EXPERIMENT NO – 5 **DATE –**

INTERNET ALGORITHMS

Internet Algorithms: A Comprehensive Theory

1. Introduction to Internet Algorithms

Internet algorithms form the backbone of modern text processing, enabling efficient operations across diverse applications such as web searching, document similarity analysis, data compression, and pattern matching in extensive datasets. These algorithms are crucial for managing massive datasets, typically measured in terabytes or petabytes, and they are designed to handle high-dimensional text data efficiently. Additionally, they must support real-time processing and leverage distributed computing to maintain performance at scale.

Key Characteristics of Internet-Scale Problems:

- **Massive Datasets:** Data sizes ranging from terabytes to petabytes, requiring algorithms that scale linearly or logarithmically with input size.
- **Real-Time Processing:** Algorithms must provide responses with minimal latency to maintain user experience, especially in search engines and recommendation systems.
- **High-Dimensional Text Data:** Data structures must be optimized for handling textual data with thousands or millions of features.
- **Distributed Computing:** Leveraging parallelism and data partitioning to improve processing efficiency and handle large datasets concurrently.

2. Fundamental String Operations

2.1 Basic Definitions

- **String:** An ordered sequence of characters from a given alphabet Σ. For example, a DNA sequence can be represented as a string over the alphabet {A, C, G, T}.
- **Substring:** A contiguous sequence of characters within a string P. If P = "GOTAACTGCTTTTATCAAACGC", a substring can be P[2..5] = "OTAA".
- **Prefix/Suffix:** Special cases of substrings that start or end at the boundaries of the string. The prefix of length 4 for the above sequence is "GOTA", while the suffix of length 3 is "CGC".
- **Pattern Matching:** The task of locating a specific pattern P within a text T. This operation is fundamental in text processing applications such as search engines and DNA analysis.

2.2 Core String Problems

Problem	Input	Output	Applications
Exact Matching	Text T, Pattern P	All positions where P occurs in T	Search engines, DNA analysis

Problem	Input	Output	Applications
Prefix Matching	String collection S, Query X	All strings in S with prefix X	Autocomplete, URL routing
Compression	Text X	Compressed representation Y	Data storage, network transmission
Similarity	Strings X, Y	Longest common subsequence	Version control, plagiarism detection

3. Pattern Matching Algorithms

3.1 Brute Force Method

- **Approach:** This method checks every possible alignment of the pattern P in the text T. The algorithm iterates through each starting position in T and attempts to match P at that position.
- **Time Complexity:** O(nm), where n is the length of T and m is the length of P.
- **Pseudocode:**

```
for i := 0 to n-m do
  j := 0
  while (j < m and T[i+j] == P[j]) do
    j := j + 1
  if j == m then return i
return -1
```

- **Limitations:** Brute force is inefficient for large inputs, making it unsuitable for large-scale text processing.

3.2 Advanced Methods

- **Knuth-Morris-Pratt (KMP):**
  - Preprocesses the pattern to build a "failure function" that helps skip redundant comparisons.
  - Time Complexity: O(n + m).
- **Boyer-Moore:**
  - Utilizes two key heuristics: "bad character" and "good suffix" to skip unnecessary comparisons.
  - Time Complexity: Best case is sublinear; worst case is O(nm).

4. Trie Data Structures

4.1 Standard Tries

- **Structure:** A tree where each edge represents a character and paths represent strings.
- **Operations:** Insert (O(d)), Search (O(d)), where d is the length of the string.
- **Applications:** Efficient for dictionary implementations, IP routing tables, and autocomplete systems.

4.2 Suffix Tries

- **Definition:** A specialized trie that contains all suffixes of a string X.
- **Compact Representation:** Stores substrings instead of individual characters, optimizing space.
- **Applications:** Full-text indexing, DNA sequence analysis, plagiarism detection.

5. Text Compression

5.1 Huffman Coding

- **Principle:** Assigns variable-length codes to characters based on their frequencies, with frequent characters receiving shorter codes.
- **Algorithm Steps:**
    - Calculate character frequencies.
    - Construct a priority queue.
    - Build a binary tree through greedy merges.
    - Generate codes from tree paths.
- **Properties:** Optimal prefix code, achieving 20-30% compression for natural language text.

