

Strings and Matching

- Strings
- Pattern Matching
- Complexity Analysis

❖ Strings

A **string** is a sequence of symbols.

An **alphabet** Σ is the set of possible symbols in strings.

Examples of strings:

- C program, HTML document, PDF document
- DNA sequence, Digitized image, Video (mp4)

Examples of alphabets:

- ASCII (1-byte chars), Unicode (multi-byte chars)
- $\{0,1\}$ (bits), $\{0..9\}$ (digits), $\{A,C,G,T\}$ (genome sequences)

Often, the "symbols" are letters (so we call them **characters** from now on)

❖ ... Strings

Notation:

- $length(P)$... #characters in P
- λ ... empty string ($length(\lambda) = 0$) $\lambda\omega = \omega = \omega\lambda$
- Σ^m ... set of all strings of length m over alphabet Σ
- Σ^* ... set of all strings over alphabet Σ
- $v\omega$ denotes the **concatenation** of strings v and ω
- **substring** of P ... any string Q such that $P = vQ\omega$, for some $v, \omega \in \Sigma^*$
- **prefix** of P ... any string Q such that $P = Q\omega$, for some $\omega \in \Sigma^*$
- **suffix** of P ... any string Q such that $P = \omega Q$, for some $\omega \in \Sigma^*$

❖ ... Strings

Example ...

The string **a/a** of length 3 over the ASCII alphabet has

- 4 prefixes: "" "a" "a/" "a/a"
- 4 suffixes: "a/a" "/a" "a" ""
- 6 substrings: "" "a" "/" "a/" "/a" "a/a"

Note: "" means the same as λ (= empty string)

❖ ... Strings

ASCII (American Standard Code for Information Interchange)

- Specifies mapping of 128 characters to integers 0..127
- The characters encoded include:
 - letters: A-Z and a-z, digits: 0-9, punctuation symbols
 - special non-printing characters: e.g. *newline* and *space*

Ascii	Char	Ascii	Char	Ascii	Char	Ascii	Char
0	Null	32	Space	64	@	96	`
1	Start of heading	33	!	65	A	97	a
2	Start of text	34	"	66	B	98	b
3	End of text	35	#	67	C	99	c
4	End of transmit	36	\$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledge	38	&	70	F	102	f
7	Audible bell	39	'	71	G	103	g
8	Backspace	40	(72	H	104	h
9	Horizontal tab	41)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift in	46	.	78	N	110	n
15	Shift out	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Neg. acknowledge	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End trans. block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitution	58	:	90	Z	122	z
27	Escape	59	;	91	[123	{
28	File separator	60	<	92	\	124	
29	Group separator	61	=	93]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	Forward del.

Often, the character with ascii code 0 is called "NUL" rather than "NULL"

❖ ... Strings

Reminder ...

In C, a string is an array of **chars** (bytes) containing ASCII codes

- these arrays have an extra value at the end containing a 0 byte
- the extra 0 can also be written '**\0**' (*null character or null-terminator*)
- convenient because don't have to track the length of the string

Because strings are so common, C provides convenient syntax:

```
char str[] = "hello";
```

```
// same as
```

```
char str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Note: **str[]** will have 6 elements either way

❖ ... Strings

C provides string manipulation functions via `#include <string.h>`

Examples:

```
// return length of string
int strlen(char *str)
// copy one string into a string buffer
char *strcpy(char *dest, char *src)
// make a copy of a string (uses malloc())
char *strdup(char *)
// concatenate two strings, into buffer holding dest
char *strcat(char *dest, char *src)
// find substring pat inside string str
char *strstr(char *str, char *pat)
// copy at most n chars from src into a dest buffer
char *strncpy(char *dest, char *src, int n)
```


❖ ... Strings

Details of one `#include <string.h>` function ...

```
char *strncat(char *dest, char *src, int n)
```

- appends string **src** to the end of **dest**
- overwrites the '**\0**' at the end of **dest**
- adds terminating '**\0**' to result string
- assumes **dest[]** is large enough to hold all of above
- returns start of string **dest**
- will never add more than **n** characters
 - if **strlen(src)** less than **n** pad with '**\0**' characters
 - otherwise, **dest** is not null-terminated

❖ Pattern Matching

strstr() provides a simple example of string matching

The general **string matching** (aka **pattern matching**) problem

- given a string T and a pattern P
- find occurrences of P in T
 - index of first occurrence (or -1 if no occurrences), or
 - pointer to first occurrence (or **NULL** if no occurrences, or
 - array of indexes/pointers to all occurrences

strstr(str, pat) returns pointer to first occurrence of **pat** in **str**

T is sometimes called "haystack", P is sometimes called "needle"

Note that *regular expression* pattern matching is quite a different beast.

❖ ... Pattern Matching

Example of simple (brute force) pattern matching:

*Pattern comparison
starting at index [0]*

<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>c</i>
<i>c</i>	<i>a</i>	<i>b</i>							

*Pattern comparison
starting at index [1]*

<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>c</i>
	<i>c</i>	<i>a</i>	<i>b</i>						

*Pattern comparison
starting at index [2]*

<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>c</i>
		<i>c</i>	<i>a</i>	<i>b</i>					

*Pattern comparison
starting at index [3]*

<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>c</i>
			<i>c</i>	<i>a</i>	<i>b</i>				

*Pattern comparison
starting at index [4]*

<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>c</i>
				<i>c</i>	<i>a</i>	<i>b</i>			

❖ ... Pattern Matching

Brute-force pattern matching algorithm:

BruteForceMatch(T,P):

Input text T of length n, pattern P of length m

Output starting index of a substring of T equal to P

 -1 if no such substring exists

for all i = 0..n-m **do**

 j = 0

 // check from left to right

while j<m \wedge T[i+j]=P[j] **do** // test ith shift of pattern

 j = j+1

if j=m **then**

return i

 // entire pattern checked

end if

end while

end for

return -1

 // no match found

❖ ... Pattern Matching

C version of above algorithm:

```
int bruteForce(char *t, char *p)
{
    int n = strlen(t); //length of string
    int m = strlen(p); //length of pattern
    // for each starting position of pattern
    for (int i = 0; i <= n-m; i++) {
        // scan pattern, comparing against string
        for (int j = 0; j < m; j++) {
            // mismatch detected, terminate scan
            if (T[i+j] != P[j]) break;
        }
        // if reached end of pattern, have a match
        if (j == m) return i;
    }
    return -1; // no match
}
```

❖ Complexity Analysis

Brute-force pattern matching runs in $O(n \cdot m)$

- considers $n-m$ starting positions for pattern
- from each starting position, scans up to m chars in the pattern

Examples of worst case (forward checking):

- $T = \text{aaa ... aaah}$
- $P = \text{aaah}$
- may occur in DNA sequences
- unlikely in English text

