

# Knuth–Morris–Pratt String Search

Lawrence C. Paulson

June 7, 2026

## Abstract

The naive algorithm to search for a pattern  $p$  within a string  $a$  compares corresponding characters from left to right, and in case of a mismatch, shifts one position along  $a$  and starts again. The worst-case time is  $O(|p||a|)$ .

Knuth–Morris–Pratt [1] exploits the knowledge gained from the partial match, never re-comparing characters that matched and thereby achieving linear time. At the first mismatched character, it shifts  $p$  as far to the right as safely possible. To do so, it consults a precomputed table, based on the pattern  $p$ . The KMP algorithm is proved correct.

## Contents

<b>1</b>	<b>Knuth-Morris-Pratt fast string search algorithm</b>	<b>3</b>
1.1	General definitions . . . . .	3
1.2	The Build-table routine . . . . .	4
1.2.1	The invariant holds after an iteration . . . . .	5
1.2.2	The build-table loop and its correctness . . . . .	6
1.2.3	The build-table loop and its correctness . . . . .	6
1.2.4	Linearity of <i>buildtabW</i> . . . . .	7
1.3	The actual string search algorithm . . . . .	7
1.4	Examples . . . . .	8
1.5	Alternative approach, expressing the algorithms as while loops	9
1.5.1	Linearity of <i>buildtabW</i> . . . . .	10
1.5.2	The actual string search algorithm . . . . .	10

**Acknowledgements** This development closely follows a formal verification of the Knuth–Morris–Pratt algorithm by Jean-Christophe Filliâtre using Why3. Christian Zimmerer reworked the algorithms and termination proofs to use while loops. Tobias Nipkow made helpful suggestions.

# 1 Knuth-Morris-Pratt fast string search algorithm

Development based on Filiâtre's verification using Why3

Many thanks to Christian Zimmerer for versions of the algorithms as while loops

**theory** *KnuthMorrisPratt* **imports** *Collections.Diff-Array HOL-Library.While-Combinator*

**begin**

## 1.1 General definitions

**abbreviation** *array*  $\equiv$  *new-array*

**abbreviation** *length-array*  $:: 'a \text{ array} \Rightarrow \text{nat} \langle \|\cdot\| \rangle$

**where** *length-array*  $\equiv$  *array-length*

**notation** *array-get* (**infixl**  $\langle !! \rangle$  100)

**notation** *array-set* ( $\langle \cdot[- ::= \cdot] \rangle$  [1000,0,0] 900)

**definition** *matches*  $:: 'a \text{ array} \Rightarrow \text{nat} \Rightarrow 'a \text{ array} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$

**where** *matches*  $a \ i \ b \ j \ n = (i+n \leq \|a\| \wedge j+n \leq \|b\|$   
 $\wedge (\forall k < n. a!!(i+k) = b!!(j+k)))$

**lemma** *matches-empty* [*simp*]: *matches*  $a \ i \ b \ j \ 0 \longleftrightarrow i \leq \|a\| \wedge j \leq \|b\|$

*<proof>*

**lemma** *matches-right-extension*:

$\llbracket \text{matches } a \ i \ b \ j \ n;$   
 $\text{Suc } (i+n) \leq \|a\|;$   
 $\text{Suc } (j+n) \leq \|b\|;$   
 $a!!(i+n) = b!!(j+n) \rrbracket \Longrightarrow$   
*matches*  $a \ i \ b \ j \ (\text{Suc } n)$

*<proof>*

**lemma** *matches-contradiction-at-first*:

$\llbracket 0 < n; a!!i \neq b!!j \rrbracket \Longrightarrow \neg \text{matches } a \ i \ b \ j \ n$

*<proof>*

**lemma** *matches-contradiction-at-i*:

$\llbracket a!!(i+k) \neq b!!(j+k); k < n \rrbracket \Longrightarrow \neg \text{matches } a \ i \ b \ j \ n$

