STRING

Algorithm Problem Solving – Samsung Vietnam R&D Center

Compose by phuong.ndp@samsung.com

Agenda

- ASCII
- Unicode
- String in C/C++
- String in Java
- Naive algorithm for Pattern Searching
- KMP (Knuth Morris Pratt) Pattern Searching
- Boyer Moore - Bad Character Heuristic
- Boyer Moore - Good Suffix heuristic
- Boyer Moore - Strong Good Suffix heuristic

ASCII / UNICODE

Characters Before ASCII/Unicode

Fundamentally, computers just deal with numbers. They store letters and other characters by assigning a number for each one. Before Unicode was invented, there were hundreds of different systems, called character encodings, for assigning these numbers.

These early character encodings were limited and could not contain enough characters to cover all the world's languages. Even for a single language like English no single encoding was adequate for all the letters, punctuation, and technical symbols in common use.

Early character encodings also conflicted with one another. That is, two encodings could use the same number for two different characters, or use different numbers for the same character. Any given computer (especially servers) would need to support many different encodings.

However, when data is passed through different computers or between different encodings, that data runs the risk of corruption.

Brief History of ASCII code

The **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, or **ASCII** code, was created in 1963 by the "American Standards Association" Committee or "**ASA**", the agency changed its name in 1969 by "American National Standards Institute" or "**ANSI**" as it is known since.

This code arises from reorder and expand the set of symbols and characters already used in telegraphy at that time by the Bell company.

At first only included capital letters and numbers , but in 1967 was added the lowercase letters and some control characters, forming what is known as US-ASCII (**Standard ASCII**), ie the characters 0 through 127.

So with this set of only 128 characters was published in 1967 as standard, containing all you need to write in English language.

Standard ASCII can represent 128 characters. It uses 7 bits to represent each character since the first bit of the byte is always 0. For instance, a capital "T" is represented by 84, or 01010100 in binary. A lowercase "t" is represented by 116 or 01110100 in binary.

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Brief History of ASCII code

In 1981, IBM developed an extension of 8-bit ASCII code, called "**code page 437**", in this version were replaced some obsolete control characters for graphic characters. Also 128 characters were added , with new symbols, signs, graphics and latin letters, all punctuation signs and characters needed to write texts in other languages, such as Spanish.

In this way was added the ASCII characters ranging from 128 to 255. (**Extended ASCII**)

The 128 (2^7) characters supported by **Standard ASCII** are enough to represent all standard English letters, numbers, and punctuation symbols.

However, it is not sufficient to represent all special characters and characters from other languages.

Extended ASCII helps solve this problem by adding an extra 128 values, for a total of 256 (2^8) characters. The additional binary values start with a 1 instead of a 0.

For example, in extended ASCII, the character "é" is represented by 233, or 11101001 in binary.

Extended ASCII is programmable; characters are based on the language of your operating system or program you are using. Foreign letters are also placed in this section.

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	^{n}
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	^{2}
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

What is Unicode?

In computer systems, characters are transformed and stored as numbers (sequences of bits) that can be handled by the processor. A code page is an encoding scheme that maps a specific sequence of bits to its character representation. The pre-Unicode world was populated with hundreds of different encoding schemes that assigned a number to each letter or character. Many such schemes included code pages that contained only 256 characters - each character requiring 8 bits of storage.

While this was relatively compact, it was insufficient to hold ideographic character sets containing thousands of characters such as **Vietnamese** and Japanese, and also did not allow the character sets of many languages to co-exist with each other.

Unicode is an attempt to include all the different schemes into one universal text-encoding standard.

A Brief Introduction to Unicode for Everybody

https://blog.daftcode.pl/a-brief-introduction-to-unicode-for-everybody-bc26f5761548

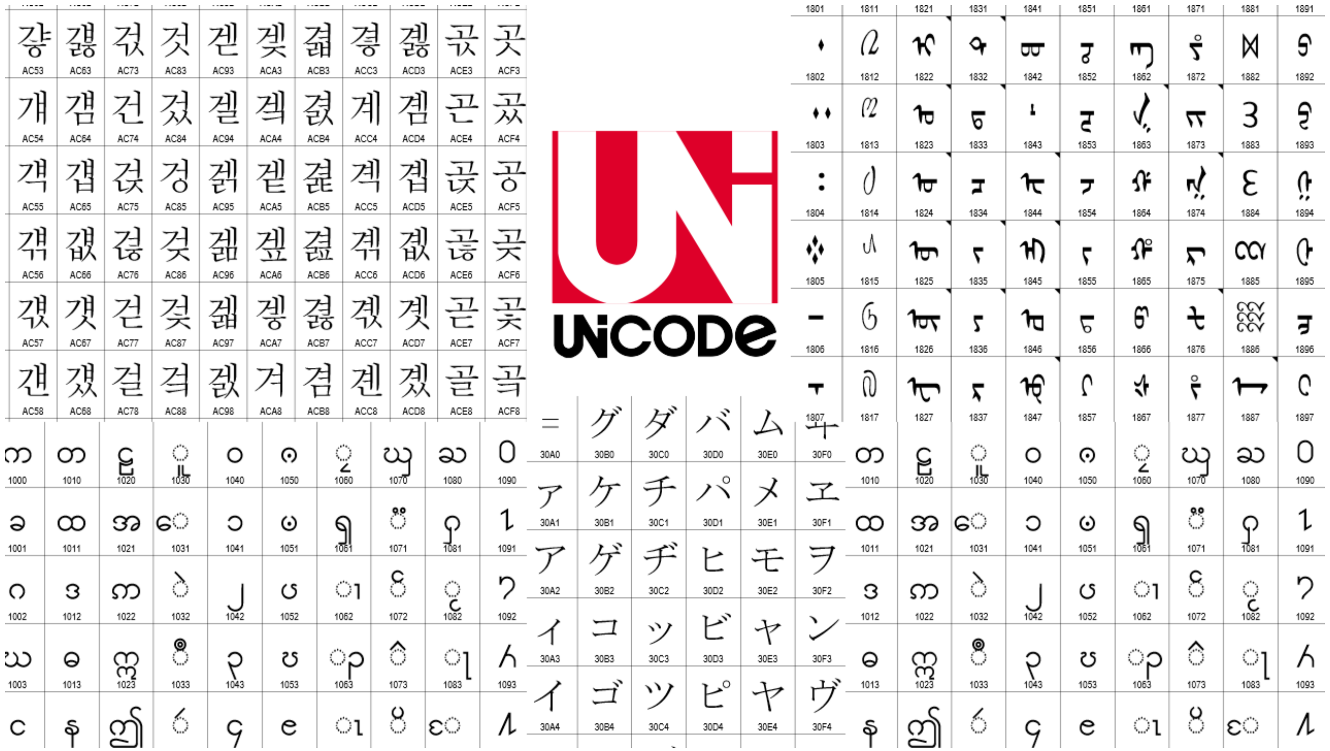
Compose by phuong.ndp

The importance of Unicode

Unicode represents a mechanism to support more regionally popular encoding systems

From a translation/localization point of view, Unicode is an important step towards standardization, at least from a tools and file format standpoint.

