# Chapter 4

# String Matching

Given a string $P_m$ called a pattern and a longer string $T_n$ called the text, the (exact) string matching problem is to find all occurrences, if any, of pattern $P_m$ in text $T_n$.

Example: $P_m = aba, T_n = bbabaxababay$
$P_m$ occurs in $T_n$ starting at locations: $3, 7, 9$. Note that two occurrences of $P_m$ may overlap. □

Some off the more common applications are in:

a) word processors

b) utilities such as grep on UNIX

c) textual information retrieval programs such as MEDLINE, LEXIS or NEXIS

d) library catalog searching programs that have replaced physical card catalogs in most large libraries

e) Internet browsers and crawlers that sift through massive amounts of text available on the Internet for material containing specific keywords

f) Internet news readers that can search for the articles for topics of interest

g) telephone directory assistance

h) on-line dictionaries and thesauri

i) numerous specialized databases, such as molecular biology databases holding raw DNA, RNA and amino acid strings, or processed patterns called motifs derived from the raw string data. Genbank is the major U.S. DNA database for example.

## 4.1 Naive String Matching

The naive method aligns the left end of $P_m$ with the left end of $T_n$ and then compares the characters of $P_m$ and $T_n$ left to right until either two unequal characters are found or until $P_m$ is exhausted, in which case an occurrence of $P_m$ is reported. In either case, $P_m$ is then shipped one place to the right, and the comparisons are restarted from the left end of $P_m$. The process is repeated until the right end of $P_m$ shifts past the right end of T.

| Naive-String |
|---|
| **for** $(i = 0; T[i] \neq' \backslash 0'; i++)$ |
|   **for** $(j = 0; (T[i+j] \neq' \backslash 0') \wedge (P[j] \neq' \backslash 0') \wedge (T[i+j] == P[j]); j++);$ |
|   **if** $(P[j] ==' \backslash 0')$ **then** output "found a match" |

Naive-String has two nested loops: The outer loop runs in $O(n)$ time, and the inner loop runs in $O(m)$ time. So the running time of Naive- String is $O(mn)$. The naive string matching algorithm can be visualized as a sliding template containing the pattern $P_m$ over the text $T_n$, and marking down the shifts for which all of this characters on the template equal the corresponding characters in the text $T_n$.

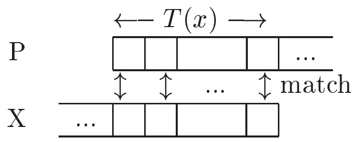
$$\begin{array}{lll} T_n & & \boxed{\text{AGRICULTURE...}} \\ P_m & \text{SHIFT} \rightarrow & \boxed{\text{INA}} \end{array}$$

Here, the template (pattern $P_m = INA$ slides next to $T_n = AGRICULTURE$, from left to right. The pattern in the figure has been shifted 3 ($s = 3$) units, and we still have no match. Naive-String is not optimal for the string matching problem. In the next section, we introduce notions and terminology that we use to investigate string matching algorithm that are more efficient than the naive algorithm.

## 4.2 String Matching Algorithms [CLR90]

**N**otation:

| | |
|---|---|
| $P[1..m]$ | array of text of length m (also denoted by $P_m$) |
| $\omega \sqsubset x$ | $\omega$ is a prefix of $x$, i.e. $x = \omega y$ for some string $y \in \sum^*$ |
| $\omega \sqsupset x$ | $\omega$ is a suffix of $x$, i.e. $x = y\omega$ for some string $y \in \sum^*$ |
| $P_k$ | $k$-character prefix $P[1..k]$ of $P1..m]$ |
| | Note: $P_0 = \epsilon$ $\qquad P_m = P = P[1..m]$ |
| $\tau$ | auxiliary function (aka suffix function corresponding to $P$) |
| | $\tau : \sum^* \rightarrow \{0, 1, ..., m\}$ |
| | $x \rightarrow \tau(x) =$ length of longest prefix of $P$ that is a suffix of $x$ |
| | i.e. $\tau(x) = max\{k \vert P_k \sqsubset x\}$ |




**E**xample: $\sum = \{a, b, c\}, P = abbac$
Then $\tau(\epsilon) = 0, \tau(caba) = 1, \tau(bbcababb) = 3$

## 4.3 Knuth-Morris-Pratt Algorithm [KMP]

KMP is a linear-time string-matching algorithm. It uses an auxiliary function $\pi[1..m]$ which is precomputed from the pattern.

The speed up in KMP is based on the following observation:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T$ | a | a | c | b | a | c | c | b | a | c | c | c | a | c | ... | |
| $P$ | | | | | a | c | c | b | a | c | c | b | | | | |
| $P$ | | | | | | | | | a | c | c | b | a | c | c | b |

$\longleftarrow q \longrightarrow$ (under the first $P$, first 7 characters) ; $\longleftarrow k \longrightarrow$ (under the second $P$, first 3 characters)

$\longleftarrow \quad s \quad \longrightarrow$

$\longleftarrow \quad s' = (s+4) \quad \longrightarrow$

The first 7 characters of $P$ match the characters of $T$ after a shift $s$ of length 4.

From the 7 (value of $q$) characters we know that we can get
<u>no</u> match by performing a $s+1$ shift,
<u>no</u> match by performing a $s+2$ shift,
<u>no</u> match by performing a $s+3$ shift,
<u>but</u> a <u>potential</u> match at a $s+4$ shift since $\boxed{\text{a}}$ of $P$ is aligned with $\boxed{\text{a}}$ of $T$.

KMP will not perform the $s+1, s+2, S+3$ shifts. It will proceed directly to $s+4$.

How?

The information it needs is contained in $\pi$ that it computes from $P$ before even starting the matching process.

Back to the strings $T$ and $P$:

Given that pattern characters $P[1..9]$ match text characters $T[s+1..s+9]$, what is the least shift $s' > s$ such that:

$P[1..k] = T[s'+1..s'+k]$ where $s'+k = s+9$ ? Such a shift $s'(s' = s+4$ in our example )is the first shift greater than $s$ that would yield a potential match. Note that *acc* (i.e. $P_k$) is a suffix of $P_q$. So an equivalent question to the one above is: What is the longest value of $k, k < 9$ such that $P_k \sqsubset P_q$ ? Once we compute $k$, we know that the next potential match might occur at a shift $s' = s + (9-k)$. The precomputed information that KMP uses is the prefix function for the pattern $P$:

$$\pi : \{1, 2, ..., m\} \rightarrow \{0, 1, ..., m-1\}$$
$$9 \rightarrow \pi[9]$$

where $\pi[9] = max\{k | (k < 9) \wedge (P_k \sqsubset P_q)\}$ i.e. $\pi[9]$ is the length of the longest prefix of $P$ that is a **proper** suffix of $P_q$.