5.2 Compression Ratios

Method	Compression	Speed	Notes
Huffman	Moderate	Fast	Requires frequency table
LZW	High	Medium	Utilized in GIF and UNIX compress
BWT	Very High	Slow	Basis for bzip2

6. Text Similarity

6.1 Longest Common Subsequence (LCS)

- **Definition:** The longest subsequence appearing in both strings, preserving character order.
- **Dynamic Programming Solution:** Employs a matrix to iteratively calculate the LCS length.
- **Time Complexity:** O(mn).

6.2 Applications

- Version control, DNA sequence alignment, and duplicate content detection in search engines.

DATE –

BOYER -MOORE ALGORITHM

Aim: C program to implement BM algorithm.

Problem Statement:

The String Pattern Matching problem (Boyer-Moore Algorithm) is to find the starting index of the first occurrence of a pattern string P in a given text string T. For each possible starting position in T, we need to determine if the substring of T matches P using an efficient approach that skips unnecessary comparisons based on information from the pattern.

Input:

- A string T (the text) of length n
- A string P (the pattern) of length m

Output:

- The starting index of the first substring of T matching P, or a message indicating P is not a substring of T

ALGORITHM

I]Algorithm Last()

Input: Pattern string P of length m
Output: Array last[c] for each character c in the alphabet, where last[c] is the largest index j such that P[j] = c (or -1 if c does not occur in P)

for each character c in the alphabet do

  last[c] ← -1

for j ← 0 to m - 1 do

  last[P[j]] ← j

Recurrence Relation

The Last() algorithm is **iterative**, not recursive, so **no recurrence relation** applies here.

Time Complexity

I] **Best Case:**

Time Complexity: O(m + d)

➞ When m is the pattern length and d is the alphabet size. Both loops must be executed regardless of input.

II] **Average Case:**

Time Complexity: O(m + d)

➞ In average scenarios, all m pattern characters and all d alphabet entries are still touched.

III] **Worst Case:**

Time Complexity: O(m + d)

➞ Always processes every alphabet character and every character of the pattern.

Space Complexity

I] **Best Case:**

Space Complexity: O(d)

➞ Only the last[] array of size d (alphabet size) is used.

II] **Average Case:**

Space Complexity: O(d)

III] **Worst Case:**

Space Complexity: O(d)

Algorithm BMMatch(T, P):

Input: Strings T (text) with n characters and P (pattern) with m characters
Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

compute function last

i ← m - 1

j ← m - 1

repeat

if P[j] = T[i] then

if j = 0 then

return i { a match! }

else

i ← i - 1

j ← j - 1

```
    else
        i ← i + m - min(j, 1 + last(T[i]))   { jump step }
        j ← m - 1
until i > n - 1
return "There is no substring of T matching P."
```

Recurrence Relation

There is **no recurrence relation** here, as this is a **non-recursive algorithm** based on string pattern matching with **bad character heuristic**.

Time Complexity

I] **Best Case:**

Time Complexity: O(n/m)

➞ When characters do not match early and pattern skips m positions on each mismatch.

➞ Very efficient for long texts and patterns with rare symbols.

II] **Average Case:**

Time Complexity: O(n)

➞ Most practical inputs yield linear performance due to good skipping using the bad character rule.

III] **Worst Case:**

Time Complexity: O(n × m)

➞ In degenerate cases where mismatches occur at the end and pattern keeps sliding one character.

➞ Example: T = "aaaaaaaaaa...", P = "aaaab"

Space Complexity

I] **Best Case:**

Space Complexity: O(1)

➞ No extra space apart from a few pointers (i, j, etc.).

II] **Average Case:**

Space Complexity: O(|Σ|)

➞ The last[] array stores position of each character in the pattern.

➞ Σ = character set (e.g., ASCII → 128 or Unicode → 256)

III] **Worst Case:**

Space Complexity: O(|Σ|)

➞ Remains same in worst case; no additional memory used per step.

