UNIT 5. SPACE AND TIME TRADEOFFS

The basic idea behind space and time tradeoffs is to preprocess the input of the problem and store the additional information obtained. This helps in solving the problem. This approach is known as ***input enhancement***. We will discuss following algorithms based on input enhancement:

- Counting methods for sorting
- Boyer-Moore Algorithm for string matching
- Horspool Algorithm for string matching

Another technique that uses space and time tradeoffs suggests using extra space to facilitate faster and/or more flexible access to the data. This approach is known as ***prestructuring.*** This indicates that, some processing is done before a problem in question is actually solved, but unlike input enhancement, it deals with access structuring. We will illustrate this approach by

- Hashing

Sorting by Counting

In this method is the application of input enhancement. Here, we will count the number of elements smaller than each element. This count is stored in a table and it will indicate the position of that element in the sorted list. This algorithm is known as ***comparison counting.***

```
ALGORITHM ComparisonCounting(A[0…n-1])
//Sorts an array by comparison counting
//Input: Array A[0…n-1]
//Output: Array S[0…n-1] of A's elements in a sorted order

for i ← 0 to n-1 do
    Count[i] ← 0

for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] < A[j]
            Count[j] ← Count[j] +1
        else
            Count[i] ← Count[i] +1

for i ← 0 to n-1 do
    S[Count[i]] ← A[i]

return S
```

Analysis:

The basic operation is comparison. Thus, the time complexity can be given as –

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

$$= \frac{n(n-1)}{2}$$

Thus,

$$C(n) \in \theta(n^2)$$

Input Enhancement in String Matching

String matching problem requires finding an occurrence of a given pattern of *m* characters in a given text of *n* characters. We have seen Brute force technique for solving this problem. Here we will study Boyer-Moore algorithm and its simplified version, Horspool algorithm for string matching.

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

```
s0, .................... c......................... sn-1
             BARBAR
```

Starting with last character *R* of the pattern, we have to keep on comparing each pair of characters in pattern and text. If all the characters match, then algorithm halts. If any mismatch is found, we need to shift our pattern towards right. The number of characters to be shifted depends on various situations as discussed here under:

Case 1. If there are no *c*'s in the pattern, shift pattern by its entire length. For example,

```
s0, - - - - - - - - - - - - S - - - - - - - - - - - - - - - - - - - - - - - sn-1
                            ≠ |
             BARBAR          ↓
                              BARBAR
```

Here, S ≠ R and also S is not present in the pattern. So, shift entire pattern.

Case 2. If there are occurrences of character *c* in pattern, but it is not last character of the pattern, then shift should align the rightmost occurrence of *c* in the pattern with the *c* in text. For example,

```
s0, - - - - - - - - - - - - B - - - - - - - - - - - - - - - - - - - - - - - sn-1
                            ≠
             BARBAR
                BARBAR
```

Case 3. If *c* is the last character in the pattern, but there are no *c*'s among other m-1 characters, then there will be entire pattern shift. For example,

```
s0, - - - - - - - - - - M A R  - - - - - - - - - - - - - - - - - - - - - - - sn-1
                        ≠  =  = |
                BAR B A  R      ↓
                                BARBAR
```

Case 4. If *c* is the last character in pattern, and also there are some other *c*'s in the pattern, then shift will be same as in case2. For example,

```
s0, - - - - - - - - - -    P R  - - - - - - - - - - - - - - - - - - - - - - - sn-1
                           ≠  =
               B A R B A R
                    B A R B A R
```

In Horspool and Boyer-Moore algorithm, we have to pre-compute the shift sizes and store them in a table. The shift table will be indexed by all possible characters that can be encountered in text. The table entries will be filled with shift sizes.

Specifically, for every character *c* in text, we compute the shift's value by the formula –

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

Following the algorithm for calculating shift-table values.

```
ALGORITHM ShiftTable(P[0…m-1])
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0…n-1] and an alphabet of possible characters
//Output: Table[0…size-1] indexed by the alphabet's characters and filled with
// shift sizes.

Initialize all the elements of Table with m

for  j ← 0 to m-2 do
        Table[P[j]] ← m-1- j

return Table
```

Now, the algorithm for Horspool technique can be summarized as below –

Step 1. For a given pattern of length *m* and the alphabet used in both the pattern and text, construct the shift table.
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 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.

