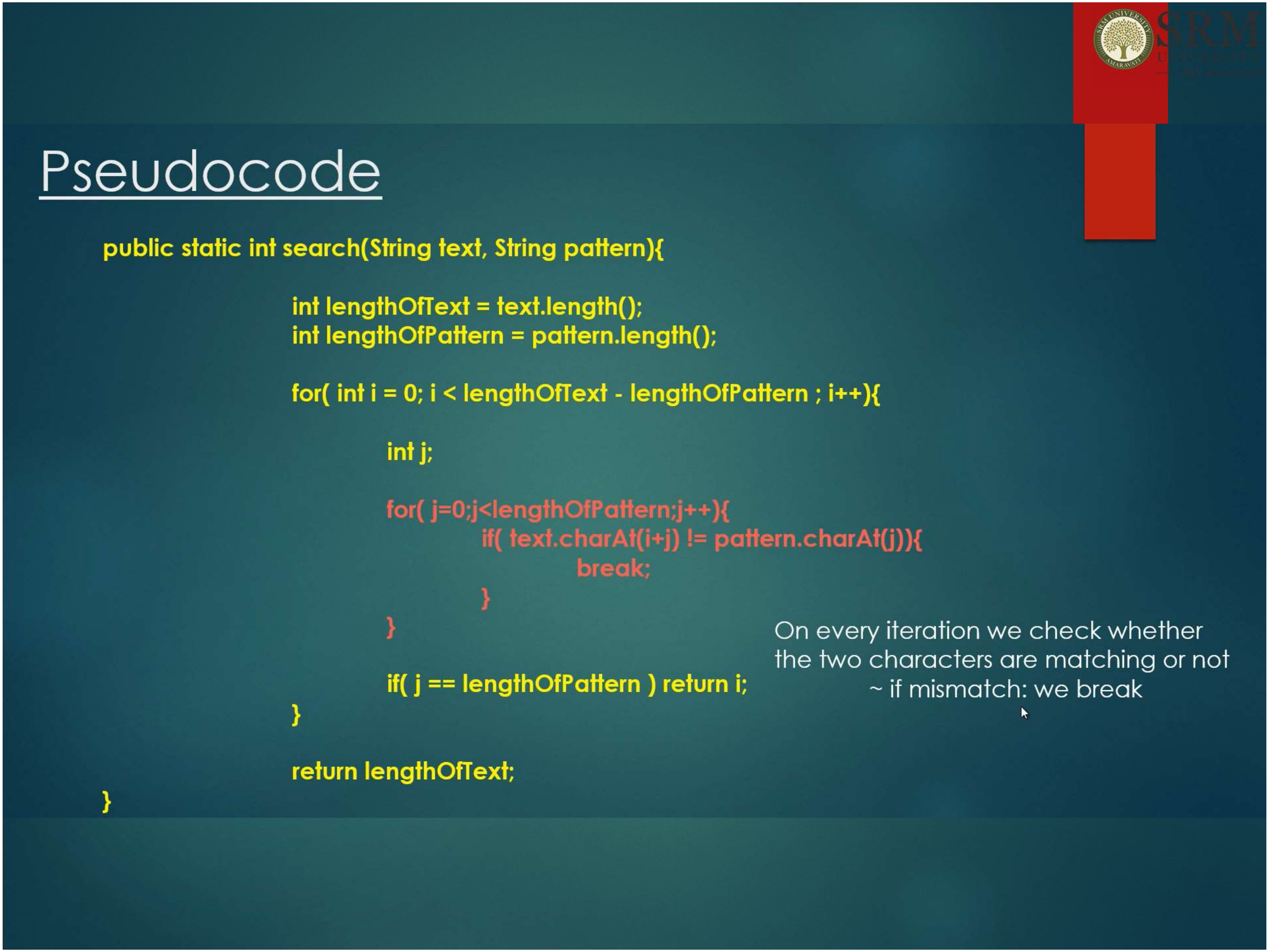
T	H	I	S		I	S		A		T	E	S	T
										T	E	S	T

