SPACE AND TIME TRADE OFFS

“

Things which matter most must never be at the mercy of things which matter less.

~ Johann Wolfgang van Goethe

WHAT IS SPACE – TIME TRADE OFF?

- The best algorithm to solve any problem is one which takes less amount of time to complete its execution and also requires less space in memory.

- This is an ideal case and hence not always possible to achieve in practice.

WHAT IS SPACE – TIME TRADE OFF?

- If time is at a premium, it is practical to choose an algorithm which takes less execution time but consumes a lot of space.
- Conversely, if space is a constraint, it is advisable to choose an algorithm which takes a long time to execute but consumes very little space.

WHAT IS SPACE – TIME TRADE OFF?

- Based on the situation, we may have to sacrifice one at the cost of another.

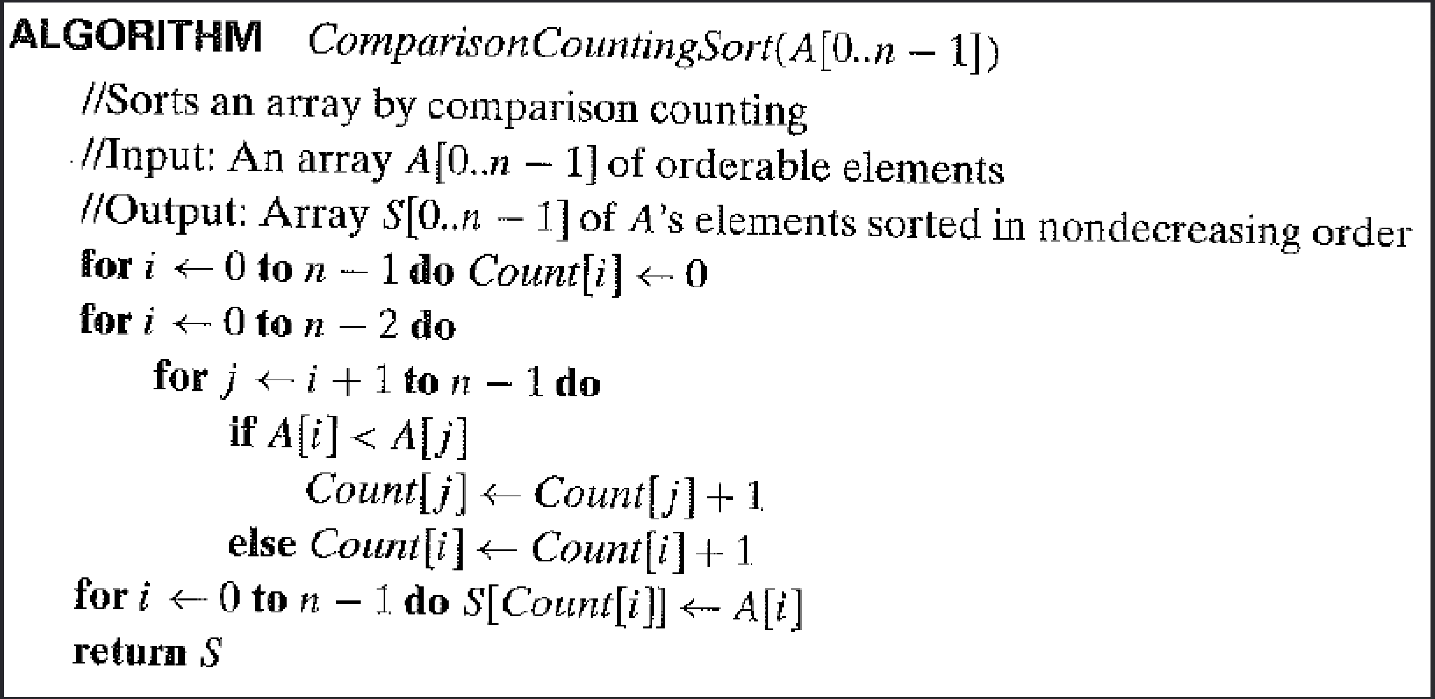
- This is a case of *SPACE – TIME TRADE OFF*.

SORTING BY COUNTING

THE IDEA

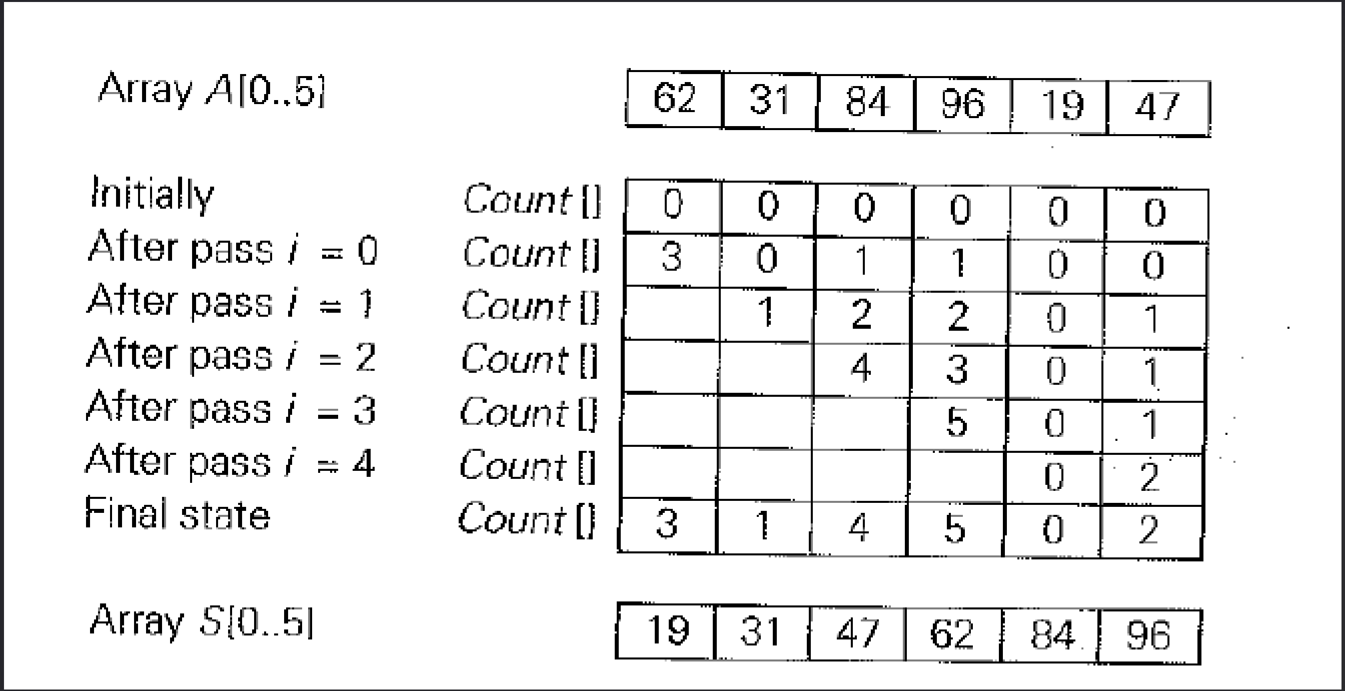
- Example of applying *Input Enhancement*.
- For each element ‘X’ of a list to be sorted, count the total number of elements in the list less than ‘X’.
- Record these results in a table.
- These numbers will indicate the positions of the elements in the sorted list.

