

# Mechanized Operational Semantics

J Strother Moore  
Department of Computer Sciences  
University of Texas at Austin

Marktoberdorf Summer School 2008

(Lecture 4: Boyer-Moore Fast String Searching)

# The Problem

One of the classic problems in computing is *string searching*: find the first occurrence of one character string ( “the *pattern*” ) in another ( “the *text*” ).

Generally, the text is *very* large (e.g., gigabytes) but the patterns are relatively small.

# Examples

Find the word “comedy” in this *NY Times* article:

Fred Armisen’s office at “Saturday Night Live” is deceptively small, barely big enough to fit a desk, a couch, and an iPod. The glorified closet, the subject of a running joke on the comedy show, now in its 31st season, can simultaneously house a wisecracking ...

AAAAAAAAAAAAACAAAGACAGGGGCAACAAAGTGAGACCCTAAAAAAAAAAAAACCCA  
AAACGGAGAACTTGGAATCCTGTGTCCAAAAAAAAAAGCAGGAAGAGAGCGTGTAGAAAC  
TGAAGCTGAAGTGGAIAAAAAAAAAAGTCGCCAGCACCTACTGTGGAGACCAGAAAGGAAAA  
AAAAAATTGGCAGTCTCGTAGCATACCAAACTAGGCTTGAAAAAAAAAACACACAAAAA  
AACACAGGCTACCCAGTATTTTATCGTCCAAAAAAAAAAGAGGGAAGAAGGACATTTATAT  
TTGCCTTCTGCCAAAAAAAAAAGTACCTCCCGCCTAGAAGAGAGTTTAGAAATCACCAAA  
AAAAAATAGAGAGTCCCAAAATGTTTCGGAATACTCAGAAAAAAAAAATCTTAGTCAGTGCT  
CACTCAGAGGGACCGGGTATTTAAAAAAAAACCTAGACCAGATGCAGCAGGTACAAATTAA  
TCAATCCCAAAAAAAAAAGACCTTCTACCCTTCCAAAAAATGATAGTTGTCTGCAATCCAAA  
AAAAAGACTCTCCGGAAGGTGGACATGCAGAACCTACCAAAAAAAAAAGAGAAGAAAGAAT  
TGCCGGGCAAAAAGTTCCACGTAAAAAAAAAAGGAAATGGGAATGGAGTGTTGTTCTCCT  
TCCTACCTAGTTTTTGAIAAAAAAAGGATGGATGTGGGTACCTGCTCACGTTCTCCAAAAA  
AAAGTGGGTGCTCTCTCACAATATTCTTAGAGGTGGCAAAAAAAAAATAAAGTTGATGGAAA  
CAGTACTGTGTGGGCCAAACAAAAAAAAAAATGGCACCACCTTTTCATTGGCTGAAAAAAAA  
AATTCAACTGAAAAACACAAGTCATACCTTCCTGTTTTATTTGCAIAAAAAAATTTTTCAA  
ACCCACGGCAACAAACGACAGTATCAIAAAAAACAACCTTCATTTGACATTCTGCTATATT  
AATGCTCTATGTGGAAAAAAAAAACCATCAAGTTGTGCCTTTTTTTCAAAGAAATCCATGCA  
AAAAAAGACCCATGAAATAATTTTCTGGATCATCCATACAGAACCAIAAAAAAAGAGGTG



Variants of the problem allow *wildcards* in the pattern and/or the text. *Exact* matching is when no wildcards are allowed.

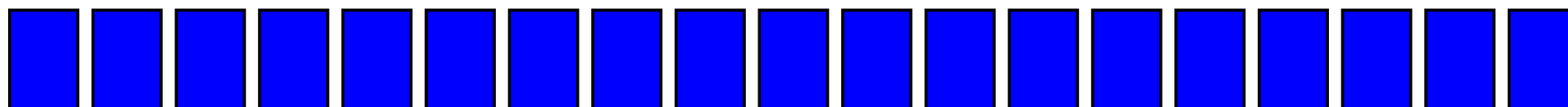
We describe the fastest sequential algorithm for solving the exact string searching problem. The algorithm is called the *Boyer-Moore fast string searching algorithm*.

# Example

Find the word “comedy” in this *NY Times* article:

Fred Armisen’s office at “Saturday Night Live” is deceptively small, barely big enough to fit a desk, a couch, and an iPod. The glorified closet, the subject of a running joke on the comedy show, now in its 31st season, can simultaneously house a wisecracking ...

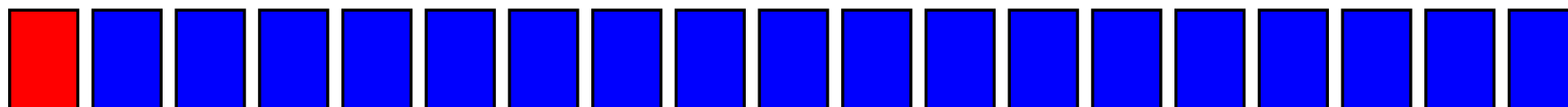
**C O M E D Y**



**J O K E      O N      T H E      C O M E D Y**

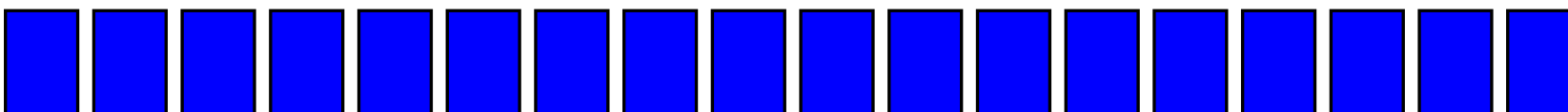


**C O M E D Y**



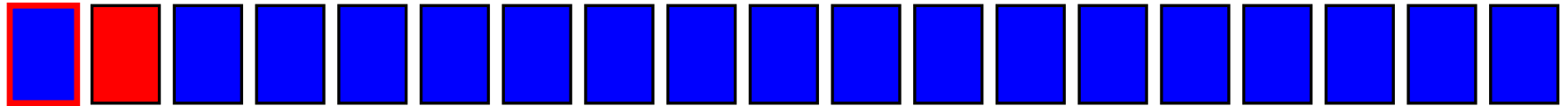
**J O K E      O N      T H E      C O M E D Y**