Examples:

1. Let $P = accbaccb$

| | $P_q$ | $P_k$ | |
|---|---|---|---|
| $P_1$ | a | | |
| | | a | $\pi[1] = 0$ |
| $P_2$ | ac | | |
| | | ac | $\pi[2] = 0$ |
| $P_3$ | acc | | |
| | | acc | $\pi[3] = 0$ |
| $P_4$ | accb | | |
| | | accb | $\pi[4] = 0$ |
| $P_5$ | accba | | |
| | a | ccba | $\pi[5] = 1$ |
| $P_6$ | accbac | | |
| | ac | cbac | $\pi[6] = 2$ |
| $P_7$ | accbacc | | |
| | acc | bacc | $\pi[7] = 3$ |
| $P_8$ | accbaccb | | |
| | accb | accb | $\pi[8] = 4$ |

2. If $P = ababababca$ then $\pi[1] = \pi[2] = \pi[9] = 0, \pi[3] = \pi[10] = 1, \pi[4] = 2, \pi[5] = 3, \pi[6] = 4, \pi[7] = 5$ and $\pi[8] = 6$.

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $P[q]$ | a | b | a | b | a | b | a | b | c | a |
| $\pi[q]$ | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 | 1 |

| | $P_q$ | $P_k$ | |
|---|---|---|---|
| $P_1$ | a | | |
| | | a | $\pi[1] = 0$ |
| $P_2$ | ab | | |
| | | ab | $\pi[2] = 0$ |
| $P_3$ | aba | | |
| | a | ba | $\pi[3] = 1$ |
| $P_4$ | abab | | |
| | ab | ab | $\pi[4] = 2$ |
| $P_5$ | ababa | | |
| | aba | ba | $\pi[5] = 3$ |
| $P_6$ | ababab | | |
| | abab | ab | $\pi[6] = 4$ |
| $P_7$ | abababa | | |
| | ababa | ba | $\pi[7] = 5$ |
| $P_8$ | abababab | | |
| | ababab | ab | $\pi[8] = 6$ |
| $P_9$ | ababababc | | |
| | | abababab | $\pi[9] = 0$ |
| $P_{10}$ | abababababca | | |
| | a | babababca | $\pi[10] = 1$ |

$P = ababababca$

For example for $q = 9$ we have:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_7$ | = | a | b | a | b | a | b | a | | | | | | | |
| | | | | | | | | a | b | a | b | a | b | a | $[k = 1]$ |
| | | | | | | a | b | a | b | a | b | a | | | $[k = 3]$ |
| | | | | a | b | a | b | a | b | a | | | | | $[k = 5]$ |

$\pi[7] = max\{k|(k < 7) \wedge (P_k \sqsubset P_7\} = max\{1, 3, 5\} = 5$

Similarly, for $q = 8$ well have:
$\pi[8] = 6$
$P_8 = abababab$
But note that for $q = 9$ we have:
$P_q = ababababc$ and so $\pi[9] = 0$
and finally $P_{10} = ababababca$ and $\pi[10] = 1$ $\square$

**Remark:**
$\pi : \{1, 2, ..., m\} \to \{0, 1, 2, ..., m-1\}$
$9 \to \pi[q]$
where $\pi[9] = max\{k|(k < q) \wedge /P_k \sqsubset P_q)\}$
i.e., $\pi[9]$ is the length of the longest prefix of $P$ that is a proper suffix of $P_q$

In EXAMPLE 2:

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $P[q]$ | a | b | a | b | a | b | a | b | c | a |
| $\pi[q]$ | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 | 1 |

The auxiliary function $\pi$ indicates how much of the last comparison can be reused if it fails. It shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison. If the comparison fails at $q = 9$ for instance, then we know that $a$ and $b$ in positions 3 and 4 are identical to $a$ and $b$ in positions $\pi[3] = 1$ and $\pi[4] = 2$. The comparison fails at $q = 9$. The portion immediately preceding $q = 9$ is: $\pi[8] = 6$, so $ababab$ in positions 3 to 8 match $ababab$ in positions 1 to 6. For t hat reason, it is sometimes called the failure function.

---

PREFIX-FUNCTION($G$)

---

$m \leftarrow$length of pattern $P$
$\pi[1] = 0$ //this is a special case
$k \leftarrow 0$
**for** $q \leftarrow 2$ **to** $m$
2     **while** 1 $\boxed{(k>0)}$ $\wedge P[k+1] \neq P[q]$) //is this the end of
        $k \leftarrow \pi[k]$
//the match ?
    **if** $P[k+1] = P[q]$ //we have a match so
//increment $k$
        **then** $k \leftarrow k + 1$
    $\pi[q] \leftarrow k$
**return** $\pi$

---

1 Note that the whole loop is activated only when $k > 0$
(and not the very first time, when $q = 2$).
First we must have a match $P[k+1] = P[q]$ to increment $k$ (to $k+1$).

2 $P[q] \neq P[k+1]$ and so now we go back in $P$ to find the
1 character that matches $P[q]$.
Note that <u>all</u> positions $q-1, q-2, q-3, q-4, \ldots$ are checked.

---

KMP($T, P$)

---

$n \leftarrow$length of $T$
$m \leftarrow$length of pattern $P$
$\pi[1] \leftarrow$ PREFIX-FUNCTION($P$)
$q \leftarrow 0$
**for** $i \leftarrow 1$ **to** $n$
    **while** $(q > 0) \wedge P[q+1] \neq T[i]$) //pattern match has failed
        $q \leftarrow \pi[q]$ //move pattern back
    **if** $P[q+1] = T[i]$ //potential pattern match
        **then** $q \leftarrow q + 1$
    **if** $q = m$ //success of pattern match
        **then print** "Pattern occurs
                with shift $i - m$"
            $q \leftarrow \pi[q]$
**end**

---

EXAMPLE:
$T = BANANORANANANANO$
$P = NANA$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | A | N | A | N | O | R | A | N | A | N | A | N | A | N | O |
| | | N | A | N | A | | | | | | | | | | |
| | | | | N | A | | | | | | | | | | |

no comparison is performed here.
$T[i] \neq P[q+1] : T[6] \neq P[3+1]$
$\Rightarrow q \leftarrow \pi[3]$    [i.e.$q \leftarrow 1$].

after $T[6] \neq P[3+1]$ we compare $T[6]$ and $P[1+1]$
because we know that $T[5] = P[1]$.