THE ALGORITHM



```
ALGORITHM  ComparisonCountingSort($A[0..n-1]$)
    //Sorts an array by comparison counting
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $S[0..n-1]$ of $A$'s elements sorted in nondecreasing order
    for $i \leftarrow 0$ to $n-1$ do $Count[i] \leftarrow 0$
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[i] < A[j]$
                $Count[j] \leftarrow Count[j]+1$
            else $Count[i] \leftarrow Count[i]+1$
    for $i \leftarrow 0$ to $n-1$ do $S[Count[i]] \leftarrow A[i]$
    return $S$
```

AN EXAMPLE



Array *A*[0..5]		62	31	84	96	19	47
Initially	*Count* []	0	0	0	0	0	0
After pass *i* = 0	*Count* []	3	0	1	1	0	0
After pass *i* = 1	*Count* []		1	2	2	0	1
After pass *i* = 2	*Count* []			4	3	0	1
After pass *i* = 3	*Count* []				5	0	1
After pass *i* = 4	*Count* []					0	2
Final state	*Count* []	3	1	4	5	0	2
Array *S*[0..5]		19	31	47	62	84	96

TIME EFFICIENCY

- Basic Operation:
  - The comparison A[i] < A[j]

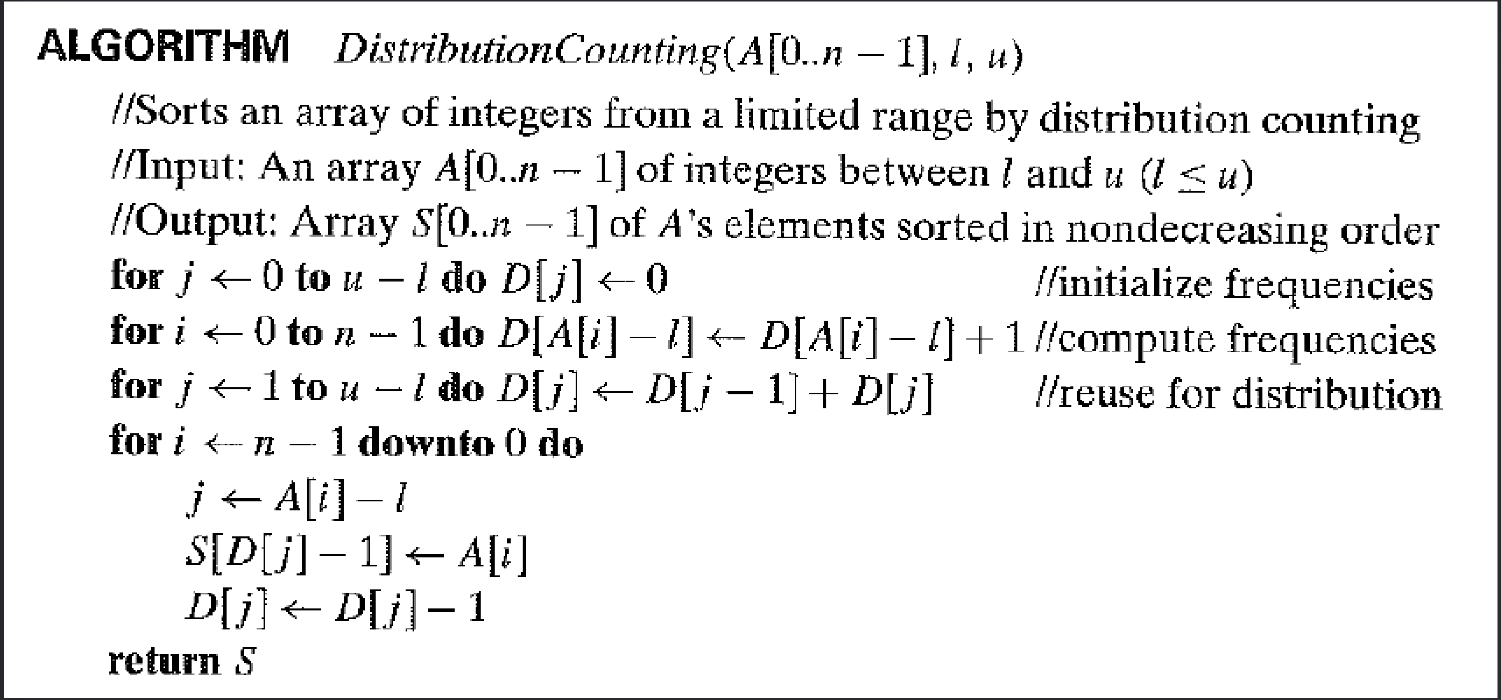
$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

VARIATION - DISTRIBUTION COUNTING

- If the list to be sorted has element values, which are integers between some lower bound *l* and upper bound *u*, we can compute the frequency of each of these elements and store them in an array F[0…u-l].

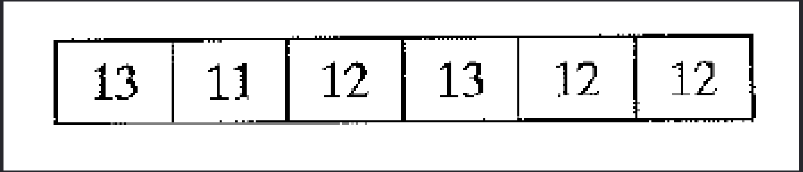
- Then, the first F[0] positions in the sorted list will contain l, the next F[1] positions will contain l + 1 and so on.

THE ALGORITHM

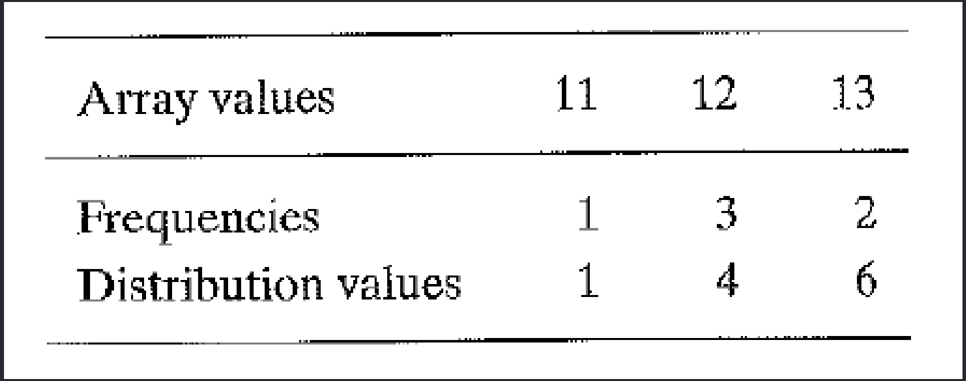


```
ALGORITHM  DistributionCounting($A[0..n-1], l, u$)
    //Sorts an array of integers from a limited range by distribution counting
    //Input: An array $A[0..n-1]$ of integers between $l$ and $u$ ($l \le u$)
    //Output: Array $S[0..n-1]$ of $A$'s elements sorted in nondecreasing order
    for $j \leftarrow 0$ to $u - l$ do $D[j] \leftarrow 0$                    //initialize frequencies
    for $i \leftarrow 0$ to $n - 1$ do $D[A[i] - l] \leftarrow D[A[i] - l] + 1$ //compute frequencies
    for $j \leftarrow 1$ to $u - l$ do $D[j] \leftarrow D[j-1] + D[j]$          //reuse for distribution
    for $i \leftarrow n - 1$ downto 0 do
        $j \leftarrow A[i] - l$
        $S[D[j] - 1] \leftarrow A[i]$
        $D[j] \leftarrow D[j] - 1$
    return $S$
```