PROGRAM

```
#include <stdio.h>
#include <string.h>
#define MAX 100
#include <sys/time.h>

char p[MAX];
char t[MAX];
int cmp[MAX] = {0};
int comparison_count = 0;
int store;
int lastoccurrence(char a) {
    int m = strlen(p);
    for (int i = m - 1; i >= 0; i--) {
        if (p[i] == a) {
            return i;
        }
    }
    return -1;
}

long long current_time_us()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}

int min(int a, int b) {
    return (a <= b) ? a : b;
}

void print_text() {
    int n = strlen(t);
    printf("Pattern: %s\n", p);

    printf("    ");
    for (int i = 0; i < n; i++) {
        printf("%4d", i);
    }
    printf("\n");

    printf("    ");
    for (int i = 0; i < n; i++) {
        printf("----");
    }
    printf("\n");

    printf("    ");
    for (int i = 0; i < n; i++) {
        printf("|%3c", t[i]);
    }
    printf("|\n");

    printf("    ");
    for (int i = 0; i < n; i++) {
        printf("----");
    }
    printf("\n");
}

void print_pattern(int i, int j, int lastocc) {
    int m = strlen(p);
    int n = strlen(t);

    printf("\n");
    for (int k = 0; k < (i - j + 1); k++) {
        printf("    ");
    }

    for (int idx = 0; idx < m; idx++) {
        printf("|%3c", p[idx]);
    }
    printf("| i = %d lastocc = %d\n", i, lastocc);

    for (int k = 0; k < (i - j + 1); k++) {
        printf("    ");
    }
    for (int idx = 0; idx < m; idx++) {
        printf("|%3d", cmp[idx]);
    }
    printf("| j = %d\n", j);
}

int BM() {
    print_text();
    int m = strlen(p);
    int n = strlen(t);
    int i = m - 1;
    int j = m - 1;
    int flag = 1;

    do {
        comparison_count++;
        if (p[j] == t[i]) {
            cmp[j]++;
            if (j == 0) {
                return i;
            } else {
```

```
            i--;
            j--;
        }
    } else {
        cmp[j]++;
        int lastocc = lastoccurrence(t[i]);
        store = n - i - (m - j);
        print_pattern( i, j, lastocc);
        i = i + m - min(j, lastocc + 1);
        j = m - 1;
    }
  } while (i <= n - 1);
  return -1;
}
int main() {
  int choice;
  long long start_time, end_time;
  do {
    printf("\nBoyer-Moore Pattern Matching Algorithm\n");
    printf("1. Enter new text and pattern\n");
    printf("2. Search pattern\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    getchar();
    switch(choice) {
      case 1:
        printf("Enter the text: ");
        fgets(t, MAX, stdin);
        t[strcspn(t, "\n")] = 0;

        printf("Enter the pattern to search: ");
        fgets(p, MAX, stdin);
        p[strcspn(p, "\n")] = 0;
        break;
      case 2:
        if (strlen(t) == 0 || strlen(p) == 0) {
          printf("Please enter text and pattern first!\n");
          break;
        }
        comparison_count = 0;
        memset(cmp, 0, sizeof(cmp));
        printf("\nText: %s\n", t);
        printf("Pattern: %s\n", p);
        start_time = current_time_us();
            int i = BM();
        print_pattern( i, 0, 0);
        end_time = current_time_us();
        printf("Time taken: %lld µs\n", end_time - start_time);
        if (i != -1) {
          printf("\nPattern found at index: %d\n", i);
        } else {
          printf("\nPattern not found in the text\n");
        }
        printf("Number of comparisons made: %d\n", comparison_count);
        break;
      case 3:
        printf("Exiting program...\n");
        break;

      default:
        printf("Invalid choice! Please try again.\n");
    }
  } while (choice != 3);
  return 0;
}
```