C O M E D Y

J 

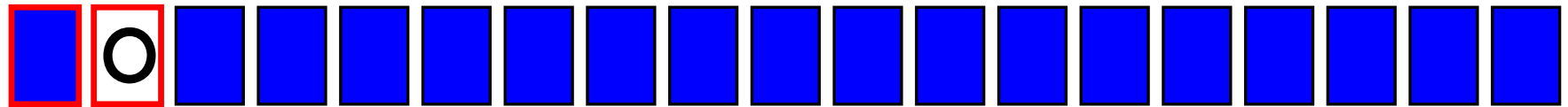
J O K E      O N      T H E      C O M E D Y

COMEDY



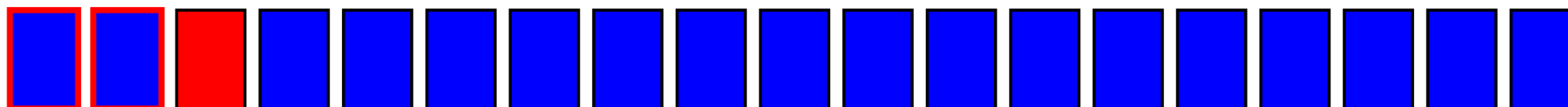
J O K E      O N      T H E      C O M E D Y

C O M E D Y



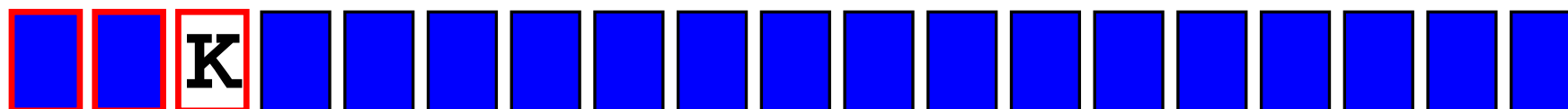
J O K E      O N      T H E      C O M E D Y

COMEDY



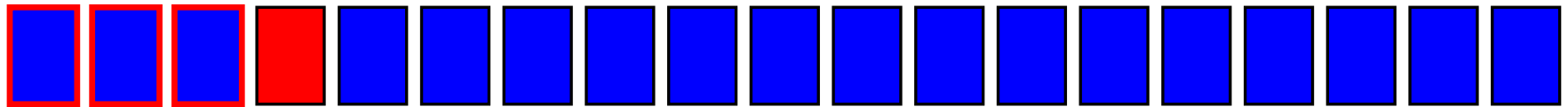
J O K E      O N      T H E      C O M E D Y

C O M E D Y



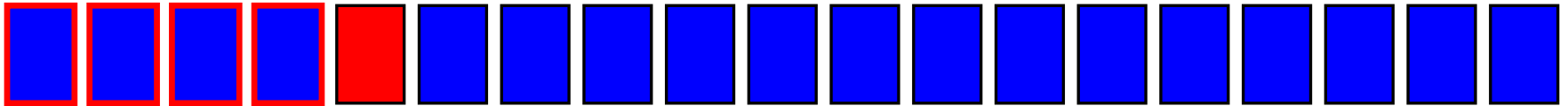
J O K E      O N      T H E      C O M E D Y

C O M E D Y



J O K E      O N      T H E      C O M E D Y

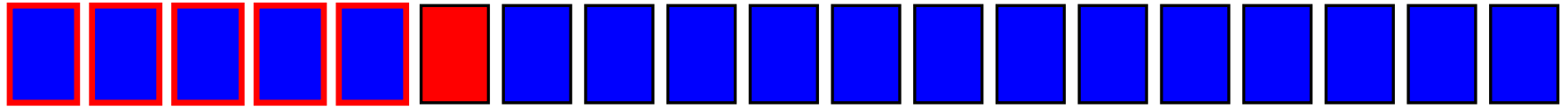
COMEDY



J O K E      O N      T H E      C O M E D Y



COMEDY

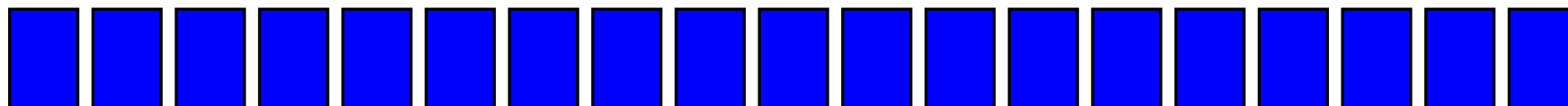


J O K E      O N      T H E      C O M E D Y

COMEDY  
COMEDY

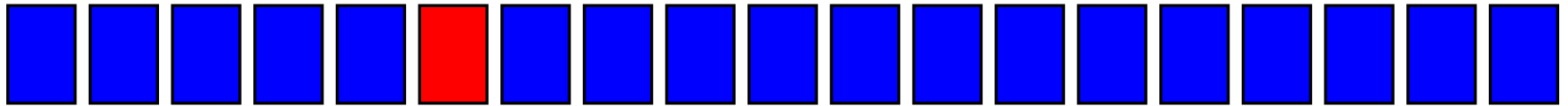
J O K E    O N    T H E    C O M E D Y

**C O M E D Y**



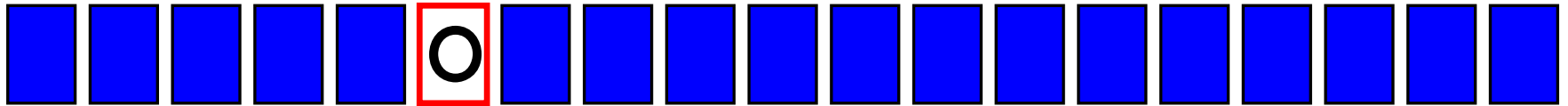
**J O K E    O N    T H E    C O M E D Y**

**C O M E D Y**



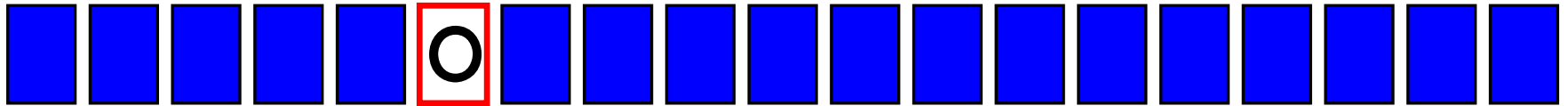
**J O K E      O N      T H E      C O M E D Y**

