


# String Searching


Opera

UCC Book of Modules, 201 x

www.ucc.ie/modules/descriptions/page014.html

net Info 1 of 2

 **UCC**  
University College Cork, Ireland  
Coláiste na hOllscoile Corcaigh



Undergraduate Postgraduate **Book of Modules** Marks and Standards

*Students should note that all of the modules below may not be available to them.*

**Undergraduate** students should refer to the relevant section of the UCC [Undergraduate Calendar](#) for their programme requirements.

**Postgraduate** students should refer to the relevant section of the UCC [Postgraduate Calendar](#) for their programme requirements.

Computer Science

- [CS1050 Fundamentals of Internet Computing](#)
- [CS1061 Programming in C](#)
- [CS1065 Computer Applications with Visual Basic](#)
- [CS1068 Introductory Programming in Python](#)
- [CS1069 Introduction to Internet Technologies](#)
- [CS1070 Introductory Python Programming for Digital Humanities](#)
- [CS1106 Introduction to Relational Databases](#)
- [CS1110 Systems Organisation I](#)
- [CS1111 Systems Organisation II](#)
- [CS1112 Foundations of Computer Science I](#)
- [CS1113 Foundations of Computer Science II](#)
- [CS1115 Web Development 1](#)
- [CS1116 Web Development 2](#)
- [CS1117 Introduction to Programming](#)
- [CS1118 Multimedia](#)
- [CS2051 Introduction to Digital Media](#)
- [CS2052 Introduction to Internet Information Systems](#)
- [CS2501 Database Design and Administration](#)
- [CS2503 Logic Design](#)

Suppose we are given a source text file, and we want to search for the first occurrence of a character string.

In CS2515, we enabled searching for words in a text file by pre-processing the file to build a binary search tree of all the words, and then just searched that tree.

If there are  $n$  distinct words in the text file, then any search can be done in time  $O(\log n)$ .

Why won't this work here?

We are asked to search for a *character string*.

This is more general than a word search

- we might be looking for a substring of the word
- we might be looking for a phrase containing multiple words

o n e e g g o r t w o ?

Find 'egg' in the above string.

There is a simple and obvious algorithm for achieving this ...

Brute force: for each char in the file, try to match the string starting there.  
(pseudocode)

Input: target (a string to find), source (a string to be searched through)

Output: the start position of the target in the source, or -1

for each possible starting character in the source

  j is the position of that character in the source

  i = 0   #the position being examined in the target

  while not failed

    if i = length(target)

      return (j-len(target)) #success

    if j = length of source

      failed                           #gone off end of source text

    else if target[i] != source[j]

      failed                           #characters didn't match

    else

      i += 1

      j += 1

return -1



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
e	g	g												
0	1	2												

j = 0

i = 0



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
e	g	g													
	0	1	2												

j = 0

i = 0



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	e	g	g											
	0	1	2											

$j = 1$

$i = 0$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	n	*	*	*	*	*	*	*	*	*	*	*	*	*
	e	g	g											
	0	1	2											

$j = 1$   
 $i = 0$





0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	n	*	*	*	*	*	*	*	*	*	*	*	*	*
		e	g	g										
		0	1	2										

$j = 2$

$i = 0$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
o n e \* \* \* \* \* \* \* \* \* \* \*  
e g g  
0 1 2



j = 2

i = 0

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
o n e \* \* \* \* \* \* \* \* \* \* \*  
e g g  
0 1 2



$j = 3$

$i = 1$




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	n	e		*	*	*	*	*	*	*	*	*	*	*
			e	g	g									
			0	1	2									

j = 3

i = 0


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
o n e \* \* \* \* \* \* \* \* \* \* \*  
e g g  
0 1 2



j = 4

i = 0

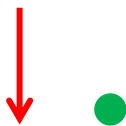
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	n	e		e	*	*	*	*	*	*	*	*	*	*
				e	g	g								
				0	1	2								



$j = 4$

$i = 0$


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	n	e		e	g	*	*	*	*	*	*	*	*	*
				e	g	g								
				0	1	2								



$j = 5$

$i = 1$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
o	n	e		e	g	g	*	*	*	*	*	*	*	*
				e	g	g								
				0	1	2								




j = 6

i = 2



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
o n e e g g o r t w o ?  
e g g  
0 1 2



j = 7

i = 3

## Complexity.

What happens when we search for "aaaaaaa" in a source consisting of repeated "aaaaaabaabaaaaaabaabaaaaaabaab"?

