Mechanized Operational Semantics

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

Marktoberdorf Summer School 2008

(Lecture 5: 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 . . .

AAAAAAAAAAAAACAAAGACAGGGGCAACAAAGTGAGACCCTAAAAAAAAAAAAACCCCA
AAACGGAGAACTTGGAATCCTGTGTCCAAAAAAAAAAGCAGGAAGAGAGCGTGTAGAAAC
TGAAGCTGAAGTGGAAAAAAAAAAGTCGCCAGCACCTACTGTGGAGACCAGAAAGGAAAA
AAAAAATTGGCAGTCTCGTAGCATACCAAAACTAGGCTTGAAAAAAAAAACACACAAAAA
AACACAGGCTACCCAGTATTTTATCGTCCAAAAAAAAAGAGGGAAGAAGGACATTTATAT
TTGCCTTCTGCCAAAAAAAAAAGTACCTCCCGCCTAGAAGAGAGTTTAGAAATCACCAAA
AAAAAATAGAGAGTCCCAAAATGTTCGGAATACTCAGAAAAAAAAATCTTAGTCAGTGCT
CACTCAGAGGGACCGGGTATTTAAAAAAAACCTAGACCAGATGCAGCAGGTACAAATTAA
TCAATCCCAAAAAAAAGACCTTCTACCCTTCCAAAAAATGATAGTTGTCTGCAATCCAAA
AAAAAGACTCTCCGGAAGGTGGACATGCAGAACCTACCAAAAAAAAAGAGAAGAAAGAAT
TGCCGGGCAAAAAGTTCCACGTAAAAAAAAAAGGAAATGGGAATGGAGTGTTGTTCTCCT
TCCTACCTAGTTTTGAAAAAAAAGGATGGATGTGGGTCACCTGCTCACGTTCTCCAAAAA
AAAGTGGGTGCTCTCTCACAATATTCTTAGAGGTGGCAAAAAAAATAAAGTTGATGGAAA
CAGTACTGTGTGGGCCAAACAAAAAAAAAATGGCACCACCTTTTCATTGGCTGAAAAAAA
AATTCAACTGAAAAACACAAGTCATACCTTCCTGTTTATTTGCAAAAAAAAATTTTTCAA
ACCCCACGGCAACAAACGACAGTATCAAAAAAACAACTTCATTTGACATTCTGCTATATT
AATGCTCTATGTGGAAAAAAAAAACCATCAAGTTGTGCCTTTTTCAAAGAAATCCATGCA
AAAAAAAGACCCATGAAATAATTTTCTGGATCATCCATACAGAACCAAAAAAAAGAGGTG