The procedure outputs:
Pattern occurs with shift 8
Pattern occurs with shift 10

KMP$(T, P)$ assumes that we have computed the auxiliary function.
PREFIX-FUNCTION$(P)$ gives:

| $P$ | N | A | N | A |
|---|---|---|---|---|
| $\pi$ | 0 | 0 | 1 | 2 |

□

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i:$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | | |
| | | | | || | | | | | | | || | | | | |
| $T$ | a | a | c | b | a | c | c | b | a | c | c | c | a | c | ... | |
| | | | | $q:$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | | | |
| $P$ | | | | | a | c | c | b | a | c | c | b | | | | |
| | | | | || | | | | | | | || | | | | |
| $P$ | | | | | | | | | a | c | c | b | a | c | c | b |

In the **while** loop:
$P[7+1] \neq T[12]$
$q \leftarrow \pi[7]$ (i.e. $q \leftarrow 3$)
Compare $P[\pi[q]+1]$ to $T[12]$
which has the effect of having shifted $\pi[9]+1$ characters
(where $q$ is the last position in $P$ where we have a character match).

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i =$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
| | | | | || text partially matched | | | | | | | || | | | |
| $T:$ | a | a | c | b | a | c | c | b | a | c | c | c | a | c | ... |
| | | | | || | | | | | | | || | | | |
| $P:$ | | | | | a | c | c | b | a | c | c | b | | | |
| | | | | $\leftarrow \pi[q] \longrightarrow$ | | | | $\longleftarrow$ | | $\longrightarrow$ || | | | | |
| | | | | | | | | □ position to which the pattern is shifted | | | | | | | |
| | | | | $\longleftarrow$ | | | | $q$ | | | $\longrightarrow$ | | | | |

Suppose $T[i-q, ..., i-1]$ matches $P[1, ..., q]$ we have a mismatch at $i$:
$T[i] \neq P[q+i]$.

At this point we know that:
$P[1..\pi[q]]$ is the longest proper prefix of $P[1..q]$ that is also a suffix of $P[1..q]$.
So $P$ is shifted $\pi[q]+1$ characters to the right so that we now have:
$T[i-\pi[q]..i-1]$ matching $P[1..\pi[q]]$ and we continue. KMP now compares $T[i]$ to $P[\pi[q]+1]$ □

Procedure KMP$(T, P)$ compares $P[q+1]$ and $T[i]$. In every iteration through the for loop, one of three things happens:

1. $q > 0$ and $P[q+1] \neq T[i]$
   which signals the end of a character match and therefore the current potential pattern match has failed. So now the question is how far back do we move the pattern to continue with the search (and to look for another potential pattern match).

2. $P[q+1] = T[i]$
   which means that we have a character match and the current potential pattern match is still possible.
   So now let us compare:
   $P[q+2]$ and $T[i+1]$ (at the next iteration of the for loop).

3. $q = m$
signals the success of a pattern match (all $m$ characters of the pattern were matched). So the procedure announced it, and as was the case with 1., we now move back the pattern to resume the search for the next possible pattern match.

### 4.3.1 Analysis of Prefix-Function

```
      PREFIX-FUNCTION(G)
      m ←length of pattern P
      π[1] = 0                              //this is a special case
(1)   k ← 0
      for q ← 2 to m
        while (k>0) ∧P[k + 1] ≠ P[q])       //is this the end of the match ?
(2)       k ← π[k]
        if P[k + 1] = P[q]                  //we have a match so increment k
(3)       then k ← k + 1
        π[q] ← k
      return π
```

**Claim:** The running time of Prefix-Function is $O(m)$.
Perform an amortized analysis where the potential is the variable $k$ of the algorithm. The potential has 0 as initial value (see (1)). Since $\pi[k] < k, k$ is decreased (see (2)) but never becomes negative since $\pi[k] > 0$ for all $k$. The other place where $k$ can change value is (3), where it can be incremented by one for each iteration of the for loop. So we have $k < 9$ upon entering the for loop, and since $q$ is incremented by one at each iteration while $k$ is either:

a) incremented by one (for some iterations (see 3)) or

b decremented (see 2)

Well still have $k < q$ (at all times). So the amortized cost of the for loop body is $O(1)$. The final potential function is at least as great as the initial potential function, and the for loop iterates $m - 1$ times. So the total actual worst-case running time of Prefix-Function is $O(m)$.

### 4.3.2 Analysis of KMP

```
      KMP(T, P)
      n ←length of T
      m ←length of pattern P
(5)   π[1] ← PREFIX-FUNCTION(P)
(1)   q ← 0
      for i ← 1 to n
        while (q > 0) ∧ P[q + 1] ≠ T[i])    //pattern match has failed
(2)       q ← π[q]                          //move pattern back
        if P[q + 1] = T[i]                  //potential pattern match
(4)       then q ← q + 1
        if q = m                            //success of pattern match
          then print "Pattern occurs
                 with shift i − m"
(3)         q ← π[q]
      end
```

**Claim:** The running tome of KMP is $O(n)$.

Perform an amortized analysis there the potential is the value of $q$. The potential has an initial value of 0 (see (1)). Since $\pi[q] < q$, $q$ is decreased (see (2 and 3)) but never becomes negative since $\pi[q] \geq 0$. The potential function is increased by at most one (see (4)), so the amortized cost of the for loop body is $O(1)$. The final potential function is at least as great as the initial potential function, and the for loop iterates $n$ times. So the total actual worst-cast running time of the for loop of KMP is $O(n)$.

But KMP makes use of PREFIX-FUNCTION (at (5)) which is $O(m)$. So the time complexity of KMP is $O(m+n)$. □

KMP solves the pattern matching problem in $O(m+n)$ instead of $O(mn)$ of the naive algorithm. The whole idea is based on shifting the pattern as far to the right as possible without bypassing potential matches, whenever a complete match occurs or a character mismatch occurs.

## 4.4 Rabin-Karp Algorithm

The Rabin-Karp algorithm interprets the symbols of the alphabet, $\Sigma$, as numbers and considers the strings, $T$ and $P$, as numbers to the base $d$, where $d = |\sum|$. So for instance the string 31415 corresponds to the decimal number $31,415$ when we let $\Sigma = \{0, 1, ..., 9\}$.

Let $p$ be the number corresponding to $P[1..m]$, and $t_s$ be the number corresponding to $T[s+1, ..s+m]$.

**Note:** $t_s = p \Leftrightarrow T[s+1..s+m]$

In other words, $s$ is a valid shift if and only if $t_s = p$. Comparison of a pattern with part of the text involves comparing 2 numbers, which takes time $O(1)$ (once the numbers are computed). Use Horner's rule to compute $p$ and $t_s$. The number associated with pattern $p$, $p = P[m] + d(P[m-1] + d(P[m-2] + ... + d(P[1])...)$ where $P[1]$ is the highest order digit, and $P[m]$ is the lowest order digit.

The time to compute $p$ is $O(m)$ assuming each arithmetic operation takes $O(1)$ time. The value $t_0$ can be similarly computed from $T[1..m]$ in time $O(m)$.