AN EXAMPLE

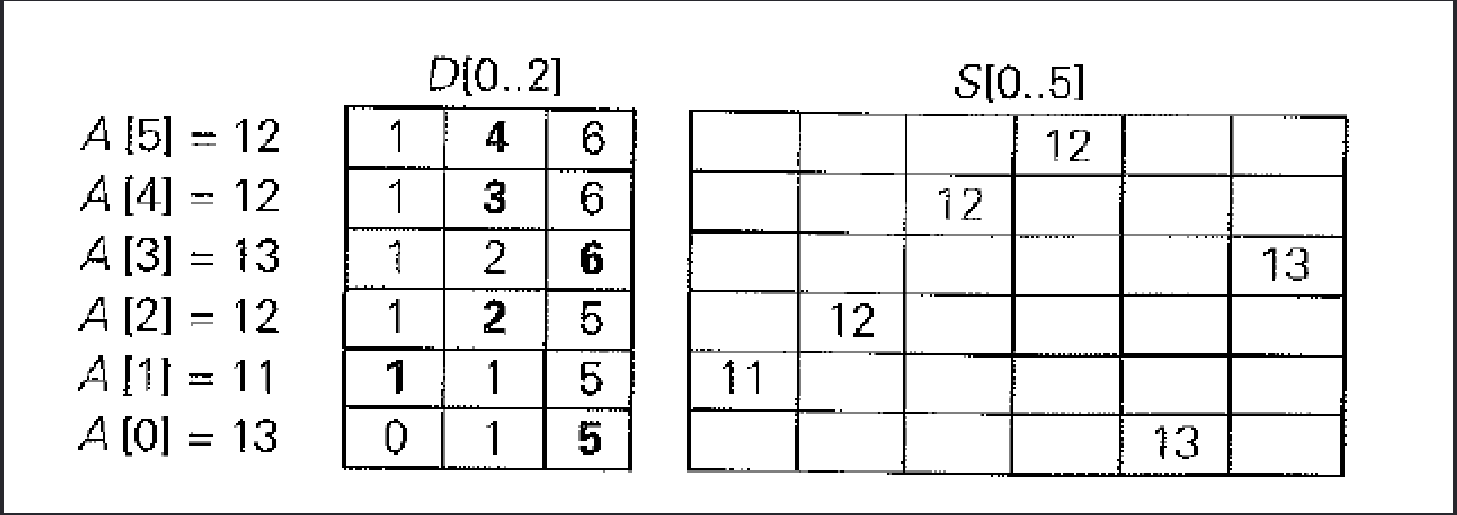


13	11	12	13	12	12



Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

AN EXAMPLE



	D[0..2]			*S*[0..5]					
A [5] = 12	1	**4**	6				12		
A [4] = 12	1	**3**	6			12			
A [3] = 13	1	2	**6**						13
A [2] = 12	1	**2**	5		12				
A [1] = 11	**1**	1	5	11					
A [0] = 13	0	1	**5**					13	

INPUT ENHANCEMENT IN STRING MATCHING

STRING MATCHING PROBLEM

Finding an occurrence of a given string of m characters called the pattern in a longer string of n characters called the text.

INPUT ENHANCEMENT IN STRING MATCHING PROBLEM

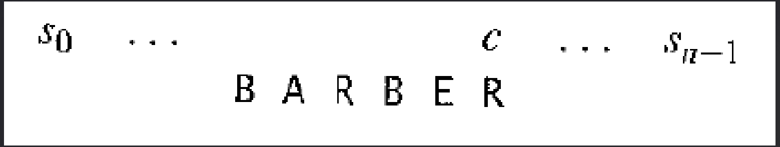
- Pre - process the pattern to get information about it.
- Store the information in a table.
- Use this information during search.

Examples:

1. Horspool’s Algorithm
2. Boyer – Moore Algorithm
3. And more

HORSPOOL'S ALGORITHM

- Consider an example of searching the pattern "BARBER" in some text.

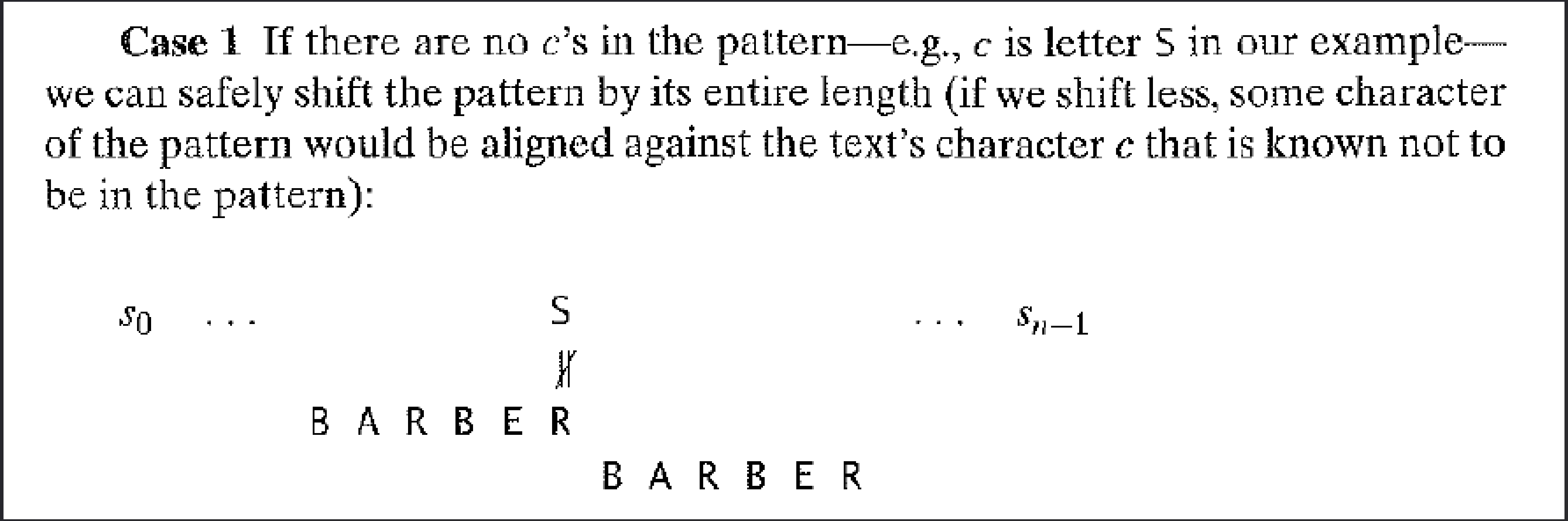


- Start with the last 'R' of the pattern, move right to left and compare corresponding characters.
- If all characters match, then a pattern is found.

HORSPOOL’S ALGORITHM

- If a mismatch is found, we should move the pattern to the right.
- And, we would like to make as large a shift as possible.
- Horspool’s algorithm determines the size of the shift by looking at the last character ‘c’ of the text which is matched against the last character of the pattern.
- In general, the following four possibilities can occur.

HORSPOOL’S ALGORITHM – CASE 1

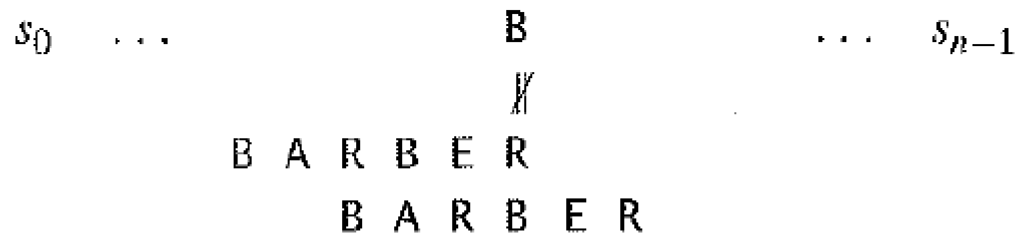


Case 1 If there are no *c*’s in the pattern—e.g., *c* is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text’s character *c* that is known not to be in the pattern):

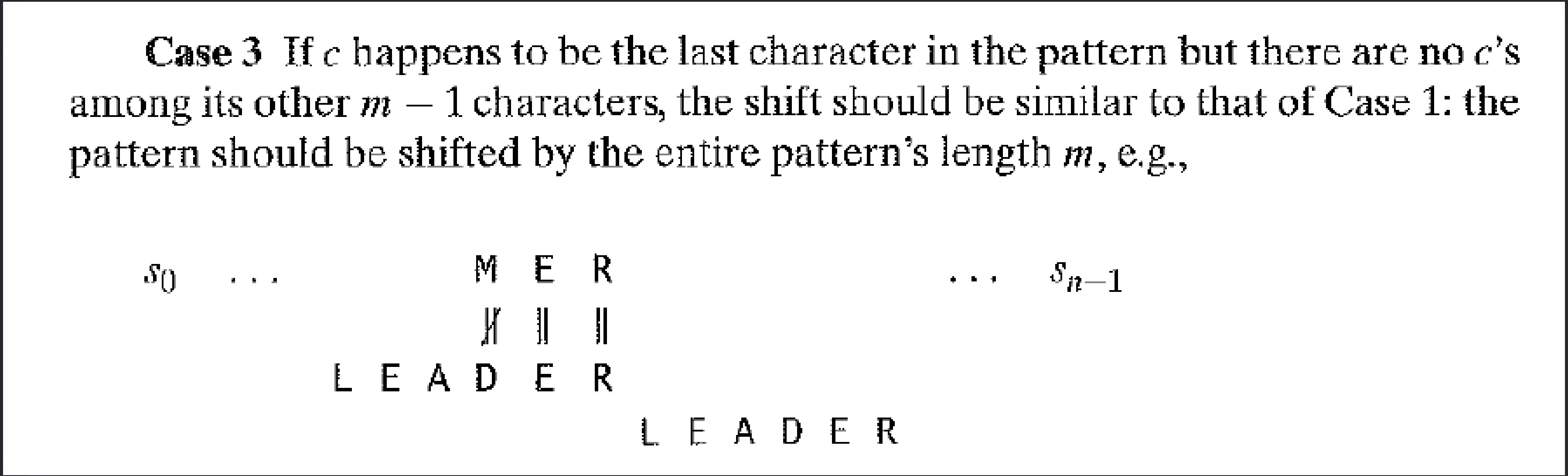
```
s0    . . .             S                  . . .   s_{n-1}
                        ≠
          B  A  R  B  E  R
                         B  A  R  B  E  R
```