- Unicode enables a single software product or a single website to be designed for multiple platforms, languages and countries (no need for re-engineering) which can lead to a significant reduction in cost over the use of legacy character sets.
- Unicode data can be used through many different systems without data corruption.
- Unicode represents a single encoding scheme for all languages and characters.
- Unicode is a common point in the conversion between other character encoding schemes. Since it is a superset of all of the other common character encoding systems, you can convert from one encoding scheme to Unicode, and then from Unicode to the other encoding scheme.
- Unicode is the preferred encoding scheme used by XML-based tools and applications.



Bottom line: Unicode is a worldwide character-encoding standard, published by the Unicode Consortium. Computers store numbers that represent a character; Unicode provides a unique number for every character.

STRING IN C/C++

String in C

Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.

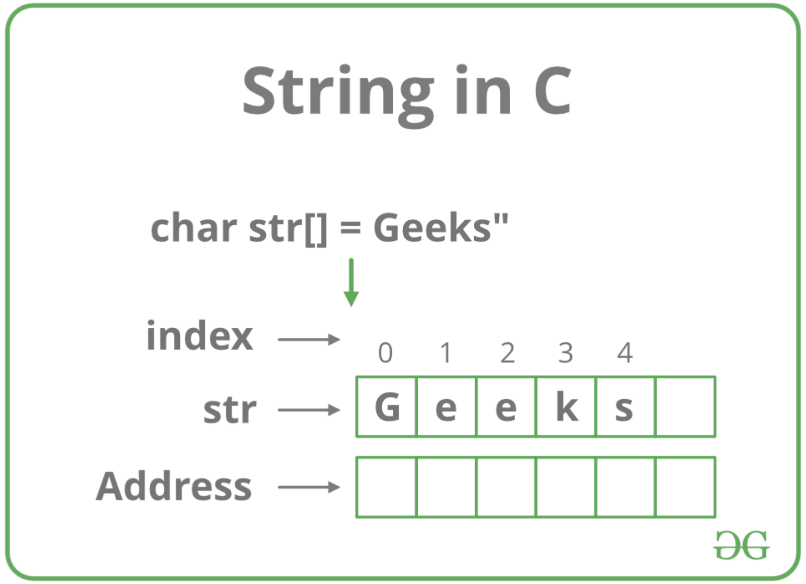
Declaration of strings: Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string.

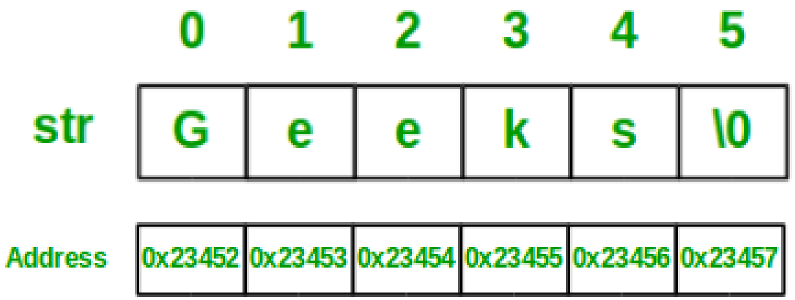
char str_name[size];

Please keep in mind that there is an extra terminating character which is the **Null** character (**'\0'**) used to indicate termination of string which differs strings from normal character arrays.

Initializing a String: A string can be initialized in different ways. We will explain this with the help of an example. Below is an example to declare a string with name as str and initialize it with "GeeksforGeeks".

```
1. char str[] = "GeeksforGeeks";
2. char str[50] = "GeeksforGeeks";
3. char str[] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
4. char str[14] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```





Reading String & Passing to function

Below is a sample program to read a string from user:

```
// C program to read strings

#include<stdio.h>

int main()
{
    // declaring string
    char str[50];

    // reading string
    scanf("%s",str);

    // print string
    printf("%s",str);

    return 0;
}
```

Passing strings to function: As strings are character arrays, so we can pass strings to function in a same way we pass an array to a function. Below is a sample program to do this:

```
// C program to illustrate how to
// pass string to functions
#include<stdio.h>

void printStr(char str[])
{
    printf("String is : %s",str);
}

int main()
{
    // declare and initialize string
    char str[] = "GeeksforGeeks";

    // print string by passing string
    // to a different function
    printStr(str);

    return 0;
}
```

Output:

```
String is : GeeksforGeeks
```

Read a Word

Notice that, in the second example only "Programming" is displayed instead of "Programming is fun".

This is because the extraction operator **>>** works as **scanf()** in C and considers a space " " has a terminating character.

Example 1: C++ String to read a word

C++ program to display a string entered by user.

```
#include <iostream>
using namespace std;

int main()
{
    char str[100];

    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: " << str << endl;

    cout << "\nEnter another string: ";
    cin >> str;
    cout << "You entered: "<<str<<endl;

    return 0;
}
```

Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Read a Line of Text

To read the text containing blank space, **cin.get** function can be used. This function takes two arguments.

First argument is the **name of the string** (address of first element of string) and second argument is the **maximum size of the array**.

In the above program, str is the name of the string and 100 is the maximum size of the array.

Example 2: C++ String to read a line of text

C++ program to read and display an entire line entered by user.