*<proof>*

**lemma** *matches-right-weakening*:

$\llbracket \text{matches } a \ i \ b \ j \ n; n' \leq n \rrbracket \Longrightarrow \text{matches } a \ i \ b \ j \ n'$

*<proof>*

**lemma** *matches-left-weakening-add*:

**assumes** *matches*  $a \ i \ b \ j \ n \ k \leq n$

**shows** *matches*  $a \ (i+k) \ b \ (j+k) \ (n-k)$

*<proof>*

**lemma** *matches-left-weakening*:

**assumes** *matches*  $a (i - (n - n')) b (j - (n - n')) n$   
**and**  $n' \leq n$   
**and**  $n - n' \leq i$   
**and**  $n - n' \leq j$   
**shows** *matches*  $a i b j n'$   
*<proof>*

**lemma** *matches-sym*: *matches*  $a i b j n \implies$  *matches*  $b j a i n$   
*<proof>*

**lemma** *matches-trans*:

$\llbracket$ *matches*  $a i b j n$ ; *matches*  $b j c k n \rrbracket \implies$  *matches*  $a i c k n$   
*<proof>*

Denotes the maximal  $n < j$  such that the first  $n$  elements of  $p$  match the last  $n$  elements of  $p[0..j - 1]$  The first  $n$  characters of the pattern have a copy starting at  $j - n$ .

**definition** *is-next* :: 'a array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool **where**

*is-next*  $p j n =$   
 $(n < j \wedge \text{matches } p (j-n) p 0 n \wedge (\forall m. n < m \wedge m < j \longrightarrow \neg \text{matches } p (j-m) p 0 m))$

**lemma** *next-iteration*:

**assumes** *matches*  $a (i-j) p 0 j$  *is-next*  $p j n j \leq i$   
**shows** *matches*  $a (i-n) p 0 n$   
*<proof>*

**lemma** *next-is-maximal*:

**assumes** *matches*  $a (i-j) p 0 j$  *is-next*  $p j n$   
**and**  $j \leq i n < m m < j$   
**shows**  $\neg \text{matches } a (i-m) p 0 m$   
*<proof>*

Filliâtre's version of the lemma above

**corollary** *next-is-maximal'*:

**assumes** *match*: *matches*  $a (i-j) p 0 j$  *is-next*  $p j n$   
**and** *more*:  $j \leq i i-j < k k < i-n$   
**shows**  $\neg \text{matches } a k p 0 \llbracket p \rrbracket$   
*<proof>*

**lemma** *next-1-0* [*simp*]: *is-next*  $p 1 0 \iff 1 \leq \llbracket p \rrbracket$   
*<proof>*

## 1.2 The Build-table routine

**definition** *buildtab-step* :: 'a array  $\Rightarrow$  nat array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat array  $\times$  nat  $\times$  nat **where**

$\text{buildtab-step } p \text{ next } i \ j =$   
 $(\text{if } p!!i = p!!j \text{ then } (\text{next}[\text{Suc } i ::= \text{Suc } j], \text{Suc } i, \text{Suc } j)$   
 $\text{else if } j=0 \text{ then } (\text{next}[\text{Suc } i ::= 0], \text{Suc } i, j)$   
 $\text{else } (\text{next}, i, \text{next}!!j))$

The conjunction of the invariants given in the Why3 development

**definition**  $\text{buildtab-invariant} :: 'a \text{ array} \Rightarrow \text{nat array} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**

$\text{buildtab-invariant } p \text{ next } i \ j =$   
 $(\| \text{next} \| = \| p \| \wedge i \leq \| p \|$   
 $\wedge j < i \wedge \text{matches } p \ (i-j) \ p \ 0 \ j$   
 $\wedge (\forall k. 0 < k \wedge k \leq i \longrightarrow \text{is-next } p \ k \ (\text{next}!!k))$   
 $\wedge (\forall k. \text{Suc } j < k \wedge k < \text{Suc } i \longrightarrow \neg \text{matches } p \ (\text{Suc } i - k) \ p \ 0 \ k))$