HORSPOOL’S ALGORITHM – CASE 2

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:



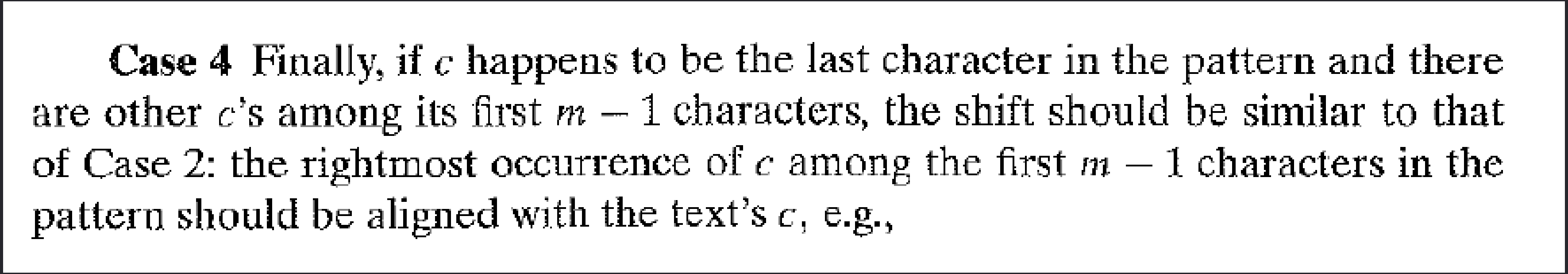
HORSPOOL'S ALGORITHM – CASE 3



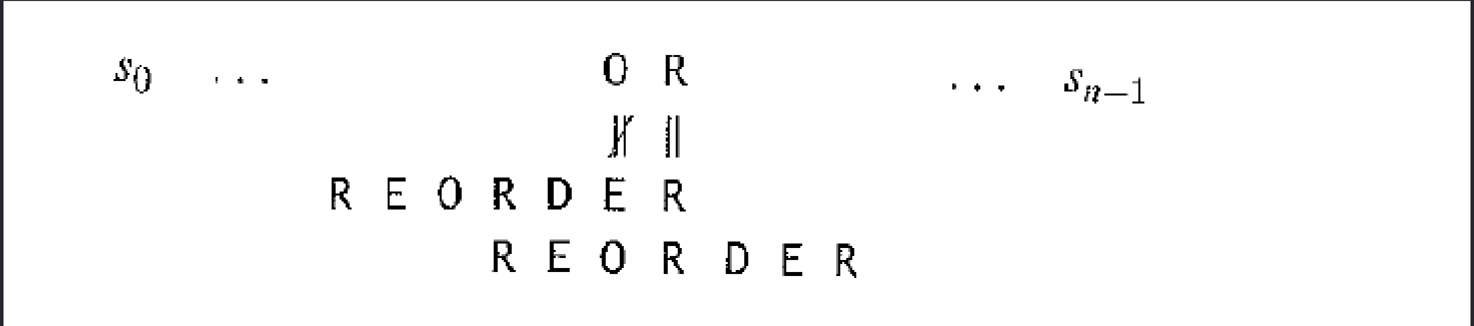
Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m-1$ characters, the shift should be similar to that of Case 1: the pattern should be shifted by the entire pattern's length m, e.g.,

$$\begin{array}{cccccccccccccccc}
s_0 & \ldots & & & M & E & R & & & & & & \ldots & s_{n-1} \\
 & & & & \not\parallel & \parallel & \parallel & & & & & & & \\
 & & L & E & A & D & E & R & & & & & & \\
 & & & & & & & L & E & A & D & E & R &
\end{array}$$

HORSPOOL'S ALGORITHM – CASE 4



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, the shift should be similar to that of Case 2: the rightmost occurrence of c among the first $m-1$ characters in the pattern should be aligned with the text's c, e.g.,

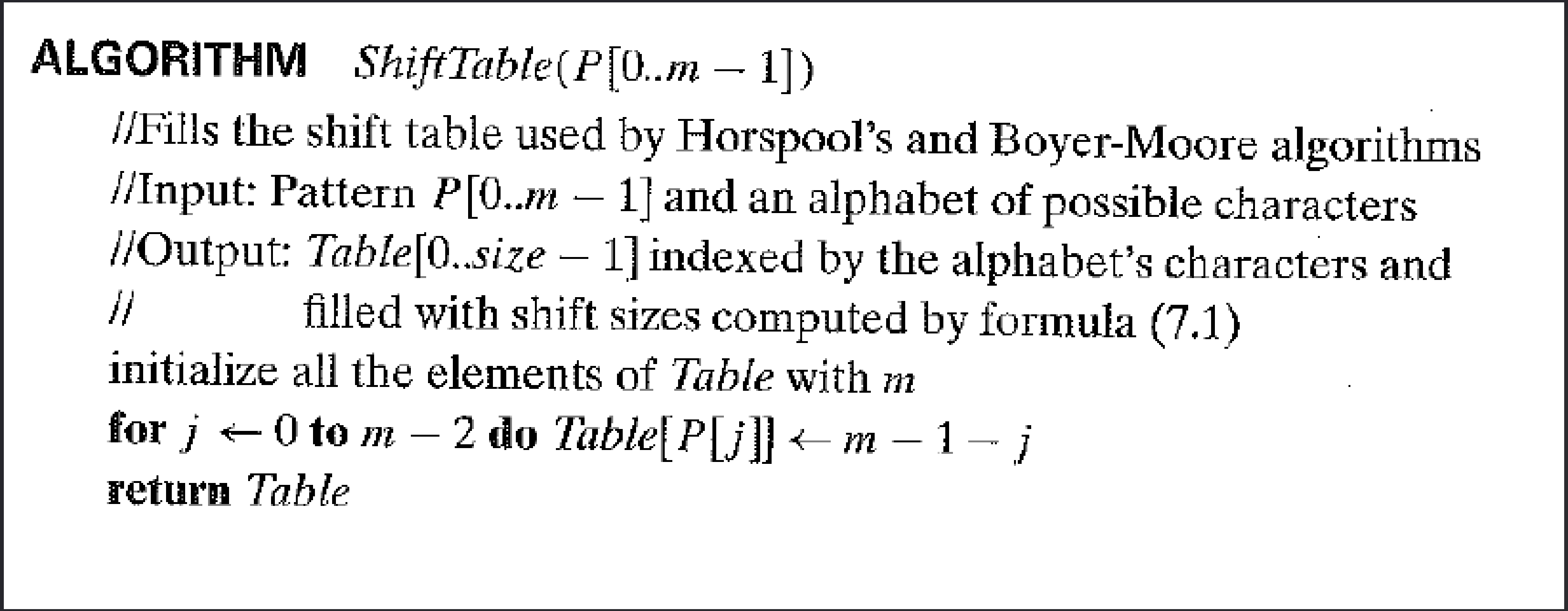


HORSPOOL’S ALGORITHM

- The shift sizes are pre – computed and stored in a table.
- The table will be indexed by all possible characters that can be encountered in a text.
- The table’s entries will indicate shift sizes calculated by:

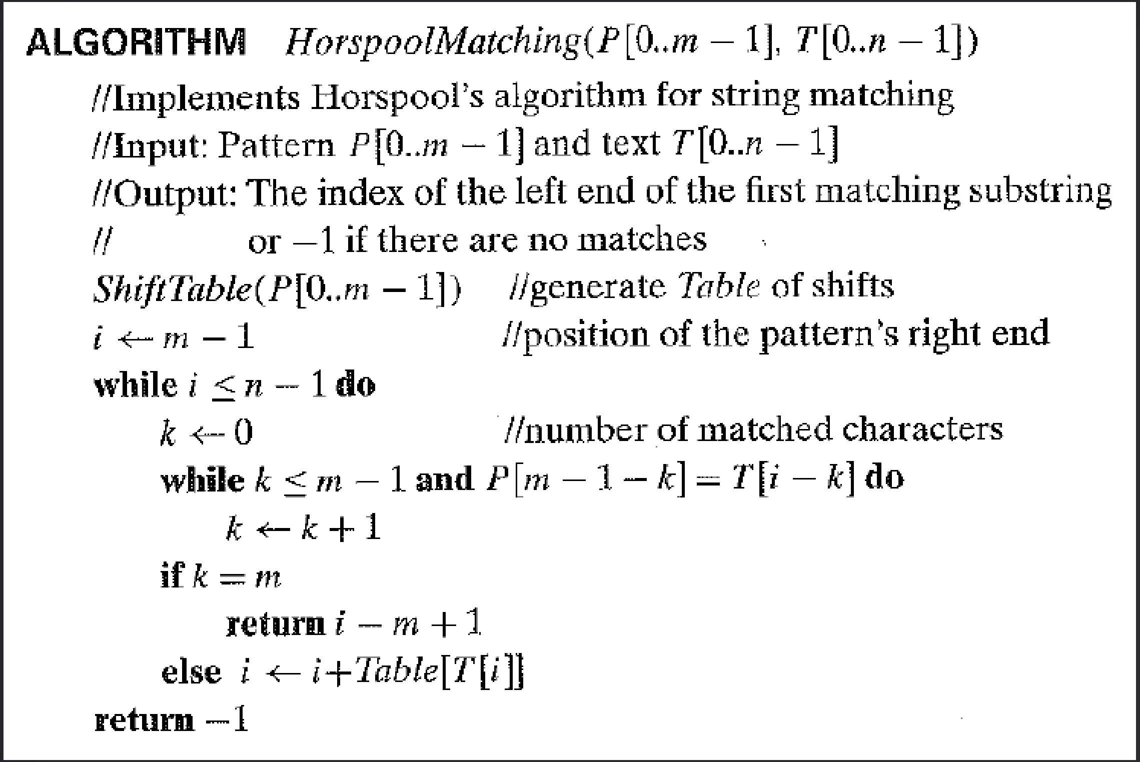
$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise} \end{cases} \qquad \textbf{(7.1)}$$

HORSPOOL'S ALGORITHM – SHIFT TABLE GENERATION



```
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 (7.1)
    initialize all the elements of $Table$ with $m$
    for $j \leftarrow 0$ to $m-2$ do $Table[P[j]] \leftarrow m-1-j$
    return $Table$
```

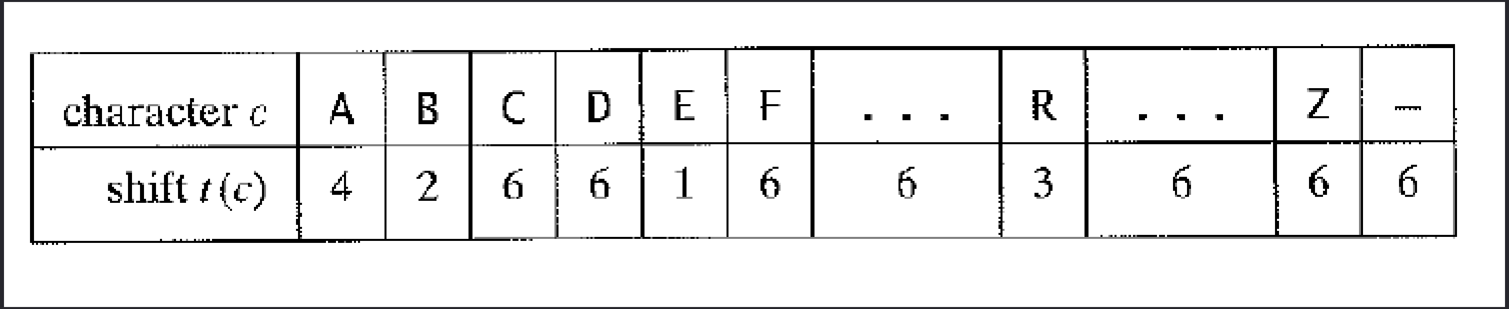
HORSPOOL'S ALGORITHM – MATCHING



```
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
    $i \leftarrow m-1$              //position of the pattern's right end
    while $i \leq n-1$ do
        $k \leftarrow 0$              //number of matched characters
        while $k \leq m-1$ and $P[m-1-k] = T[i-k]$ do
            $k \leftarrow k+1$
        if $k = m$
            return $i-m+1$
        else $i \leftarrow i + Table[T[i]]$
    return $-1$
```