C O M E D Y



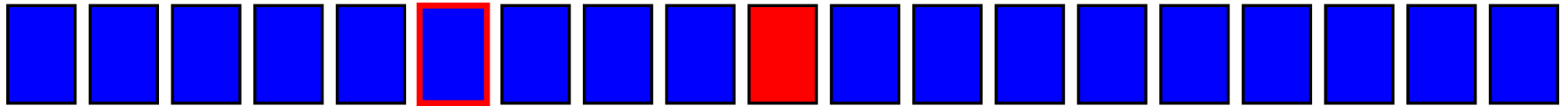
J O K E    O N    T H E    C O M E D Y

C O M E D Y



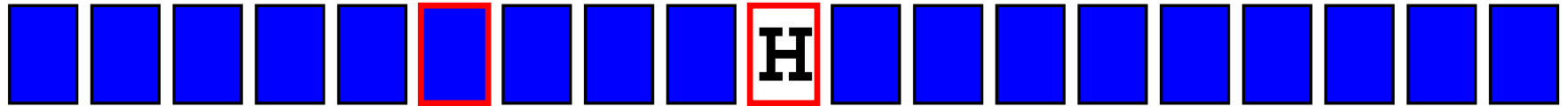
J O K E    O N    T H E    C O M E D Y

C O M E D Y



J O K E    O N    T H E    C O M E D Y

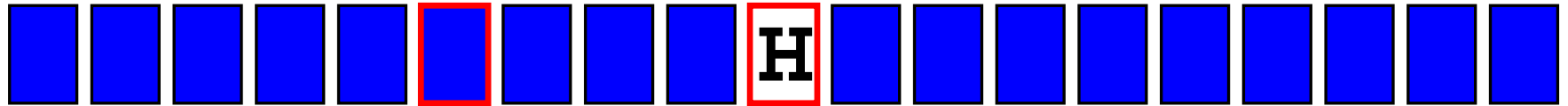
C O M E D Y



J O K E    O N    T H E    C O M E D Y

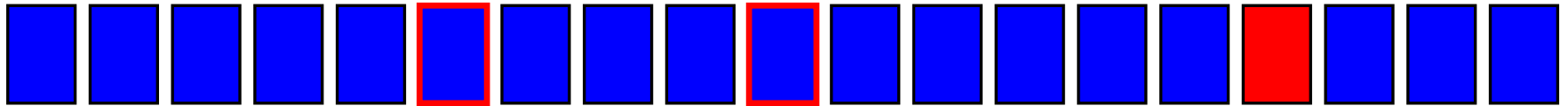


C O M E D Y



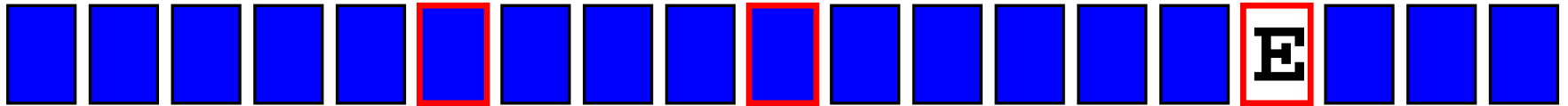
J O K E    O N    T H E    C O M E D Y

C O M E D Y



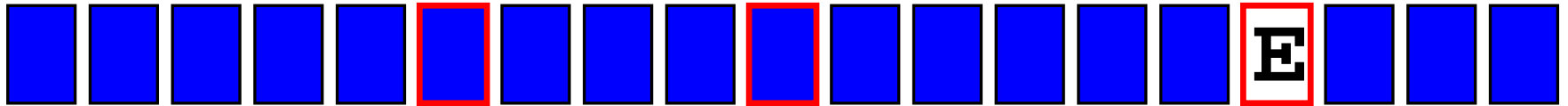
J O K E    O N    T H E    C O M E D Y

C O M E D Y



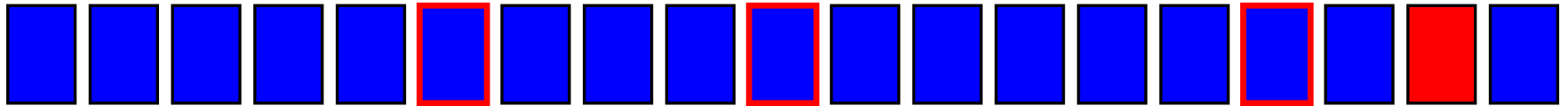
J O K E    O N    T H E    C O M E D Y

C O M E D Y

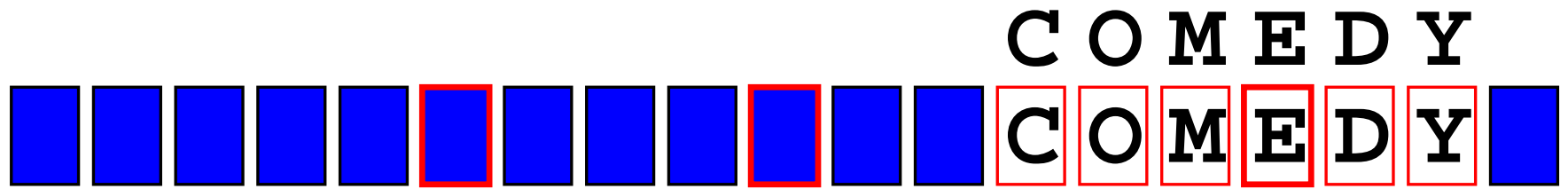


J O K E      O N      T H E      C O M E D Y

C O M E D Y



J O K E    O N    T H E    C O M E D Y



J O K E    O N    T H E    C O M E D Y

Key Property: The longer the pattern, the faster the search!

# Pre-Computing the Skip Distance

*pat*: 543210

COMEDY

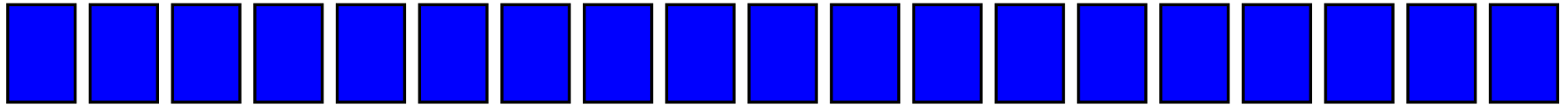
*txt*: xxxxx0xxxxxxxxxxxxx...

↑

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

This is a 1-dimensional array, `skip[c]`, as big as the alphabet.