We do 7 checks for the 1<sup>st</sup> 'a' (6 succeed, and the 7<sup>th</sup> fails on 'b')  
6 checks for the 2<sup>nd</sup> 'a', 5 checks for the 3<sup>rd</sup> 'a', and so on, and  
then again 7 checks for the 'a' in 8<sup>th</sup> position, 6 checks for the 'a' in  
9<sup>th</sup> position, etc

Let  $n$  be the length of the source text.

Let  $m$  be the length of the target.

We might need  $0.5 * m * (m+1)$  checks, repeated  $n/m$  times.

$n/m * O(m^2)$ , which is  $O(n * m)$

Or:

$n$  times round the outer loop, and each loop could be  $m$  checks,  
so  $O(n * m)$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
a	c	c	o	m	m	o	d	a	t	e					

Can we do any better?




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
a	c	c	o	m	m	o	d	a	t	e					



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	a	c	c	o	m	m	o	d	a	t	e				

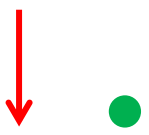


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	y	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	a	c	c	o	m	m	o	d	a	t	e				




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	y	*	*	*	*	*	*	*	*	*	*	*	*	*	*
		a	c	c	o	m	m	o	d	a	t	e			

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  
m y a \* \* \* \* \* \* \* \* \* \* \* \* \*  
a c c o m m o d a t e






(match the next 9 characters successfully)




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	y	a	c	c	o	m	m	o	d	a	t	*	*	*	*
		a	c	c	o	m	m	o	d	a	t	e			



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	y	a	c	c	o	m	m	o	d	a	t	i	*	*	*
		a	c	c	o	m	m	o	d	a	t	e			




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	y	a	c	c	o	m	m	o	d	a	t	i	*	*	*
			a	c	c	o	m	m	o	d	a	t	e		

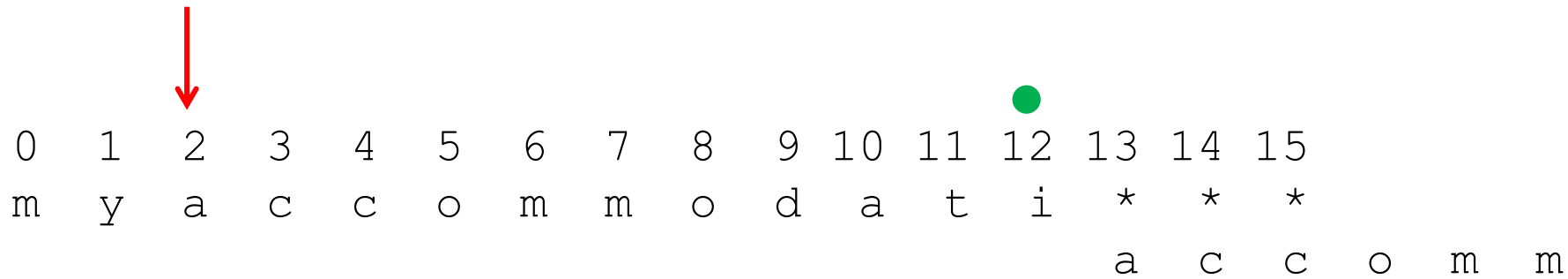


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	y	a	c	c	o	m	m	o	d	a	t	i	*	*	*
				a	c	c	o	m	m	o	d	a	t	e	

If we have partially matched the first  $k$  characters of our target, and then we fail on the  $(k+1)^{\text{th}}$  character, we should be able to work out whether or not we need to check the next  $k$  positions for the start of the string.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m	y	a	c	c	o	m	m	o	d	a	t	i	*	*	*
		a	c	c	o	m	m	o	d	a	t	e			



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15				
m	y	a	c	c	o	m	m	o	d	a	t	i	*	*	*				
													a	c	c	o	m	m	

jump past the sequence of characters that we failed to match?

but is this guaranteed not to miss any early matches?



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
a	m	a	l	g	a	m	a	t	e						





0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
a	m	a	l	g	a	m	a	t	e						

(match the next 6 characters successfully)




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	m	a	l	g	a	m	a	*	*	*	*	*	*	*	*
a	m	a	l	g	a	m	a	t	e						

(match the next 6 characters successfully)




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	m	a	l	g	a	m	a	l	*	*	*	*	*	*	*
a	m	a	l	g	a	m	a	t	e						



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
a	m	a	l	g	a	m	a	l	*	*	*	*	*	*	*			
									a	m	a	l	g	a	m	a	t	e

Jump forward to here?



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
a	m	a	l	g	a	m	a	l	g	a	m	a	t	e				
									a	m	a	l	g	a	m	a	t	e

Jump forward to here?

