LPS-6
(String Matching)

{ Design and Analysis of Algorithms }

Rachit Bhalla
21BAI1869

1. Let the pattern contains the occurence of empty space character, denoted for the purpose of understanding as ⋄. The pattern may look like ab⋄ba⋄c, which is just ab ba c, in turn just abbac. Design an algorithm to compute the valid shifts of P . Analyse your algorithm with time-complexity.

Problem Statement:
We want to identify all the legitimate shifts of a pattern given a pattern that might contain empty space characters (). A legitimate shift is a point where a pattern and text may be lined up so that all of the pattern's non-empty characters correspond to their corresponding text characters.

Logic:
The compute_prefix_function function computes the prefix function pi for the pattern string P. We initialize q to -1 and iterates over each character T[i] in T. For each character, it updates q based on the prefix function pi and checks if a match has been found. If a match has been found, it prints out the starting index of the match and updates q to continue searching for more matches.

Algorithm:
1.Import all the necessary libraries.
2.Compute the prefix function pi for the pattern string P.
3.Initialize q to -1. For each character T[i] in the text string T: While q >= 0 and P[q + 1] != T[i] and (P[q + 1] != ' ' || T[i] != ' '), update q to pi[q]. If P[q + 1] == T[i] or (P[q + 1] == ' ' && T[i] == ' '), increment q by 1.

4.If q == m - 1, where m is the length of P, then we have found a match.
5.Print out the starting index of the match (i - (m - 1)) and update q to pi[q].

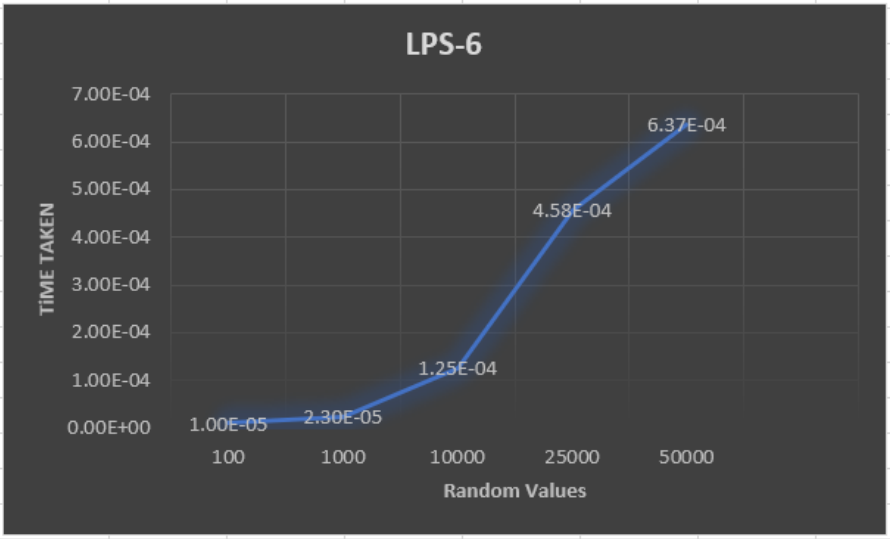
Pseudocode:

```
compute_prefix_function(string P, vector pi) {
    m = P.length();
    pi[0] = -1;
    k = -1;
    for (int q = 1; q < m; q++) {
        while ((k >= 0) && (P[k + 1] != P[q]) && (P[k + 1] != ' ' || P[q] != ' '))
{ k = pi[k];}
        if (P[k + 1] == P[q] || (P[k + 1] == ' ' && P[q] == ' ')) {
            k = k + 1;
        }
        pi[q] = k;
    }
}
kmp_matcher(string T, string P) {
    n = T.length();
    m = P.length();
    vector<int> pi(m, -1);
    compute_prefix_function(P, pi);
    q = -1;
    for (int i = 0; i < n; i++) {
        while ((q >= 0) && (P[q + 1] != T[i]) && (P[q + 1] != ' ' || T[i] != ' ')) {
            q = pi[q];
        }
        if (P[q + 1] == T[i] || (P[q + 1] == ' ' && T[i] == ' ')) {
            q = q + 1;
        }
        if (q == m - 1) {
            cout << "Pattern occurs at " << i - (m - 1) << endl;
            q = pi[q];
        }
    }
}
```