Brute-force Search

T	H	I	S		I	S		A		T	E	S	T
										T	E	S	T

Pseudocode

```
public static int search(String text, String pattern){

        int lengthOfText = text.length();
        int lengthOfPattern = pattern.length();

        for( int i = 0; i < lengthOfText - lengthOfPattern ; i++){

            int j;

            for( j=0;j<lengthOfPattern;j++){
                if( text.charAt(i+j) != pattern.charAt(j)){
                    break;
                }
            }

            if( j == lengthOfPattern ) return i;
        }

        return lengthOfText;
}
```



Pseudocode

```
public static int search(String text, String pattern){

        int lengthOfText = text.length();
        int lengthOfPattern = pattern.length();

        for( int i = 0; i < lengthOfText - lengthOfPattern ; i++){

            int j;

            for( j=0;j<lengthOfPattern;j++){
                if( text.charAt(i+j) != pattern.charAt(j)){
                    break;
                }
            }

            if( j == lengthOfPattern ) return i;
        }

        return lengthOfText;
}
```

On every iteration we check whether the two characters are matching or not
~ if mismatch: we break

Pseudocode

```
public static int search(String text, String pattern){

        int lengthOfText = text.length();
        int lengthOfPattern = pattern.length();

        for( int i = 0; i < lengthOfText - lengthOfPattern ; i++){

            int j;

            for( j=0;j<lengthOfPattern;j++){
                if( text.charAt(i+j) != pattern.charAt(j)){
                    break;
                }
            }

            if( j == lengthOfPattern ) return i;
        }

        return lengthOfText;
}
```

It means we have found the pattern in the text: because no mismatching character has been found !!!