C O M E D Y

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

COMEDY

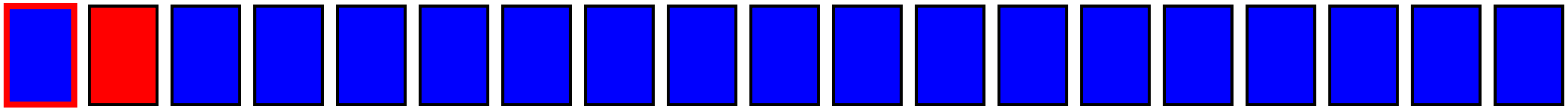
JOKE ON THE COMEDY

COMEDY

J

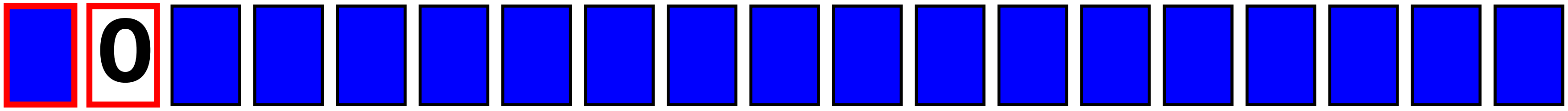
JOKE ON THE COMEDY

C O M E D Y



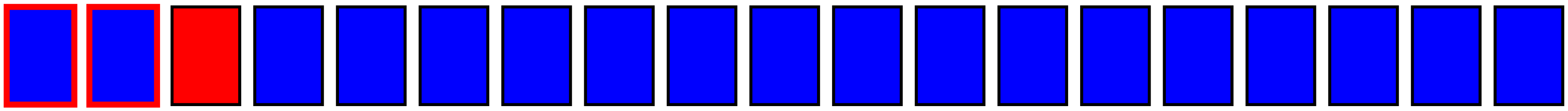
J O K E   O N   T H E   C O M E D Y

COMEDY



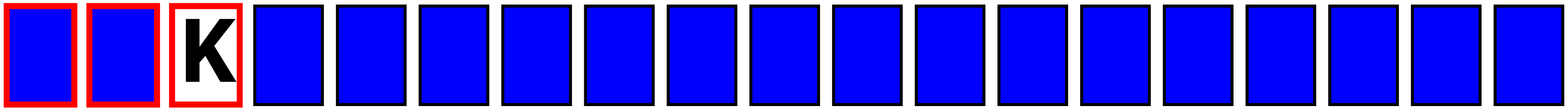
JOKE ON THE COMEDY

COMEDY



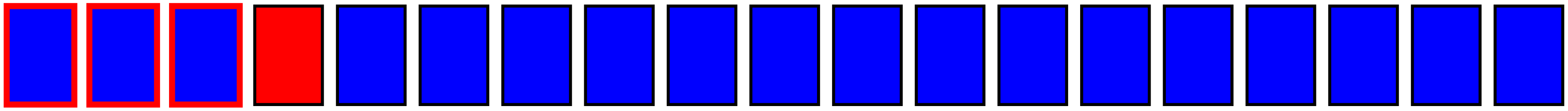
JOKE ON THE COMEDY

C O M E D Y



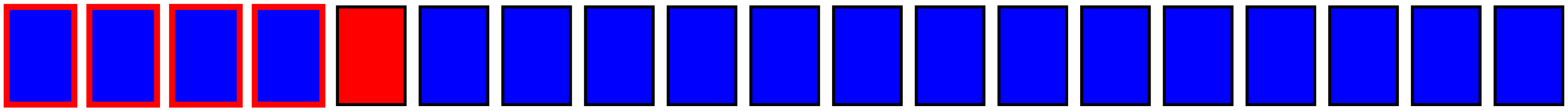
J O K E O N T H E C O M E D Y

C O M E D Y



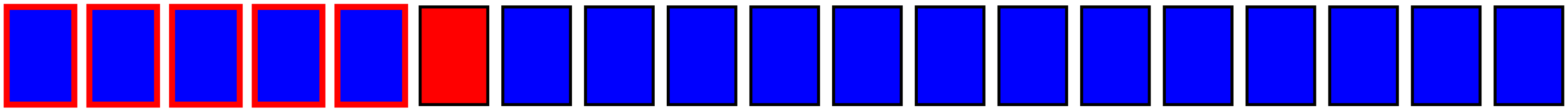
J O K E   O N   T H E   C O M E D Y

COMEDY



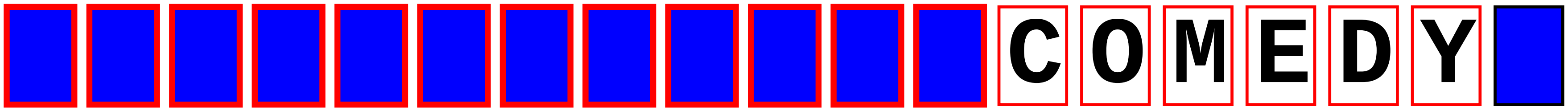
JOKE ON THE COMEDY

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

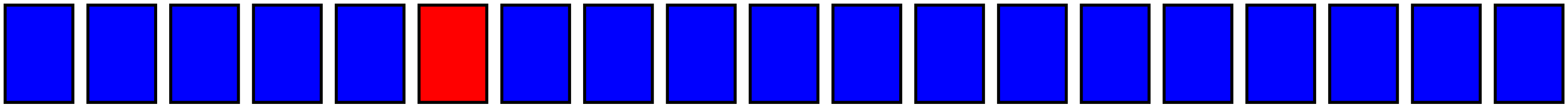