Missed the first (and maybe only) appearance of the target

## Computing the correct maximum jump:

..	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	a	m	a	l	g	a	m	a	X	*	*	*	*	*	*	*	*
	a	m	a	l	g	a	m	a	t	e							
	0	1	2	3	4	5	6	7	8	9							

mismatch at position 35

previous 3 chars match 1<sup>st</sup> 3 characters of target

..	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	a	m	a	l	g	a	m	a	X	*	*	*	*	*	*	*	*
						a	m	a	l	g	a	m	a	t	e		

## Computing the correct maximum jump:

..	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	a	m	a	l	g	a	m	a	X	*	*	*	*	*	*	*	*
	a	m	a	l	g	a	m	a	t	e							
	0	1	2	3	4	5	6	7	8	9							

mismatch at position 35

previous 3 chars match 1<sup>st</sup> 3 characters of target

..	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	a	m	a	l	g	a	m	a	X	*	*	*	*	*	*	*	*
						a	m	a	l	g	a	m	a	t	e		

next potential match starts at position 32

If the match failed at position  $j$  in the source find the longest *prefix* of the target that matches if we move it up to finish at position  $j-1$  (and we must move it up at least one position)

# Computing the jump:

a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
a	b	a	c	a	b	a	d	a	b	c									
0	1	2	3	4	5	6	7	8	9	10									

X at 7, prev. 3 match start, so  
move target 0 to what was  $7-3 = 4$

a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
				a	b	a	c	a	b	a	d	a	b	c					
				0	1	2	3	4	5	6	7	8	9	10					

X at 5, prev. 1 matches  
start, so move 0 to what  
was  $5-1 = 4$

a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
								a	b	a	c	a	b	a	d	a	b	c	
								0	1	2	3	4	5	6	7	8	9	10	

X at 1, no prefix matches  
(apart from the same  
start) so move 0 up one  
place to what was 1

a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
									a	b	a	c	a	b	a	d	a	b	c
									0	1	2	3	4	5	6	7	8	9	10



If we fail at position  $i$  in the target and position  $j$  in the source,  
then we know that position 0 to  $(i-1)$  in the target have matches to positions  
 $(j-i)$  to  $(j-1)$  in the source

									j										
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
				a	b	a	c	a	b	a	d	a	b	c					
				0	1	2	3	4	5	6	7	8	9	10					
									i										

so to decide the jump position, we don't need to look at the source ...

If the match fails at position  $i$  in the target, just find the longest prefix of the  
target that matches if we move it up to finish at position  $i-1$  in the *target*.

Since we are not looking at the source, we can do all of this offline before  
we begin. For each position  $i$  in the target, if we were to fail at this position,  
compute the number of positions we must shift the target forward.

	a	m	a	l	g	a	m	a	t	e	
X											
a	m	a	l	g	a	m	a	t	e		
0	1	2	3	4	5	6	7	8	9		1 position

	a	m	a	l	g	a	m	a	t	e	
a	X										
a	m	a	l	g	a	m	a	t	e		
0	1	2	3	4	5	6	7	8	9		1 position (maybe X = 'a'?)

		a	m	a	l	g	a	m	a	t	e	
a	m	X										
a	m	a	l	g	a	m	a	t	e			
0	1	2	3	4	5	6	7	8	9		2 positions (maybe X = 'a'?) (actually, we know it can't be 'a', because it failed with 'a', but in general ...?)	

		a	m	a	l	g	a	m	a	t	e	
a	m	a	X									
a	m	a	l	g	a	m	a	t	e			
0	1	2	3	4	5	6	7	8	9		2 positions	
											(maybe X = 'm'?)	

				a	m	a	l	g	a	m	a	t	e
a	m	a	l	X									
a	m	a	l	g	a	m	a	t	e				
0	1	2	3	4	5	6	7	8	9		4 positions		
											(maybe X = 'a'?)		

and so on ...

					a	m	a	l	g	a	m	a	t	e
a	m	a	l	g	a	m	a	X						
a	m	a	l	g	a	m	a	t	e					
0	1	2	3	4	5	6	7	8	9		5 positions			
											(maybe X = 'l'?)			