Problem

D	D	D	D	D	D	S

D	D	D	S

Problem

D	D	D	D	D	D	S

D	D	D	S

Problem

D	D	D	D	D	D	S

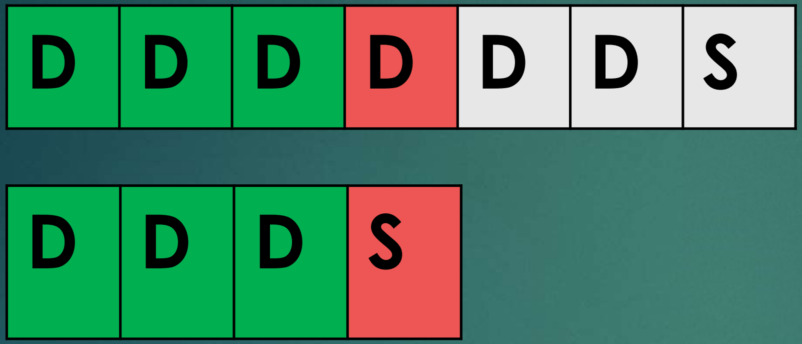
D	D	D	S

Problem

D	D	D	D	D	D	S

D	D	D	S

Problem



D	D	D	D	D	D	S

D	D	D	S

Problem

D	D	D	D	D	D	S

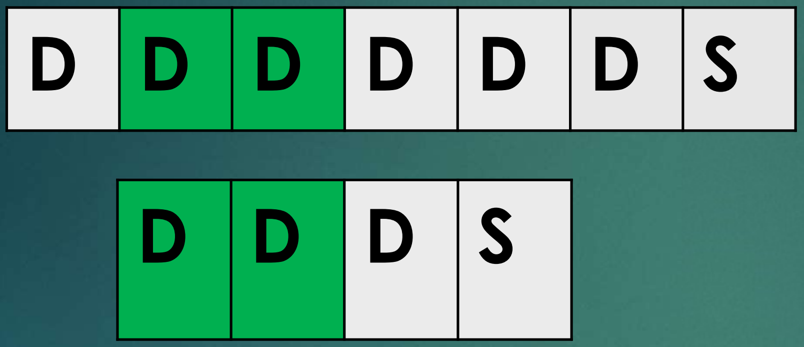
D	D	D	S

Problem

D	D	D	D	D	D	S

D	D	D	S

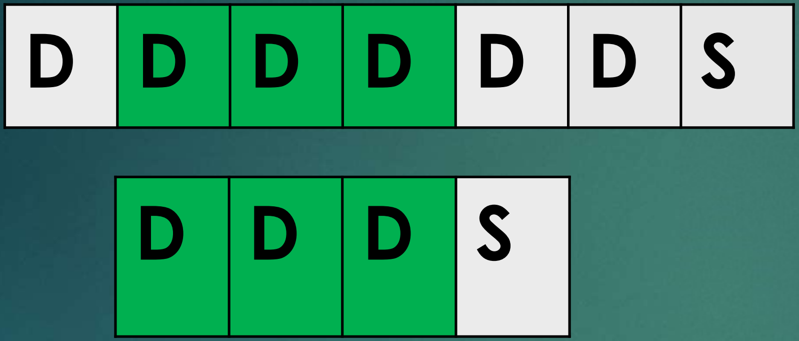
Problem



D	D	D	D	D	D	S

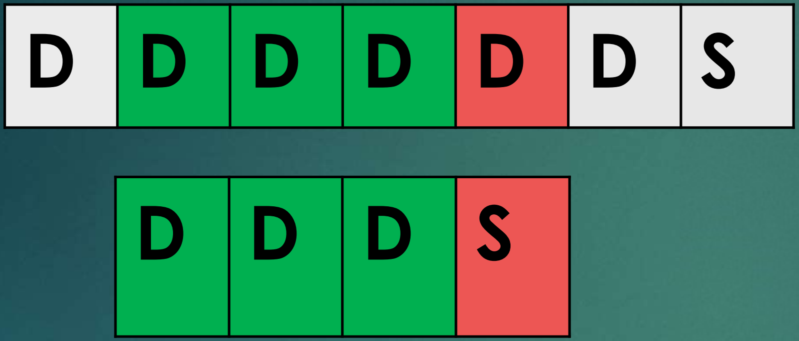
D	D	D	S

Problem



D	D	D	D	D	D	S
	D	D	D	S		

Problem



D	D	D	D	D	D	S

D	D	D	S

Problem

D	D	D	D	D	D	S

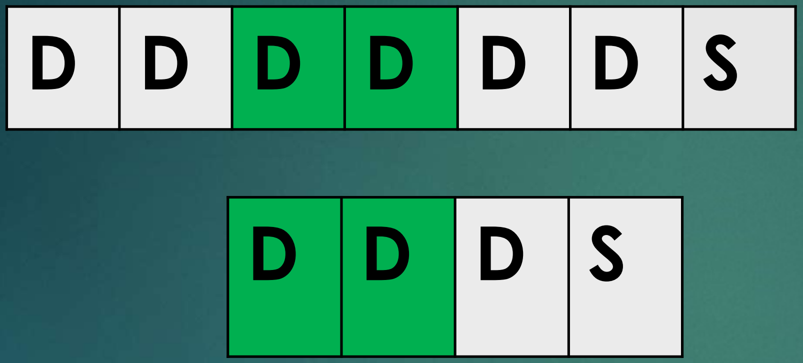
D	D	D	S

Problem

D	D	D	D	D	D	S

D	D	D	S

Problem



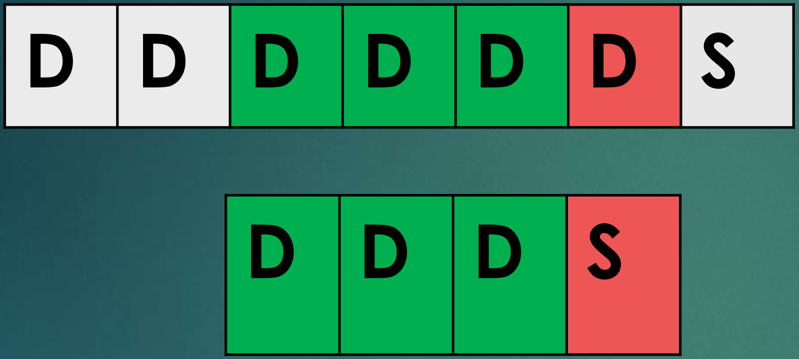
D	D	D	D	D	D	S

D	D	D	S

Problem

D	D	D	D	D	D	S
		D	D	D	S	

Problem



D	D	D	D	D	D	S
		D	D	D	S	

Problem

D	D	D	D	D	D	S

D	D	D	S

Problem

D	D	D	D	D	D	S
			D	D	D	S

Problem

D	D	D	D	D	D	S
			D	D	D	S

Problem

D	D	D	D	D	D	S
			D	D	D	S

Problem

D	D	D	D	D	D	S
			D	D	D	S

Boyer-Moore Algorithm

Boyer-Moore search

- Problem with brute force search → we keep considering too many bad options as well ~ maybe we can eliminate a lot of possibilities
- Thats why Boyer-Moore algorithm came to be
- Very efficient string search algorithm
- The algorithm needs to preprocess the pattern, but not the whole text !!!
- The algorithm runs faster as the length of the pattern increases
- The key features of the algorithm are to match on the tail of the pattern rather than the head
- Why is it good? We can skip multiple characters at the same time rather than searching every single character in the text

Boyer-Moore search

- We have to construct a „**bad match table**": this is the preprocessing stage
- This table never has elements smaller than **1**
