Unit V

**Pattern Matching:**

String-matching is a very important subject in the wider domain of text processing. String-matching algorithms are basic components used in implementations of practical softwares existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (system or software design). Finally, they also play an important role in theoretical computer science by providing challenging problems.

Although data are memorized in various ways, text remains the main form to exchange information. This is particularly evident in literature or linguistics where data are composed of huge corpus and dictionaries. This apply as well to computer science where a large amount of data are stored in linear files. And this is also the case, for instance, in molecular biology because biological molecules can often be approximated as sequences of nucleotides or amino acids. Furthermore, the quantity of available data in these fields tend to double every eighteen months. This is the reason why algorithms should be efficient even if the speed and capacity of storage of computers increase regularly.

String-matching consists in finding one, or more generally, all the occurrences of a string (more generally called a *pattern*) in a *text*. The pattern is denoted by p=p[0 .. m-1]; its length is equal to m. The text is denoted by t=t[0 .. n-1]; its length is equal to n.

**Brute Force Algorithm**

**Description:** The brute force algorithm consists in checking, at all positions in the text between 0 and n-m, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the text character comparisons can be done in any order. The time complexity of this searching phase is O(mn)

**/\* Brute Force Pattern Matching implementation \*/**

#include<stdio.h>

#include<string.h>

int BruteForce(char\*t, char \*p,int n, int m){

int i,j;

for(j=0; j<=n-m; j++){

i=0;

while(i < m && p[i] == t[j+i])

i++;

if(i >= m)

return j;

}

return -1;

}

int main(){

char p[80], t[80];

int m,n,i;

printf("Enter a String:");

gets(t);

printf("Enter a pattern:");

gets(p);

m=strlen(p);

n=strlen(t);

i=BruteForce(t,p,n,m);

if(i>0)

printf("Pattern found at inded %d",i);

else

printf("Pattern not found");

}

**Main features of Brute Force**

* no preprocessing phase;
* constant extra space needed;
* always shifts the window by exactly 1 position to the right;
* comparisons can be done in any order;
* searching phase in O(mn) time complexity;
* 2n expected text characters comparison

**Boyer Moore Algorithm**

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the «search» and «substitute» commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the good-suffix shift (also called matching shift and the bad-character shift (also called the occurrence shift).

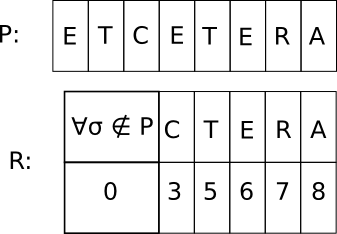
Assume that a mismatch occurs between the character t[i]=a of the pattern and the character y[i+j]=b of the text during an attempt at position j.

Then, t[i+1 .. m-1]=p[i+j+1 .. j+m-1]=u and t[i] neq p[i+j]. The good-suffix shift consists in aligning the segment p[i+j+1 .. j+m-1]=t[i+1 .. m-1] with its rightmost occurrence in x that is preceded by a character different from t[i]

Algorithm Skeleton

1. Align P with the beginning of T and match from right to left.
2. If whole P was match report occurrence.
3. Otherwise shift P by the maximal amount between the ones given by the *bad character shift* and the *good suffix shift*.

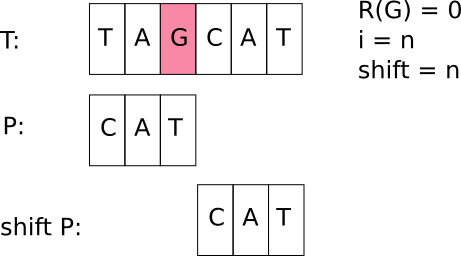
Conditional correctness: If the two shifts never go beyond an occurrence of P in T, the algorithm will report all occurrences.

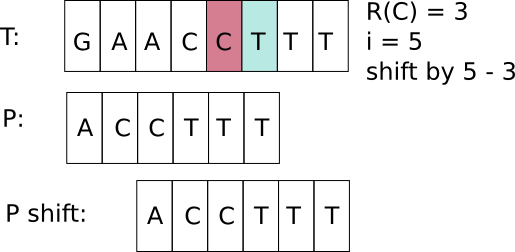
Bad Character rule

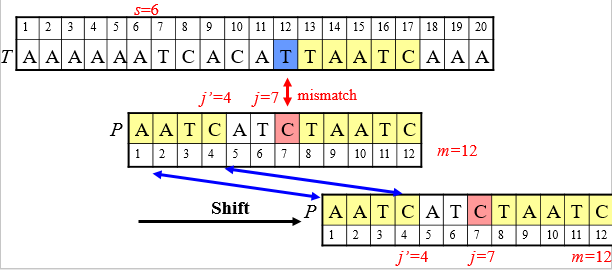
* *Definition* For each character x, let R(x) be the position of the right-most occurrence of character x in P. R(x) is defined to be zero if x does not occur in P.