For position  $i$  ( $>0$ ) in the target,  $\pi(i)$  is the length of prefix of target that ends in position  $i$

		a	b	a	c	a	b	a	d	a	b	c						
i	$\pi$	0	1	2	3	4	5	6	7	8	9	10						
0	0	.																
1	0		.															
2	1			a	b	a	c	a	b	a	d	a	b	c				
3	0				.													
4	1					a	b	a	c	a	b	a	d	a	b	c		
5	2					a	b	a	c	a	b	a	d	a	b	c		
6	3					a	b	a	c	a	b	a	d	a	b	c		
7	0								.									
8	1									a	b	a	c	a	b	a	d	a
9	2									a	b	a	c	a	b	a	d	a
10	0											.						

$\pi(i)$  is either 0 or  $\pi(i-1)+1$  and so  $\forall i \pi(i) \leq i$  (we will use this later)

$\pi(i) == \pi(i-1)+1$  if and only if  $\text{target}[i] == \text{target}[\pi(i-1)]$

We can use this to compute  $\pi(i)$  in an algorithm

## Computing the $\pi$ table for a target string (pseudocode)

```
 $\pi[0] = 0$   
for each  $i$  from 1 to  $\text{length}(\text{target})-1$   
  if  $\text{target}[i] == \text{target}[\pi[i-1]]$   
     $\pi[i] = \pi[i-1] + 1$   
  else  
     $\pi[i] = 0$ 
```

Complexity:  $O(m)$ , where  $m$  is the length of the target

## Knuth Morris Pratt algorithm for string matching (pseudocode)

Input: target (a string), source (the source text)

Output: the start position of the first occurrence of target in source, or -1

j = 0 #the current index in source being checked

i = 0 #the current index in target being checked

pi = [0]\*length(target)

compute\_pi(target,pi)

while i < length(target) and j < length(source)

    if target[i] == source[j]      #current chars match

        i += 1

        j += 1

    else if i == 0                  #failed match with start of target

        j += 1                      # so force a step forward

    else

        i = pi[i-1]                #jump forward (note: j does not move)

if i == length(target)

    return j-length(target)

else

    return -1

j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
	a	b	a	c	a	b	a	d	a	b	c									
i:	0	1	2	3	4	5	6	7	8	9	10									

$j = 7 \otimes i = 7$ , so set  $i = \pi(6) = 3$

j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
					a	b	a	c	a	b	a	d	a	b	c					
				i:	0	1	2	3	4	5	6	7	8	9	10					

$j = 9 \otimes i = 5$ , so set  $i = \pi(4) = 1$

j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
									a	b	a	c	a	b	a	d	a	b	c	
								i:	0	1	2	3	4	5	6	7	8	9	10	

$j = 9 \otimes i = 1$ , so set  $i = \pi(0) = 0$

j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	a	b	a	c	a	b	a	c	a	a	b	a	c	a	b	a	d	a	b	c
										a	b	a	c	a	b	a	d	a	b	c
									i:	0	1	2	3	4	5	6	7	8	9	10

	a	b	a	c	a	b	a	d	a	b	c
$\pi$	0	1	2	3	4	5	6	7	8	9	0
	0	0	1	0	1	2	3	0	1	2	0

## Knuth Morris Pratt algorithm for string matching (pseudocode)

Input: target (a string), source (the source text)

Output: the start position of the first occurrence of target in source, or -1

j = 0 #the current index in source being checked

i = 0 #the current index in target being checked

pi = [0]\*length(target)

compute\_pi(target,pi)

while i < length(target) and j < length(source)

    if target[i] == source[j]      #current chars match

        i += 1

        j += 1

    else if i == 0                  #failed match with start of target

        j += 1                      # so force a step forward

    else

        i = pi[i-1]                #jump forward (note: j does not move)

if i == length(target)

    return j-length(target)

else

    return -1



Complexity ( $m$  is the length of the target,  $n$  is the length of the source)

Inside the loop there are  $O(1)$  operations.

We progress round the loop when  $j$  increases by 1, or  $i$  decreases (since  $i$  is assigned to  $\pi(i-1)$ , and  $\pi(i) \leq i$ , and  $\pi(i) \leq \pi(i-1) + 1$ ).

$j$  can only increase  $n$  times.

$i$  either increases in step with  $j$ , or stays the same, or decreases, but never goes below 0, and never goes above  $m$ .

The worst we can do is always change  $i$  each time round the loop.

Since  $i$  only increases when  $j$  increases, it has a maximum of  $n$  steps up.

Therefore it has a maximum of  $n$  steps down.

Therefore the loop is executed a maximum of  $2n$  times, which is  $O(n)$ .

The initialisation was  $O(m)$ . Therefore algorithm is  $O(n+m)$

Note – this is 'optimal' for the worst case.

We have to look at every character in the target at least once, and every character in the source text at least once, so at least  $O(n+m)$

# Next Lecture

More text processing