Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture 9 – String processing: String searching and Pattern matching

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

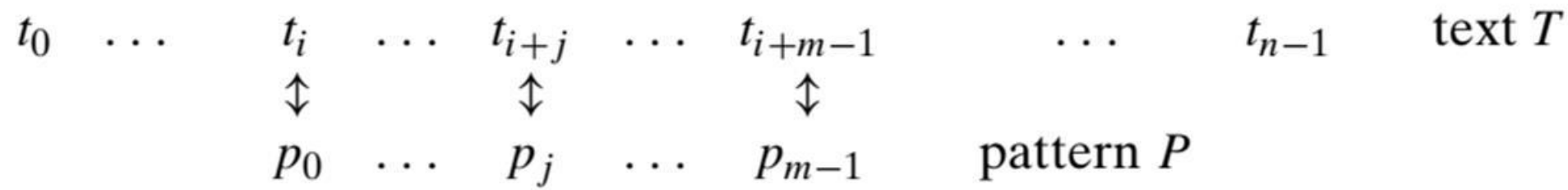
Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

String searching and Pattern matching

- Given a string of *n* characters called the ***text*** and a string of *m* characters *(m ≤ n)* called the ***pattern***, find a substring of the text that matches the pattern.
- To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching substring in the text—such that

$$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$$



Applications

- Web Searches
- Database Queries
- Detecting Plagiarism

String searching and Pattern matching

- If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.
- A brute-force algorithm for the string-matching problem is quite obvious:
  - Align the pattern against the first *m* characters of the text.
  - Start matching the corresponding pairs of characters from left to right until either all the *m* pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.
  - In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text.