Bad character shift

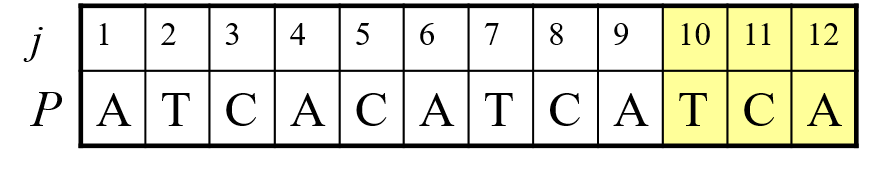
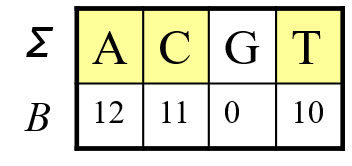
* Definition: Suppose a particular alignment of *P* against *T*, the rightmost *n-i* characters of *P* match their counterparts in *T*, but the character *P(i)* mismatches with its counterpart, say in position *k* of *T*. If the right-most position of the character *T(k)* in *P* is *j*, *j < i*, then shift so that character *j* of *P* is below character *k* of *T*, otherwise shift by 1.
* The shift would be *max[1, i-R(T(k))]*.
* Simple case: The character aligned with *P(n)*, *T(k)* does not appear in P: P is shifted by *n* (to start after *k*).



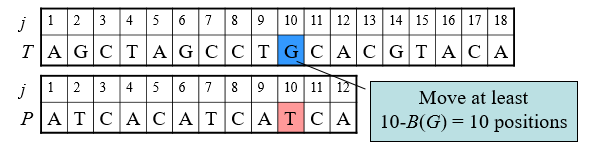
* General case: Shift by *i – R(x)*. Trivial to prove correctness.
* Ex: Suppose that *P1* is aligned to *T6* now. We compare pair-wise between *P* and *T* from right to left. Since *T12* ≠ *P7* and there is no substring *P8,12* in left of *P8*to exactly match *T13,17*. We find a longest suffix “AATC” of substring *T13,17*, the longest suffix is also prefix of *P*. We shift the window such that the last character of prefix substring to match the last character of the suffix substring. Therefore, we can shift at least 12-4=8 positions.



* Let ***Bc*(*a*)** be the rightmost position of *a* in *P*. The function will be used for applying ***bad character rule***.

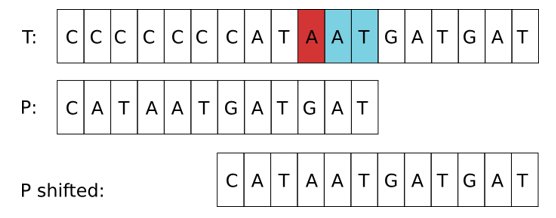


* We can move our pattern right at least ***j*-*B*(*Ts*+*j*-1)** position by above ***Bc*** function.

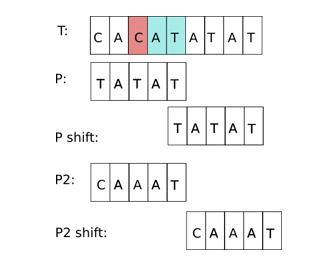


**Good Suffix Rule**

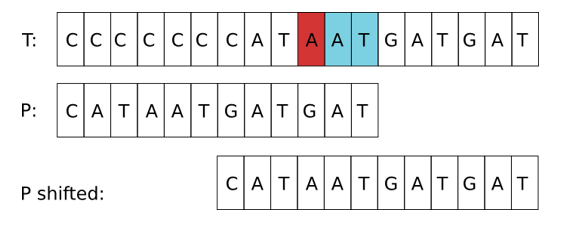
* *Definition:* Suppose for a given alignment of and , a substring of matches a suffix of , but a mismatch occurs to the next character to the left. Then find, if exists, the **rightmost** copy of in , such as is not a suffix of , and the character to the left of in differs from the one to the left of in . Shift to the right, so that substring in is below substring in .

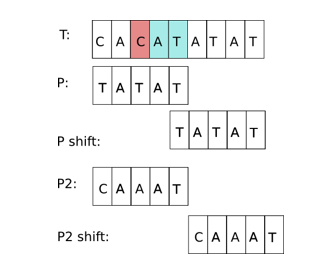


* If does not exist, then shift the left end of past the left end of in by the **least** amount, so that a prefix of matches a suffix of t in . If no such shift is possible then shift by n places to the right.



**Good Suffix Rule**

* *Definition:* Suppose for a given alignment of and , a substring of matches a suffix of , but a mismatch occurs to the next character to the left. Then find, if exists, the **rightmost** copy of in , such as is not a suffix of , and the character to the left of in differs from the one to the left of in . Shift to the right, so that substring in is below substring in .
* If does not exist, then shift the left end of past the left end of in by the **least** amount, so that a prefix of matches a suffix of t in . If no such shift is possible then shift by n places to the right.



**Knuth – Morris – Pratt Algorithm**

Introduction: The design of the Knuth-Morris-Pratt algorithm follows a tight analysis of the *Morris and Pratt* algorithm. Let us look more closely at the Morris-Pratt algorithm. It is possible to improve the length of the shifts.