OUTPUT:

```
Boyer-Moore Pattern Matching Algorithm
1. Enter new text and pattern
2. Search pattern
3. Exit
Enter your choice: 1
Enter the text: aabaacbbaabaacaabaabacaccaca
Enter the pattern to search: aabacac

Boyer-Moore Pattern Matching Algorithm
1. Enter new text and pattern
2. Search pattern
3. Exit
Enter your choice: 2

Text: aabaacbbaabaacaabaabacaccaca
Pattern: aabacac
Pattern: aabacac
      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27
    -----------------------------------------------------------------------------------------------------------------
    | a|  a|  b|  a|  a|  c|  b|  b|  a|  a|  b|  a|  a|  c|  a|  a|  b|  a|  a|  b|  a|  c|  a|  c|  c|  a|  c|  a|
    -----------------------------------------------------------------------------------------------------------------

    |  a|  a|  b|  a|  c|  a|  c| i = 6 lastocc = 2
    |  0|  0|  0|  0|  0|  0|  1| j = 6

                  |  a|  a|  b|  a|  c|  a|  c| i = 10 lastocc = 2
                  |  0|  0|  0|  0|  0|  0|  2| j = 6

                                |  a|  a|  b|  a|  c|  a|  c| i = 14 lastocc = 5
                                |  0|  0|  0|  0|  0|  0|  3| j = 6

                                   |  a|  a|  b|  a|  c|  a|  c| i = 15 lastocc = 5
                                   |  0|  0|  0|  0|  0|  0|  4| j = 6

                                      |  a|  a|  b|  a|  c|  a|  c| i = 16 lastocc = 2
                                      |  0|  0|  0|  0|  0|  0|  5| j = 6

                                                  |  a|  a|  b|  a|  c|  a|  c| i = 20 lastocc = 5
                                                  |  0|  0|  0|  0|  0|  0|  6| j = 6

                                                     |  a|  a|  b|  a|  c|  a|  c| i = 19 lastocc = 2
                                                     |  0|  0|  0|  0|  1|  1|  7| j = 4

                                                           |  a|  a|  b|  a|  c|  a|  c| i = 17 lastocc = 0
                                                           |  1|  1|  1|  1|  2|  2|  8| j = 0
Time taken: 21574 µs

Pattern found at index: 17
Number of comparisons made: 16

Boyer-Moore Pattern Matching Algorithm
1. Enter new text and pattern
2. Search pattern
3. Exit
Enter your choice: 3
Exiting program...
```

Conclusion: BM algorithm was implemented successfully in C .

DATE –

KMP ALGORITHM

Aim: C program to implement KMP algorithm.

Problem Statement:

The String Pattern Matching problem is to find the starting index of the first occurrence of a pattern string P in a given text string T. For each possible starting position in T, we need to determine if the substring of T matches P.

Input:

- A string T (the text) of length n
- A string P (the pattern) of length m

Output:

- The starting index of the first substring of T matching P, or a message indicating P is not a substring of T

ALGORITHMS –

Algorithm KMPFailureFunction(P):

Input: String P (pattern) with m characters
Output: The failure function f for P, which maps *j* to the length of the longest prefix of *P* that is a suffix of P[1..j]

```
i ← 1
j ← 0
f(0) ← 0
while i < m do
   if P[j] = P[i] then
      {we have matched j + 1 characters}
      f(i) ← j + 1
      i ← i + 1
      j ← j + 1
   else if j > 0 then
      {j indexes just after a prefix of P that must match}
```

j ← f(j - 1)

else

{we have no match here}

f(i) ← 0

i ← i + 1

Recurrence Relation

There is **no recurrence**, as this is a **linear iterative algorithm** that builds the failure function array for the pattern.

Time Complexity

I] **Best Case:**

Time Complexity: O(m)

➞ When all characters in the pattern match and the loop runs with simple increments of i and j.

II] **Average Case:**

Time Complexity: O(m)

➞ On average, each character is processed a constant number of times due to efficient backtracking using the failure function.

III] **Worst Case:**

Time Complexity: O(m)

➞ Even in the worst-case scenario, each character is accessed at most twice — once for match check and once for failure fallback

Space Complexity

I] **Best Case:**

Space Complexity: O(m)

➞ Space is required to store the failure function array of size m.