String searching and Pattern matching

- Note that the last position in the text that can still be a beginning of a matching substring is *n* - *m* (provided the text positions are indexed from 0 to *n* - 1).
- Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
N  O  B  O  D  Y  _  N  O  T  I  C  E  D  _  H  I  M
N  O  T
   N  O  T
      N  O  T
         N  O  T
            N  O  T
               N  O  T
                  N  O  T
                     N  O  T
                        N  O  T
```

Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

String searching and Pattern matching

ALGORITHM *BruteForceStringMatch*$(T[0..n-1],\ P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

String searching and Pattern matching

- The algorithm shifts the pattern almost always after a single character comparison.
- The worst case is much worse: the algorithm may have to make all *m* comparisons before shifting the pattern, and this can happen for each of the *n* - *m* + 1 tries.
- Thus, in the worst case, the algorithm makes *m(n* - *m* + 1*)* character comparisons, which puts it in the *O(nm)* class.
- For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons.
- Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e.,O*(n+m)*.

String searching and Pattern matching

- Several faster algorithms have been discovered.
- Most of them exploit the input-enhancement idea:
  - preprocess the pattern to get some information about it, store this information in a table, and
  - then use this information during an actual search for the pattern in a given text.
- This is exactly the idea behind the two best known algorithms of this type:
  - The Knuth-Morris-Pratt algorithm and
  - The Boyer-Moore algorithm

String searching and Pattern matching

- The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text:
  - The Knuth Morris-Pratt algorithm does it left to right, whereas the Boyer-Moore algorithm does it right to left.
  - The latter idea leads to simpler algorithms, it is the only one that we will pursue here.
  - Although the underlying idea of the Boyer-Moore algorithm is simple, its actual implementation in a working method is less so.
  - Therefore, we start our discussion with a simplified version of the Boyer-Moore algorithm suggested by R. Horspool.
  - In addition to being simpler, Horspool's algorithm is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

Horspool's Algorithm

- Consider, as an example, searching for the pattern BARBER in some text:

- Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text.
- If all the pattern's characters match successfully, a matching substring is found.
- Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired.

Horspool's Algorithm: Example

- As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$											

B	A	R	B	E	R
0	1	2	3	4	5

Length = 6

Value=length-index-1

Horspool's Algorithm: Example

- As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

B	A	R	B	E	R
0	1	2	3	4	5

Length = 6

Value=length-index-1

Horspool's Algorithm: Example

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

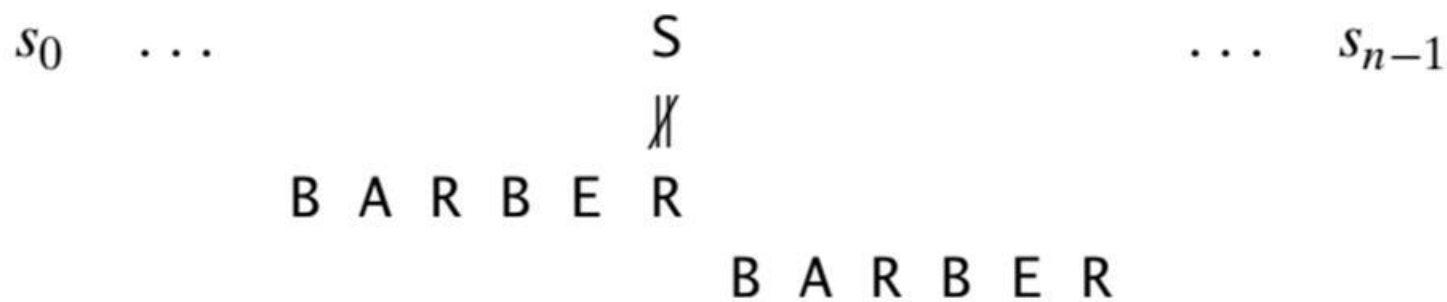
```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                   B A R B E R
```

Horspool's Algorithm

- If a mismatch occurs, we need to shift the pattern to the right.
- Horspool's algorithm determines the size of such a shift by looking at the character *c* of the text that is aligned against the last character of the pattern.
- This is the case even if character *c* itself matches its counterpart in the pattern.
- In general, the following four possibilities can occur.

Horspool's Algorithm

- **Case 1**
  - If there are no *c*'s in the pattern—e.g., *c* is letter S in our example— we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character *c* that is known not to be in the pattern):



Horspool's Algorithm

- **Case 2**