HORSPOOL'S ALGORITHM – EXAMPLE

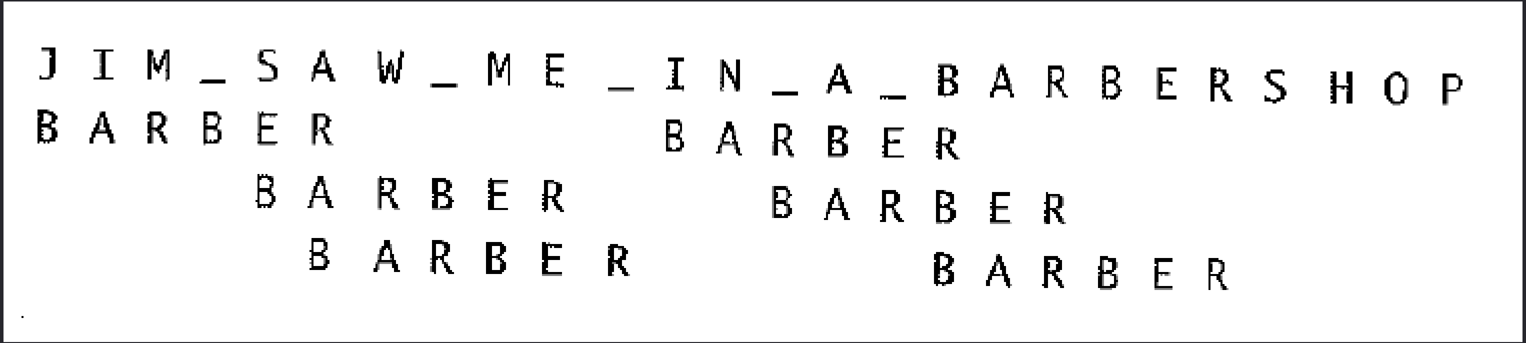
Pattern = BARBER



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

HORSPOOL'S ALGORITHM – EXAMPLE

Pattern = BARBER



```
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 - EFFICIENCY

- Worst – Case Efficiency: $\Theta(nm)$
- For random texts: $\Theta(n)$

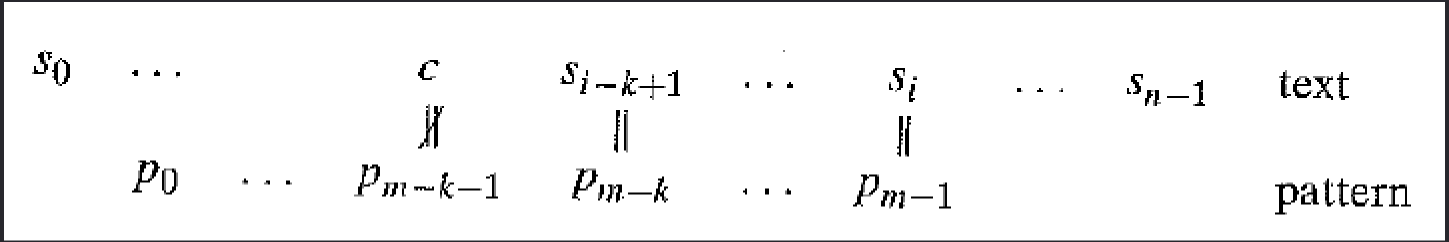
BOYER – MOORE ALGORITHM

- The Boyer – Moore algorithm is similar to the Horspool's algorithm.

- If the first comparison of the rightmost character in the pattern with the corresponding character c in the text fails, it shifts the pattern to the right by the number of characters retrieved from the precomputed table.

BOYER – MOORE ALGORITHM

- The Boyer – Moore algorithm acts differently when compared to the Horspool's algorithm if some positive number k (0 < k < m) of the pattern's characters match the text before a mismatch is encountered.



s_0 ... c s_{i-k+1} ... s_i ... s_{n-1} text
$\not\parallel$ $\parallel$ $\parallel$
p_0 ... p_{m-k-1} p_{m-k} ... p_{m-1} pattern

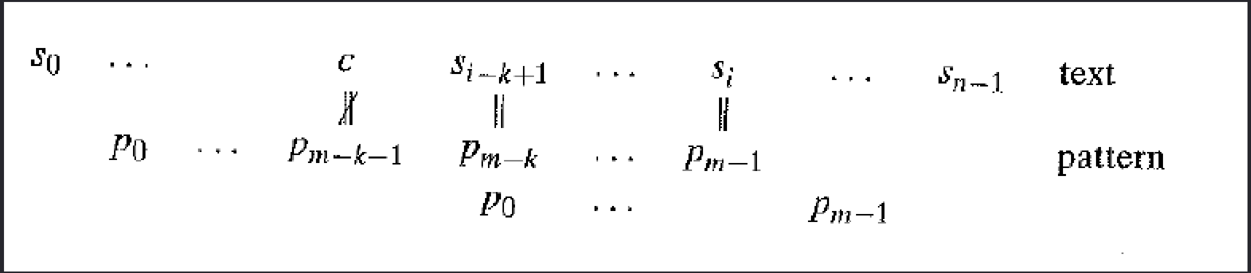
BOYER – MOORE ALGORITHM

- In such a situation, the shift size is calculated by considereing two quantities.
  - The text's character c that caused the mismatch with its counterpart in the pattern. – BAD SYMBOL SHIFT.
  - If there is a successful match of the last k > 0 characters of the pattern. The ending portion of the pattern is referred to as its suffix. – GOOD SUFFIX SHIFT.

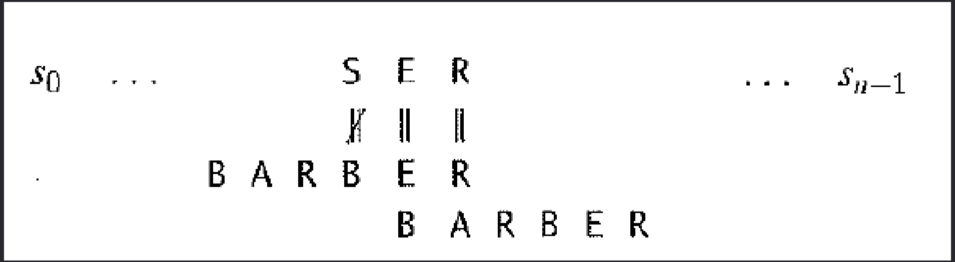
BAD SYMBOL SHIFT

- If c is not in the pattern, we shift the pattern to just pass this character in the text.
- The size of this shift is computed by the formula: $t_1(c) - k$
- $t_1(c)$ - Entry in the precomputed table
- k – Number of matched characters

BAD SYMBOL SHIFT



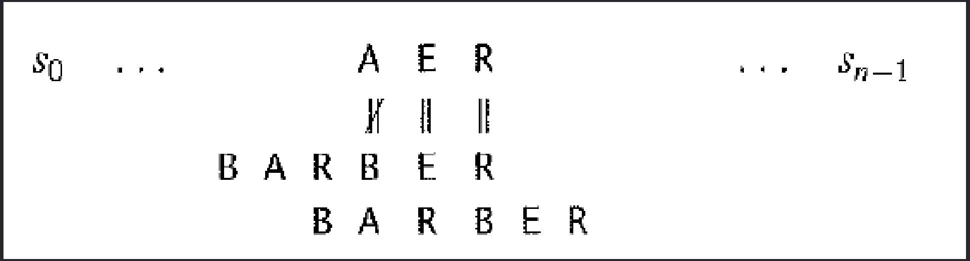
s_0 … c s_{i-k+1} … s_i … s_{n-1} text
p_0 … p_{m-k-1} p_{m-k} … p_{m-1} pattern
p_0 … p_{m-1}



- $t_1(S) = 6 - 2 = 4$

BAD SYMBOL SHIFT

- The same formula can be applied when the mismatching character c of the text occurs in the pattern if t1(c) – k > 0.
- $t_1(A) - k = 4 - 2 = 2$



BAD SYMBOL SHIFT - SUMMARY

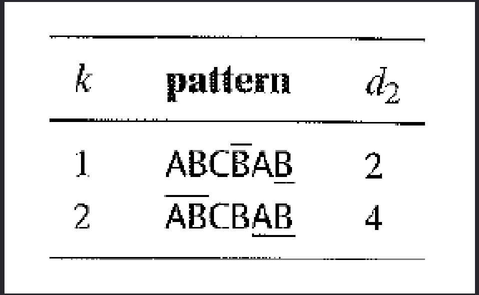
- The bad symbol shift d1 is computed by the Boyer Moore algorithm as follows:

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

[1 if t1(c) – k <= 0]

GOOD SUFFIX SHIFT

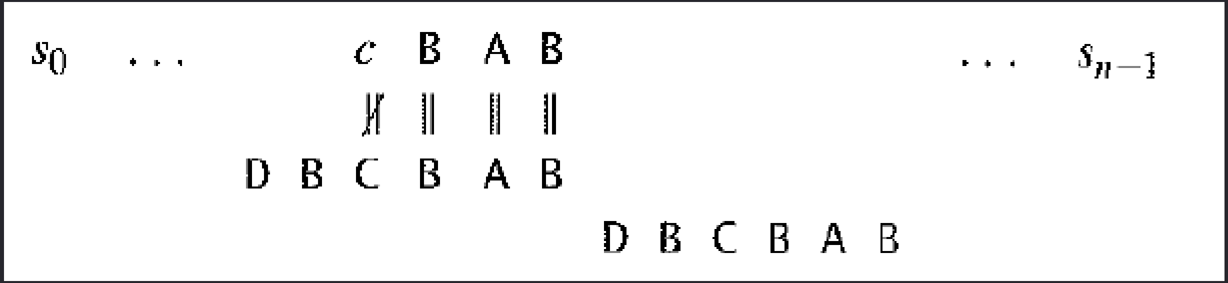
- If there is another occurrence of suff(k) not preceded by the same character as its rightmost occurrence: the pattern is shifted by d_2 between such a second rightmost occurrence of suff(k) and its rightmost occurrence.



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

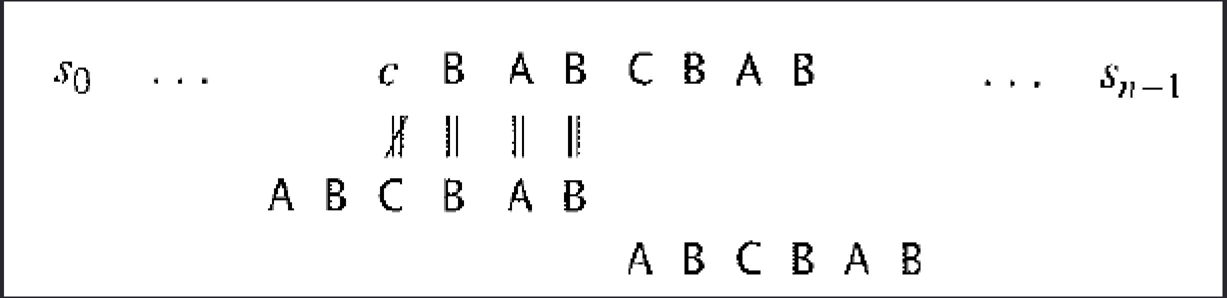
GOOD SUFFIX SHIFT

- If there is no occurrence of suff(k) not preceded by the same character as its rightmost occurrence: the pattern is shifted by its entire length m.



GOOD SUFFIX SHIFT

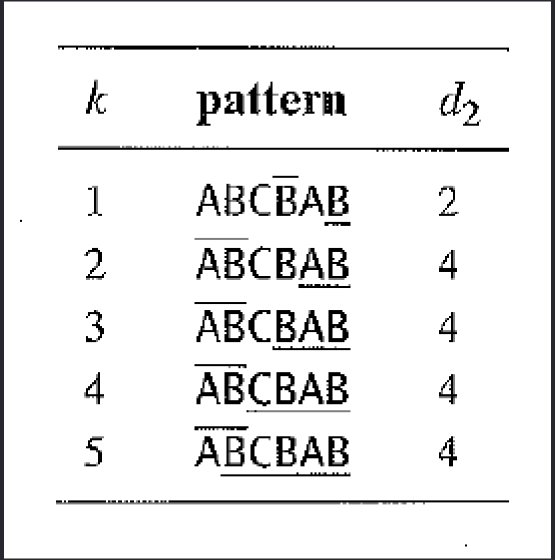
- Shifting by the entire pattern length when there is no other occurrence of suff(k) not preceded by the same character will lead to incorrect solutions.



GOOD SUFFIX SHIFT

- To avoid such an erroneous shift based on a suffix of size *k*, for which there is no other occurrence in the pattern not preceded by the same character as in its last occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l.
- If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern's length *m.*

GOOD SUFFIX SHIFT