The invariant trivially holds upon initialisation

**lemma**  $\text{buildtab-invariant-init}: \| p \| \geq 2 \implies \text{buildtab-invariant } p \ (\text{array } 0 \ \| p \|) \ 1 \ 0$

$\langle \text{proof} \rangle$

### 1.2.1 The invariant holds after an iteration

each conjunct is proved separately

**lemma**  $\text{length-invariant}$ :

**shows**  $\text{let } (\text{next}', i', j') = \text{buildtab-step } p \ \text{next } i \ j \ \text{in } \| \text{next}' \| = \| \text{next} \|$

$\langle \text{proof} \rangle$

**lemma**  $i$ -invariant:

**assumes**  $\text{Suc } i < m$

**shows**  $\text{let } (\text{next}', i', j') = \text{buildtab-step } p \ \text{next } i \ j \ \text{in } i' \leq m$

$\langle \text{proof} \rangle$

**lemma**  $ji$ -invariant:

**assumes**  $\text{buildtab-invariant } p \ \text{next } i \ j$

**shows**  $\text{let } (\text{next}', i', j') = \text{buildtab-step } p \ \text{next } i \ j \ \text{in } j' < i'$

$\langle \text{proof} \rangle$

**lemma**  $\text{matches-invariant}$ :

**assumes**  $\text{buildtab-invariant } p \ \text{next } i \ j$  **and**  $\text{Suc } i < \| p \|$

**shows**  $\text{let } (\text{next}', i', j') = \text{buildtab-step } p \ \text{next } i \ j \ \text{in } \text{matches } p \ (i' - j') \ p \ 0 \ j'$

$\langle \text{proof} \rangle$

**lemma**  $\text{is-next-invariant}$ :

**assumes**  $\text{buildtab-invariant } p \ \text{next } i \ j$  **and**  $\text{Suc } i < \| p \|$

**shows**  $\text{let } (\text{next}', i', j') = \text{buildtab-step } p \ \text{next } i \ j \ \text{in } \forall k. 0 < k \longrightarrow k \leq i' \longrightarrow \text{is-next } p \ k \ (\text{next}'!!k)$

$\langle \text{proof} \rangle$

**lemma**  $\text{non-matches-aux}$ :

**assumes**  $\text{Suc } (\text{Suc } j) < k \text{ matches } p \ (\text{Suc } (\text{Suc } i) - k) \ p \ 0 \ k$

**shows**  $\text{matches } p \ (\text{Suc } i - (k - 1)) \ p \ 0 \ (k - 1)$

*<proof>*

**lemma** *non-matches-invariant*:

**assumes** *bt*: *buildtab-invariant* *p* *next* *i* *j* **and**  $\|p\| \geq 2$  *Suc* *i*  $< \|p\|$   
**shows** *let* (*next'*,*i'*,*j'*) = *buildtab-step* *p* *next* *i* *j* *in*  $\forall k. \text{Suc } j' < k \longrightarrow k < \text{Suc } i'$   
 $\longrightarrow \neg \text{matches } p (\text{Suc } i' - k) p 0 k$   
*<proof>*

**lemma** *buildtab-invariant*:

**assumes** *ini*: *buildtab-invariant* *p* *next* *i* *j*  
**and** *Suc* *i*  $< \|p\|$  (*next'*,*i'*,*j'*) = *buildtab-step* *p* *next* *i* *j*  
**shows** *buildtab-invariant* *p* *next'* *i'* *j'*  
*<proof>*

## 1.2.2 The build-table loop and its correctness

Declaring a partial recursive function with the *tailrec* option relaxes the need for a termination proof, because a tail-recursive recursion equation can never cause inconsistency.