II] **Average Case:**

Space Complexity: O(m)

➞ No additional dynamic space is used beyond the f[] array.

III] **Worst Case:**

Space Complexity: O(m)

➞ Same space used in all cases; no recursion or auxiliary structures.

Algorithm KMPMatch(T, P):

Input: Strings T (text) with n characters and P (pattern) with m characters
Output: Starting index of the first substring of T matching P, or an indication that *P* is not a substring of T

```
f ← KMPFailureFunction(P)    {construct the failure function f for P}
i ← 0
j ← 0
while i < n do
   if P[j] = T[i] then
      if j = m - 1 then
         return i - m + 1    {a match!}
      i ← i + 1
      j ← j + 1
   else if j > 0 then
      {no match, but we have advanced in P}
      j ← f(j - 1)    {j indexes just after prefix of P that must match}
   else
      i ← i + 1
return "There is no substring of T matching P."
```

Recurrence Relation

There is **no recurrence relation**, as the algorithm follows an **iterative linear approach** using the failure function array.

Time Complexity

I] **Best Case:**
Time Complexity: O(n)
➞ When there is an early full match or mismatch is detected early with fast jumps using the failure function.

II] **Average Case:**
Time Complexity: O(n)
➞ Most characters in the text T are compared at most once due to smart backtracking via failure function.

III] **Worst Case:**

Time Complexity: O(n)

➞ Even in the worst case, due to the failure function f, the algorithm never backtracks on i and progresses through the text linearly.

Space Complexity

I] **Best Case:**

Space Complexity: O(m)

➞ Only space used is for the failure function f[0...m-1].

II] **Average Case:**

Space Complexity: O(m)

➞ Space remains the same, as only the pattern length affects auxiliary memory.

III] **Worst Case:**

Space Complexity: O(m)

➞ No recursion or stack usage; space is dominated by the f[] array for the pattern.