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

THE 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.

Step 3 Align the pattern against the beginning of the text.

THE ALGORITHM

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* 2: 0 character pairs are matched successfully.

In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where cis the text's mismatched character.

THE ALGORITHM

If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula:

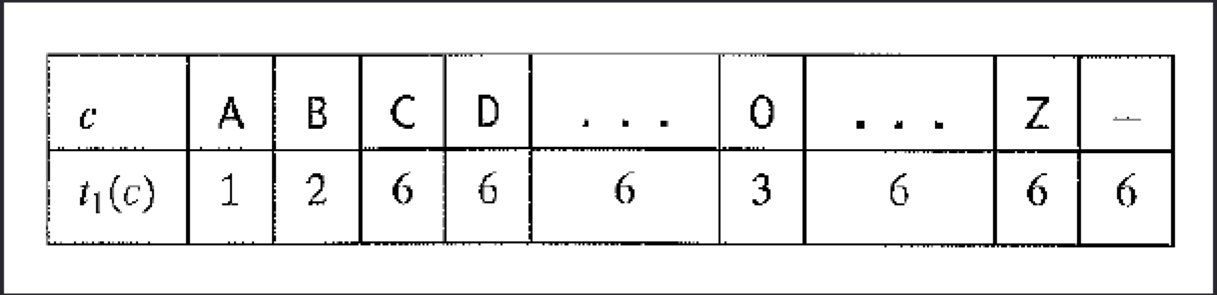
$$d = \begin{cases} d_1 & \text{if } k = 0 \\ \max\{d_1, d_2\} & \text{if } k > 0 \end{cases},$$

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

THE EXAMPLE

Pattern: BAOBAB

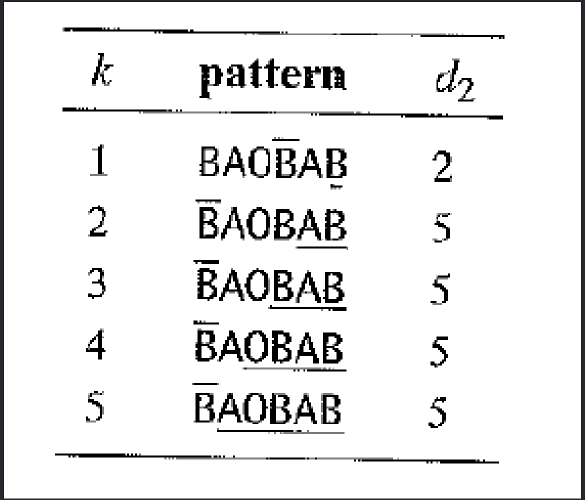
Bad Symbol Shift Table:



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

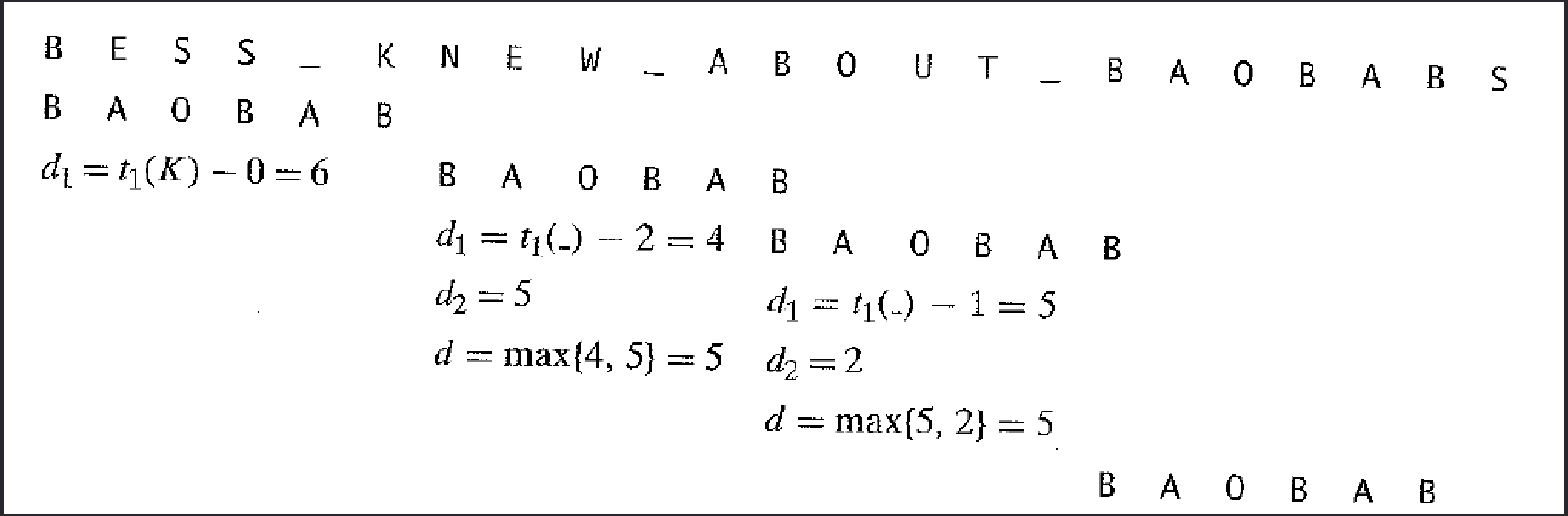
THE EXAMPLE

GOOD SUFFIX TABLE:



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

THE EXAMPLE



B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(K) - 0 = 6$ B A O B A B

$d_1 = t_1(\_) - 2 = 4$ B A O B A B

$d_2 = 5$ $d_1 = t_1(\_) - 1 = 5$

$d = \max\{4, 5\} = 5$ $d_2 = 2$

$d = \max\{5, 2\} = 5$

B A O B A B

THE EFFICIENCY

The worst case efficiency of Boyer Moore algorithm is linear.

#######

HASHING

DICTIONARY

- A dictionary is an abstract data type.
- It is a set with insertion, deletion and searching operations defined on its elements.
- The elements of this set are arbitrary in nature: Numbers, Characters, Strings and so on.
- Practical Uses: Student Records in an educational institution, citizen records in government office, book records in a library.

DICTIONARY

- Records generally have many fields.
- Each field represents information about the entity the record represents.
- Ex: In a student record, we might have ID, Name, Semester, Age and so on.
- But in a record, there will be one field, the key, which uniquely identifies it.

HASHING

- Hashing is a very efficient way of implementing dictionaries.
- The idea is to distribute the keys among a one – dimensional array H[0..m-1] called the HASH TABLE.
- The distribution is done by computing, for each of the keys, the value of some predefined function called the HASH FUNCTION.
- This function assigns an integer between 0 and m-1 called the HASH ADDRESS.

HASHING

- Hashing is a very efficient way of implementing dictionaries.
- The idea is to distribute the keys among a one – dimensional array H[0..m-1] called the HASH TABLE.
- The distribution is done by computing, for each of the keys, the value of some predefined function called the HASH FUNCTION.
- This function assigns an integer between 0 and m-1 called the HASH ADDRESS.

HASHING - EXAMPLE

- If keys are non – negative integers, then a hash function is generally of the form, h(K) = K mod m.

- If keys are letters of some alphabet, we assign a position to the character in the alphabet, denoted by ord(K) and then apply the function we applied on integers.

HASHING - EXAMPLE

- If the key is a character string $c_0c_1 \ldots c_{s-1}$, we use a function of the form Σ (ord(c_i)), i goes from 0 to s-1.
- Another option is:

$$h \leftarrow 0;\ \textbf{for}\ i \leftarrow 0\ \textbf{to}\ s-1\ \textbf{do}\ h \leftarrow (h * C + ord(c_i)) \bmod m,$$

- C is a larger constant than every ord(c_i).

HASH FUNCTIONS – PROPERTIES

- A hash table's size should not be excessively large compared to the number of keys.