  - If there are occurrences of character *c* in the pattern but it is not the last one there—e.g., *c* is letter B in our example—the shift should align the rightmost occurrence of *c* in the pattern with the *c* in the text:

Horspool's Algorithm

- **Case 3**
  - If *c* happens to be the last character in the pattern but there are no *c*'s among its other *m* - 1 characters—e.g., *c* is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length *m*

Horspool's Algorithm

- **Case 4**
  - Finally, if *c* happens to be the last character in the pattern and there are other *c*'s among its first *m* - 1 characters—e.g., *c* is letter R in our example— the situation is similar to that of Case 2 and the rightmost occurrence of *c* among the first *m* - 1 characters in the pattern should be aligned with the text's *c*:

Horspool's Algorithm

- These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position always made by the brute-force algorithm.
- However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority.
- Fortunately, the idea of input enhancement makes repetitive comparisons unnecessary.
- We can precompute shift sizes and store them in a table.
- The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters.

Horspool's Algorithm

- The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively

Horspool's Algorithm

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Horspool's Algorithm

- Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step m - 1 times:
  - for the j^{th} character of the pattern $(0 \leq j \leq m - 2)$, overwrite its entry in the table with $m - 1 - j$, which is the character's distance to the last character of the pattern.
  - Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

Horspool's Algorithm

- **Step 1**
  - For a given pattern of length *m* and the alphabet used in both the pattern and text, construct the shift table as described above.
- **Step 2**
  - Align the pattern against the beginning of the text.
- **Step 3**
  - Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text.
  - Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all *m* characters are matched (then stop) or a mismatching pair is encountered.
  - In the latter case, retrieve the entry *t(c)* from the *c*'s column of the shift table where *c* is the text's character currently aligned against the last character of the pattern, and shift the pattern by *t(c)* characters to the right along the text.

Horspool's Algorithm

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//          or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shifts
i ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

Horspool's Algorithm: Example

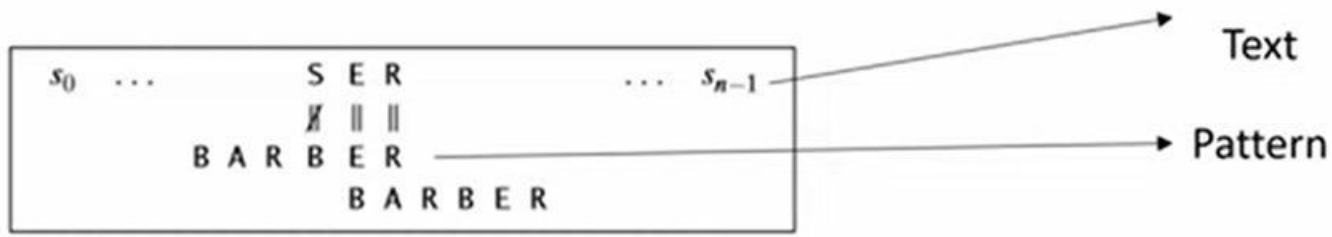
- A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in *O(nm)*.
- Best Case complexity is *O(m/n)*
- But for random texts, it is in *O(n)*, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm because the algorithm may have to make (n - m + 1) tries.
- In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.

Boyer-Moore Algorithm

- If the first comparison of the rightmost character in the pattern with the corresponding character *c* in the text fails, the algorithm does exactly the same thing as Horspool's algorithm.
- Namely, it shifts the pattern to the right by the number of characters retrieved from the table precomputed as explained earlier.

BOYER MOORE

- It is a **pattern matching algorithm**
- Compute **bad shift** using **d1=t1(c) - k**, where **'c'** is the **text character** that caused **mismatch**, t1(c) is the shift table value of 'c' character, and **'k'** is the number of matched characters
  - Bad Shift table of BARBER is same as Horspool's Algorithm. i.e.
    - ✓ Shift values of E=1, B=2, R=3, A=4, B=2.
    - ✓ Shift values of other characters = length of pattern = 6
  - E.g.: For example, if we search for the **pattern BARBER** in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by **t1(c)-k= t1(S)−2=6−2=4** positions



Boyer-Moore Algorithm