**C O M E D Y**



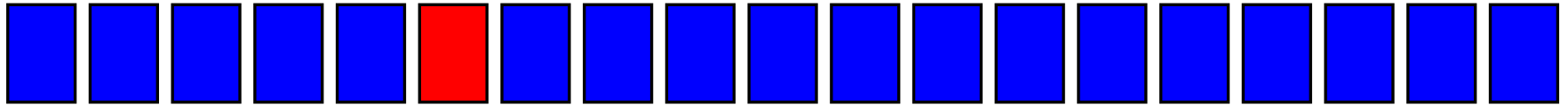
**J O K E      O N      T H E      C O M E D Y**

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6



**C O M E D Y**

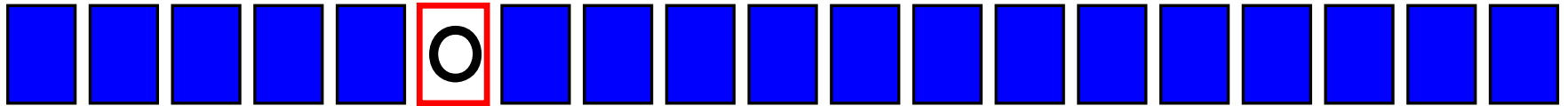


**J O K E      O N      T H E      C O M E D Y**

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

**C O M E D Y**

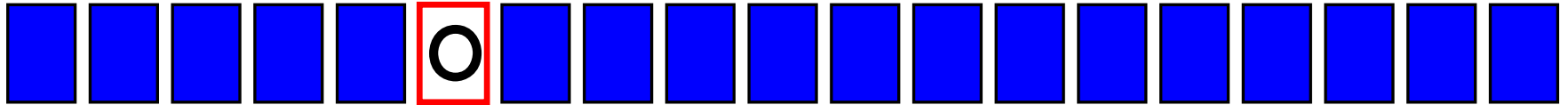


**J O K E      O N      T H E      C O M E D Y**

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

C O M E D Y

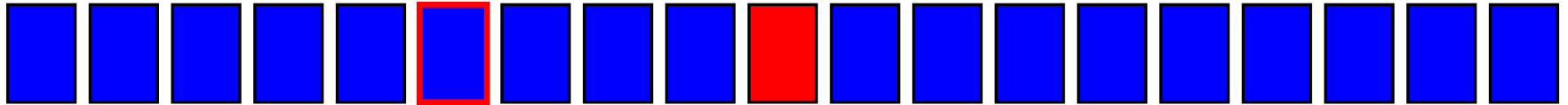


J O K E      O N      T H E      C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

**C O M E D Y**

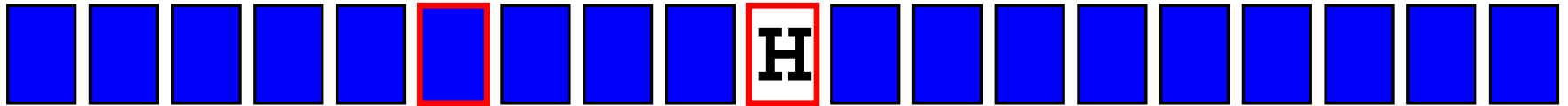


**J O K E      O N      T H E      C O M E D Y**

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

C O M E D Y

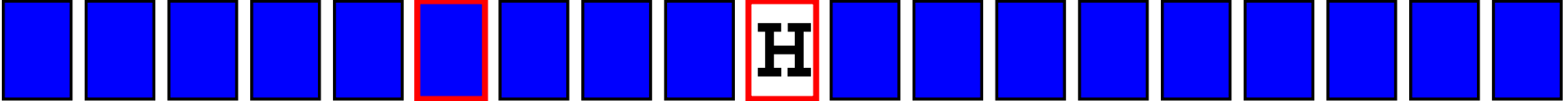


J O K E      O N      T H E      C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

COMEDY

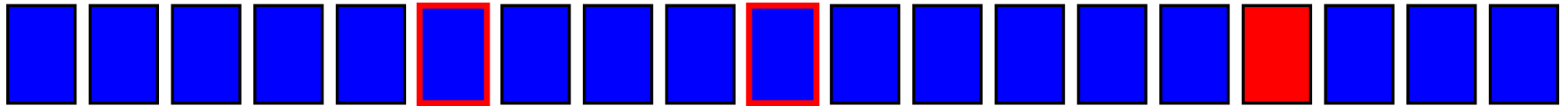


J O K E    O N    T H E    C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

C O M E D Y

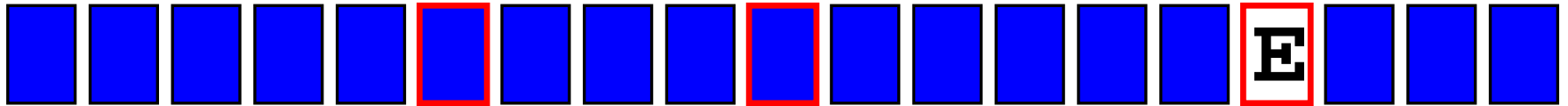


J O K E      O N      T H E      C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

C O M E D Y

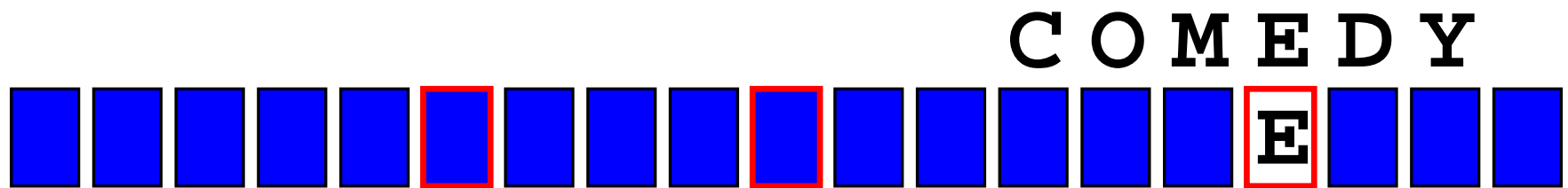


J O K E      O N      T H E      C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