$t_{s+1} = d(t_s - d^{(m-1)}T[s+1]) + T[s+m+1]$, where

| | | |
|---|---|---|
| $d$ | : | shifts left |
| $t_s - d^{(m-1)}T[s+1]$ | : | removes the high order digit |
| $T[s+m+1]$ | : | adds the low order digit. |

EXAMPLE:

Suppose $d = 10, m = 5$ and $t_s = 31415$. Also suppose that $T[s+5+1] = 2$; in other words, the text has '2' right after 5 of 31415.

$t_{s+1} = d(t_s - d^{(m-1)}T[s+1]) + T[s+m+1]$

$t_{s+1} = 10(31415 - 10^3(3)) + 2$

since $t_s = 31415 = T[s+1..s+m]$

i.e. $T[s+1] = 3$ So $t_{s+1} = 10(31415 - 30000) + 2$

$= 10(1415) + 2$

$= 14152$, which is what we have.

$$\overbrace{3 \quad 1 \quad 4 \quad 1 \quad 5}^{t_s} \quad 2$$
$$3 \quad \underbrace{1 \quad 4 \quad 1 \quad 5 \quad 2}_{t_{s+1}}$$

□

Since $t_{s+1}$ can be computed from $t_1$ in $O(1)$ time, $p$ and $t_0, t_1, ..., t_{n-m}$ can all be computed in $O(n+m)$ and we can thus find all occurrences of the pattern

$P = P[1..m]$ in the text $T = T[1..n]$ in time $O(n+m)$. However, if $m$ and $d$ are large, this running time is unrealistic. Perform all the arithmetic $mod\ q$, for some prime number $q$ of base $d$, such that $dq$ fits within one computer word. This will allow all of the necessary computations to be performed with single-precision arithmetic.

So the recurrence equation with $mod\ q$ is:
$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod\ q$
where $h \equiv d^{(m-1)}(mod\ q)$.

EXAMPLE: $T = 5\quad 5\quad 3\quad 1\quad 2\quad 2\quad 7\quad 3\quad 1$
$\Sigma = \{0, 1, ..., 9\}$ $\qquad P = 12$ and $q = 3$.
Note $P\ mod\ q = 0$.

$$\underbrace{5\quad 5}_{55\ mod\ 3=1}\quad 3\quad 1\quad 2\quad 2\quad 7\quad 3\quad 1$$

$$5\quad \underbrace{5\quad 3}_{2}\quad 1\quad 2\quad 2\quad 7\quad 3\quad 1$$

$$5\quad 5\quad \underbrace{3\quad 1}_{1}\quad 2\quad 2\quad 7\quad 3\quad 1$$

$$5\quad 5\quad 3\quad 1\quad \underbrace{2\quad 2}_{0}\quad 7\quad 3\quad 1$$

match at position 4

$$5\quad 5\quad 3\quad 1\quad 2\quad \underbrace{2\quad 7}_{0}\quad 3\quad 1$$

spurious hit at position 6

$$5\quad 5\quad 3\quad 1\quad 2\quad 2\quad \underbrace{7\quad 3}_{1}\quad 1$$

$$5\quad 5\quad 3\quad 1\quad 2\quad \underbrace{2\quad 7}_{1}\quad 3\quad 1$$

$$5\quad 5\quad 3\quad 1\quad 2\quad 2\quad 7\quad \underbrace{3\quad 1}_{1}$$

Note that
$t_6 = T[6,7] = 27$ and $27 \equiv p(mod\ 3)$
But $27 \neq p$
So we might have $t_s = p(mod\ q)$ but $T[s+1..s+m] \neq P[1..m]$ This is called a spurious hit. In this case, we need to explicitly compare $P[1..m]$ and $T[s+1..s+m]$ which takes $O(m)$ time. Thus, whenever we have $t_s \equiv p(mod\ q)$ we should compare $t_s$ and $p$ to determine if we have a valid pattern match or just a spurious hit. □

```
RABIN-KARP(T, P, d, q)
n ←length of T
//p mod q and t_s mod q
//are the fingerprints of P and t_s
m ←length of pattern P
h ← d^(m−1) mod q
p ← 0
t_0 ← 0
for i ← 1 to m                              //loop computes:
     p ← (dp + P[i]) mod q                  //p ← P[1..m] mod q
     t_0 ← (dt_0 + T[i]) mod q              //t_0 ← T[1..m] mod q
for s ← 0 to n − m                          //iterate through all pos. shifts
(*)if p = t_s                               //is p = t_s where
//if p = t_s, and P[1..m] ≠ T[s + 1..s + m]
//then we have a spurious hit.              //t_s ≡ T[s + 1..s + m] mod q
     then if P[1..m] = T[s + 1..s + m]
          then print "Pattern occurs
          with shift s"                     //we have a pattern match
          if s < n − m then                 //success of pattern match
             t_(s+1) ← (d(t_s − T[s + 1]h)
                  +T[s + m − 1]) mod q
end
```

To make sure that $t_s = T[s+1..s+m] mod\ q$ whenever we check $p = t_s$ at (*), we need the last **if** statement in the **for** loop (**if** $s < n-m$) and obviously $t_{s+1}$ should not be updated if we reach the end of $T$.

EXAMPLE: $P = 31415$

```
RABIN-KARP(T, P, d, q)
n ←length of T                              T = 235902314152...
m ←length of pattern P                      P = 31415, q = 13
h ← d^(m−1) mod q                           h = 10^4 mod 13
p ← 0
t_0 ← 0
for i ← 1 to m                              31415 ≡ 7 mod 13
     p ← (dp + P[i]) mod q                  p ≡ 7 mod 13
     t_0 ← (dt_0 + T[i]) mod q              23590 = t_0 ≡ 8 mod 13
for s ← 0 to n − m                          t_0 = T[1..m] = 23590
  if p = t_s                                p = 7 and t_0 = 8
     then if P[1..m] = T[s + 1..s + m]
          then print "Pattern occurs
          with shift s"
          if s < n − m
             then t_(s+1) ← (d(t_s − T[s + 1]h)
                     +T[s + m − 1]) mod q
end
```

$t_1 = 10[23590 - 2(1000)] + T[6] = 10(3590) + 2 = 35902$
So next time through the loop check $p = 7$ and $t_1 = 9$ (since $35902 \equiv 9\ mod\ 13$).