PROGRAM

```
#include <stdio.h>
#include <string.h>
#define MAX 100
#include <sys/time.h>
char p[MAX];
char t[MAX];
int cmp[MAX] = {0};
int comparison_count = 0;
int f[MAX];
int store;
long long current_time_us(){
   struct timeval tv;
   gettimeofday(&tv, NULL);
   return tv.tv_sec * 1000000LL + tv.tv_usec;
}
void failureFunction(int m){
   f[0] = 0;
   int i = 1, j = 0;
   printf("\nFailure Function (f[]): ");
   while (i < m)   {
      if (p[i] == p[j])      {
         f[i] = j + 1;
         i++;
         j++;
      }
      else if (j > 0)      {
         j = f[j - 1];
      }
      else      {
         f[i] = 0;
         i++;
      }
   }
   printf("\n");
   for (int k = 0; k < m; k++)
      printf("%2c ", p[k]);
   printf("\n");
   for (int k = 0; k < m; k++)
      printf("---", f[k]);
   printf("-\n|");
   for (int k = 0; k < m; k++)
      printf("%d |", f[k]);
   printf("\n");
   for (int k = 0; k < m; k++)
      printf("---", f[k]);
   printf("-\n");
   printf("\n");
}
void print_pattern(int i, int j){
   int m = strlen(p);
   int n = strlen(t);
   printf("\n");
   for (int k = 0; k < (i - j + 1); k++)   {
      printf("    ");
   }
   for (int idx = 0; idx < m; idx++)   {
      printf("|%3c", p[idx]);
   }
   printf("| i = %d\n", i);
   for (int k = 0; k < (i - j + 1); k++)   {
      printf("    ");
   }
   for (int idx = 0; idx < m; idx++)   {
      printf("|%3d", cmp[idx]);
   }
   printf("| j = %d\n", j);
}
void print_text(){
   int n = strlen(t);
   printf("Pattern: %s\n", p);
   printf("    ");
   for (int i = 0; i < n; i++)   {
      printf("%4d", i);
   }
   printf("\n");
   printf("    ");
   for (int i = 0; i < n; i++)   {
      printf("----");
   }
   printf("\n");
   printf("    ");
   for (int i = 0; i < n; i++)   {
      printf("|%3c", t[i]);
   }
   printf("|\n");
   printf("    ");
   for (int i = 0; i < n; i++)   {
      printf("----");
   }
   printf("\n");
}

int KMP(){
   int m = strlen(p);
```

```
    int n = strlen(t);
    failureFunction(m);
    print_text( );
    int i = 0, j = 0;
    while (i < n){
        comparison_count++;
        if (t[i] == p[j])      {
            cmp[j]++;
            if (j == m - 1)
            {
                return i - m + 1;
            }
            i++;
            j++;
        }
        else if (j > 0)   {
            store = i;
            cmp[j]++;
            print_pattern( i, j);
            j = f[j - 1];
        }
        else      {
            cmp[j]++;
            print_pattern( i, j);
            i++;
        }
    }
    return -1;
}
int main(){
    int choice;
    long long start_time, end_time;
    do    {
        printf("\nKnuth-Morris-Pratt Pattern Matching Algorithm\n");
        printf("1. Enter new text and pattern\n");
        printf("2. Search pattern\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar();
        switch (choice)       {
        case 1:
            printf("Enter the text: ");
            fgets(t, MAX, stdin);
            t[strcspn(t, "\n")] = 0;

            printf("Enter the pattern to search: ");
            fgets(p, MAX, stdin);
            p[strcspn(p, "\n")] = 0;
            break;
        case 2:
            if (strlen(t) == 0 || strlen(p) == 0)
            {
                printf("Please enter text and pattern first!\n");
                break;
            }
            comparison_count = 0;
            memset(cmp, 0, sizeof(cmp));
            printf("\nText: %s\n", t);
            printf("Pattern: %s\n", p);
            start_time = current_time_us();
            int i = KMP();
            print_pattern( i, 0);
            end_time = current_time_us();
            printf("Time taken: %lld µs\n", end_time - start_time);
            if (i != -1)          {
                printf("\nPattern found at index: %d\n", i);
            }
            else          {
                printf("\nPattern not found in the text\n");
            }
            printf("Number of comparisons made: %d\n", comparison_count);
            break;
        case 3:
            printf("Exiting program...\n");
            break;
        default:
            printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 3);
    return 0;
}
```

OUTPUT:

```
Knuth-Morris-Pratt Pattern Matching Algorithm
1. Enter new text and pattern
2. Search pattern
3. Exit
Enter your choice: 1
Enter the text: aabaacbbaabaacaabaabacaccaca
Enter the pattern to search: aabacac

Knuth-Morris-Pratt Pattern Matching Algorithm
1. Enter new text and pattern
2. Search pattern
3. Exit
Enter your choice: 2

Text: aabaacbbaabaacaabaabacaccaca
Pattern: aabacac

Failure Function (f[]):
 a  a  b  a  c  a  c
----------------------
|0 |1 |0 |1 |0 |1 |0 |
----------------------
```

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
 ---------------------------------------------------------------------------------------
 | a| a| b| a| a| c| b| b| a| a| b| a| a| c| a| a| b| a| a| b| a| c| a| c| c| a| c| a|
 ---------------------------------------------------------------------------------------

 | a| a| b| a| c| a| c| i = 4
 | 1| 1| 1| 1| 1| 0| 0| j = 4

          | a| a| b| a| c| a| c| i = 5
          | 1| 2| 2| 1| 1| 0| 0| j = 2

              | a| a| b| a| c| a| c| i = 5
              | 1| 3| 2| 1| 1| 0| 0| j = 1

                  | a| a| b| a| c| a| c| i = 5
                  | 2| 3| 2| 1| 1| 0| 0| j = 0

                      | a| a| b| a| c| a| c| i = 6
                      | 3| 3| 2| 1| 1| 0| 0| j = 0

                          | a| a| b| a| c| a| c| i = 7
                          | 4| 3| 2| 1| 1| 0| 0| j = 0

                              | a| a| b| a| c| a| c| i = 12
                              | 5| 4| 3| 2| 2| 0| 0| j = 4