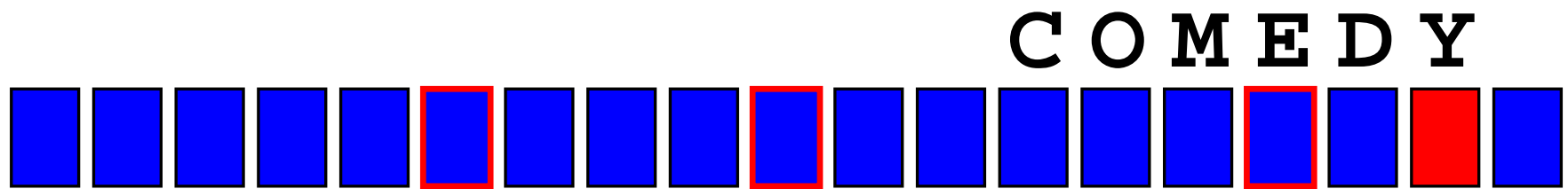




**J O K E      O N      T H E      C O M E D Y**

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6



**J O K E      O N      T H E      C O M E D Y**

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

J O K E      O N      T H E      C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6
B 6	G 6	L 6	Q 6	V 6
C 5	H 6	M 3	R 6	W 6
D 1	I 6	N 6	S 6	X 6
E 2	J 6	O 4	T 6	Y 0
				Z 6

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----

|

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----R-----  
                  |

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----A-----  
                          |

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----P-----  
                  |

## But Wait! There's More!

*pat*:      NONPARTIPULAR

*txt*: -----P-----  
                  |

Slide 2 to match the discovered character.



# But Wait! There's More!

*pat*:      NONPARTIPULAR

*txt*: -----P??-----  
                          |

# But Wait! There's More!

*pat*:      NONPARTIPULAR

*txt*:    -----PAR-----  
                                  |

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----

|

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----R-----  
                  |

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----AR-----

|

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----PAR-----  
                          |

# But Wait! There's More!

*pat*: NONPARTIPULAR

*txt*: -----PAR-----  
                          |

## But Wait! There's More!

*pat*:                   NONPARTIPULAR  
*txt*: -----PAR-----  
                          |

Slide 7 to match the *discovered substring*!



	$j$	$ pat $
$pat:$	NONPARTIPULAR	
$txt:$	-----PAR-----	
	$i$	
$dt:$	$txt[i]$	$pat[j + 1] \dots pat[ pat ]$
	P	A R

$dt: \text{txt}[i] \text{ pat}[j+1] \dots \text{pat}[|\text{pat}|]$

$dt$  can be computed given  $\text{txt}[i]$  and index  $j$  in  $\text{pat}$ !

There are only  $|\alpha| \times |\text{pat}|$  combinations, where  $|\alpha|$  is the alphabet size.

# The Skip Distance – Delta

Given  $pat$ , the skip can be pre-computed for every combination of character read,  $c$ , and pattern index,  $j$ , by finding how far we must slide to find the *last* occurrence of  $dt$  in  $pat$ .

*pat*: NONPARTIPULAR

*txt*: -----PAR-----

|

*pat*:                   NONPARTIPULAR  
*txt*: -----PAR-----  
                          |

*pat*: BC-ABC-BBC-CBC

*txt*: -----BBC-----

|

*pat*:           BC-ABC-BBC-CBC

*txt*:   -----BBC-----

|

*pat*: BC-ABC-BBC-CBC

*txt*: -----ABC-----

|



*pat*: BC-ABC-BBC-CBC

*txt*: -----ABC-----

|

*pat*: BC-ABC-BBC-CBC

*txt*: -----DBC-----  
                  |

*pat*: BC-ABC-BBC-CBC

*txt*: -----DBC-----

|

*pat*: EE-ABC-BBC-CBC

*txt*: -----DBC-----

|

*pat*: EE-ABC-BBC-CBC

*txt*: -----DBC-----

|

# The Delta Array

$\text{delta}[c,j]$  is an array of size  $|\alpha| \times |\text{pat}|$  that gives the skip distance when a mismatch occurs after comparing  $c$  from  $\text{txt}$  to  $\text{pat}[j]$ .

# The Algorithm

*fast(pat, txt)*

**if** *pat* = ""

**then**

**if** *txt* = ""

**then return** *Not-Found*;

**else return** 0; **end**;

**end**;

*preprocess pat to produce delta;*

$j := |pat| - 1;$

$i := j;$



```
while  $(0 \leq j \wedge i < |txt|)$   
do  
  if  $pat[j] = txt[i]$   
    then  
       $i := i - 1;$   
       $j := j - 1;$   
    else  
       $i := i + delta[txt[i], j];$   
       $j := |pat| - 1;$   
    end;
```

```
If ( $j < 0$ )  
    then return  $i + 1$ ;  
    else return Not-Found; end;  
  
end;
```

# Performance

How does the algorithm perform?

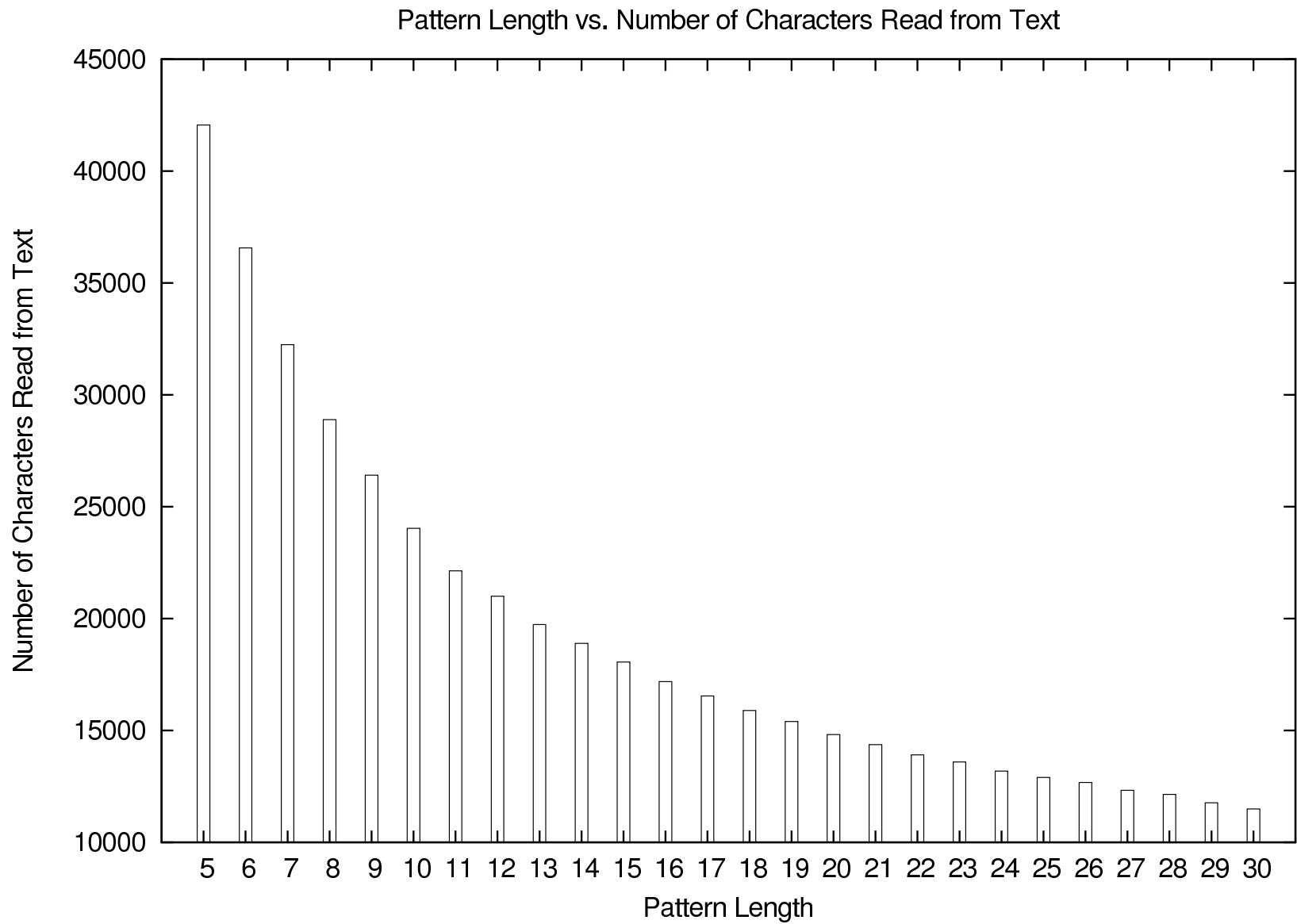
This depends on the size of the alphabet. We only have data on English text right now.