```
#include <iostream>
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

Functions for manipulating C strings

No	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

```
#include <iostream>
#include <cstring>
using namespace std;
int main () {
char str1[10] = "Hello";
char str2[10] = "World";
char str3[10];
int len ;

// copy str1 into str3
strcpy( str3, str1);
cout << "strcpy( str3, str1) : " << str3 << endl;

// concatenates str1 and str2
strcat( str1, str2); cout << "strcat( str1, str2): "
<< str1 << endl;

// total lenghth of str1 after concatenation
len = strlen(str1);
cout << "strlen(str1) : " << len << endl;

return 0;
}
```

String Class in C++

C++ provides following two types of string representations –

- The C-style character string.
- The string class type introduced with Standard C++.

C++ has in its definition a way to represent sequence of characters as an object of class. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing access to single byte character.

```
#include <iostream>
#include <string>

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

Character Array vs std:: string

A character array is simply **an array of characters** can terminated by a null character. A string is a **class which defines objects** that be represented as stream of characters.

Size of the character array has to **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is wasted** in case of character array. In case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated**, no memory is wasted**.

Implementation of **character array is faster** than std:: string. **Strings are slower** when compared to implementation than character array.

Character array **do not offer** much **inbuilt functions** to manipulate strings. String class defines **a number of functionalities** which allow manifold operations on strings.

Input String Data Type

In this program, a **string** str is declared. Then the string is asked from the user.

Instead of using **cin>>** or **cin.get()** function, you can get the entered line of text using **getline()**.

getline() function takes the input stream as the first parameter which is **cin** and str as the location of the line to be stored.

Example 3: C++ string using string data type

```
#include <iostream>
using namespace std;

int main()
{
    // Declaring a string object
    string str;
    cout << "Enter a string: ";
    getline(cin, str);

    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

STRING IN JAVA

String in Java

In Java, strings are special. For example, to create the **string** objects you **need not to use 'new'** keyword. Where as to create **other type of objects** you **have to use 'new'** keyword.

JVM divides the allocated memory to a Java program into two parts. one is **Stack** and another one is **heap**. Stack is used for **execution purpose** and heap is used for **storage purpose**.

In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called **String Constant Pool**.

Whenever you create a string object **using string literal**, that object is stored in the **string constant pool** and whenever you create a string object using **new** keyword, such object is stored in the **heap memory**.

String Constant Pool.

```java
String s1 = "abc";

String s2 = "xyz";

String s3 = "123";

String s4 = "A";
```

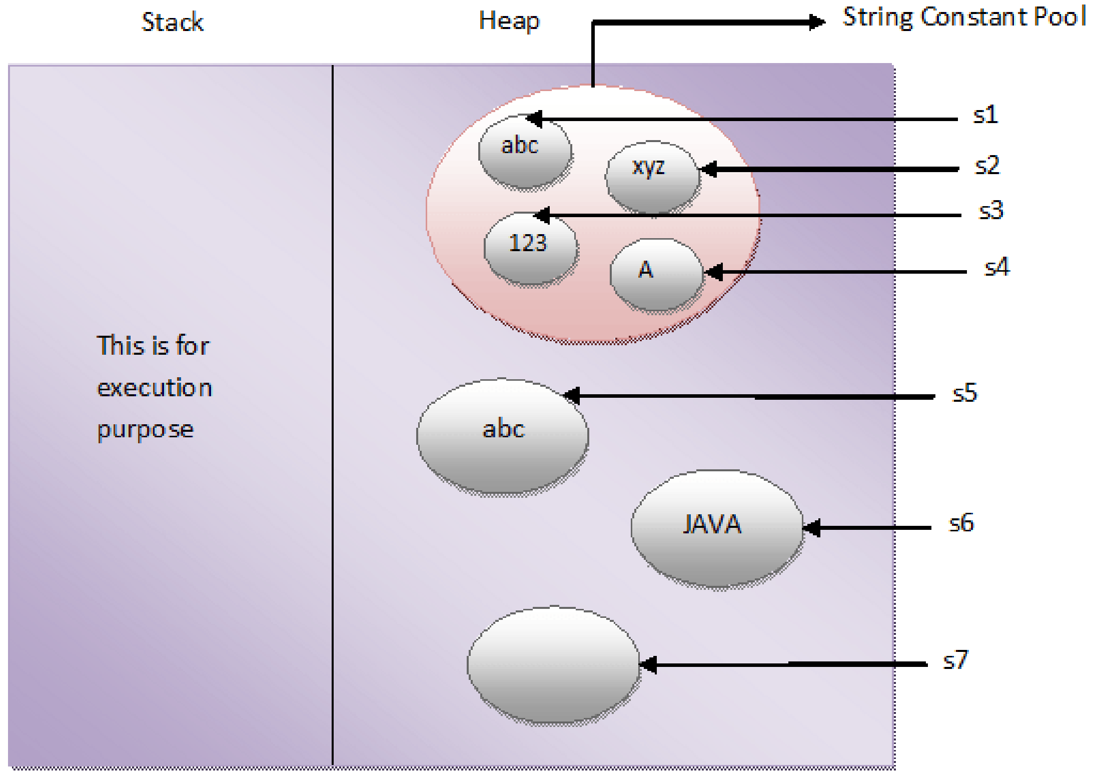
Heap memory.

```java
String s5 = new String("abc");

char[] c = {'J', 'A', 'V', 'A'};

String s6 = new String(c);

String s7 = new String(new StringBuffer());
```



String in Java

Pool space is allocated to an object depending upon it's content. There will be **no two objects** in the pool having the **same content**.

This is what happens when you create string objects using string literal,

"When you create a string object using string literal, JVM first checks the content of to be created object. If there exist an object in the pool with the same content, then it returns the reference of that object. It doesn't create new object. If the content is different from the existing objects then only it creates new object."

When you create string objects using new keyword, a new object is created **whether the content is same or not**.

This can be proved by using "==" operator. As "==" operator returns true if two objects have **same physical address** in the memory otherwise it will return false.