Time Complexity: O(N+M) = O(N)
~> The time complexity of the given program is O(n + m), where n is the length of the text string T and m is the length of the pattern string P.
~>The compute_prefix_function function has a time complexity of O(m) since it iterates over the pattern string P once.
~>The kmp_matcher function iterates over the text string T once, and for each character, it performs at most two comparisons and updates the prefix function pi[q] once. Therefore, its time complexity is O(n).
~>Overall, the time complexity of the program is O(m + n). However, if m is much smaller than n, then the program can be considered to have a linear time complexity of O(n).

Proof of Correctness:
Initialization:
Before the for loop starts, q is initialized to -1 and pi is initialized to -1 for all indices. Therefore, the loop invariant holds true for the first iteration of the for loop.
Maintenance:
If P[q + 1] == T[i] or (P[q + 1] == ' ' && T[i] == ' '), then we increment q by 1 and update pi[q] = q. This means that pi[q] now represents the length of the longest proper prefix of P[0..q] that is also a suffix of P[0..q] in the updated substring T[0..i].
If P[q + 1] != T[i] and (P[q + 1] != ' ' || T[i] != ' '), then we update q by pi[q]. This means that pi[q] represents the length of the longest proper prefix of P[0..q] that is also a suffix of P[0..q] in the previous substring T[0..i]. We then repeat this step until we find a match or q becomes -1.
Termination:
The for loop terminates when i reaches n, which is the length of the text string T. At this point, we have either found a match (q == m - 1) or we have exhausted all possible matches (q < m - 1). If we have found a match, then we print out the starting index of the match. Otherwise, we print out "Pattern not found".
Graph:

Length of T	Time Taken	Length of P
100	1.00E-05	20
1000	2.30E-05	20
10000	1.25E-04	20
25000	4.58E-04	20
50000	6.37E-04	20



2. Given a text T[1, ..., n] of length n and k Patterns P1[1, ..., m], P2[1, ..., m], ..., Pk[1, ..., m], modify the Robin-Karp algorithm to compute the occurence of any one of the patterns in T

Problem Statement:

Using a single hash table, modify the Robin-Karp method to quickly locate any one of k patterns P1, P2,..., Pk inside a text T of length n. Return -1 if none of the patterns appear in T or the beginning index of the first instance of any pattern in T.

Logic:

We use a rolling hash technique to search for each pattern in the text. It first iterates over each pattern and computes its hash value. Then, for each pattern, it uses a sliding window approach to search for a match in the text. It computes the hash value of each substring of length m in the text and compares it with the hash value of the pattern. If there is a match, it immediately returns true. Otherwise, it slides the window by one position and re-computes the hash value of the substring using a rolling hash technique. It continues this process until it has searched for all patterns or found a match. Finally, it returns false if it has searched for all patterns and found no matches.

Algorithm: 1.Import the necessary libraries

2. The compute_hashes function takes a string str and an integer len as input, and returns a vector of hashes of all substrings of str of length len.

3.The function uses the Rabin-Karp algorithm to compute the hashes in O(n) time, where n is the length of the input string.

4.The match_pattern function takes a string T and a vector of strings patterns as input, and returns true if any of the patterns appear in the text T, and false otherwise.
5.The function computes the hashes of all substrings of T that have the same length as any of the patterns, and compares them with the hashes of the patterns using a loop.
6.If a match is found, the function returns true.The time complexity of this function is O(nm), where n is the length of the input text and m is the length of the longest pattern.