- Keep comparing the pattern to the text starting from the rightmost character in the pattern
- When mismatch occurs we have to shift the pattern to the right corresponding to the value in the „**bad match table**"
- **WHY?**
- Because we can skip several characters unlike brute-force search → the algorithm will be faster !!!

„bad match table”

- Make a table of the characters
- Make sure the table does not contains repetitive characters // if there is several **a** letters in the pattern, the bad table only contains one **a** letter
- **Max(1, lengthOfPattern-actualIndex-1)** // this is the formula we use
- We iterate over the pattern and compute the values to the bad match table → we keep updating the old values for the same characters !!!

„bad match table”

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values				

„bad match table”

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values	3			

„bad match table"

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values	3			

„bad match table”

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values	3	2		

„bad match table”

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values	3	2		

„bad match table”

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values	3	2	1	

„bad match table"

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values	1	2	1	

„bad match table”

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

max(1, lengthOfPattern – indexOfActualCharacter – 1)

Letters	T	E	S	*
Values	1	2	1	4

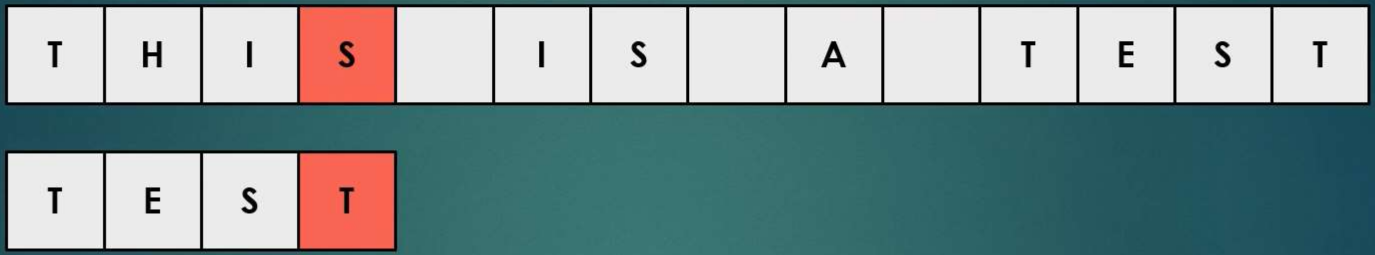
Boyer-Moore search

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search



T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

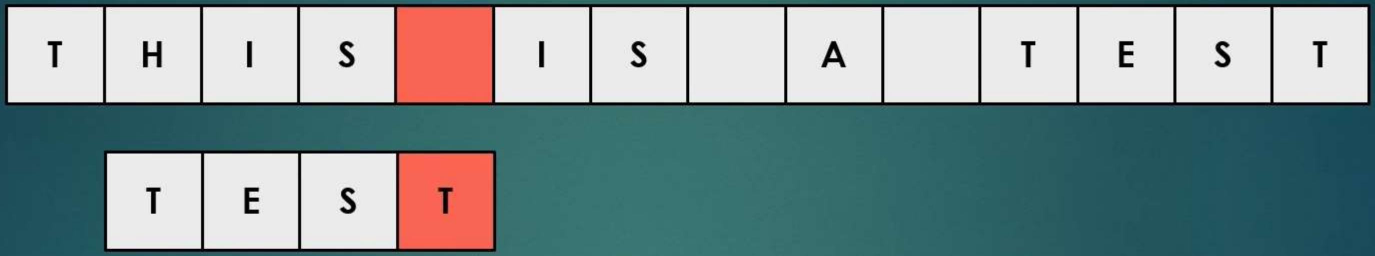
Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search