### 4.4.1 Analysis of Rabin-Karp

A) Worst-case analysis:

RABIN-KARP$(T, P, d, q)$

$n \leftarrow$length of $T$
$m \leftarrow$length of pattern $P$
$h \leftarrow d^{m-1} mod\ q$
$p \leftarrow 0$
$t_0 \leftarrow 0$
**for** $i \leftarrow 1$ **to** $m$
    $p \leftarrow (dp + P[i]) mod$ q
    $t_0 \leftarrow (dt_0 + T[i]) mod$ q

**for** $s \leftarrow 0$ **to** $n - m$
  **if** $p = t_s$
    **then if** $P[1..m] = T[s+1..s+m]$
        **then print** "Pattern occurs with shift $s$"
        **if** $s < n - m$
          **then** $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m-1]) \bmod q$
**end**

Example (of worst case)
$P = a^m$ and $T = a^n$

Since each one of the $n - m + 1$ shifts is valid, the verification will take $\Theta((n - m + 1)m)$ time ($m$ characters per iteration)

The running time of RABIN-KARP is $\Theta((n - m + 1)m)$ in the worst case since we verify **every** valid shift. Here $p = t_s$ is always true.

B) Average case analysis:
RABIN-KARP$T, P, d, q)$ has an expected time of $O(n + m)$ plus the time required to process spurious hits (when we expect few valid shifts - perhaps as low as $O(1)$. Let us assume that reducing values modulo $q$ is like a random mapping from $\Sigma^*$ to $Z_q$ $(Z_q = \{0, ..., q-1\})$. Then the expected number of spurious hits is $O(\frac{n}{q})$ since the probability that for any given $t_s$ we have: $t_s = p\ mod\ q$ is $\frac{1}{q}$. So the expected running time is:
$\underbrace{O(n)}_{without\ spurious\ hits} + O(m(v + \frac{n}{q}))$, where $s$ is the number of pattern matches.
Note that if $q \geq$ ($q$ is larger than $P$) and the expected number of the valid shifts is small ($O(1)$ let's say), then RABIN-KARP runs in $O(n + m)$ time. $\square$

The prime number $q$ should be as large as possible, consistent with $(|\sum| + 1)q$ not causing arithmetic overflow. Sedgewick uses a value of: $q = 33554393$ for $|\Sigma| = 32$ [see Algorithms, Addison-Wesley, 1983]. Pirklbauer uses a value of: $q = 8355967$ for $|\Sigma| = 256$ [ see "A study of pattern-matching algorithms" in Structured programming, Volume 13, 1992]. Gonnet and Baeza-Yates present an implementation of the algorithm that goes away with the mod operations by virtue of the implicit modular arithmetic of the target hardware. Spurious hits are also known as signature collisions. A more recent implementation of the algorithm reduces the probability of a large number of collisions by randomly reselecting the prime $q$ (after a collision), reinitializing, and then continuing with the search.

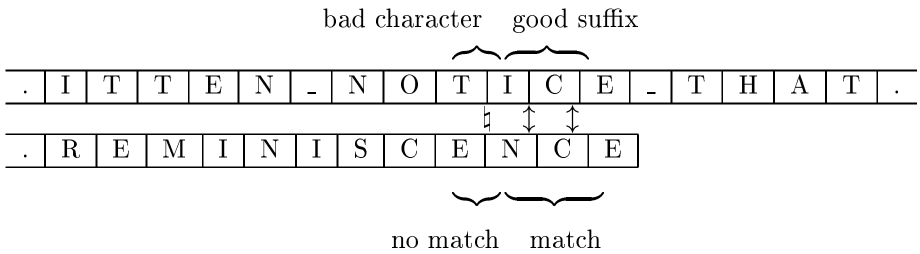
## 4.5 Boyer-Moore

As is the case with KMP, the efficiency of BOYER-MOORE depends on determing the location of the next matching attempt, given that the previous attempt was unsuccessful. Once again, following a mismatch at the $q + 1$ position of the pattern, the preceding $q$ symbols of the pattern and their structure give insight as to where the next matching attempt should begin.For the KMP and for the BOYER-MOORE algorithm, knowing how far to shift the pattern is the most important factor. In both algorithms, the pattern is preprocessed to determine the shift for a failure at the $q^{th}$ symbol of $P$ for each $q = 1, 2, ..., m$. BOYER-MOORE is more efficient than KNUTHMORRISPRATT algorithm, on the average, especially when the pattern $P$ is long and the alphabet $\sum$ is large.

As with KMP, we slide $P$ along $T$ from left to right, checking corresponding characters. But with the BOYER-MOORE algorithm (BM), the characters of $P$ are checked right to left after each movement of the pattern. The information gained at the end of the pattern (i.e., on the right) often allows the algorithm to proceed in large jumps through the text being searched. Thus BM has the unusual property that, in most cases, not all of the first $q$ characters of $T$ are inspected. The number of characters actually inspected (on the average) decreases as a function of the length of the pattern $P$.

BM uses two heuristics that allow it to avoid much of the work that KMP performs. The heuristics are: the bad-character heuristic and the good suffix heuristic. The heuristics allow BM to skip the examination of many text characters. When a mismatch occurs, each heuristic proposes an amount by which the pattern can be safely shifted without missing a match. BM chooses the larger amount of the two and shifts the pattern by that amount. The two heuristics can be viewed as operating independly in parallel.

EXAMPLE:




Shift $s$ is invalid; although

i) a "good suffix" $\subset E$ of $P$ matched in $T$, we have:

ii) a "bad character" in $T$, which did not match pattern character $n$ □

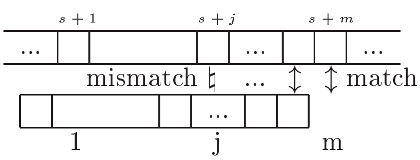
### 4.5.1 The Bad-Character Heuristic

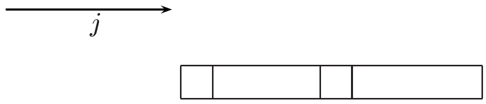
Suppose we have just found a mismatch: $P[j] \neq T[s+j]$ for some $j, 1 \leq j \leq m$. Let $k$ be the largest index, $1 \leq k \leq m$, such that $T[s+j] = P[k]$, if any such $k$ exists. If such $k$ does not exist (if the pattern does not contain a character such that $T[s+j] = P[k], 1 \leq k \leq m$) then let $k = 0$.

**Claim:** Increase $s$ by $j - k$. We have three cases to consider:

- Case one: $k = 0$
- Case two: $k < j$
- Case three: $k > j$

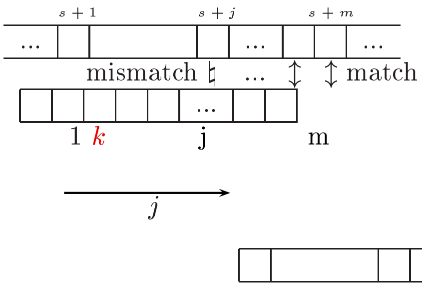
**Case one:** $k = 0$






Here, the bad character, $T[s+j]$ is not found in $p$. $k=0$. So we shift $p$ until it has passed over the bad character) (i.e., just to the right of the bad character). We increase $s$ by $j-k=j$.