J O K E   O N   T H E   C O M E D Y

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



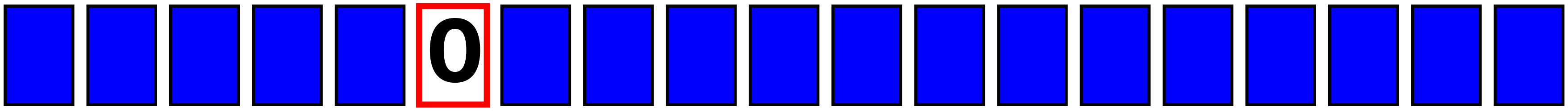
J O K E O N T H E C O M E D Y

COMEDY

					O													

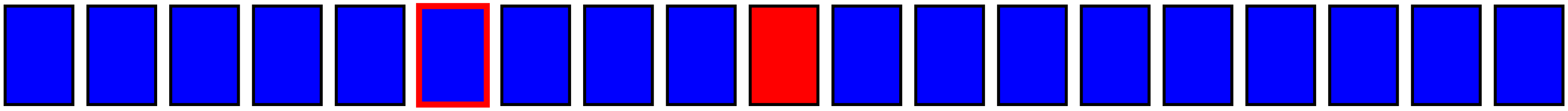
JOKE ON THE COMEDY

COMEDY



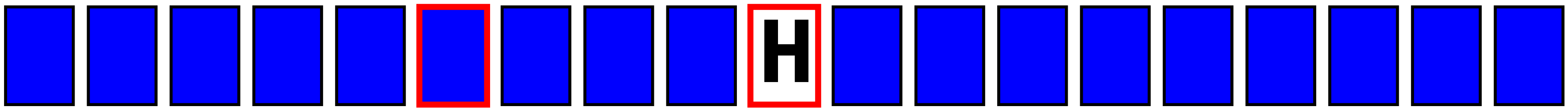
JOKE ON THE COMEDY

C O M E D Y



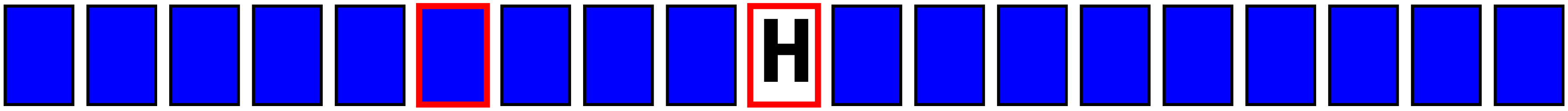
J O K E O N T H E C O M E D Y

COMEDY



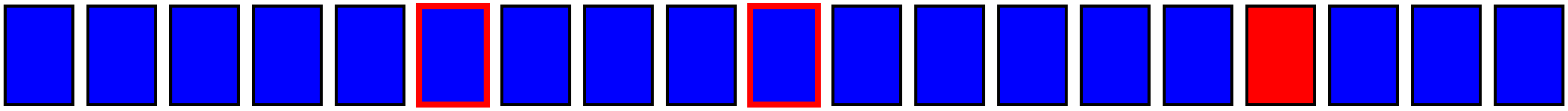
JOKE ON THE COMEDY

COMEDY



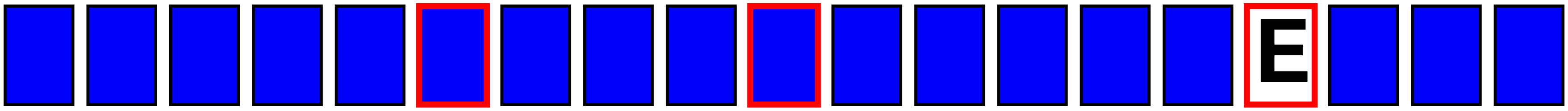
JOKE ON THE COMEDY

C O M E D Y



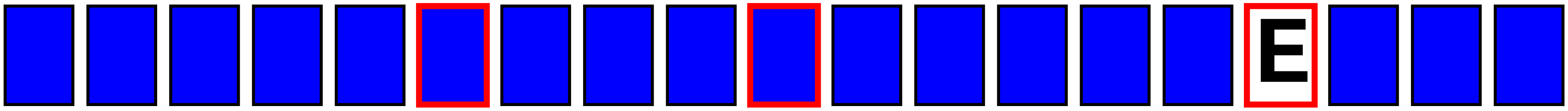
J O K E   O N   T H E   C O M E D Y

C O M E D Y



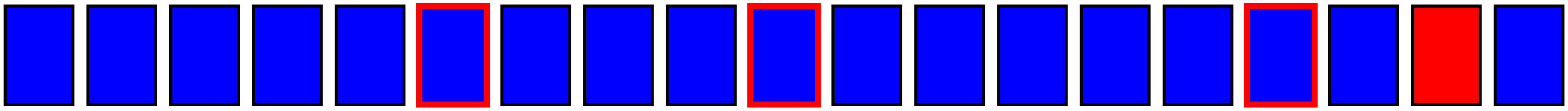
J O K E   O N   T H E   C O M E D Y

C O M E D Y

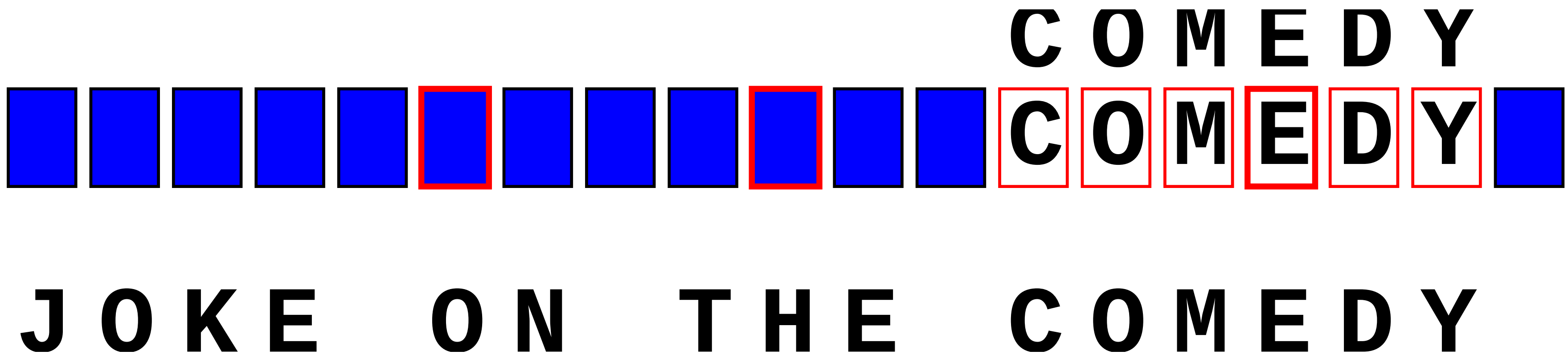


J O K E O N T H E C O M E D Y