Consider an attempt at a left position *j*, that is when the the window is positioned on the text factor *y*[*j* .. *j*+*m*-1]. Assume that the first mismatch occurs between *x*[*i*] and *y*[*i*+*j*] with 0 < i < *m*. Then, *x*[0 .. *i*-1] = *y*[*j* .. *i*+*j*-1] =*u* and *a* = *x*[*i*] neq *y*[*i*+*j*]=*b*.

When shifting, it is reasonable to expect that a prefix *v* of the pattern matches some suffix of the portion *u* of the text. Moreover, if we want to avoid another immediate mismatch, the character following the prefix *v* in the pattern must be different from *a*. The longest such prefix *v* is called the ***tagged border*** of *u* (it occurs at both ends of *u* followed by different characters in *x*).

This introduces the notation: let *kmpNext*[*i*] be the length of the longest border of *x*[0 .. *i*-1] followed by a character *c* different from *x*[*i*] and -1 if no such tagged border exits, for 0 < *i* leq *m*. Then, after a shift, the comparisons can resume between characters *x*[*kmpNext*[*i*]] and *y*[*i*+*j*] without missing any occurrence of *x* in *y*, and avoiding a backtrack on the text (see figure 7.1). The value of *kmpNext*[0] is set to -1.

The table *kmpNext* can be computed in ***O***(*m*) space and time before the searching phase, applying the same searching algorithm to the pattern itself, as if *x*=*y*.

The searching phase can be performed in ***O***(*m*+*n*) time. The Knuth-Morris-Pratt algorithm performs at most 2*n*-1 text character comparisons during the searching phase. The **delay** (maximal number of comparisons for a single text character) is bounded by logPhi(*m*) where Phi is the golden ratio ( golden ratio ).

**Main Features**

* performs the comparisons from left to right;
* preprocessing phase in ***O***(*m*) space and time complexity;
* searching phase in ***O***(*n*+*m*) time complexity (independent from the alphabet size);
* delay bounded by logPhi(*m*) where Phi is the golden ratio ( golden ratio ).

Code:

void preKmp(char \*x, int m, int kmpNext[]) {

int i, j;

i = 0;

j = kmpNext[0] = -1;

while (i < m) {

while (j > -1 && x[i] != x[j])

j = kmpNext[j];

i++;

j++;

if (x[i] == x[j])

kmpNext[i] = kmpNext[j];

else

kmpNext[i] = j;

}

}

void KMP(char \*x, int m, char \*y, int n) {

int i, j, kmpNext[XSIZE];

/\* Preprocessing \*/

preKmp(x, m, kmpNext);

/\* Searching \*/

i = j = 0;

while (j < n) {

while (i > -1 && x[i] != y[j])

i = kmpNext[i];

i++;

j++;

if (i >= m) {

OUTPUT(j - i);

i = kmpNext[i];

}

}

}

**Tries**

All the search trees are used to store the collection of numerical values but they are not suitable for storing the collection of words or strings. Trie is a data structure which is used to store the collection of strings and makes searching of a pattern in words more easy. The term ***trie*** came from the word **retrieval**. Trie data structure makes retrieval of a string from the collection of strings more easily. Trie is also called as **Prefix Tree** and some times **Digital Tree**. Trie is a tree like data structure used to store collection of strings.

The trie data structure provides fast pattern matching for string data values. Using trie, we bring the search complexity of a string to the optimal limit. A trie searches a string in O(m) time complexity, where m is the length of the string.

In trie, every node except the root stores a character value. Every node in trie can have one or a number of children. All the children of a node are alphabetically ordered. If any two strings have a common prefix then they will have the same ancestors.

**Types of Tries:**

* Standard Tries
* Suffix Tries
* Compressed Tries

**Standard Trie:**

* The standard trie for a set of strings S is an ordered tree such that:
  + Each node but the root is labeled with a character
  + The children of a node are alphabetically ordered
  + The paths from the external nodes to the root yield the strings of S
* Example: standard trie for the set of strings

S = { bear, bell, bid, bull, buy, sell, stock, stop }



* A standard trie uses ***O***(***n***) space and supports searches, insertions and deletions in time ***O***(***dm***), where:

***n*** total size of the strings in S

***m*** size of the string parameter of the operation

***d*** size of the alphabet

**Word Matching with a Trie**

* We insert the words of the text into a trie
* Each leaf stores the occurrences of the associated word in the text





**Compressed Trie**

* A compressed trie has internal nodes of degree at least two
* It is obtained from standard trie by compressing chains of “redundant” nodes





**Suffix Trie**

* The suffix trie of a string ***X*** is the compressed trie of all the suffixes of ***X***



* Compact representation of the suffix trie for a string ***X*** of size ***n*** from an alphabet of size ***d***
  + Uses ***O***(***n***) space
  + Supports arbitrary pattern matching queries in ***X*** in ***O***(***dm***) time, where ***m*** is the size of the pattern