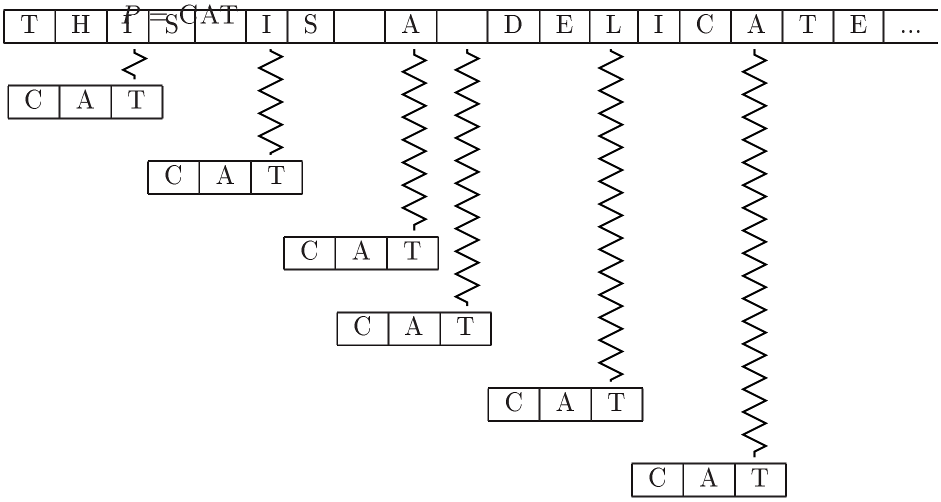
**Case two** : $k < j$




The rightmost occurrence of the bad character is in the pattern to the left of position $j$.
$T[s+j] = P[k]$
We slide $p$ to the right by $j-k$ characters to align $T[s+j]$ with $P[k]$. Since $k$ is the largest index, $1 \le k \le m$, we can safely increase $s$ by $j-k$ without missing any valid shifts.

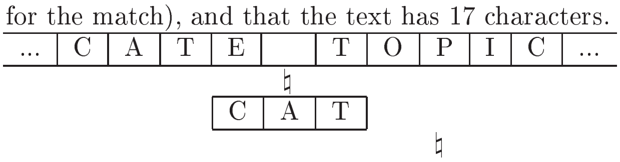
**Case three:** $k > j$
Here $j-k < 0$, and so the bad-character heuristic is proposing to decrease $s$. BM will ignore this recommendation and will choose the recommendation suggested by the good-suffix heuristic.

Example: $T$ = THIS IS A DELICATE TOPIC
$P$ = CAT




and here we find a match: C A T

Note that we have made 9 comparisons up to this point (the 6 mismatches shown +3 for the match), and that the text has 17 characters.



| | | |
|---|---|---|
| C | A | T |

□

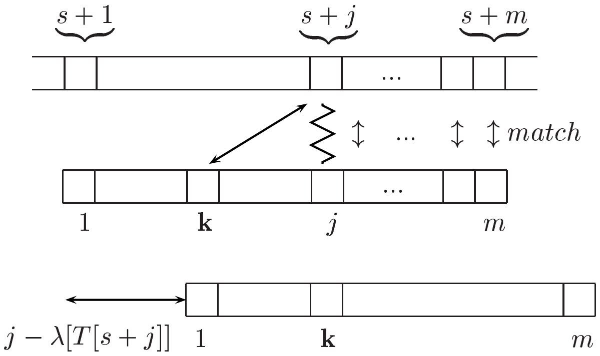
COMPUTE-LAST-OCCURRENCE($P, m, \sum$) computes the rightmost position in the pattern $P$ (of length $m$) at which each character of $P$ occurs. The results are saved in $\lambda$. In other words, $\lambda[a]$ is the index of the rightmost position at which $a$ occurs for each $a \in \sum$.

```
COMPUTE-LAST-OCCURRENCE(P, m, ∑)
for each character a ∈ ∑
    λ[a] ← 0
for j ← 1 to m
    λ[P[j]] ← j
return λ
```

The running time of COMPUTE-LAST-OCCURRENCE is: $O(|\sum| + m)$.
EXAMPLE: $P = reminiscence$

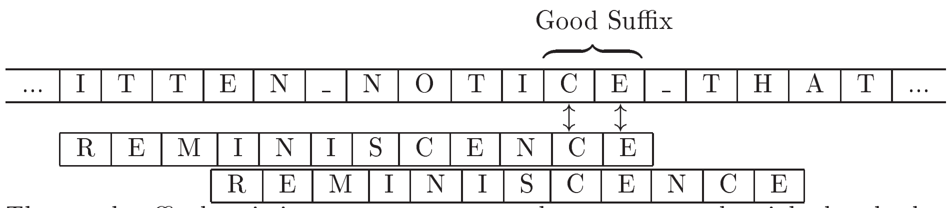
| a | $\lambda[a]$ |
|---|---|
| r | 1 |
| e | 2 9 12 |
| m | 3 |
| i | 4 6 |
| n | 5 10 |
| s | 7 |
| c | 8 11 |

Recall that BM is interested in $\lambda$ in case two and case three. In case two, we increase $s$ by $j - k$ where $k$ is the largest index, $1 \leq k \leq m$, where $T[s+j] = P[k]$. In other words, $k = \lambda[T[s+j]]$. So the bad-character heuristic will recommend to increase $s$ by $j - \lambda[T[s+j]]$.




As for case three, $j - k < 0$, i.e. $j - \lambda[T[s+j]]$ since the bad character $T[s+j]$ occurs to the right of position $j$. In this case, BM relies on the recommendation given by the good-suffix heuristic (which is always positive).

### 4.5.2 The Good-Suffix Heuristic




The good-suffix heuristic proposes to move the pattern to the right by the least amount that guarantees that any pattern characters that align with the good suffix previously found in the text will match those suffix characters. So in out example, we move $p$ 3 characters to the right to align CE of $T$ with the new CE of $P$. The shifts are computed by procedure COMPUTE-GOOD-SUFFIX and are saved in $\gamma$. If $P[j] \neq T[s+j]$, with $j < m$, then the good-suffix heuristic assures that we can safely advance $s$ by $\gamma[j]$ where: $\gamma[j] = m - max\{k|0 \leq k < m$ and $P_k \sqsubset P[j+1..m]$ or $P[j+1..m] \sqsubset P_k$. In other words, $\gamma[j]$ is the least amount we can advance $s$ and not cause any characters in the good-suffix: $T[s+j+1..s+m]$ to be mismatched against the new alignment of the pattern.