In simple words, there can not be two string objects with same content in the string constant pool. But, there can be two string objects with the same content in the heap memory.

```java
public class StringExamples
{
    public static void main(String[] args)
    {
        //Creating string objects using literals

        String s1 = "abc";

        String s2 = "abc";

        System.out.println(s1 == s2);         //Output : true

        //Creating string objects using new operator

        String s3 = new String("abc");

        String s4 = new String("abc");

        System.out.println(s3 == s4);         //Output : false
    }
}
```

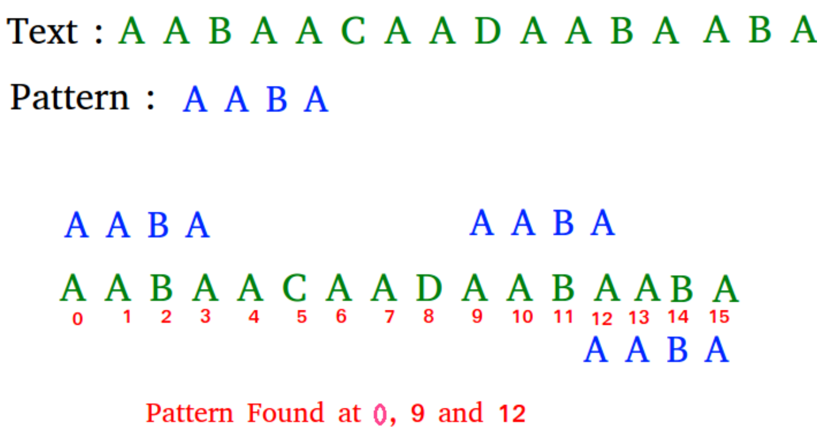
SEARCH

Problem

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10

Input:  txt[] =  "AABAACAADAABAABA"
        pat[] =  "AABA"
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```



Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

Naive algorithm for Pattern Searching

Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

```cpp
// C++ program for Naive Pattern
// Searching algorithm
#include <bits/stdc++.h>
using namespace std;

void search(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++) {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            cout << "Pattern found at index "
                 << i << endl;
    }
}

// Driver Code
int main()
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";
    search(pat, txt);
    return 0;
}

// This code is contributed
// by Akanksha Rai
```

```java
// Java program for Naive Pattern Searching
public class NaiveSearch {

    public static void search(String txt, String pat)
    {
        int M = pat.length();
        int N = txt.length();

        /* A loop to slide pat one by one */
        for (int i = 0; i <= N - M; i++) {

            int j;

            /* For current index i, check for pattern
              match */
            for (j = 0; j < M; j++)
                if (txt.charAt(i + j) != pat.charAt(j))
                    break;

            if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
                System.out.println("Pattern found at index " + i);
        }
    }

    public static void main(String[] args)
    {
        String txt = "AABAACAADAABAAABAA";
        String pat = "AABA";
        search(txt, pat);
    }
}
// This code is contributed by Harikishore
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

KMP (Knuth Morris Pratt) Pattern Searching

The Naive pattern searching algorithm doesn’t work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] =  "ABABAC" (not a worst case, but a bad case for Naive)
```

The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

KMP (Knuth Morris Pratt) Pattern Searching

TEXT	S	V	M	C		S	V	B	C	D	E	G	F
PATTERN	S	V	M	C		S	V	V					
						S	V	M	C		S	V	V

Matching Overview
txt = "AAAAABAAABA"
pat = "AAAA"

We compare first window of txt with pat
txt = "**AAAA**ABAAABA"
pat = "**AAAA**" [Initial position]
We find a match. This is same as Naive String Matching.

In the next step, we compare next window of txt with pat.
txt = "**AAAAA**BAAABA"
pat = "**AAAA**" [Pattern shifted one position]

This is where KMP does optimization over Naive. In this second window, we only compare fourth **A** of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

Need of Preprocessing?
An important question arises from the above explanation, how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array lps[] that tells us the count of characters to be skipped.

KMP (Knuth Morris Pratt) Pattern Searching

Preprocessing Overview:

- KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m (same as size of pattern) which is used to skip characters while matching.
- name lps indicates longest proper prefix which is also suffix.. A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". **Proper prefixes** are "", "A" and "AB". **Suffixes** of the string are "", "C", "BC" and "ABC".
- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern **pat**[0..i].

```
lps[i] = the longest proper prefix of pat[0..i]
              which is also a suffix of pat[0..i].
```

Note: lps[i] could also be defined as **longest prefix** which is also **proper suffix**. We need to use properly at one place to make sure that the whole substring is not considered.

KMP (Knuth Morris Pratt) Pattern Searching

Preprocessing Overview:

Examples of **lps**[] construction:
For the pattern “AAAA”,
lps[] is [0, 1, 2, 3]

For the pattern “ABCDE”,
lps[] is [0, 0, 0, 0, 0]

suffix
prefix
index

For the pattern “AABAACAABAA”,
lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

suffix
prefix
index

For the pattern “AAACAAAAAC”,
lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

suffix
prefix
index

For the pattern “AAABAAA”,
lps[] is [0, 1, 2, 0, 1, 2, 3]

KMP (Knuth Morris Pratt) Pattern Searching

Searching Algorithm:

Unlike Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use **lps**[] to decide next positions (or to know a number of characters to be skipped)?

- We start comparison of **pat**[j] with j = 0 with characters of current window of text.
- We keep matching characters **txt**[i] and **pat**[j] and keep incrementing i and j while **pat**[j] and **txt**[i] keep **matching**.
- When we see a **mismatch**
    - We know that characters **pat**[0..j-1] match with **txt**[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
    - We also know (from above definition) that **lps**[j-1] is count of characters of **pat**[0…j-1] that are both proper prefix and suffix.
    - From above two points, we can conclude that we do not need to match these **lps**[j-1] characters with **txt**[i-j…i-1] because we know that these characters will anyway match.

KMP (Knuth Morris Pratt) Pattern Searching

```
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
lps[] = {0, 1, 2, 3}

i = 0, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 1, j = 1
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 2, j = 2
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
pat[i] and pat[j] match, do i++, j++

i = 3, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++
```

KMP (Knuth Morris Pratt) Pattern Searching

```
i = 4, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3

Here unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above
step) gave us index of next character to match.
i = 4, j = 3
txt[] = "AAAAABAAABA"
pat[] =  "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 5, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3

Again unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above
step) gave us index of next character to match.
i = 5, j = 3
txt[] = "AAAAABAAABA"
pat[] =   "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[2] = 2
```