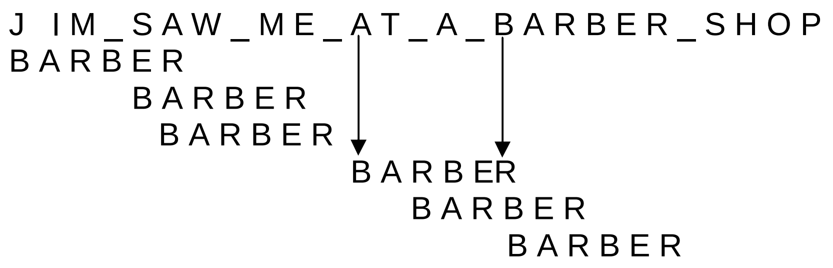
The pseudocode can be given as –

```
ALGORITHM Horspool(P[0…m-1], T[0…n-1])
//Implement Horspool's algorithm for string matching
//Input: Pattern P[0…m-1] and text T[0…n-1]
//Output: The position of first matching, if successful, otherwise, -1

ShiftTable(P[0…m-1])

i←m-1
while i ≤ n-1 do
    k ← 0
    while k ≤ m-1 and P[m-1-k] ==T[i-k]
        k ← k+1
    if k==m
        return i-m+1
    else
        i ← i+ Table[T[i]]

return -1
```

Example:
Let us consider an example of search a pattern BARBER in the text –
JIM_SAW_ME_AT_A_BARBER_SHOP

The text consists of the alphabets A to Z and a character _.
Let us construct a shift-table.

Character *c*	A	B	C	D	E	F	….	R	….	Z	---
Shift *t(c)*	4	2	6	6	1	6	6	3	6	6	6



HASHING

Hashing is a way of representing dictionaries. Dictionary is an abstract data type with a set of operations searching, insertion and deletion defined on its elements. The elements of dictionary can be numeric or characters or most of the times, records.

Usually, a record consists of several fields; each may by different data types. For example, student record may contain student id, name, gender, marks etc. Every record is usually identified by some ***key***. Here we will consider the implementation of a dictionary of *n* records with keys k1, k2 …kn.

Hashing is based on the idea of distributing keys among a one-dimensional array H[0…m-1], called ***hash table***. For each key, a value is computed using a predefined function called ***hash function***. This function assigns an integer, called ***hash address***, between 0 to m-1 to each key. Based on the hash address, the keys will be distributed in a hash table.

For example, if the keys k1, k2, …., kn are integers, then a hash function can be –
h(K) = K mod m.

Let us take keys as 65, 78, 22, 30, 47, 89. And let hash function be , h(k) = k %10.
Then the hash address may be any value from 0 to 9 and hash table may look like –

0	1	2	3	4	5	6	7	8	9

For each key, hash address will be computed as –
h(65) = 65 %10 = 5
h(78) = 78%10 = 8
h(22)= 22 % 10 =2
h(30)= 30 %10 =0
h(47) = 47 %10 = 7
h(89)=89 % 10 = 9

Now, each of these keys can be hashed into a hash table as –

0	1	2	3	4	5	6	7	8	9
30		22			65		47	78	89

In general, a hash function should satisfy the following requirements:

- A hash function needs to distribute keys among the cells of hash table as evenly as possible.
- A hash function has to be easy to compute.

Hash Collisions

Let us have *n* keys and the hash table is of size *m* such that *m<n.* As each key will have an address with any value between 0 to m-1, it is obvious that more than one key will have same hash address. That is, two or more keys need to be hashed into the same cell of hash table. This situation is called as ***hash collision***. In the worst case, all the keys may be hashed into same cell of hash table. But, we can avoid this by choosing size of hash table and hast function properly. Anyway, every hashing scheme must have a mechanism for resolving hash collision.