                                        | a| a| b| a| c| a| c| i = 13
                                        | 5| 5| 4| 2| 2| 0| 0| j = 2

                                            | a| a| b| a| c| a| c| i = 13
                                            | 5| 6| 4| 2| 2| 0| 0| j = 1

                                                | a| a| b| a| c| a| c| i = 13
                                                | 6| 6| 4| 2| 2| 0| 0| j = 0

                                                    | a| a| b| a| c| a| c| i = 18
                                                    | 7| 7| 5| 3| 3| 0| 0| j = 4

                                                              | a| a| b| a| c| a| c| i = 17
                                                              | 7| 8| 6| 4| 4| 1| 1| j = 0
```

```
Time taken: 26318 µs

Pattern found at index: 17
Number of comparisons made: 31

Knuth-Morris-Pratt Pattern Matching Algorithm
1. Enter new text and pattern
2. Search pattern
3. Exit
Enter your choice: 3
```

Conclusion: KMP algorithm was implemented successfully in C .

DATE –

LONGEST COMMON SUBSEQUENCE

Aim: C program to implement LCS algorithm.

Problem Statement

The Longest Common Subsequence (LCS) problem is to find the length of the longest subsequence common to two given strings X and Y. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguously, in both strings.

Input:

- Strings X and Y with n and m elements, respectively

Output:

- For i = 0, ..., n-1 and j = 0, ..., m-1, the length L[i, j] of a longest common subsequence of X[0..i] and Y[0..j]

ALGORITHMS:

Algorithm LCS(X, Y)

Input: Strings X and Y with n and m elements, respectively
Output: For i = 0, ..., n-1, j = 0, ..., m-1, the length L[i, j] of a longest common subsequence of X[0..i] and Y[0..j]

```
for i ← -1 to n - 1 do
  L[i, -1] ← 0

for j ← 0 to m - 1 do
  L[-1, j] ← 0
for i ← 0 to n - 1 do
  for j ← 0 to m - 1 do
    if X[i] = Y[j] then
      L[i, j] ← L[i - 1, j - 1] + 1
    else
      L[i, j] ← max{ L[i - 1, j], L[i, j - 1] }

return array L
```

Recurrence Relation

Let L[i][j] be the length of the LCS of X[0..i] and Y[0..j].

$$L[i][j] = \begin{cases} 0 & \text{if } i = -1 \text{ or } j = -1 \\ L[i-1][j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(L[i-1][j], L[i][j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

Time Complexity

I] **Best Case:**

Time Complexity: O(n × m)

➞ Even if characters match early, the table L must still be completely filled.

II] **Average Case:**

Time Complexity: O(n × m)

➞ All entries in the n × m table must be computed.

III] **Worst Case:**

Time Complexity: O(n × m)

➞ No characters match at all; still must fill all table cells.

Space Complexity

I] **Best Case:**

Space Complexity: O(n × m)

➞ Full 2D array L is used.

II] **Average Case:**

Space Complexity: O(n × m)

III] **Worst Case:**

Space Complexity: O(n × m)

PROGRAM

```
#include <string.h>
#include <stdio.h>
#include <sys/time.h>

#define MAX 100
#define UP_ARROW '^'
#define LEFT_ARROW '<'
#define DIAGONAL_ARROW '\\'

char X[MAX];
char Y[MAX];
int L[MAX][MAX];
int sub1[MAX];
int leastSequence[MAX][MAX] = {0};

int maxleast(int a, int b) {
    return (a > b) ? a : b;
}

long long current_time_us()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}

void LCS() {
    int n = strlen(X);
    int m = strlen(Y);
    for (int i = 0; i <= n; i++) {
        L[i][0] = 0;
    }
    for (int j = 0; j <= m; j++) {
        L[0][j] = 0;
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (X[i - 1] == Y[j - 1]) {
                L[i][j] = L[i - 1][j - 1] + 1;
            } else {
                L[i][j] = maxleast(L[i - 1][j], L[i][j - 1]);
            }
        }
    }
}