## 1.2.3 The build-table loop and its correctness

**partial-function** (*tailrec*) *buildtab* :: '*a* array  $\Rightarrow$  nat array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat array **where**

*buildtab* *p* *next* *i* *j* =  
(*if* *Suc* *i*  $< \|p\|$   
  *then* *let* (*next'*,*i'*,*j'*) = *buildtab-step* *p* *next* *i* *j* *in* *buildtab* *p* *next'* *i'* *j'*  
  *else* *next*)

**declare** *buildtab.simps*[*code*]

Nevertheless, termination must eventually be shown: to use induction to reason about executions. We do so by defining a well founded relation. Termination proofs are by well-founded induction.

**definition** *rel-buildtab* *m* = *inv-image* (*lex-prod* (*measure* ( $\lambda i. m - i$ )) (*measure id*)) *snd*

**lemma** *wf-rel-buildtab*: *wf* (*rel-buildtab* *m*)

*<proof>*

**lemma** *buildtab-correct*:

**assumes** *k*:  $0 < k \wedge k < \|p\|$  **and** *ini*: *buildtab-invariant* *p* *next* *i* *j*  
**shows** *is-next* *p* *k* (*buildtab* *p* *next* *i* *j* !! *k*)  
*<proof>*

Before building the table, check for the degenerate case

**definition** *table* :: '*a* array  $\Rightarrow$  nat array **where**

*table* *p* = (*if*  $\|p\| > 1$  *then* *buildtab* *p* (*array* 0  $\|p\|$ ) 1 0  
  *else* *array* 0  $\|p\|$ )

**declare** *table-def*[code]

**lemma** *is-next-table*:

**assumes**  $0 < j \wedge j < \|p\|$

**shows** *is-next*  $p\ j$  (*table*  $p\ !!j$ )

*<proof>*

### 1.2.4 Linearity of *buildtab* $W$

**partial-function** (*tailrec*) *T-buildtab* :: 'a array  $\Rightarrow$  nat array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat **where**

*T-buildtab*  $p\ next\ i\ j\ t =$

(if *Suc*  $i < \|p\|$

then let ( $next', i', j'$ ) = *buildtab-step*  $p\ next\ i\ j$  in *T-buildtab*  $p\ next'\ i'\ j'$  (*Suc*  $t$ )

else  $t$ )

**lemma** *T-buildtab-correct*:

**assumes** *ini*: *buildtab-invariant*  $p\ next\ i\ j$

**shows** *T-buildtab*  $p\ next\ i\ j\ t \leq 2*\|p\| - 2*i + j + t$

*<proof>*

**lemma** *T-buildtab-linear*:

**assumes**  $2 \leq \|p\|$

**shows** *T-buildtab*  $p$  (*array*  $0\ \|p\|$ )  $1\ 0\ 0 \leq 2*(\|p\| - 1)$

*<proof>*

## 1.3 The actual string search algorithm

**definition**

*KMP-step*  $p\ next\ a\ i\ j =$

(if  $a!!i = p!!j$  then (*Suc*  $i$ , *Suc*  $j$ )

else if  $j=0$  then (*Suc*  $i$ ,  $0$ ) else ( $i$ , *next!!j*))

The conjunction of the invariants given in the Why3 development

**definition** *KMP-invariant* :: 'a array  $\Rightarrow$  'a array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool **where**

*KMP-invariant*  $p\ a\ i\ j =$

$(j \leq \|p\| \wedge j \leq i \wedge i \leq \|a\| \wedge \text{matches } a\ (i-j)\ p\ 0\ j$