T	H	I	S		I	S		A		T	E	S	T
	T	E	S	T									

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search



T H I S   I S   A   T E S T

T E S T

Letters	T	E	S	*
Values	1	2	1	4

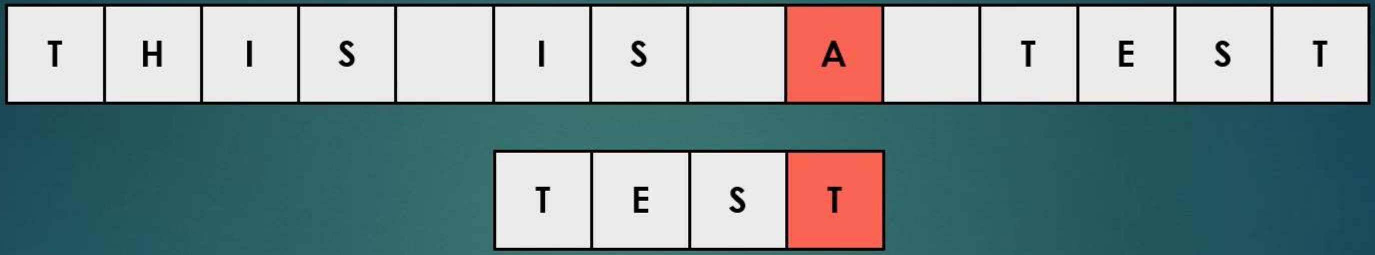
Boyer-Moore search

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search



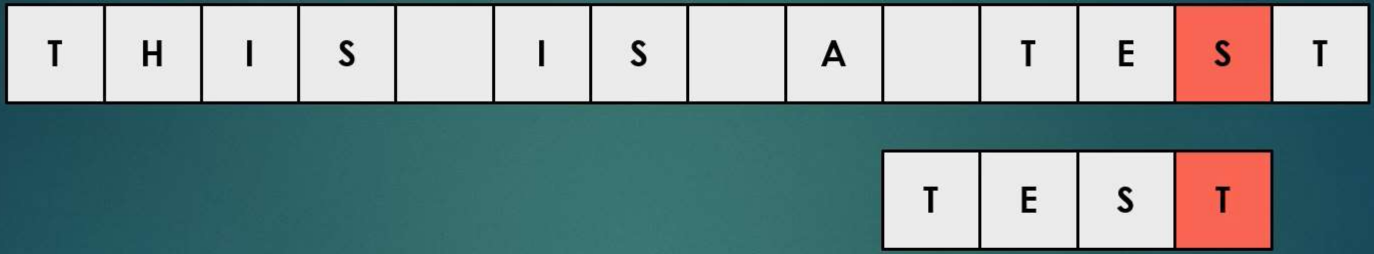
Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search