- A hash function needs to evenly distribute the keys among the cells of the hash table.
- A hash function should be easy to compute.

COLLISIONS

- In Hashing, there will arise situations when two or more keys are hashed to the same cell. This phenomenon is called collision.
- So, every hashing scheme should have a collision resolution mechanism.

OPEN HASHING

- Also called: Separate Chaining.
- In this mechanism, keys are stored in linked lists attached to cells of a hash table.
- Each list contains all keys hashed to the cell.

OPEN HASHING - EXAMPLE

Keys:

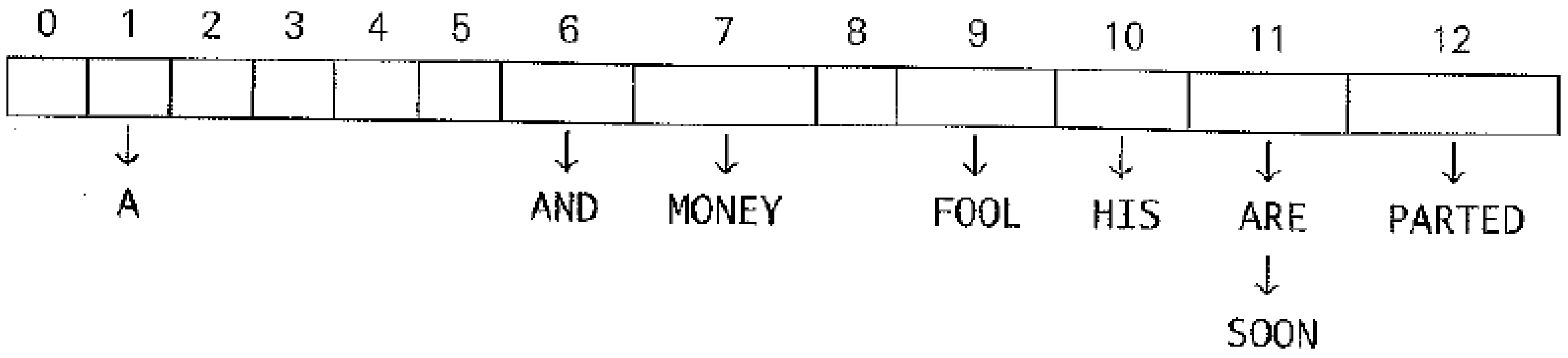
A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

Hash Function:

Add positions of a word’s letters in the alphabet and compute the sum’s remainder by 13.

OPEN HASHING – EXAMPLE

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12



OPEN HASHING - EXAMPLE

Load Factor:

If the hash function distributes n keys among m cells of the hash table evenly, each list will be n/m keys long.

Ratio $\alpha = n / m$ is called the Load Factor of the hash table.

OPEN HASHING - EXAMPLE

The average number of pointers inspected in successful searches S and unsuccessful searches U turns out to be:

$S = 1 + \alpha / 2$

$U = \alpha$

OPEN HASHING - EXAMPLE

Insertion: Done at the end of a list.

Deletion: Search for the key to be deleted and remove it.

Both operations have efficiency of $\Theta(1)$.

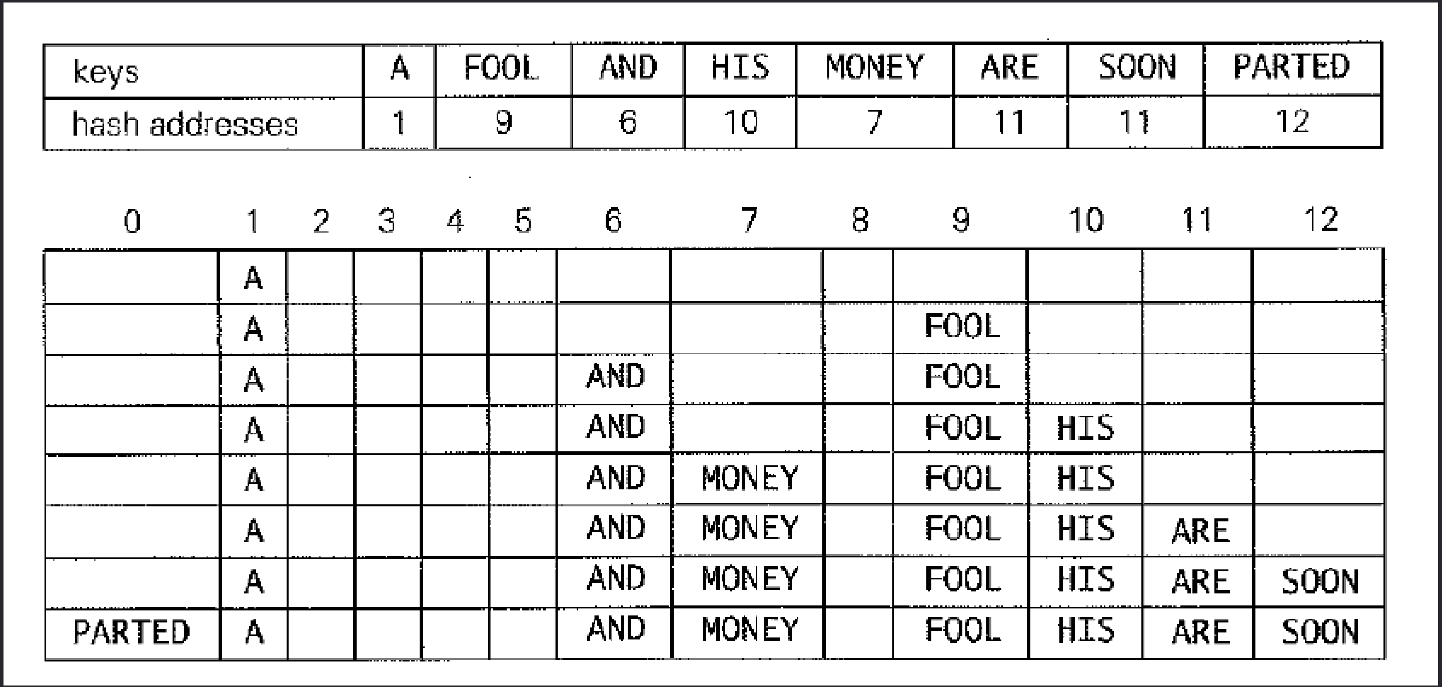
CLOSED HASHING

- All keys are stored in the hash table itself without the use of linked lists.
- Different strategies can be employed for collision resolution.
- The simplest one is Linear Probing.

CLOSED HASHING

- If a collision occurs, check the cell following the one where collision occurs.
- If it is empty, insert the element there.
- If it is already occupied, check the cell following this one and so on.
- If the end of the table is reached, the search is wrapped to the beginning of the table.

CLOSED HASHING - EXAMPLE



keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

CLOSED HASHING

- Search and Insertion operations are straightforward.
- If we delete a key from the hash table, we will be unable to find other keys which are hashed to this cell but stored elsewhere because of Linear Probing.
- Solution: Lazy Deletion – Mark previously occupied locations by a special symbol to distinguish them from locations that have not been occupied.

CLOSED HASHING

- Number of successful searches (S) and number of unsuccessful searches (U):

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \text{ and } U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

CLOSED HASHING

- Linear Probing leads to a problem called Clustering.
- A cluster in linear probing is a sequence of contiguously occupied cells.
- Clusters make dictionary operations less efficient.
- As clusters increase, the probability that a new element will be added to the cluster increases.

DOUBLE HASHING

- In Double Hashing, we use another hash function s(K) to determine a fixed increment for the probing sequence to be used after a collision at location l = h (k):

$$(l + s(K)) \bmod m, \;\; (l + 2s(K)) \bmod m, \; \ldots \; .$$

- To guarantee that every location in the table is probed by the above sequence, the increment s(K) and table size m must be relatively prime.

DOUBLE HASHING

- Double hashing will lead to a problem when the table gets close to being full.
- Solution to this is Rehashing: All the current keys are scanned and moved to a new table.