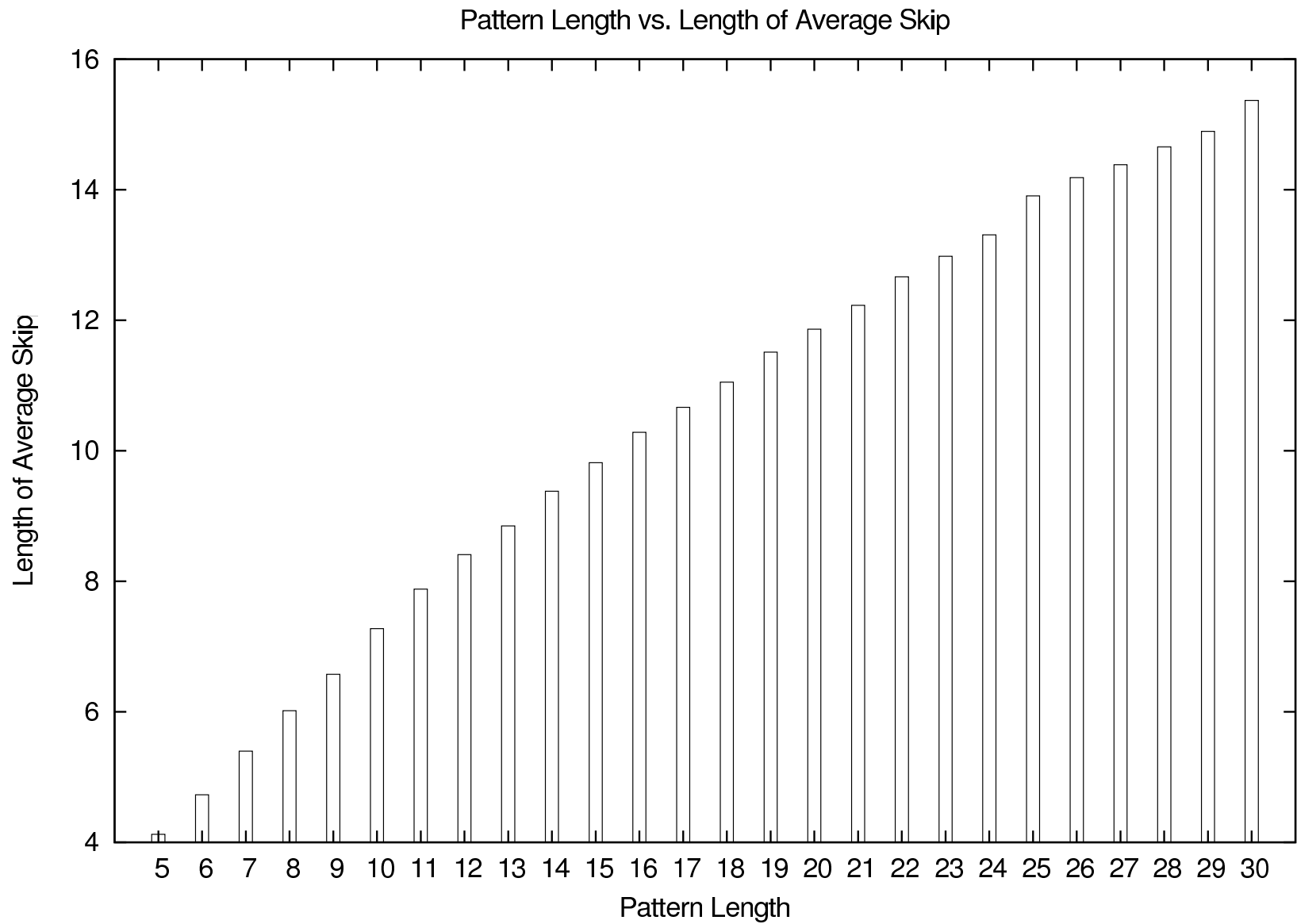
In our test:

txt: English text of length 177,985.

pat: 100 randomly chosen patterns of length 5 – 30, chosen from another English text and filtered so they do not occur in the search text.

The naive string searching algorithm would look at all 177,985 characters of the search text. In fact, it would look at some characters more than once.





# Goal

Prove the correctness of an M1 program for the Boyer-Moore fast string searching algorithm.

We will not code the preprocessing in M1.

We will write code for the Boyer-Moore algorithm that assumes that the contents of a certain local contains a 2-dimensional delta array.

We will initialize the array variable with ACL2 code, not M1 code.

We will proceed as previously advised:

- Step 1: prove that the code implements the algorithm
- Step 2: prove that the algorithm implements the spec

We'll do Step 2 *first*. It's *always* the hardest.