T	H	I	S		I	S		A		T	E	S	T
									T	E	S	T	

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search

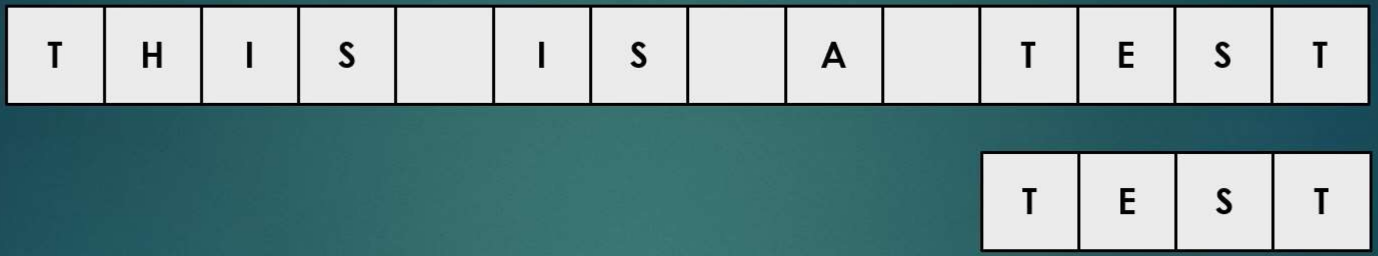


T H I S I S A T E S T

T E S T

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search

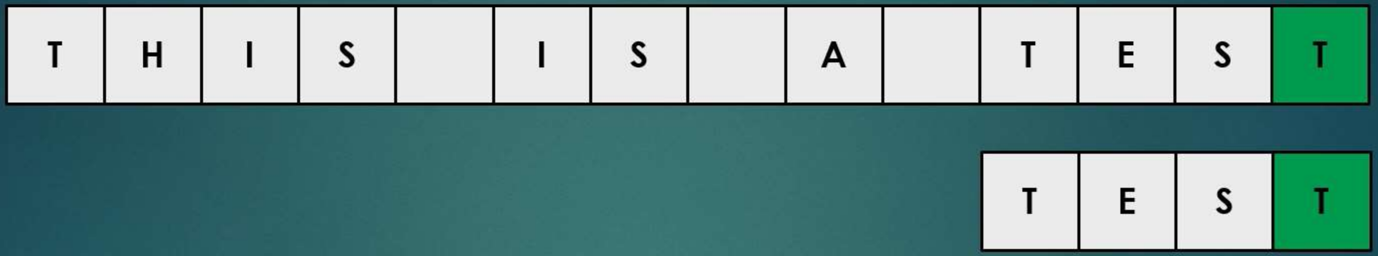


T H I S   I S   A   T E S T

T E S T

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search



T H I S I S A T E S T

T E S T

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search

T	H	I	S		I	S		A		T	E	S	T

T	E	S	T

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search

T	H	I	S		I	S		A		T	E	S	T
										T	E	S	T

Letters	T	E	S	*
Values	1	2	1	4

Boyer-Moore search

T	H	I	S		I	S		A		T	E	S	T
										T	E	S	T

Letters	T	E	S	*
Values	1	2	1	4

Pseudocode

```
public void precomputeShifts() {

                int lengthOfPattern = this.pattern.length();

                for (int index = 0; index < lengthOfPattern; index++) {
                        char actualCharacter = this.pattern.charAt(index);
                        int maxShift = Math.max(1, lengthOfPattern - index - 1);
                        this.mismatchShiftsTable.put(actualCharacter, maxShift);
                }
}
```

Pseudocode

```
public void precomputeShifts() {

        int lengthOfPattern = this.pattern.length();

        for (int index = 0; index < lengthOfPattern; index++) {
            char actualCharacter = this.pattern.charAt(index);
            int maxShift = Math.max(1, lengthOfPattern - index - 1);
            this.mismatchShiftsTable.put(actualCharacter, maxShift);
        }
}
```

We calculate the length of the pattern in advance

Pseudocode

```
public void precomputeShifts() {

        int lengthOfPattern = this.pattern.length();

        for (int index = 0; index < lengthOfPattern; index++) {
                char actualCharacter = this.pattern.charAt(index);
                int maxShift = Math.max(1, lengthOfPattern - index - 1);
                this.mismatchShiftsTable.put(actualCharacter, maxShift);
        }
}
```

We consider every character in the pattern and keep building up the „bad match table" as a hashtable !!!

Pseudocode

```
for (int i = 0; i <= lengthOfText - lengthOfPattern; i += numOfSkips) {

        numOfSkips = 0;

        for (int j = lengthOfPattern - 1; j >= 0; j--) {
            if (pattern.charAt(j) != text.charAt(i + j)) {

                if ( this.mismatchShiftsTable.get(text.charAt(i+j)) != null ) {
                    numOfSkips = this.mismatchShiftsTable.get(text.charAt(i+j));
                    break;
                } else {
                    numOfSkips = lengthOfPattern;
                    break;
                }
            }
        }

        if (numOfSkips == 0) return i;
}
```