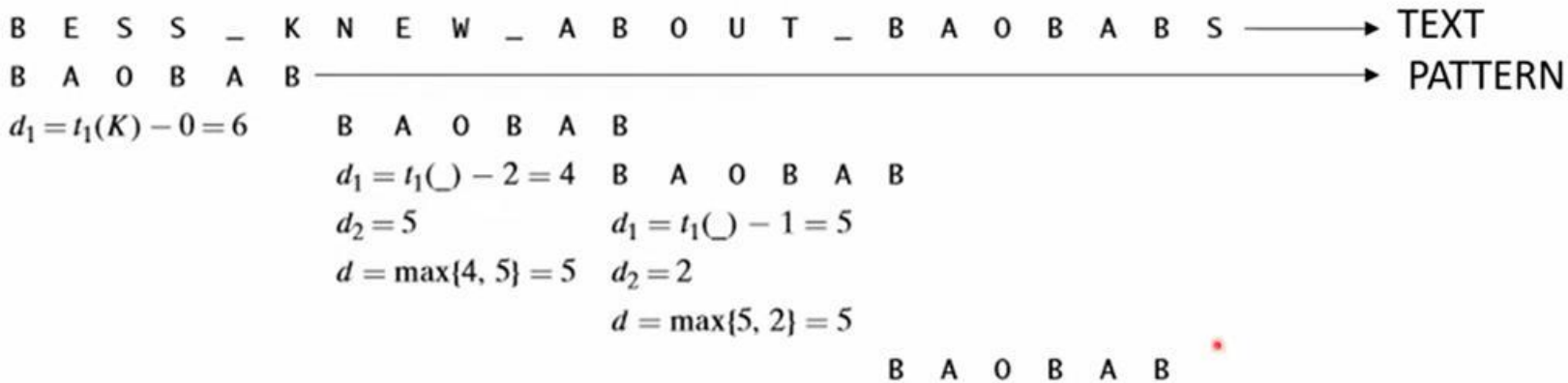
Compute Good-suffix table

k	**pattern**	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4
3	$\overline{\text{AB}}$C$\underline{\text{BAB}}$	4
4	$\overline{\text{AB}}$$\underline{\text{CBAB}}$	4
5	A$\underline{\text{B}}$$\underline{\text{CBAB}}$	4

Shift the pattern by max(d1, d2)

BOYER MOORE Problems

- Example of string matching with the Boyer-Moore algorithm is as follows:



- The bad-symbol table: BAOBAB: A=1,B=2,O=3,A=1,B=2,

other characters=pattern length=6

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

- The Good suffix table

k	pattern	d_2
1	BAO$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{B}}$AOB$\underline{\text{AB}}$	5
3	$\overline{\text{B}}$AO$\underline{\text{BAB}}$	5
4	$\overline{\text{B}}$A$\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}$A$\underline{\text{OBAB}}$	5

Boyer-Moore Algorithm

- Boyer-Moore algorithm determines the shift size by considering two quantities.
  - The first one is guided by the text's character *c* that caused a mismatch with its counterpart in the pattern.
  - Accordingly, it is called the ***bad symbol shift***.
  - The reasoning behind this shift is the reasoning we used in Horspool's algorithm.
  - If *c* is not in the pattern, we shift the pattern to just pass this *c* in the text.
  - Conveniently, the size of this shift can be computed by the formula *t1(c)* - *k* where *t1(c)* is the entry in the precomputed table used by Horspool's algorithm and *k* is the number of matched characters .

Boyer-Moore Algorithm

- For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by *t*1*(*S*)* - 2 = 6 - 2 = 4 positions:

Boyer-Moore Algorithm

- The same formula can also be used when the mismatching character *c* of the text occurs in the pattern, provided *t*1*(c)* - *k* > 0.
- For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by *t*1*(*A*)* - 2 = 4 - 2 = 2 positions

Boyer-Moore Algorithm

- If $t1(c) - k \leq 0$, we obviously do not want to shift the pattern by 0 or a negative number of positions.
- Rather, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.
- To summarize, the bad-symbol shift $d1$ is computed by the Boyer-Moore algorithm either as $t1(c) - k$ if this quantity is positive and as 1 if it is negative or zero.
- This can be expressed by the following compact formula:

$$d1 = \max\{t1(c) - k, 1\}$$

Boyer-Moore Algorithm

- The second type of shift is guided by a successful match of the last $k > 0$ characters of the pattern.
- We refer to the ending portion of the pattern as its suffix of size k and denote it *suff (k).*
- Accordingly, we call this type of shift the ***good-suffix shift***.
- We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern's suffixes of sizes $1, \ldots, m - 1$ to fill in the good-suffix shift table.

Boyer-Moore Algorithm

- The case when there is another occurrence of *suff (k)* in the pattern or, to be more accurate, there is another occurrence of *suff (k)* not preceded by the same character as in its rightmost occurrence.
- In this case, we can shift the pattern by the distance *d*2 between such a second rightmost occurrence of *suff (k)* and its rightmost occurrence.
- For example, for the pattern ABCBAB, these distances for k = 1 and 2 will be 2 and 4, respectively

k	**pattern**	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CBA$\underline{\text{B}}$	4

The Boyer-Moore algorithm

- **Step 1**
  - For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.
- **Step 2**
  - Using the pattern, construct the good-suffix shift table as described earlier.
- **Step 3**
  - Align the pattern against the beginning of the text.
- **Step 4**
  - Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text.
  - Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully.
  - In the latter case, retrieve the entry $t1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character.