# Caveat

In this talk I will ignore hypotheses and distracting arithmetic details.

The ACL2 proof scripts provide the complete details.

# The Obviously Correct Algorithm

```
(defun matchp (pat j txt i)
  (cond ((not (natp j)) nil)
        ((>= j (length pat)) t)
        ((>= i (length txt)) nil)
        ((equal (char pat j)
                  (char txt i))
         (matchp pat (+ 1 j)
                  txt (+ 1 i)))
        (t nil)))
```

# The Obviously Correct Algorithm

```
(defun matchp (pat j txt i)
  (cond
    ((>= j (length pat)) t)
    ((>= i (length txt)) nil)
    ((equal (char pat j)
             (char txt i))
     (matchp pat (+ 1 j)
              txt (+ 1 i)))
    (t nil)))
```

```
(defun correct-loop (pat txt i)
  (cond ((>= i (length txt)) nil)
        ((matchp pat 0 txt i) i)
        (t (correct-loop pat txt (+ 1 i)))))
```

```
(defun correct (pat txt)
  (correct-loop pat txt 0))
```

# The Fast Algorithm

```
(defun fast-loop (pat j txt i)
  (cond
    ((< j 0) (+ 1 i))
    ((<= (length txt) i) nil)
    ((equal (char pat j) (char txt i))
     (fast-loop pat (- j 1) txt (- i 1)))
    (t (fast-loop pat
                  (- (length pat) 1)
                  txt
                  (+ i (delta (char txt i)
                              j pat))))))
```

```

(defun fast-loop (pat j txt i)
  (declare
    (xargs :measure (measure pat j txt i)
            :well-founded-relation 1<))
  (cond ...
    ((equal (char pat j) (char txt i))
     (fast-loop pat (- j 1) txt (- i 1)))
    (t (fast-loop pat
                  (- (length pat) 1)
                  txt
                  (+ i (delta (char txt i)
                              j pat)))))))

```

Note Above:

In this formalization of the algorithm, we do not pre-compute  $\delta$  but instead compute the skip distance as a function of the char from  $\text{txt}$ , the index  $j$  in  $\text{pat}$ , and  $\text{pat}$ .

The M1 code will use a 2-dimensional array initialized by an ACL2 function.

We will prove the ACL2 preprocessing correct.

But at the algorithmic level, we needn't think about arrays.

```
(defun fast (pat txt)
  (if (equal pat "")
      (if (equal txt "")
          nil
          0)
      (fast-loop pat
                  (- (length pat) 1)
                  txt
                  (- (length pat) 1))))
```



## “Pre-Processing”

```
(defun delta (v j pat)
  (let* ((pat~ (coerce pat 'list))
         (dt (cons v (nthcdr (+ j 1) pat~))))
    (+ (- (len pat~) 1)
       (- (find-pmatchp dt pat~ (- j 1))))))

(defun find-pmatchp (dt pat~ j)
  (cond ((pmatchp dt pat~ j) j)
        (t (find-pmatchp dt pat~ (- j 1)))))
```

pat: BC-ABC-BBC-CBC  
dt: BBC  
pmatchp: BBC

pat: BC-ABC-BBC-CBC  
dt: ABC  
pmatchp: ABC

pat: BC-ABC-BBC-CBC  
dt: GBC  
pmatchp: GBC

# Goal

```
(defthm fast-is-correct
  (implies (and (stringp pat)
                 (stringp txt))
    (equal (fast pat txt)
           (correct pat txt))))
```

# Observation 1 – List Counterparts

Every string processing function has a list processing counterpart.

```
(char str i) = (nth i (coerce str 'list))
```

# Observation 1 – List Counterparts

Let  $\text{pat}^\sim$  be  $(\text{coerce pat 'list})$ .

$(\text{equal } (\text{correct-loop pat txt i})$   
           $(\text{correct-loop}^\sim \text{pat}^\sim \text{txt}^\sim i))$

# Observation 1 – List Counterparts

```
(defun delta (v j pat)
  (let* ((pat~ (coerce pat 'list))
        (dt~ (cons v (nthcdr (+ j 1) pat~))))
    (+ (- (len pat~) 1)
       (- (find-pmatchp dt~ pat~ (- j 1))))))
```

## Observation 2 – Matching is Equality

```
(defun matchp (pat j txt i)
  (cond ((>= j (length pat)) t)
        ((>= i (length txt)) nil)
        ((equal (char pat j) (char txt i))
         (matchp pat (+ 1 j) txt (+ 1 i)))
        (t nil)))
```

j

pat:       abcUVW

txt:       xxxxxUVWxxxxx

i

## Observation 2 – Matching is Equality

```
(equal (matchp pat j txt i)
      (equal (firstn (len (nthcdr j pat~))
                    (nthcdr i txt~))
              (nthcdr j pat~)))
```

```
      j
pat:   abcUVW
txt:   xxxxxUVWxxxxx
      i
```



## Observation 3 – Destructor Elimination

$(\text{append } (\text{firstn } n \ x) \ (\text{nthcdr } n \ x)) = x$

So to prove:

$\psi(x, (\text{firstn } n \ x), (\text{nthcdr } n \ x))$

it is sufficient to prove

$\psi(\text{append } a \ b, a, b)$

# Goal

```
(defthm fast-is-correct
  (implies (and (stringp pat)
                 (stringp txt))
    (equal (fast pat txt)
           (correct pat txt))))
```

# The Crux

```
(implies
  (equal (firstn (len (nthcdr (+ 1 j) pat~))
              (nthcdr (+ 1 i) txt~))
    (nthcdr (+ 1 j) pat~))
  (equal
    (correct-loop~ pat~ txt~
      (+ i (- (find-pmatchp
                (cons (car (nthcdr i txt~))
                      (nthcdr j (cdr pat~)))
                pat~ (+ -1 j))))))
    (correct-loop~ pat~ txt~ (+ i (- j)))))
```

# Decomposition

The crux is to prove that `correct-loop` can skip ahead in big steps (like `fast` does).

But we can decompose this into two parts.