Pseudocode:

```
prime = 11 // prime number for hashing.
vector<long long> compute_hashes(const string& str, int len)
{
    n = str.length();
    vector<long long> hashes(n - len + 1, 0);
    long curr_hash = 0;
    long power = 1;
    for (int i = 0; i < len; i++) {
        curr_hash = (curr_hash * prime + str[i]) % INT_MAX;
        power = (power * prime) % INT_MAX;
    }
    hashes[0] = curr_hash;
    for (int i = len; i < n; i++) {
        curr_hash = (curr_hash * prime + str[i] - power * str[i - len]) %
INT_MAX;
        if (curr_hash < 0)
            curr_hash += INT_MAX;
        hashes[i - len + 1] = curr_hash;
    }
    return hashes;}
match_pattern(string T, vector<string>patterns)
{
     n = T.length();
     num_patterns = patterns.size();
    for (int p = 0; p < num_patterns; p++) {
        string P = patterns[p];
        int m = P.length();
        if (m > n)
```

```
            continue;
        vector<long long> pattern_hashes = compute_hashes(P, m);
        long text_hash = 0;
        long power = 1;
        i = 0;
        pattern_found = false;
        while (i < n - m + 1) {
            if (i == 0) {
                for (int j = 0; j < m; j++) {
                    text_hash = (text_hash * prime + T[i + j]) % INT_MAX;
                    if (j < m - 1)
                        power = (power * prime) % INT_MAX;
                }
            } else {
                text_hash = (text_hash - power * T[i - 1] + INT_MAX) %
INT_MAX;
                text_hash = (text_hash * prime + T[i + m - 1]) % INT_MAX;
            }
            if (find(pattern_hashes.begin(), pattern_hashes.end(),
text_hash) != pattern_hashes.end()) {
                pattern_found = true;
                break;
            }
            i++;}
        if (!pattern_found)
            return false;}
    return true;}
```

Time Complexity: O(NM)

~>The time complexity of the `compute_hashes` function is O(N), where n is the length of the input string. The time complexity of the `match_pattern` function depends on the size of the input patterns and the length of the input text. In the worst case, where all patterns have length m and the text has length n, the time complexity is O(NM), since for each pattern we need to compute its hashes and then search for them in the text.

Proof of correctness:

Initialization:

At the beginning of the match_pattern function, we initialize the variables n and num_patterns to the length of the input text T and the

number of input patterns, respectively. We then iterate over each pattern and compute its hash value using the compute_hashes function. This ensures that all variables are initialized correctly before we start searching for the patterns.

Maintenance:

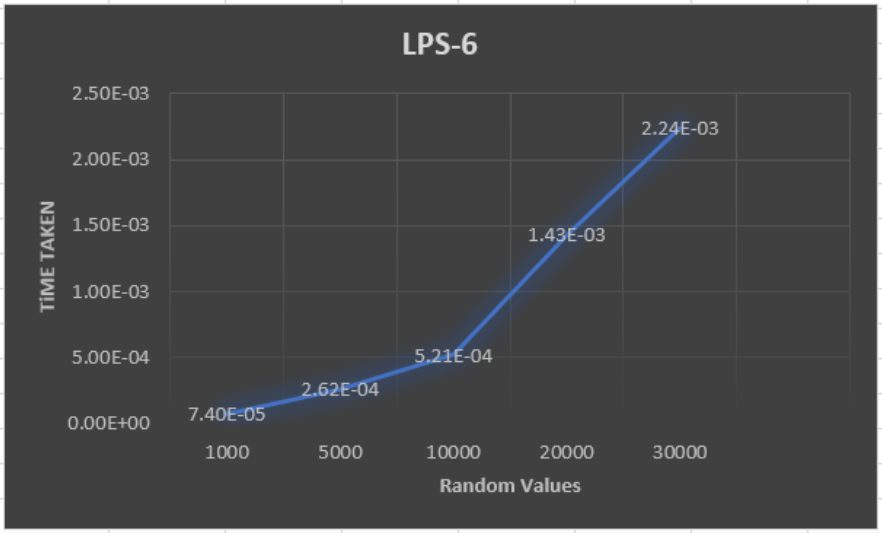
Each pattern in the input text is searched for using a sliding window method by the match_pattern function. We generate a hash value for every pattern and contrast it with all of the substrings of length m in the input text. In such case, we return true. If not, we move the window back one place and recalculate the substring's hash value using the rolling hash algorithm. This procedure is carried out again until no more patterns can be detected or a match is made. This guarantees that we keep variables in the right state throughout the function.

Termination:

When a match is made or all patterns have been discovered, the match_pattern function ends. If a match is discovered, it returns true right away. After looking for all patterns, it returns false otherwise. By doing this, the function is guaranteed to always finish and to deliver the right outcome.