$\wedge (\forall k < i-j. \neg \text{matches } a\ k\ p\ 0\ \|p\|))$

The invariant trivially holds upon initialisation

**lemma** *KMP-invariant-init*: *KMP-invariant*  $p\ a\ 0\ 0$

*<proof>*

The invariant holds after an iteration

**lemma** *KMP-invariant*:

**assumes** *ini*: *KMP-invariant*  $p\ a\ i\ j$

**and**  $j < \|p\|$  **and**  $i < \|a\|$

**shows** let ( $i', j'$ ) = *KMP-step*  $p$  (*table*  $p$ )  $a\ i\ j$  in *KMP-invariant*  $p\ a\ i'\ j'$

*<proof>*

The first three arguments are precomputed so that they are not part of the inner loop.

```
partial-function (tailrec) search :: nat ⇒ nat ⇒ nat array ⇒ 'a array ⇒ 'a array
⇒ nat ⇒ nat ⇒ nat * nat where
  search m n next p a i j =
    (if j < m ∧ i < n then let (i',j') = KMP-step p next a i j in search m n next p
a i' j'
    else (i,j))
declare search.simps[code]
```

**definition** *rel-KMP* n = *lex-prod* (*measure* (λi. n-i)) (*measure* id)

**lemma** *wf-rel-KMP*: *wf* (*rel-KMP* n)  
⟨*proof*⟩

Also expresses the absence of a match, when  $r = \|a\|$

```
definition first-occur :: 'a array ⇒ 'a array ⇒ nat ⇒ bool
where first-occur p a r = ((r < \|a\| → matches a r p 0 \|p\|) ∧ (∀ k < r. ¬
matches a k p 0 \|p\|))
```

**lemma** *KMP-correct*:

```
assumes ini: KMP-invariant p a i j
defines [simp]: next ≡ table p
shows let (i',j') = search \|p\| \|a\| next p a i j in first-occur p a (if j' = \|p\| then
i' - \|p\| else i')
⟨proof⟩
```

```
definition KMP-search :: 'a array ⇒ 'a array ⇒ nat × nat where
  KMP-search p a = search \|p\| \|a\| (table p) p a 0 0
declare KMP-search-def[code]
```

**lemma** *KMP-search*:

```
(i,j) = KMP-search p a ⇒ first-occur p a (if j = \|p\| then i - \|p\| else i)
⟨proof⟩
```

## 1.4 Examples

Building the table, examples from the KMP paper and from Cormen et al.

```
definition Knuth-pattern = array-of-list [1,2,3,1,2,3,1,3,1,2::nat]
```

```
value list-of-array (table Knuth-pattern)
```

```
definition CLR-pattern = array-of-list [1,2,1,2,1,2,1,2,3,1::nat]
```

```
value list-of-array (table CLR-pattern)
```

Worst-case string searches

```
definition bad-list :: nat ⇒ nat list
```

**where** *bad-list*  $n = \text{replicate } n \ 0 \ @ \ [1]$

**definition** *bad-pattern* = *array-of-list* (*bad-list* 1000)

**definition** *bad-string* = *array-of-list* (*bad-list* 2000000)

**definition** *worse-string* = *array-of-list* (*replicate* 2000000 ( $0::\text{nat}$ ))

**definition** *lousy-string* = *array-of-list* (*concat* (*replicate* 2002 (*bad-list* 999)))

**value** *list-of-array* (*table* *bad-pattern*)

A successful search

**value** *KMP-search* *bad-pattern* *bad-string*

The search above from the specification alone, i.e. brute-force

**lemma** *matches* *bad-string* ( $2000001-1001$ ) *bad-pattern* 0 1001  
<proof>

Unsuccessful searches

**value** *KMP-search* *bad-pattern* *worse-string*

The search above from the specification alone, i.e. brute-force

**lemma**  $\forall k < 2000000. \neg \text{matches } \textit{worse-string } k \ \textit{bad-pattern } 0 \ 1001$   
<proof>

**value** *KMP-search* *lousy-string* *bad-string*

**lemma**  $\forall k < \|\textit{lousy-string}\|. \neg \text{matches } \textit{lousy-string } k \ \textit{bad-pattern } 0 \ 1001$   
<proof>

## 1.5 Alternative approach, expressing the algorithms as while loops

**definition** *builddtabW*:: 'a array  $\Rightarrow$  nat array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat array option  
**where**