KMP (Knuth Morris Pratt) Pattern Searching

```
i = 5, j = 2
txt[] = "AAAAABAAABA"
pat[] =    "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[1] = 1

i = 5, j = 1
txt[] = "AAAAABAAABA"
pat[] =     "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[0] = 0

i = 5, j = 0
txt[] = "AAAAABAAABA"
pat[] =      "AAAA"
txt[i] and pat[j] do NOT match and j is 0, we do i++.
```

```
i = 6, j = 0
txt[] = "AAAAABAAABA"
pat[] =       "AAAA"
txt[i] and pat[j] match, do i++ and j++

i = 7, j = 1
txt[] = "AAAAABAAABA"
pat[] =       "AAAA"
txt[i] and pat[j] match, do i++ and j++

We continue this way...
```

KMP (Knuth Morris Pratt) Pattern Searching

Preprocessing Algorithm:

In the preprocessing part, we calculate values in **lps**[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize **lps**[0] and len as 0. If **pat**[len] and **pat**[i] match, we increment len by **1** and assign the incremented value to **lps**[i]. If **pat**[i] and **pat**[len] do not match and len is not 0, we update len to **lps**[len-**1**]. See computeLPSArray () in the below code for details.

```
pat[] = "AAACAAAA"

len = 0, i  = 0.
lps[0] is always 0, we move
to i = 1

len = 0, i  = 1.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 1, lps[1] = 1, i = 2

len = 1, i  = 2.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 2, lps[2] = 2, i = 3
```

KMP (Knuth Morris Pratt) Pattern Searching

```
len = 2, i  = 3.
Since pat[len] and pat[i] do not match, and len > 0,
set len = lps[len-1] = lps[1] = 1

len = 1, i  = 3.
Since pat[len] and pat[i] do not match and len > 0,
len = lps[len-1] = lps[0] = 0

len = 0, i  = 3.
Since pat[len] and pat[i] do not match and len = 0,
Set lps[3] = 0 and i = 4.
We know that characters pat
len = 0, i  = 4.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 1, lps[4] = 1, i = 5

len = 1, i  = 5.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 2, lps[5] = 2, i = 6
```

```
len = 2, i  = 6.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 3, lps[6] = 3, i = 7

len = 3, i  = 7.
Since pat[len] and pat[i] do not match and len > 0,
set len = lps[len-1] = lps[2] = 2

len = 2, i  = 7.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 3, lps[7] = 3, i = 8

We stop here as we have constructed the whole lps[].
```

Problem Solving

Implement KMP (Knuth Morris Pratt) Pattern Searching

Boyer Moore Algorithm for Pattern Searching

Boyer Moore Algorithm

Boyer Moore is a combination of following two approaches.

1) Bad Character Heuristic

2) Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm.

It processes the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by the max of the slides suggested by the two heuristics. So it **uses best of the two heuristics at every step**.

Boyer Moore - Bad Character Heuristic

Unlike the previous pattern searching algorithms, Boyer Moore algorithm **starts matching from the last character** of the pattern.

The idea of bad character heuristic is simple. The character of the text which doesn’t match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –

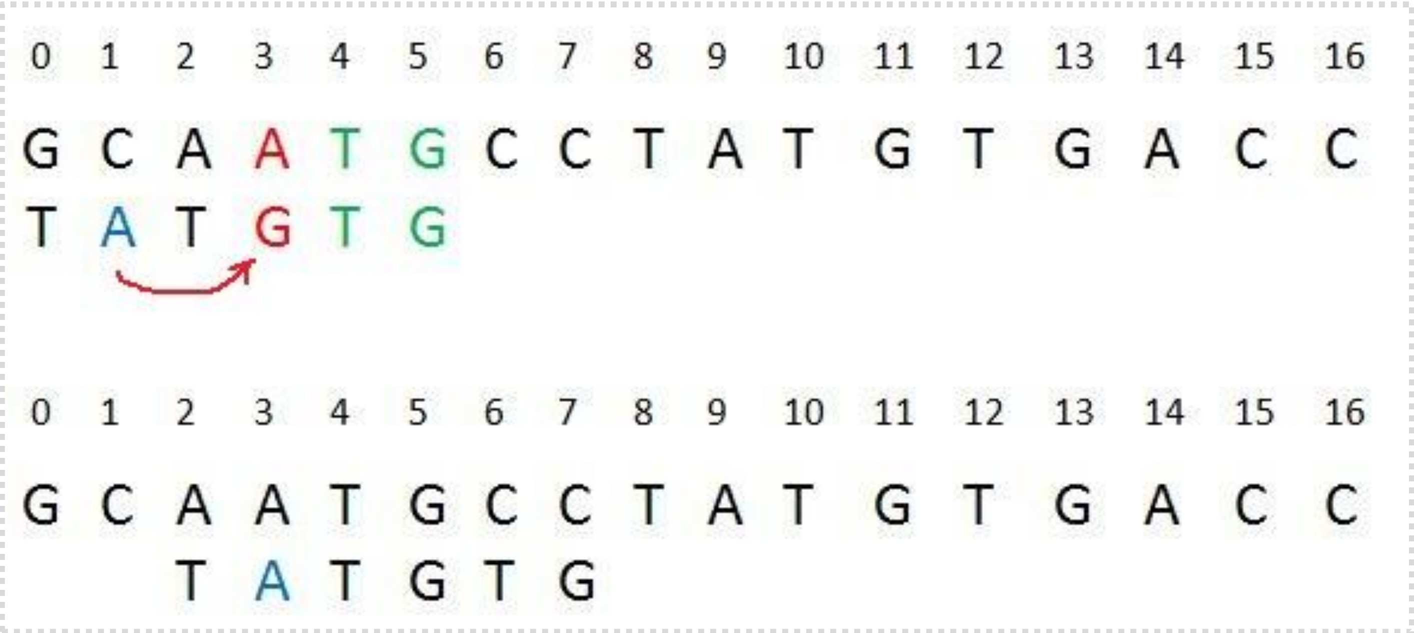
1) The mismatch becomes a match

2) Pattern P move past the mismatched character.

Case 1 – Mismatch become match

We will lookup the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then we’ll shift the pattern such that it get aligned to the mismatching character in text T.