# Decomposition

- (a) `correct-loop` can skip ahead if there are no matches in the region skipped
- (b) there are no matches in the region skipped by `find-pmatchp`

## Summary of Step 2

A total of 9 definitions and lemmas are proved to establish

```
(defthm fast-is-correct
  (implies (and (stringp pat)
                 (stringp txt))
            (equal (fast pat txt)
                   (correct pat txt))))
```

# Step 1

```
(defconst *m1-boyer-moore-program*

; Allocation of locals

; pat    0
; j      1
; txt    2
; i      3
; pmax   4 = (length pat)
; tmax   5 = (length txt)
; array  6 = (preprocess pat)
; c      7 = temp - last char read from txt

' (

      (load 0)          ; 0      (load pat)
      (push "")         ; 1      (push "")
```

(ifane 5)	; 2	(ifane loop)
(load 2)	; 3	(load txt)
(push "")	; 4	(push "")
(ifane 40)	; 5	(ifane win)
(goto 43)	; 6	(goto lose)
; loop:		
(load 1)	; 7	(load j)
(iflt 37)	; 8	(iflt win))
(load 5)	; 9	(load tmax)
(load 3)	; 10	(load i)
(sub)	; 11	(sub)
(ifle 37)	; 12	(ifle lose)
(load 0)	; 13	(load pat)
(load 1)	; 14	(load j)
(aload)	; 15	(aload)
(load 2)	; 16	(load txt)
(load 3)	; 17	(load i)
(aload)	; 18	(aload)
(store 7)	; 19	(store v)



(load 7)	; 20	(load v)
(sub)	; 21	(sub)
(ifne 10)	; 22	(ifne skip)
(load 1)	; 23	(load j)
(push 1)	; 24	(push 1)
(sub)	; 25	(sub)
(store 1)	; 26	(store j)
(load 3)	; 27	(load i)
(push 1)	; 28	(push 1)
(sub)	; 29	(sub)
(store 3)	; 30	(store i)
(goto -24)	; 31	(goto loop)
; skip:		
(load 3)	; 32	(load i)
(load 6)	; 33	(load array)
(load 7)	; 34	(load v)
(aload)	; 35	(aload)
(load 1)	; 36	(load j)
(aload)	; 37	(aload)

(add)	; 38	(add)
(store 3)	; 39	(store i)
(load 4)	; 40	(load pmax)
(push 1)	; 41	(push 1)
(sub)	; 42	(sub)
(store 1)	; 43	(store j)
(goto -37)	; 44	(goto loop)
; win:		
(load 3)	; 45	(load i)
(push 1)	; 46	(push 1)
(add)	; 47	(add)
(return)	; 48	(return)
; lose:		
(push nil)	; 49	(push nil)
(return) )	; 50	(return))
)		

# The Schedule

How do we define the schedule for such a complicated piece of code?

# The Schedule

```
(defun m1-boyer-moore-loop-sched (pat j txt i)
  (cond
    ((< j 0) (repeat 0 6))
    ((<= (length txt) i) (repeat 0 8))
    ((equal (char-code (char pat j))
            (char-code (char txt i)))
     (append (repeat 0 25)
              (m1-boyer-moore-loop-sched pat (- j 1)
                                           txt (- i 1)))))
  (t (append (repeat 0 29)
              (m1-boyer-moore-loop-sched
               pat (- (length pat) 1)
               txt (+ i (delta (char txt i) j pat)))))))
```

# The Schedule

```
(defun m1-boyer-moore-loop-sched (pat j txt i)
  (cond
    ((< j 0) (repeat 0 6))
    ((<= (length txt) i) (repeat 0 8))
    ((equal (char-code (char pat j))
            (char-code (char txt i)))
     (append (repeat 0 25)
              (m1-boyer-moore-loop-sched pat (- j 1)
                                           txt (- i 1))))
    (t (append (repeat 0 29)
                (m1-boyer-moore-loop-sched
                 pat (- (length pat) 1)
                 txt (+ i (delta (char txt i) j pat)))))))
```

```
(defun m1-boyer-moore-sched (pat txt)
  (if (equal pat "")
      (if (equal txt "")
          (repeat 0 9)
          (repeat 0 10))
      (append (repeat 0 3)
                (m1-boyer-moore-loop-sched
                 pat (- (length pat) 1)
                 txt (- (length pat) 1))))))
```

# The Schedule

Defining the schedule is trivial if you have verified the algorithm.

They have identical recursive structure and justification.

```

(defthm m1-boyer-moore-is-fast
  (implies
    (and (stringp pat) (stringp txt))
    (equal (top (stack
      (run (m1-boyer-moore-sched pat txt)
        (make-state 0
          (list pat (- (length pat) 1)
            txt (- (length pat) 1)
            (length pat) (length txt)
            (preprocess pat)
            0)
          nil *m1-boyer-moore-program*))))))
    (fast pat txt))))

```



```

(defthm m1-boyer-moore-halts
  (implies
    (and (stringp pat) (stringp txt))
    (haltedp
      (run (m1-boyer-moore-sched pat txt)
        (make-state 0
          (list pat (- (length pat) 1)
            txt (- (length pat) 1)
            (length pat) (length txt)
            (preprocess pat)
            0)
          nil *m1-boyer-moore-program*)))))

```

# Main Theorem

Given

```
(defthm fast-is-correct
  (implies (and (stringp pat)
                 (stringp txt))
            (equal (fast pat txt)
                   (correct pat txt))))
```

and

```

(defthm m1-boyer-moore-is-fast
  (implies
    (and (stringp pat) (stringp txt))
    (equal (top (stack
      (run (m1-boyer-moore-sched pat txt)
        (make-state 0
          (list pat (- (length pat) 1)
            txt (- (length pat) 1)
            (length pat) (length txt)
            (preprocess pat)
            0)
          nil *m1-boyer-moore-program*))))))
    (fast pat txt))))

```

it is trivial to show:

```

(defthm m1-boyer-moore-is-correct
  (implies
    (and (stringp pat) (stringp txt))
    (equal (top (stack
      (run (m1-boyer-moore-sched pat txt)
        (make-state 0
          (list pat (- (length pat) 1)
            txt (- (length pat) 1)
            (length pat) (length txt)
            (preprocess pat)
            0)
          nil *m1-boyer-moore-program*))))))
    (correct pat txt))))

```

# Demo 1

# Conclusion

Mechanized operational (interpretive) semantics

- are entirely within a logical framework and so permit logical analysis of programs by traditional formal proofs, without introduction of meta-logical transformers (VCGs)
- are generally *executable*
- are easily related to implementations
- allow derivation of language properties

- may allow derivation of intensional properties (e.g., how many steps a program takes to terminate)
- allow verification of system hierarchies (multiple layers of abstraction can be formalized and related within the proof system)

# Thank You