  - If $k > 0$, also retrieve the corresponding $d2$ entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

The Boyer-Moore algorithm

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

BOYER MOORE

- Example-2:

Text: **STOU_TOR_STOR**

Pattern: STOR

Good suffix table for STOR

k	Pattern	d2
1	STOR	4
2	STOR	4
3	STOR	4

Bad Shift table for STOR , O=1, T=2, S=3, others=pattern-len=4

S	T	O	U	_	T	O	R	_	S	T	O	R
S	**T**	**O**	**R**									
d1=t(U)-k=4-0=4				**S**	**T**	**O**	**R**					
d1=t(_)-k=4-3=1 d2=4								**S**	**T**	**O**	**R**	
d1=t(O)-0=1-0=1									**S**	**T**	**O**	**R**

The Boyer-Moore algorithm

- When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear i.e., O ((n - m + 1) + |∑|)) where ∑ - set of characters used in a pattern (often called the alphabet ∑of a pattern).
- Though this algorithm runs very fast, especially on large alphabets (relative to the length of the pattern), many people prefer its simplified versions, such as Horspool's algorithm, when dealing with natural-language–like strings.

Rabin-Karp Algorithm

- This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number.
- For expository purposes, let us assume that ∑={a=1, b=2,c=3,..j=10}, so that each character is a decimal digit.
- We can then view a string of k consecutive characters as representing a length-k decimal number.
- The character string **cadae** thus corresponds to the decimal number 31415.
  - Check the substring in the text of length 5 characters in the text and verify hash.
  - If the hash is same, then we compare and verify the pattern is matching.

Rabin-Karp Algorithm: Examples

Rabin-Karp Algorithm

- Given a pattern P[1…m], let p denote its corresponding decimal value.
- In a similar manner, given a text T[1…n], let t_s denote the decimal value of the length-m substring T[S+1….S+m], for s = 0, 1,,,, n - m.
- Certainly, t_s = p if and only if T[S+1….S+m]= P[1…m]; thus, s is a valid shift if and only if t_s = p.
- If we could compute p in time O(m) and all the t_s values in a total of O(n-m+1) time, then we could determine all valid shifts s in time O(m)+O(n-m+1)=O(n), by comparing p with each of the t_s values.

Rabin-Karp Algorithm

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m−1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                          // preprocessing
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n − m                      // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n − m
14          t_{s+1} = (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

Rabin-Karp Algorithm

- RABIN-KARP-MATCHER takes O(m) preprocessing time, and its matching time is ,O((n - m + 1)m) in the worst case.
- In many applications, we expect few valid shifts—perhaps some constant c of them. In such applications, the expected matching time of the algorithm is only O((n - m + 1)+cm)= O(n + m), plus the time required to process spurious hits.
- We can then expect that the number of spurious hits is O(n/q), since we can estimate the chance that an arbitrary t_s will be equivalent to p, modulo q, as 1/q, where q is chosen randomly from integers of the appropriate size.
- Since there are O(n) positions at which the test of line 10 fails and we spend O(m) time for each hit, the expected matching time taken by the Rabin-Karp algorithm is O(n)+O(m(v+n/q)), where v is the number of valid shifts.
- This running time is O(n) if v=O(1) and we choose $q \geq m$. That is, if the expected number of valid shifts is small (O(1)) and we choose the prime q to be larger than the length of the pattern, then we can expect the Rabin-Karp procedure to use only O(n+m) matching time. Since $m \leq n$, this expected matching time is O(n).

Complexity

Algorithm	Preprocessing Time	Matching Time
Naive	O	(O (n - m + 1)m)
Rabin-Karp	O(m)	(O (n - m + 1)m)
Finite Automata	O(m|∑|)	O (n)
Knuth-Morris-Pratt	O(m)	O (n)
Boyer-Moore	O(|∑|)	(O ((n - m + 1) + |∑|))

∑ - set of characters used in a pattern (often called the alphabet ∑of a pattern).