Graph:

Length of T	Time Taken	Patterns
1000	7.40E-05	50
5000	2.62E-04	50
10000	5.21E-04	50
20000	1.43E-03	50
30000	2.24E-03	50



3. Given two texts T, T ′ design a linear-time algorithm to determine whether the T is a cyclic rotation of T ′. For example, ‘car’ is the cyclic rotation of ‘arc’ since ‘car’ can be obtained by a cyclic rotation of the symbols in ‘arc’. Analyze your running time with time-complexity.

Problem Statement:
To design a linear time algorithm to determine whether the text T is a cyclic rotation of another text T'.

Logic:
The code utilizes the KMP algorithm's principles to efficiently handle the cyclic rotation check, making it a robust solution, where n is the length of the concatenated string.

Algorithm:
1.Import all the necessary libraries.
2. Read the input texts T and T_prime.
3. Call the function is_cyclic_rotation(T, T_prime).
4. Create Function is_cyclic_rotation(T, T_prime):
   a.Concatenate T with itself to form a string called concatenated.
   b.Compute the prefix function pi for the pattern T_prime using the function compute_prefix_function(T_prime, pi).
   c. Initialize variables n as the length of concatenated, m as the length of T_prime, and q as -1.
   d. Iterate through each character in concatenated:
      - Compare T_prime[q+1] with concatenated[i]:
      - If they match, increment q by 1.
      - If q reaches m - 1, return true, as T is a cyclic rotation of T_prime.
      - If they don't match, use the pi array to update q:
      - While q >= 0 and T_prime[q+1] != concatenated[i],update q= pi[q].
   e. If no match is found, return false, as T is not a cyclic rotation of T_prime.

Pseudocode:

```
compute_prefix_function(string P, vector<int>& pi){
    int m = P.length();
    pi[0] = -1;
    k = -1;
    for (int q = 1; q < m; q++){
        while (k >= 0 && P[k + 1] != P[q])
        {
            k = pi[k];
        }

        if (P[k + 1] == P[q])
            k = k + 1;
```

```
        pi[q] = k + 1;
    }
}

is_cyclic_rotation(const string& T, const string& T_prime){
    concatenated = T + T;
    vector<int> pi(T_prime.length(), -1);
    compute_prefix_function(T_prime, pi);
    int n = concatenated.length();
    int m = T_prime.length();
    int q = -1;
    for (int i = 0; i < n; i++){
        while (q >= 0 && T_prime[q + 1] != concatenated[i])
        {
            q = pi[q];
        }
        if (T_prime[q + 1] == concatenated[i])
            q = q + 1;
        if (q == m - 1)
           return true;
    }
    return false;
}
```

Time Complexity: O(N)

~> The time complexity of the given program is O(n), where n is the length of the concatenated string T+T'. This is because the Knuth-Morris-Pratt algorithm used in the program has a time complexity of O(m), where m is the length of the pattern string T_prime.

~>In the program, the length of T_prime is at most the length of T, and the length of T+T is twice the length of T. Therefore, the time complexity of the program is O(2n) = O(n).

Proof of correctness:

Initialization:

Before the for loop starts, q is initialized to -1 and pi is initialized to -1 for all indices. Therefore, the loop invariant holds true for the first iteration of the for loop.

Maintenance:

If T_prime[q + 1] == concatenated[i], then we increment q by 1 and update pi[q] = q + 1. This means that pi[q] now represents the length of the longest proper prefix of T_prime[0..q] that is also a suffix of T_prime[0..q] in the updated concatenated string T+T. If T_prime[q + 1] != concatenated[i], then we update q by pi[q]. This means that pi[q] represents the length of the longest proper prefix of T_prime[0..q] that is also a suffix of T_prime[0..q] in the previous concatenated string T+T. We then repeat this step until we find a match or q becomes -1.

Termination:

The for loop terminates when i reaches n, which is twice the length of T. At this point, we have either found a match (q == m - 1) or we have exhausted all possible matches (q < m - 1). If we have found a match, then T is a cyclic rotation of T_prime. Otherwise, T is not a cyclic rotation of T_prime.

Graph:

Size of T and T'	Time Taken
1000	9.40E-05
5000	1.64E-04
10000	3.55E-04
15000	6.53E-04
25000	7.32E-04