EXAMPLE:
In the previous example: $T[s+j] = \text{I} \neq \text{N} = P[j]$
The good suffix is: $T[s+j+1..m] = \text{CE}$
$P = \text{REMINISCENCE}$
CE is a suffix of REMINISCE
So $P_q = \text{REMINISCE}$ and $k = 9$
So shift by $m - k = 12 - 9 = 3$ characters.

EXAMPLE:

$T[s+j]$ ↘ ↙ $T[s+m]$

| ... | I | T | T | E | N | _ | N | O | T | I | C | E | _ | T | H | A | T | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$P[j]$ $\updownarrow$ $\updownarrow$

| R | E | M | I | N | I | S | C | E | N | C | E | $P[j+1..m]$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| R | E | M | I | N | I | S | C | E | N | C | E |
|---|---|---|---|---|---|---|---|---|---|---|---|

CE $\sqsubset$ REMINISCE $\Rightarrow P_q$ so $k = 9$. So shift $P$ by $m - k = 12 - 9 = 3$ characters

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| R | E | M | I | N | I | S | C | E | N | C | E |

We just showed that $\gamma[10] = 3$ since $\gamma[10] = 12 - max\{9\}$

Note that:
$\gamma[11] = 12 - max\{k|E \sqsubset P_k$ or $P_k \sqsubset E\}$
$E \sqsubset RE$ and $E \sqsubset REMINISCE$
$\Rightarrow E \sqsubset P_2$ and $E \sqsubset P_q$
So $\gamma[11] = 12 - max\{2, 9\} \Rightarrow \gamma[11] = 3$

```
                                  ...  N   E
                                   ♮   ↕
  R   E   M   I   N   I   S   C   E   N   C   E
              R   E   M   I   N   I   S   C   E   N   C   E
```

□

It can be shown that:
$\gamma[j] = m - max\{k|0 \leq k < m$ and $P_k \sqsubset P[j+1..m]$ or $P[j+1..m] \sqsubset P_k$
is equivalent to:
$\gamma[j] = min(\{m - \pi[m]\} \cup \{l - \pi'[l] | 1 \leq l \leq m$ and $j = m - \pi'[l]\})$;
where $\pi'$ is the prefix function of $P'$, the reverse of $P$.

Procedure COMPUTE-GOOD-SUFFIX is a straightforward implementation of the last equation.

```
COMPUTE-GOOD-SUFFIX(P, m)
π ← PREFIX-FUNCTION(P)
P' ← REVERSE(P)
π' ← PREFIX-FUNCTION(P')
for j ← 0 to m
   γ[j] ← m − π[m]
for l ← 1 to m
   j ← m − π'[l]
      if γ[j] > l − π'[l]
         then γ[j] ← l − π'[l]
return γ
```

EXAMPLE:

| $P$ | R | E | M | I | N | I | S | C | E | N | C | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\pi$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| $P'$ | E | C | N | E | C | S | I | N | I | M | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\pi'$ | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

| $j$ | R | E | M | I | N | I | S | C | E | N | C | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\gamma[j]$ | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 3 | 3 | 1 |

The running time of COMPUTE-GOOD-SUFFIX is $O(m)$. □

Having covered the bad character procedure and the good suffix procedure, we are now ready to tackle the BOYER-MOORE algorithm.

```
BOYER-MOORE(T, P, ∑)
n ← length of T
m ← length of P
λ ← COMPUTE-LAST-OCCURENCE(P, m, ∑)
γ ← COMPUTE-GOOD-SUFFIX(P, m)
s ← 0
while  s ≤ n − m
  j ← m
    while  j > 0 and P[j] = T[s + j]
      j ← j − 1
    if  j = 0
      then output  "Pattern occurs at shift" s
        s ← s = γ[0]
      else s ← s + max(γ[j], j − λ[T[s + j]])
end
```

### 4.5.3 Analysis of Boyer-Moore

```
BOYER-MOORE(T, P, ∑)
n ← length of T
m ← length of P
λ ← COMPUTE-LAST-OCCURENCE(P, m, ∑)         O(|e| + m)
γ ← COMPUTE-GOOD-SUFFIX(P, m)               O(m)
s ← 0
while  s ≤ n − m                            O(n − m + 1)
  j ← m
    while  j > 0 and P[j] = T[s + j]        O(m)
      j ← j − 1
    if  j = 0
      then output  "Pattern occurs at shift" s
        s ← s = γ[0]
      else s ← s + max(γ[j], j − λ[T[s + j]])
end
```

The worst-case running time of the BOYER-MOORE algorithm is $O((n-m+1)m + |\sum|)$. In practice, BM performs better than other string matching algorithms.

### 4.5.4 Longest Common Subsequence

Suppose we have the same problem as before, a text $T$ of length $n$ and a pattern $P$ of length $m$. We would like to know if the letters of $P$ appear in order (but possibly separated) in $T$. If they do, we say that $P$ is a subsequence of $Y$.

EXAMPLE:

- $T$ = POLYTECHNIC INSTITUTES

- $P$ = TENNIS

Is $P$ a subsequence of $T$? Yes.

POLYTECHNIC INSTITUTES □

What if the pattern does not occur in $T$? It makes sense to ask for the longest subsequence that occurs both in the pattern as well as in the text. This is the common subsequence problem. Note that here the text and the pattern have similar roles (or symmetric roles), so that we don't need to call them "text" and "pattern" anymore. Instead, call them $X =< x_1, x_2, ...x_m >$ and $Y =< y_1, y_2, ..., y_n >$.

There are several areas where we would want to solve the longest common subsequence problem:

1) molecular biology: DNA sequences (genes) can be represented as sequences of 4 letters: $A, C, G, T$, corresponding to the 4 submolecules forming the DNA. When biologists find a new sequence, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.

2) file comparison: The UNIX program "diff" compares 2 different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the 2 files. Note: Any line in the subsequence has not been changed. In other words, what is displayed is indeed the remaining set of lines (that have changed). Here it is convenient to think of each line of a file as being a single character in a string.