COMEDY



JOKE ON THE COMEDY



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

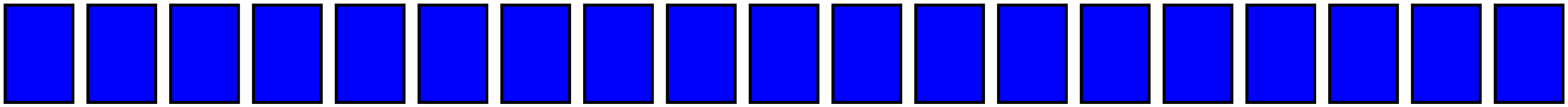
Pre-Computing the Skip Distance

```
pat: 543210
     COMEDY
txt: xxxxxOxxxxxxxxxxx...
           ↑
```

A 6	F 6	K 6	P 6	U 6	<space> 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

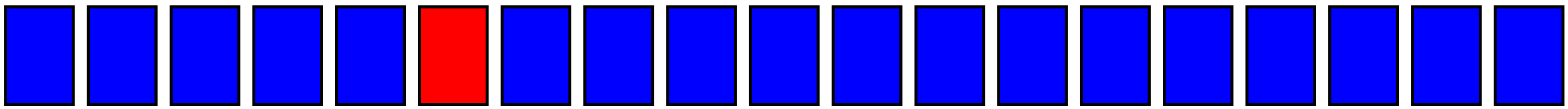


J O K E   O N   T H E   C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6	<space> 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

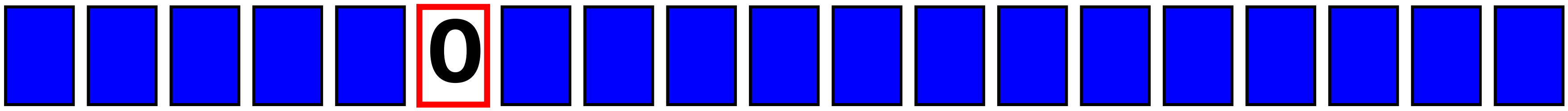


J O K E   O N   T H E   C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6	<space> 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	

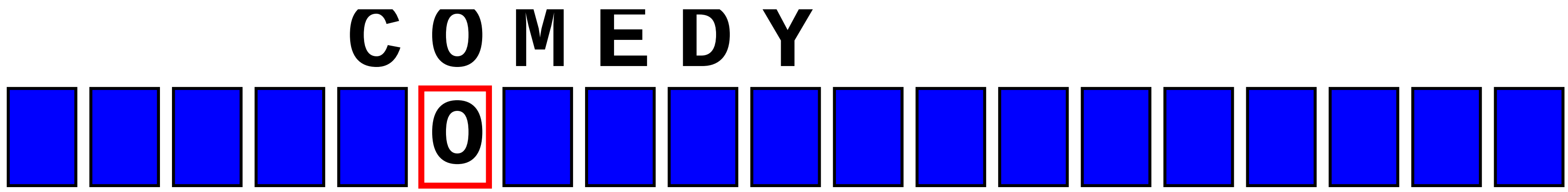
C O M E D Y



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

skip[c]:

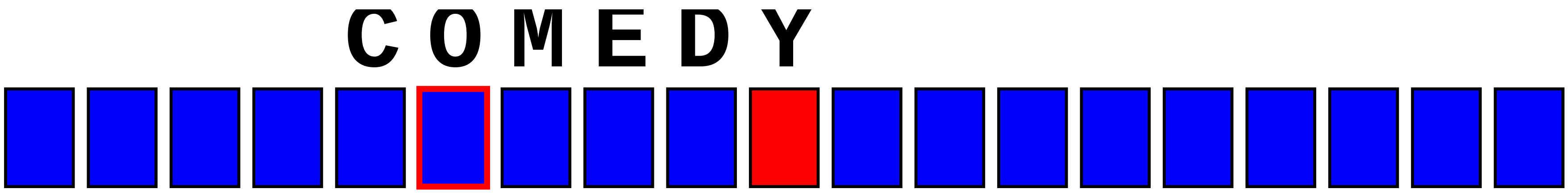
A 6	F 6	K 6	P 6	U 6	<space> 6
B 6	G 6	L 6	Q 6	V 6	
C 5	H 6	M 3	R 6	W 6	
D 1	I 6	N 6	S 6	X 6	
E 2	J 6	O 4	T 6	Y 0	
				Z 6	



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

skip[c]:

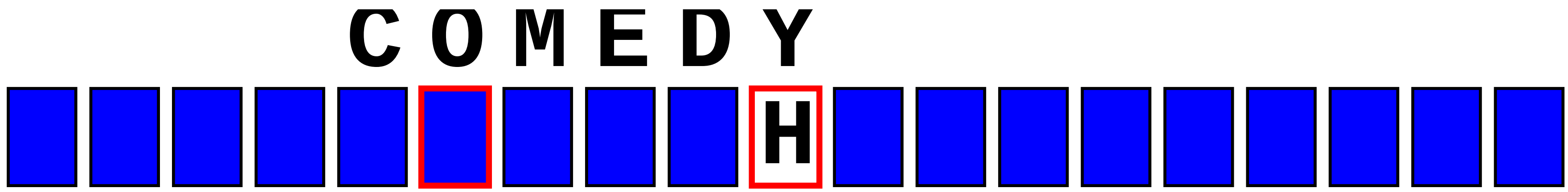
A 6	F 6	K 6	P 6	U 6	<space> 6
B 6	G 6	L 6	Q 6	V 6	
C 5	H 6	M 3	R 6	W 6	
D 1	I 6	N 6	S 6	X 6	
E 2	J 6	O 4	T 6	Y 0	
				Z 6	



JOKE ON THE COMEDY

skip[c]:

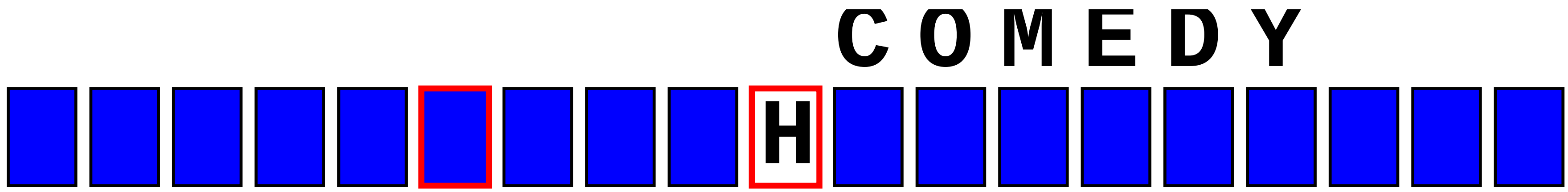
A 6	F 6	K 6	P 6	U 6	<space> 6
B 6	G 6	L 6	Q 6	V 6	
C 5	H 6	M 3	R 6	W 6	
D 1	I 6	N 6	S 6	X 6	
E 2	J 6	O 4	T 6	Y 0	
				Z 6	



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

skip[c]:

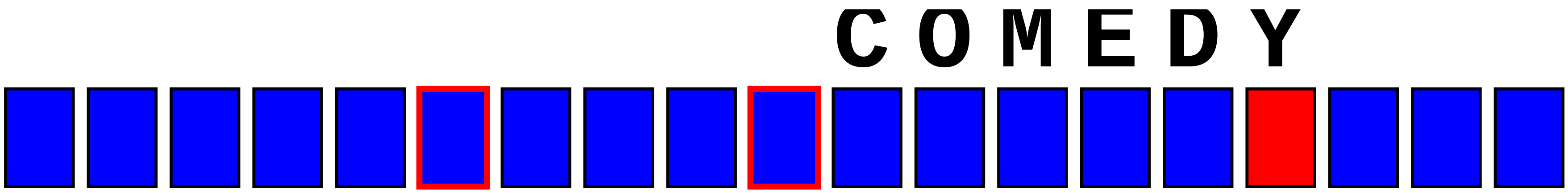
A 6	F 6	K 6	P 6	U 6	<space> 6
B 6	G 6	L 6	Q 6	V 6	
C 5	H 6	M 3	R 6	W 6	
D 1	I 6	N 6	S 6	X 6	
E 2	J 6	O 4	T 6	Y 0	
				Z 6	