*builddtabW*  $p \ \textit{next} \ i \ j \equiv$   
*map-option* *fst* (*while-option* ( $\lambda(-, i', -). \text{Suc } i' < \|p\|$ )  
( $\lambda(\textit{next}', i', j'). \text{builddtab-step } p \ \textit{next}' \ i' \ j'$ )  
( $\textit{next}, i, j$ ))

**lemma** *builddtabW-halts*:

**assumes** *builddtab-invariant*  $p \ \textit{next} \ i \ j$

**shows**  $\exists y. \text{builddtabW } p \ \textit{next} \ i \ j = \text{Some } y$

<proof>

**lemma** *builddtabW-correct*:

**assumes**  $k: 0 < k \wedge k < \|p\|$  **and** *ini*: *builddtab-invariant*  $p \ \textit{next} \ i \ j$

**shows** *is-next*  $p \ k$  (the (*builddtabW*  $p \ \textit{next} \ i \ j$ ) !!  $k$ )

<proof>

### 1.5.1 Linearity of *buildtabW*

**definition** *T-buildtabW* :: 'a array ⇒ nat array ⇒ nat ⇒ nat ⇒ nat ⇒ nat option  
**where**

*T-buildtabW* p n*xt* i j t ≡ map-option (λ(-, -, -, r). r)  
 (while-option (λ(-, i, -, -). Suc i < ||p||)  
 (λ(nxt, i, j, t). let (nxt', i', j') = buildtab-step p n*xt* i j in (nxt', i',  
 j', Suc t))  
 (nxt, i, j, t))

**lemma** *T-buildtabW-halts*:

**assumes** *buildtab-invariant* p n*xt* i j  
**shows** ∃ y. *T-buildtabW* p n*xt* i j t = Some y  
 ⟨*proof*⟩

**lemma** *T-buildtabW-correct*:

**assumes** *ini*: *buildtab-invariant* p n*xt* i j  
**shows** the (*T-buildtabW* p n*xt* i j t) ≤ 2\*||p|| - 2\*i + j + t  
 ⟨*proof*⟩

**lemma** *T-buildtabW-linear*:

**assumes** 2 ≤ ||p||  
**shows** the (*T-buildtabW* p (array 0 ||p||) 1 0 0) ≤ 2\*(||p|| - 1)  
 ⟨*proof*⟩

### 1.5.2 The actual string search algorithm

**definition** *searchW* :: nat ⇒ nat ⇒ nat array ⇒ 'a array ⇒ 'a array ⇒ nat ⇒  
 nat ⇒ (nat \* nat) option **where**

*searchW* m n n*xt* p a i j = while-option (λ(i, j). j < m ∧ i < n) (λ(i, j). *KMP-step*  
 p n*xt* a i j) (i, j)

**lemma** *searchW-halts*:

**assumes** *KMP-invariant* p a i j  
**shows** ∃ y. *searchW* ||p|| ||a|| (table p) p a i j = Some y  
 ⟨*proof*⟩

**lemma** *KMP-correctW*:

**assumes** *ini*: *KMP-invariant* p a i j  
**defines** [*simp*]: n*xt* ≡ table p  
**shows** let (i', j') = the (*searchW* ||p|| ||a|| n*xt* p a i j) in *first-occur* p a (if j' =  
 ||p|| then i' - ||p|| else i')  
 ⟨*proof*⟩

**definition** *KMP-searchW* :: 'a array ⇒ 'a array ⇒ nat × nat **where**

*KMP-searchW* p a = the (*searchW* ||p|| ||a|| (table p) p a 0 0)

**declare** *KMP-searchW-def*[code]

**lemma** *KMP-searchW*:

(i, j) = *KMP-searchW* p a ⇒ *first-occur* p a (if j = ||p|| then i - ||p|| else i)

*<proof>*

**end**

## **References**

- [1] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.