Pseudocode

```
for (int i = 0; i <= lengthOfText - lengthOfPattern; i += numOfSkips) {

        numOfSkips = 0;

        for (int j = lengthOfPattern - 1; j >= 0; j--) {
            if (pattern.charAt(j) != text.charAt(i + j)) {

                if ( this.mismatchShiftsTable.get(text.charAt(i+j)) != null ) {
                    numOfSkips = this.mismatchShiftsTable.get(text.charAt(i+j));
                    break;
                } else {
                    numOfSkips = lengthOfPattern;
                    break;
                }
            }
        }

        if (numOfSkips == 0) return i;
}
```

Its very important that we increment the loop index accoding to the „bad match table" values !!!

Pseudocode

```
for (int i = 0; i <= lengthOfText - lengthOfPattern; i += numOfSkips) {

        numOfSkips = 0;

        for (int j = lengthOfPattern - 1; j >= 0; j--) {
            if (pattern.charAt(j) != text.charAt(i + j)) {

                if ( this.mismatchShiftsTable.get(text.charAt(i+j)) != null ) {
                    numOfSkips = this.mismatchShiftsTable.get(text.charAt(i+j));
                    break;
                } else {
                    numOfSkips = lengthOfPattern;
                    break;
                }
            }
        }

        if (numOfSkips == 0) return i;
}
```

We iterate through the pattern in reverse order, so we start at the rightmost character!!!

Pseudocode

```
for (int i = 0; i <= lengthOfText - lengthOfPattern; i += numOfSkips) {

            numOfSkips = 0;

            for (int j = lengthOfPattern - 1; j >= 0; j--) {
                  if (pattern.charAt(j) != text.charAt(i + j)) {

                        if ( this.mismatchShiftsTable.get(text.charAt(i+j)) != null ) {
                              numOfSkips = this.mismatchShiftsTable.get(text.charAt(i+j));
                              break;
                        } else {
                              numOfSkips = lengthOfPattern;
                              break;
                        }
                  }
            }

            if (numOfSkips == 0) return i;
}
```

We iterate through the pattern in reverse order, so we start at the rightmost character!!!

Pseudocode

```
for (int i = 0; i <= lengthOfText - lengthOfPattern; i += numOfSkips) {

        numOfSkips = 0;

        for (int j = lengthOfPattern - 1; j >= 0; j--) {
            if (pattern.charAt(j) != text.charAt(i + j)) {

                if ( this.mismatchShiftsTable.get(text.charAt(i+j)) != null ) {
                    numOfSkips = this.mismatchShiftsTable.get(text.charAt(i+j));
                    break;
                } else {
                    numOfSkips = lengthOfPattern;
                    break;
                }
            }
        }

        if (numOfSkips == 0) return i;
}
```

If there is a mismatch: we update the number of skips, we get it from the „bad match table" or if it is not in the table → we shift the pattern with the patternLength !!!

Analysis

- Turns out to be very efficient !!!
- Mismatched character heuristics takes about ~ **N / M** character compares, where **M** is the length of the pattern and **N** is the length of the text
- It is not even linear: it is sublinear
- So the longer the pattern → the faster the algorithm become
- BUT worst case scenario can as bas as brute force when for example pattern is like **CDDDDDD** and text is **DDDDDDDDDDD**

The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt Algorithm

- **Substring:** substring of an m-character string P to refer to a string of the form $P[i]\ P[i+1]\ P[i+2]\ldots p[j]$ for some $0 \leq i \leq j \leq m-1$ that is, the string formed by the characters in P from index i to index j, inclusive.
- **Prefix:** any substring of the form $P[0 \ldots i]$, for $0 \leq i \leq m-1$, is a prefix of P
- **Suffix:** any substring of the form $P[i \ldots m-1]$, for $0 \leq i \leq m-1$, is suffix of P