JOKE ON THE COMEDY

skip[c]:

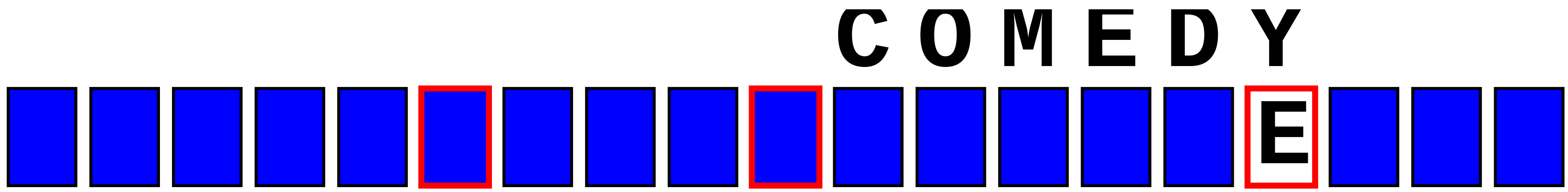
A 6	F 6	K 6	P 6	U 6	<space> 6
B 6	G 6	L 6	Q 6	V 6	
C 5	H 6	M 3	R 6	W 6	
D 1	I 6	N 6	S 6	X 6	
E 2	J 6	O 4	T 6	Y 0	
				Z 6	



JOKE ON THE COMEDY

skip[c]:

A 6	F 6	K 6	P 6	U 6	<space> 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	

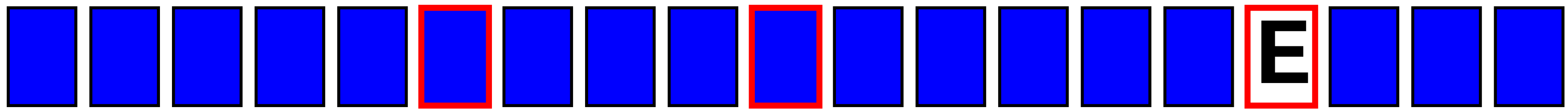


J O K E   O N   T H E   C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6	<space> 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

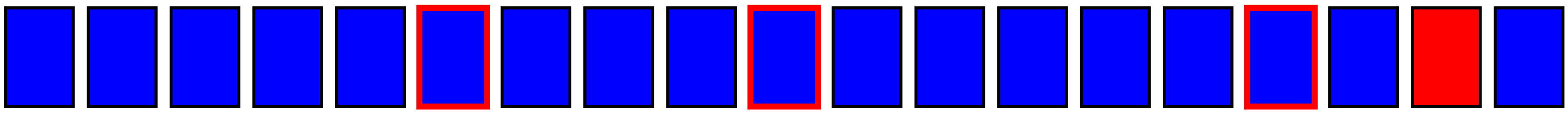


J O K E   O N   T H E   C O M E D Y

skip[c]:

A 6	F 6	K 6	P 6	U 6	<space> 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	

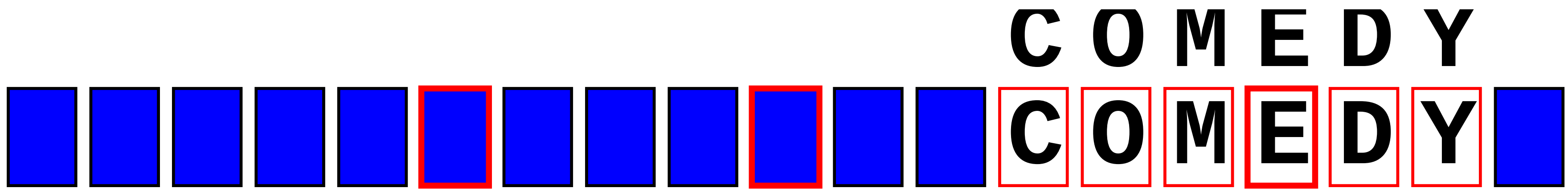
C O M E D Y



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	<space> 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	



JOKE ON THE COMEDY

skip[c]:

A 6	F 6	K 6	P 6	U 6	<space> 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------------
               |
```

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*!

But Wait! There's More!