3) screen redisplay:
Many text editors, like "Emacs", display part of a file on the screen, updating the screen image as the file is changed. For slow–in terminals, these programs want to send the terminal as few characters as possible to cause it to update its display correctly. Here, the common subsequence is the part of the display that has been corrected and does not not need to be changed. □

To solve the longest common subsequence, we use dynamic programming (recall the string alignment problem and the edit problem). Procedure LCS$(X, Y)$ takes 2 sequences $X =< x_1, x_2, ..., x_m >$ and $Y =< y_1, y_2, ..., y_n >$ as input. It saves the $c[i, j]$ values in a table $c[0..m, 0..n]$ whose entries are computed in a row-major order, where $c[i, j]$ is the length of an LCS of the sequences $X_i$ and $Y_j$. If either $u = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. In general, we have:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i > 0, j > 0 \text{ and } x_i = y_j \\ max(c[i, j-1].c[i-1, j]) & \text{if } i > 0, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Procedure LCS$(X, Y)$ also maintains a table $b[1..m, 1..n]$ to simplify the construction of an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. Procedure LCS$X, Y$ returns the $b$ and $c$ tables, and $c[m, n]$ contains the length of an LCS of $X$ and $Y$. Note that the elements for which we have " $\nwarrow$ " are the characters that belong to an LCS. Also note that LCS$(X, Y)$ could have been designed without the explicit use of table $b$.

```
LCS(X, Y)
m ← length of X
n ← length of Y
for i ← 0 to m
   c[i,0] ← 0
for j ← 0 to n
   c[0,j] ← 0
for i ← 1 to m
   for j ← 1 to n
      if x_i = y_j
         then c[i,j] ← c[i−1,j−1] + 1
            b[i,j] ← "↖"
            else if c[i,j] ≥ c[i−1,j]
               then c[i,j] ← c[i−1,j]
                  b[i,j] ← "↑"
               else c[i,j] ← c[i,j−1]
                  b[i,j] ← "←"
return b and c
```

The running time of LCS$(X,Y)$ is $O(mn)$,
since we are assuming that each table entry takes $O(1)$ time to compute.

EXAMPLE:

| | F | I | T | G | E | I | S | T |
|---|---|---|---|---|---|---|---|---|
| **R** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **E** | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| **T** | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 2 |
| **E** | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| **R** | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| **I** | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 3 |
| **T** | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 4 |

### 4.5.5 More Applications: Computer Viruses

**1) Computer Viruses**

- From: http://www.mactech.com/articles/mactech/Vol.08/08.02/VirusDetection/

- Recently, there has been an explosion in the development rate of computer virus code.
- Computer viruses have attracted the attention of the public, the media, and the scientific community.
- Detection of the virus is simply considered a pattern string to be searched for in a larger text (a possibly infected program).

**There are 3 categories of anti-viral products**

- Infection Prevention Products: halt the virus replication and prevent the initial infection from occurring. Example: cdev (Control Panel Device) Vaccine on Macintosh.
- Infection Detection Products: detect infection right after it has occurred and mark specific components of system segments that have become infected. It is done by periodically inspecting executable files, and may or may not attempt repair of an infected file. Examples: INIT package GateKeeper and GateKeeper Aid on Macintosh.
- Infection Identification Products: identify specific viral strains on systems that are already infected and may remove the virus, returning the system to its state prior to infection. This antivirus is the most common and its effectiveness dependents on the frequency of the user invocation. Examples: McAfee's MS-DOS program SCAN and the Macintosh Disinfectant series.

**Remarks**

- Their anti-virus package uses a modification of the Boyer-Moore algorithm. More specifically, they use a randomized system of string selection.
- Work in string matching, like work in virus detection, is by no means complete.
- Modern algorithms make use of parallel hardware and improved data structures, such as suffix trees (which may be respectively applied to randomized matching and dynamic programming).

**2) Detecting Software Plagiarism Systems**

- A) From: http://wwwipd.ira.uka.de/jplag/
- JPlan can detect software plagiarism.
- JPlag is a system that finds similarities among multiple sets of source code files.
- JPlag currently supports Java, C, C++, and Scheme.
- B) Moss (**M**easure **O**f **S**oftware **S**imilarity) is an automatic system for determining the similarity of C, C++, Java, Pascal, Ada, ML, Lisp, or Scheme programs.

**1) Common Subsequences**

- From: "The Complexity of Some Problems on Subsequences and Supersequences", by D. Maier.
- $S = s_1 s_2 ... s_m$ is a sequence with m elements; $|S| = m$.
  $S'$ is a subsequence of $S$, $S' < S$, if it is a sequence which consists of $S$ with a number of terms between 0 & m deleted.

- $R = \{S_1, ..., S_p\}$ is a set of sequences with alphabet A, i.e., the set of values the different $s_i's$ can take.
- LCS(R), the longest common subsequence of R, is the longest sequence S, such that $S < S_i$ for i=1,2,...,p.
- Similarly, SCS(R), the shortest common sequence of R, is the shortest sequence $S'$, such that $S_i < S'$ for i=1,2,...,p.
- Example If R={ababe, cabe, abdec} then
  (a) A = {a,b,c,d,e}, and $|A| = 5$,
  (b) LCS(R) = abe, and
  (c) SCS(R) = cabeabdec.

### 4.5.6 The Longest Upsequence [by E.W.Dijkstra]

Define an upsequence of length $k$ in an array $L = L_1 L_2 \cdots L_n$ of numbers to be a sequence $i_1, \ldots, i_k$ with $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, for which $L_{i_1} \leq L_{i_2} \leq \cdots \leq L_{i_k}$, i.e., to be a not necessarily contiguous subsequence of $L$ which is non-decreasing from left to right.

The problem, given an array $L$, is to find the greatest length of any upsequence which occurs in $L$. For example, if $L$ is sorted in decreasing order, the answer is 1. Note that for any prefix $B = L_1 \cdots L_m$ of $L$ $(0 \leq m \leq n)$, and for any length $k$ of upsequence which occurs at all in $B$, define the record economical top element (RETE) with respect to $k$ and $B$ to be the smallest number which is the last (i.e., largest) member of an upsequence of length $k$ in $B$. Proceed from left to right, letting $B$ grow from the empty sequence to all of $L$, and at all times keep track of all the RETEs which exist. At the end, the biggest $k$ for which a RETE exists will be the answer.

As we consider each of the numbers $L_1, \ldots, L_n$ in turn, we will find that it is a new RETE in exactly one way. For example, suppose that $L_1 = 3$, $L_2 = 5$, $L_3 = 2$, and $L_4 = 7$. RETE w.r.t $B$ and $k$:

| $B = 3$ | | RETE | UPSEQUENCES |
|---|---|---|---|
| | 1-upsequence | 3 | (3) |
| $B = 35$ | | | |
| | 1-upsequence | 3 (3) | |
| | 2-upsequence | 5 | (3,5) |
| $B = 352$ | | | |
| | 1-upsequence | 2 | (2) [since 2 is smaller than 3] |
| | 2-upsequence | 5 | (3,5) |
| $B = 3527$ | 1-upsequence | 2 (2) | |
| | 2-upsequence | 5 | (3,5) |
| | 3-upsequence | 7 | (3,5,7) |

Note that 2 and 5 and 7 are RETEs for one subsequence (each). $L[5]$ will also be a new RETE in exactly one way.

# Bibliography