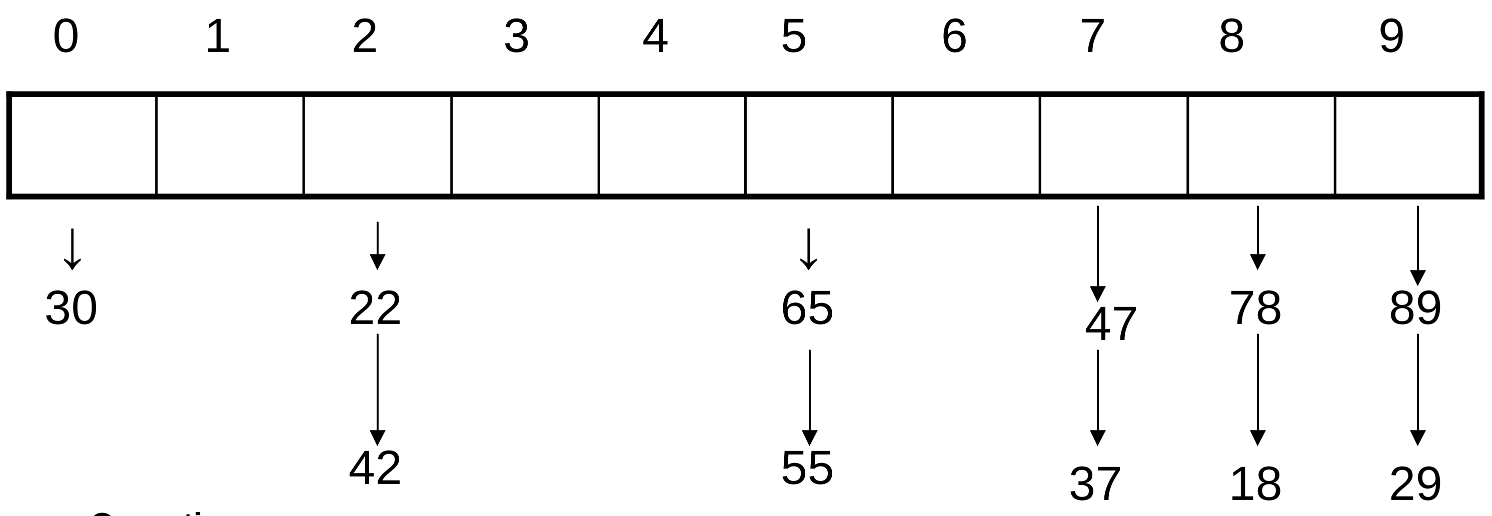
There are two methods for hash collision resolution, viz Open hashing and closed hashing.

Open Hashing (Separate Chaining)

- In open hashing, keys are stored in linked lists attached to cells of a hash table.
- Each list contains all the keys hashed to its cell.
- For example, consider the elements 65, 78, 22, 30, 47, 89, 55, 42, 18, 29, 37.
- If we take the hash function as h(k)= k%10, then the hash addresses will be –

h(65) = 65 %10 = 5	h(78) = 78%10 = 8	h(18)=18%10 =8
h(22)= 22 % 10 =2	h(30)= 30 %10 =0	h(29)=29%10=9
h(47) = 47 %10 = 7	h(89)=89 % 10 = 9	h(37)=37%10 =7
h(55)=55%10 =5	h(42)=42%10 =2	

- Now, the hashing is done as below –



Operations:

- **Searching:** Now, if we want to search for the key element in a hash table, we need to find the hash address of that key using same hash function.
- Using the obtained hash address, we need to search the linked list by tracing it, till either the key is found or list gets exhausted.
- **Insertion:** Insertion of new element to hash table is also done in similar manner.
- Hash key is obtained for new element and is inserted at the end of the list for that particular cell.
- **Deletion:** Deletion of element is done by searching that element and then deleting it from a linked list.

Efficiency:

- If the hash function distributes *n* keys among *m* cells of the hash table about evenly, then each linked list will be about n/m keys long.
- The ratio $\alpha = n/m$ is called as ***load factor***.
- The average number of comparisons done for a successful search, $S \approx 1 + \alpha/2$
- And for unsuccessful search, $U = \alpha$

Closed Hashing (Open Addressing)

In this technique, all keys are stored in the hash table itself without using linked lists. Different methods can be used to resolve hash collisions. The simplest technique is ***linear probing.*** This method suggests to check the next cell from where the collision occurs. If that cell is empty, the key is hashed there. Otherwise, we will continue checking for the empty cell in a circular manner. Thus, in this technique, the hash table size must be at least as large as the total number of keys.
Consider the elements 65, 78, 18, 22, 30, 89, 37, 55, 42

Let us take the hash function as h(k)= k%10, then the hash addresses will be –

h(65) = 65 %10 = 5	h(78) = 78%10 = 8	h(18)=18%10 =8
h(22)= 22 % 10 =2	h(30)= 30 %10 =0	h(89)=89 % 10 = 9
h(37)=37%10 =7	h(55)=55%10 =5	h(42)=42%10 =2

Now, hashing is done as below –

0	1	2	3	4	5	6	7	8	9
30	89	22	42		65	55	37	78	18

Efficiency:

- If the hash function distributes *n* keys among *m* cells of the hash table about evenly, then each linked list will be about n/m keys long.
- The ratio α =n/m is called as ***load factor***.
- The average number of comparisons done for a successful search,

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

- And for unsuccessful search,

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