```
pat:            NONPARTIPULAR
txt: ----------PAR----------
                 |
```

Slide 7 to match the *discovered substring*!

There are only $|\alpha| \times |pat|$ combinations, where $|\alpha|$ is the alphabet size. We can still pre-compute the skip distance.

The Delta Array

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

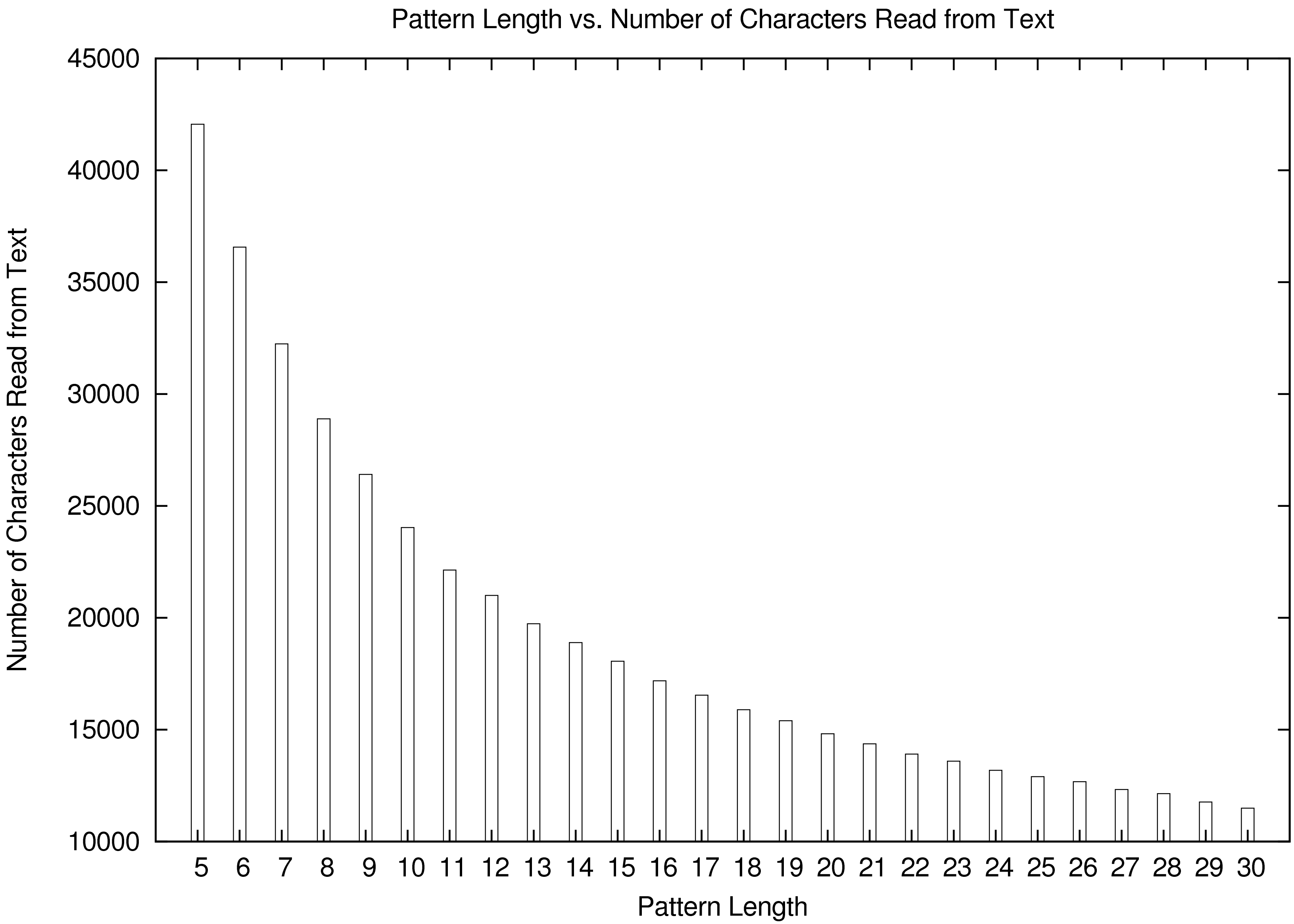
Performance

How does the algorithm perform?

In our test:

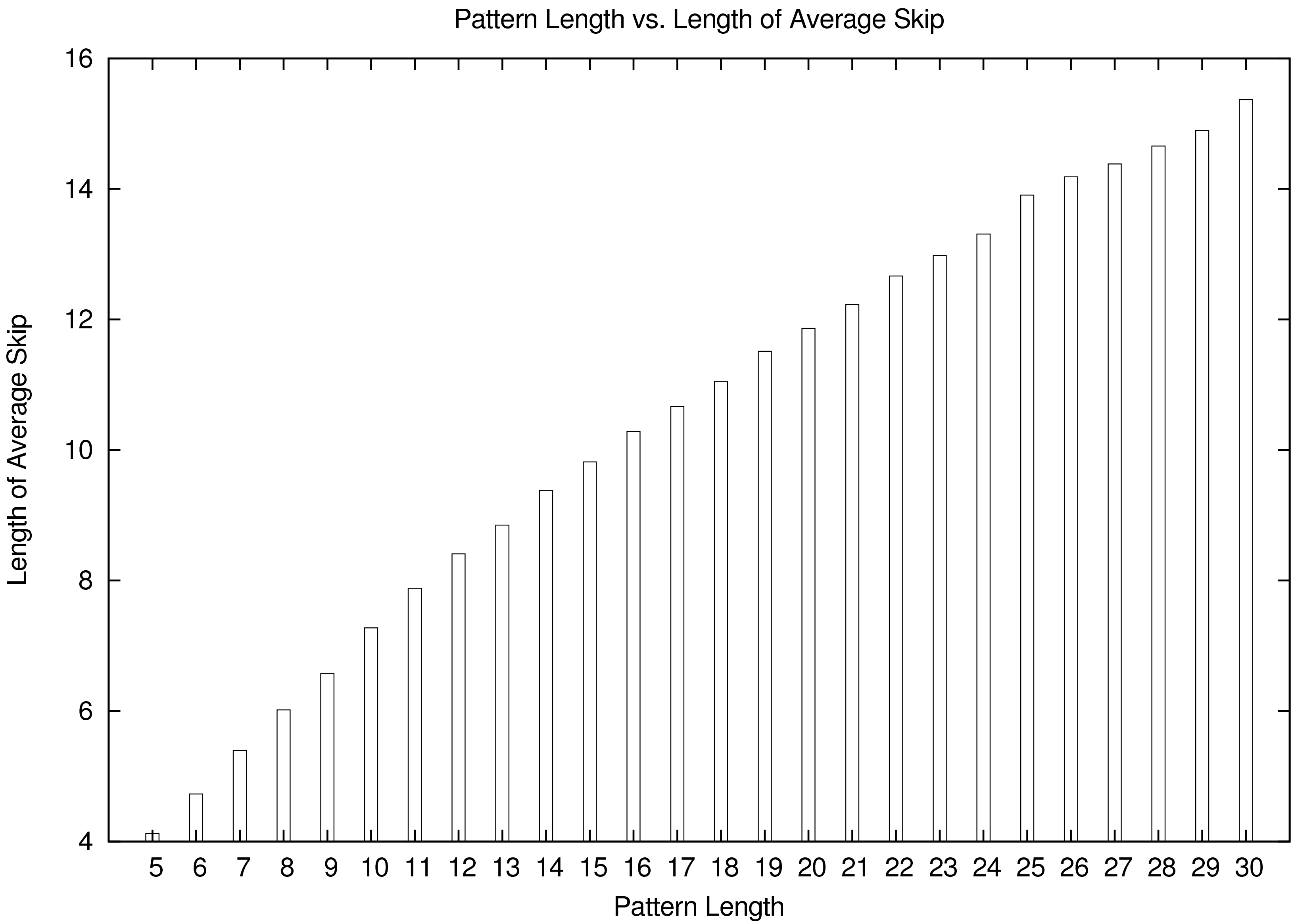
`txt`: English text of length 177,985.

`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.



Pattern Length vs. Number of Characters Read from Text
45000
40000
35000
30000
25000
20000
15000
10000
Number of Characters Read from Text
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Pattern Length

Naive algorithm would be a line at $\sim$180,000 reads.


Pattern Length vs. Length of Average Skip
Length of Average Skip
4
6
8
10
12
14
16
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Pattern Length

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.

Demo 1

The Obviously Correct Algorithm

```
(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))
```

(I omit type-like tests here.)

The Fast Algorithm

```
(defun fast-loop (pat j txt i)
  (declare :measure (measure pat j txt i)
           :well-founded-relation l<))
  (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))))))
```

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

Step 2: Fast Algorithm is Correct

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

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 the `delta` computation.

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))))
```

(On top of a library of useful utilities having nothing to do with this problem.)

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 c)
```

```
    (load 7)         ; 20    (load c)
    (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 c)
    (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 the two steps:

Step 1: The code computes the same thing as the function `fast`

Step 2: The function `fast` computes the same as `correct`

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))))
```

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