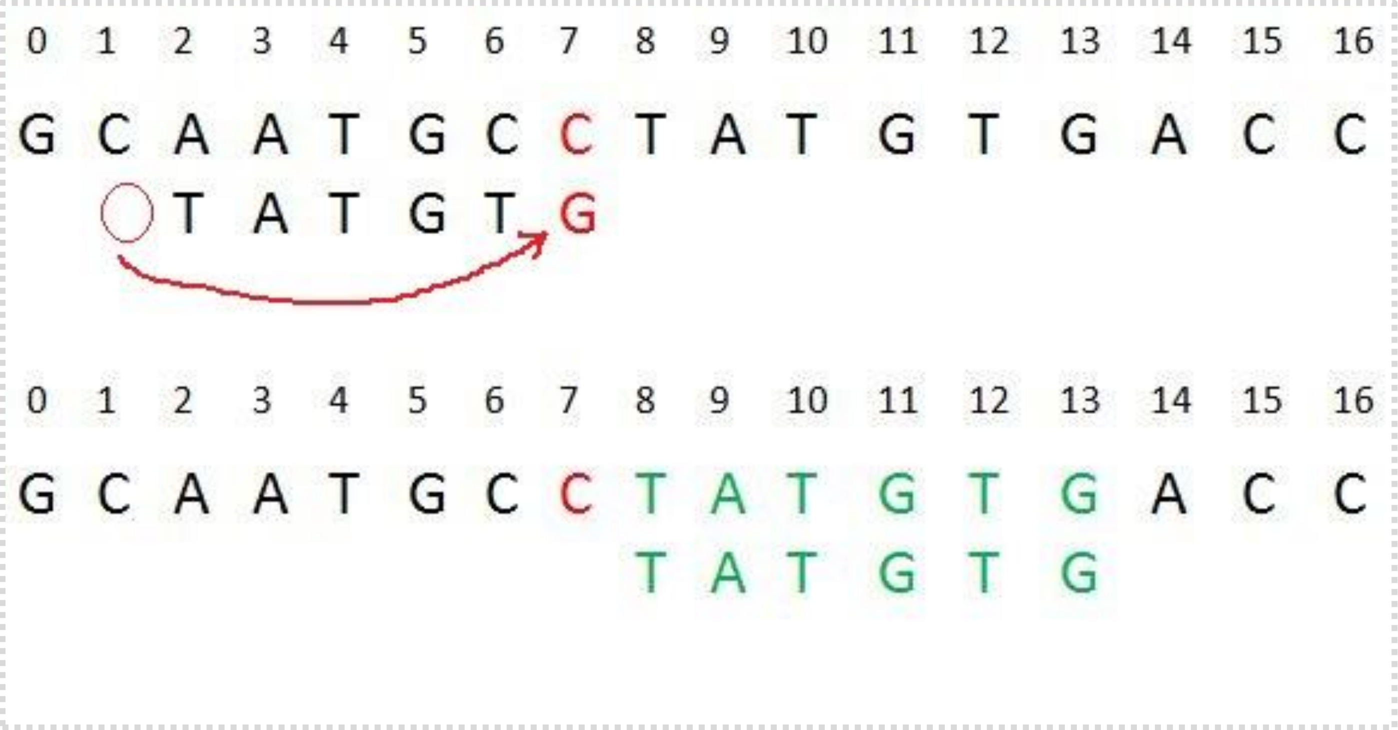
Explanation: In the above example, we got a mismatch at position 3. Here our mismatching character is “A”. Now we will search for last occurrence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.



Boyer Moore - Bad Character Heuristic

Case 2 – Pattern move past the mismatch character

We'll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.



Explanation: Here we have a mismatch at position 7. The mismatching character "C" does not exist in pattern before position 7 so we'll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because, "C" do not exist in pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

Boyer Moore

We preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern).

```
/* Driver code */
int main()
{
    string txt= "ABAAABCD";
    string pat = "ABC";
    search(txt, pat);
    return 0;
}

 // This code is contributed by rathbhupendra
```

```
// The preprocessing function for Boyer Moore's
// bad character heuristic
void badCharHeuristic( string str, int size,
                        int badchar[NO_OF_CHARS])
{
    int i;

    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;

    // Fill the actual value of last occurrence
    // of a character
    for (i = 0; i < size; i++)
        badchar[(int) str[i]] = i;
}
```

```
/* A pattern searching function that uses Bad
Character Heuristic of Boyer Moore Algorithm */
void search( string txt, string pat)
{
    int m = pat.size();
    int n = txt.size();

    int badchar[NO_OF_CHARS];

    /* Fill the bad character array by calling
    the preprocessing function badCharHeuristic()
    for given pattern */
    badCharHeuristic(pat, m, badchar);

    int s = 0; // s is shift of the pattern with
                // respect to text
    while(s <= (n - m))
    {
        int j = m - 1;

        /* Keep reducing index j of pattern while
        characters of pattern and text are
        matching at this shift s */
        while(j >= 0 && pat[j] == txt[s + j])
            j--;

        /* If the pattern is present at current
        shift, then index j will become -1 after
        the above loop */
        if (j < 0)
        {
            cout << "pattern occurs at shift = " <<  s << endl;

            /* Shift the pattern so that the next
            character in text aligns with the last
            occurrence of it in pattern.
            The condition s+m < n is necessary for
            the case when pattern occurs at the end
            of text */
            s += (s + m < n)? m-badchar[txt[s + m]] : 1;

        }

        else
            /* Shift the pattern so that the bad character
            in text aligns with the last occurrence of
            it in pattern. The max function is used to
            make sure that we get a positive shift.
            We may get a negative shift if the last
            occurrence of bad character in pattern
            is on the right side of the current
            character. */
            s += max(1, j - badchar[txt[s + j]]);
    }
}
```

Boyer Moore | Good Suffix heuristic

Just like bad character heuristic, a preprocessing table is generated for good suffix heuristic.

Let t be substring of text T which is matched with substring of pattern P. Now we shift pattern until :

1) Another occurrence of t in P matched with t in T.

2) A prefix of P, which matches with suffix of t

3) P moves past t

Boyer Moore | Good Suffix heuristic

Case 1: Another occurrence of t in P matched with t in T

Pattern P might contain few more occurrences of t. In such case, we will try to shift the pattern to align that occurrence with t in text T. For example:

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

Explanation: In the above example, we have got a substring t of text T matched with pattern P (in **green**) before mismatch at index 2. Now we will search for occurrence of t ("AB") in P. We have found an occurrence starting at position 1 (in **yellow** background) so we will right shift the pattern 2 times to align t in P with t in T. This is weak rule of original Boyer Moore and not much effective, we will discuss a Strong Good Suffix rule shortly.