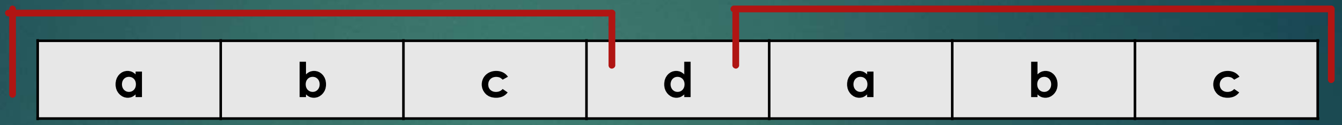
String: abcdabc

Prefix: a, ab, abc, abcd …

Suffix: c, bc, abc, dabc …

The Knuth-Morris-Pratt Algorithm

- Is the beginning part of the pattern appearing again anywhere in the pattern.



a	b	c	d	a	b	c

- π Table: In KMP algorithm we generate a table which stores similarity index values of the pattern

a	b	c	d	a	b	e	a	b	f
0	0	0	0	1	2	0	1	2	0

a	b	c	d	e	a	b	f	a	b	c
0	0	0	0	0	1	2	0	1	2	3

The prefix function, Π

Following pseudocode computes the prefix fucnction, Π:

```
Compute-Prefix-Function (p)
m ← length[p]              //'p' pattern to be matched
Π[1] ← 0
k ← 0
    for q ← 2 to m
          do while k > 0 and p[k+1] != p[q]
                 do k ← Π[k]
               If p[k+1] = p[q]
                 then k ← k +1
               Π[q] ← k
   return Π
```

The Knuth-Morris-Pratt Algorithm

String

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

1	2	3	4	5
a	b	a	b	d

The Knuth-Morris-Pratt Algorithm

String

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

The Knuth-Morris-Pratt Algorithm

String i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

		i												
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

	j				
0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] \neq P[j+1]$

$i = i$

$j = \pi[j]$

The Knuth-Morris-Pratt Algorithm

String

				i										
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

		j			
0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] \neq P[j+1]$

$i = i$

$j = \pi[j]$

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] \neq P[j+1]$

$i = i+1$

$j = 0$

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]
i = i+1
j = j+1

The Knuth-Morris-Pratt Algorithm

String

						i								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

	j				
0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] == P[j+1]$

$i = i+1$

$j = j+1$

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] \neq P[j+1]$

$i = i$

$j = \pi[j]$

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] \neq P[j+1]$

$i = i+1$

$j = 0$

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] == P[j+1]$

$i = i+1$

$j = j+1$

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] == P[j+1]$

$i = i+1$

$j = j+1$

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] \neq P[j+1]$

$i = i$

$j = \pi[j]$

The Knuth-Morris-Pratt Algorithm

String

												i		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

		j			
0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

$S[i] == P[j+1]$

$i = i+1$

$j = j+1$

The Knuth-Morris-Pratt Algorithm

String

													i	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

			j		
0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

S[i] == P[j+1]

i = i+1

j = j+1

The Knuth-Morris-Pratt Algorithm

String

i

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern

j

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

When we reach the end of the table we conclude that the pattern is present in the String

KMP Matcher Algorithm

```
n ← length[S]
m ← length[p]
Π ← Compute-Prefix-Function(p)
q ← 0
for i ← 1 to n
    do while  q > 0 and p[q+1] != S[i]
            do  q ← Π[q]
    if p[q+1] = S[i]
        then q ← q + 1
    if q = m
      then print "Pattern occurs with shift" i – m
                q ← Π[ q]
```

Running - time analysis

Compute-Prefix-Function (Π)

```
1  m ← length[p]            //'p' pattern to be matched
2  Π[1] ← 0
3  k ← 0
4           for q ← 2 to m
5                do while k > 0 and p[k+1] != p[q]
6                do k ← Π[k]
7                   If p[k+1] = p[q]
8                     then k ← k +1
9                   Π[q] ← k
10          return Π
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

KMP Matcher

```
1.     n ← length[S]
2.     m ← length[p]
3.     Π ← Compute-Prefix-Function(p)
4.     q ← 0
5.     for i ← 1 to n
6.       do while  q > 0 and p[q+1] != S[i]
7.             do  q ← Π[q]
8.        if p[q+1] = S[i]
9.           then q ← q + 1
10.       if q = m
11.         then print "Pattern occurs with shift" i – m
12.                   q ← Π[ q]
```

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S'. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

Complexity = O(m+n)