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

Boyer Moore | Good Suffix heuristic

Case 2: A prefix of P, which matches with suffix of t in T

It is not always likely that we will find the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes we can search for some suffix of t matching with some prefix of P and try to align them by shifting P. For example:

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

Explanation: In above example, we have got t ("BAB") matched with P (in green) at index 2-4 before mismatch . But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t. We have found prefix "AB" (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t "AB" starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Figure – Case 2

Boyer Moore | Good Suffix heuristic

Case 3: P moves past t

If the above two cases are not satisfied, we will shift the pattern past the t. For example:

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

Explanation: If above example, there exist no occurrence of t ("AB") in P and also there is no prefix in P which matches with the suffix of t. So, in that case, we can never find any perfect match before index 4, so we will shift the P past the t ie. to index 5.

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Figure – Case 3

Boyer Moore | Strong Good Suffix heuristic

Suppose substring q = P[i to n] got matched with t in T and c = P[i-1] is the mismatching character. Now unlike case 1 we will search for t in P which is not preceded by character c. The closest such occurrence is then aligned with t in T by shifting pattern P. For example:

Explanation: In above example, **q = P[7 to 8]** got matched with t in T. The mismatching character **c** is "C" at position P[6]. Now if we start searching t in P we will get the first occurrence of t starting at position 4. But this occurrence is preceded by "C" which is equal to **c**, so we will skip this and carry on searching. At position 1 we got another occurrence of t (in the **yellow** background). This occurrence is preceded by "A" (in **blue**) which is not equivalent to **c**. So we will shift pattern P 6 times to align this occurrence with t in T. We are doing this because we already know that character **c = "C"** causes the mismatch. So any occurrence of t preceded by c will again cause mismatch when aligned with t, so that's why it is better to skip this.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P	A	A	C	C	A	C	C	A	C									

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P							A	A	C	C	A	C	C	A	C			

Figure – strong suffix rule

Preprocessing for Good Suffix Heuristic

As a part of preprocessing, an array **shift** is created. Each entry **shift[i]** contain the distance pattern will shift if **mismatch** occur at position **i-1**. That is, the suffix of pattern starting at position **i** **is matched** and a **mismatch** occur at position **i-1**. Preprocessing is done separately for strong good suffix and case 2 discussed above.

1) Preprocessing for Strong Good Suffix

Before discussing preprocessing, let us first discuss the idea of border. A **border** is a substring which is both **proper suffix** and **proper prefix**. For example, in string "**ccacc**", "**c**" is a border, "**cc**" is a border because it appears in both end of string but "**cca**" is not a border.

As a part of preprocessing an array **bpos** (border position) is calculated. Each entry **bpos[i]** contains the starting index of border for suffix starting at index **i** in given pattern P.

The suffix **φ** beginning at position m has no border, so **bpos[m]** is set to **m+1** where **m** is the **length of the pattern**.

The shift position is obtained by the borders which cannot be extended to the left. Following is the code for preprocessing.

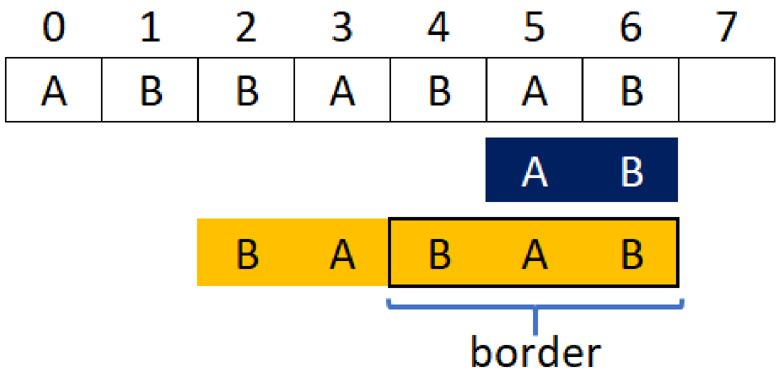
Preprocessing for Good Suffix Heuristic

Explanation: Consider pattern **P = "ABBABAB"**, **m = 7**.

The widest border of suffix "AB" beginning at position i = 5 is φ(nothing) starting at position 7 so **bpos[5] = 7**.

At position i = 2 the suffix is "**BABAB**". The widest border for this suffix is "**BAB**" starting at position **4**, so **j = bpos[2] = 4**.

We can understand **bpos[i] = j** using following example




```
void preprocess_strong_suffix(int *shift, int *bpos,
                              char *pat, int m)
{
    int i = m, j = m+1;
    bpos[i] = j;
    while(i > 0)
    {
        while(j <= m && pat[i-1] != pat[j-1])
        {
            if (shift[j] == 0)
                shift[j] = j-i;
            j = bpos[j];
        }
        i--; j--;
        bpos[i] = j;
    }
}
```

i	0	1	2	3	...	(i-1)	i	...	(j-1)	j	...	m
P	A	A	B	A	...	#	x	...	?	x	...	Φ

Preprocessing for Good Suffix Heuristic

If character # Which is at position **i-1** is equivalent (==) to character ? at position **j-1**, we know that **border of suffix at i-1** begin **at j-1** or **bpos[i-1] = j-1** or in the code.

```
i--;
j--;
bpos[i] = j
```

i	0	1	2	3	...	(i-1)	i	...	(j-1)	j	...	m
P	A	A	B	A	...	#	x	...	?	x	...	Φ

Preprocessing for Good Suffix Heuristic

But if character # at position i-1 do not match with character ? at position j-1 then we continue our search to the right. Now we know that.

1. Border width will be smaller than the border starting at position j ie. smaller than **x…φ**
2. Border has to begin with # and end with **φ** or could be empty (no border exist).

With above two facts we will continue our search in sub string **x…φ** from position **j to m**. The next border should be at **j = bpos[j]** (because bpos[j] is the border position of j => **p[j]==p[bpos[j]]**). After updating **j**, we again compare character at position j-1 (?) with # and if they are equal then we got our border otherwise we continue our search to right **until j>m** (out of pattern). This process is shown by code

```
while(j <= m && pat[i-1] != pat[j-1])
{
    j = bpos[j];
}
i--; j--;
bpos[i]=j;
```

i	0	1	2	3	…	(i-1)	i	…	(j-1)	j	…	m
P	A	A	B	A	…	#	x	…	?	x	…	Φ

Preprocessing for Good Suffix Heuristic

In above code look at these conditions:

```
pat[i-1] != pat[j-1]
```

t

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P	A	A	C	C	A	C	C	A	C									

P[j-1] occurrence of t P[i-1] ~ mismatch

This is the condition which we discussed in case 2 (in Strong Good Suffix Heuristic). When the **character preceding** the **occurrence** of **t** in pattern P is different than mismatching character in P (**P[i-1]**), we **stop** skipping the occurrences and shift the pattern. So here **P[i] == P[j]** but **P[i-1] != P[j-1]** so we shift pattern **from i to j**. So **shift[j] = j-i** is recorder for **j**. So whenever any mismatch occur at position j we will shift the pattern **shift[j+1]** positions to the right.

			i						j			
i	0	1	2	3	4	5	6	7	8	9	10	11
T		B	A	C	A	A	B	A	B	A		
P		A	A	C	C	A	C	C	A	C		

t

occurrence of t

P[i-1] ~ mismatch

P[j-1]

Preprocessing for Good Suffix Heuristic

In above code the following condition is very important.

```
if (shift[j] == 0 )
```

					i			j		
i	0	1	2	3	4	5	6	7	8	
P	a	d	d	b	d	d	c	d	d	
					d	d	c	d	d	shift[j]=j-i ~ shift[7]=7-4=3
		d	d	b	d	d	c	d	d	shift[j]=j-i ~ shift[7]=7-1=6
		i						j		

This condition prevent modification of **shift[j]** value from suffix having same border. For example, Consider pattern **P = "addbddcdd"**, when we calculate **bpos[i-1]** for i = 4 then j = 7 in this case. we will be eventually setting value of **shift[7] = 3**.

Now if we calculate **bpos[i-1]** for i = 1 then j = 7 and we will be setting value **shift[7] = 6** again, if there is no test shift[j] == 0. This mean if we have a **mismatch at position 6** we will **shift pattern P 3 positions** to right **not 6 position**.

=> DO NOT MISS ANY CASES.

Preprocessing for Good Suffix Heuristic

2) Preprocessing for Case 2

In the preprocessing for case 2, for each suffix **the widest border of the whole pattern** that is contained in that suffix is determined.

The starting position of the widest border of the pattern at all is stored in **bpos[0]**

In the following preprocessing algorithm, this value **bpos[0]** is stored initially in all free entries of array shift. But when the suffix of the pattern becomes shorter than **bpos[0]**, the algorithm continues with the next-wider border of the pattern, i.e. with **bpos[j]**.

Boyer Moore | Code of Strong Good Suffix

```
// preprocessing for strong good suffix rule
void preprocess_strong_suffix(int *shift, int *bpos,
                                char *pat, int m)
{
    // m is the length of pattern
    int i=m, j=m+1;
    bpos[i]=j;

    while(i>0)
    {
        /*if character at position i-1 is not equivalent to
          character at j-1, then continue searching to right
          of the pattern for border */
        while(j<=m && pat[i-1] != pat[j-1])
        {
            /* the character preceding the occurrence of t in
               pattern P is different than the mismatching character in P,
               we stop skipping the occurrences and shift the pattern
               from i to j */
            if (shift[j]==0)
                shift[j] = j-i;

            //Update the position of next border
            j = bpos[j];
        }
        /* p[i-1] matched with p[j-1], border is found.
           store the  beginning position of border */
        i--;j--;
        bpos[i] = j;
    }
}
```

```
//Preprocessing for case 2
void preprocess_case2(int *shift, int *bpos,
                      char *pat, int m)
{
    int i, j;
    j = bpos[0];
    for(i=0; i<=m; i++)
    {
        /* set the border position of the first character of the pattern
           to all indices in array shift having shift[i] = 0 */
        if(shift[i]==0)
            shift[i] = j;

        /* suffix becomes shorter than bpos[0], use the position of
           next widest border as value of j */
        if (i==j)
            j = bpos[j];
    }
}
```

Boyer Moore | Code of Strong Good Suffix

```
//Driver
int main()
{
    char text[] = "ABAAAABAACD";
    char pat[] = "ABA";
    search(text, pat);
    return 0;
}
```

```
/*Search for a pattern in given text using
  Boyer Moore algorithm with Good suffix rule */
void search(char *text, char *pat)
{
    // s is shift of the pattern with respect to text
    int s=0, j;
    int m = strlen(pat);
    int n = strlen(text);

    int bpos[m+1], shift[m+1];

    //initialize all occurrence of shift to 0
    for(int i=0;i<m+1;i++) shift[i]=0;

    //do preprocessing
    preprocess_strong_suffix(shift, bpos, pat, m);
    preprocess_case2(shift, bpos, pat, m);

    while(s <= n-m)
    {

        j = m-1;

        /* Keep reducing index j of pattern while characters of
             pattern and text are matching at this shift s*/
        while(j >= 0 && pat[j] == text[s+j])
            j--;

        /* If the pattern is present at the current shift, then index j
             will become -1 after the above loop */
        if (j<0)
        {
            printf("pattern occurs at shift = %d\n", s);
            s += shift[0];
        }
        else
            /*pat[i] != pat[s+j] so shift the pattern
              shift[j+1] times  */
            s += shift[j+1];
    }

}
```

Output:

```
pattern occurs at shift = 0
pattern occurs at shift = 5
```

Thank you!

Algorithm Problem Solving – Samsung Vietnam R&D Center

Compose by phuong.ndp@samsung.com

Source

https://theasciicode.com.ar/

https://techterms.com/definition/ascii

https://www.computerhope.com/jargon/a/ascii.htm

https://www.interproinc.com/blog/unicode-101-introduction-unicode-standard

https://unicode.org/standard/WhatIsUnicode.html

https://www.geeksforgeeks.org/strings-in-c-2/

https://www.tutorialspoint.com/cplusplus/cpp_strings.htm

https://www.programiz.com/cpp-programming/strings

https://javaconceptoftheday.com/how-the-strings-are-stored-in-the-memory/#:~:text=This%20part%20of%20the%20heap,stored%20in%20the%20heap%20memory.

https://thuytrangcoding.wordpress.com/2018/02/11/string-match-kmp/

https://stackjava.com/algorithm/thuat-toan-tim-kiem-knuth-morris-pratt.html

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/?ref=rp