void traverse() {
    int n = strlen(X);
    int m = strlen(Y);
    int i = n;
    int j = m;
    int c = L[n][m];
    while (c > 0) {
        if (X[i - 1] == Y[j - 1]) {
            leastSequence[i][j] = DIAGONAL_ARROW;
            sub1[c] = X[i - 1];
            c--;
            i--;
            j--;
        } else {
            if (L[i - 1][j] > L[i][j - 1]) {
                leastSequence[i][j] = UP_ARROW;
                i--;
            } else {
                leastSequence[i][j] = LEFT_ARROW;
                j--;
            }
        }
    }
}

int main() {
    int choice;
    while (1) {
        printf("\n=== Longest Common Subsequence Menu ===\n");
        printf("1. Find LCS of two strings\n");
        printf("2. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                // Clear input buffer
                while (getchar() != '\n');

                printf("Enter X: ");
                scanf("%s", X);
                printf("Enter Y: ");
```

```
        scanf("%s", Y);


        int n = strlen(X);
        int m = strlen(Y);
        long long start_time
= current_time_us();


        LCS();
        traverse();


        long long end_time
= current_time_us();
        printf("Time taken:
%lld µs\n", end_time -
start_time);


        printf("\n    -1 ");
        for (int j = 0; j < m;
j++) {
            printf("%3d ", j);
        }
        printf("\n       ");
        for (int j = 0; j < m;
j++) {
            printf("%3c ",
Y[j]);
        }
        printf("\n");

        for (int i = 0; i <= n;
i++) {
            if (i == 0) {
                printf("-1  ");
            } else {
                printf("%2d
%c", i-1, X[i-1]);
            }

            for (int j = 0; j <=
m; j++) {
                if
(leastSequence[i][j] ==
DIAGONAL_ARROW) {
                    printf("%2c%
d ", DIAGONAL_ARROW,
L[i][j]);
                } else if
(leastSequence[i][j] ==
UP_ARROW) {
                    printf("%2c%
d ", UP_ARROW, L[i][j]);
                } else if
(leastSequence[i][j] ==
LEFT_ARROW) {
                    printf("%2c%
d ", LEFT_ARROW, L[i][j]);
                } else {
                    printf("%3d ",
L[i][j]);
                }
            }
            printf("\n");
        }


        printf("\nLongest
Common Subsequence: ");
        for (int i = 1; i <=
L[n][m]; i++) {
            printf("%c ",
sub1[i]);
        }
        printf("\n");
        break;
      }
      case 2:
        printf("Exiting
program...\n");
        return 0;
      default:
        printf("Invalid
choice! Please try again.\n");
    }
  }
  return 0;
}
```

OUTPUT:

```
=== Longest Common Subsequence Menu ===
1. Find LCS of two strings
2. Exit
Enter your choice: 1
Enter X: GCTAGTTACG
Enter Y: ATGACTAAGCCTAGT
Time taken: 0 µs

     -1   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
          A   T   G   A   C   T   A   A   G   C   C   T   A   G   T
-1    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 0 G  0   0   0  \1  <1   1   1   1   1   1   1   1   1   1   1   1
 1 C  0   0   0   1   1  \2   2   2   2   2   2   2   2   2   2   2
 2 T  0   0   1   1   1   2  \3  <3   3   3   3   3   3   3   3   3
 3 A  0   1   1   1   2   2   3   4  \4   4   4   4   4   4   4   4
 4 G  0   1   1   2   2   2   3   4   4  \5   5   5   5   5   5   5
 5 T  0   1   2   2   2   2   3   4   4  ^5  <5  <5   6   6   6   6
 6 T  0   1   2   2   2   2   3   4   4   5   5   5  \6   6   6   7
 7 A  0   1   2   2   3   3   3   4   5   5   5   5   6  \7   7   7
 8 C  0   1   2   2   3   4   4   4   5   5   6   6   6  ^7   7   7
 9 G  0   1   2   3   3   4   4   4   5   6   6   6   6   7  \8  <8

Longest Common Subsequence: G C T A G T A G

=== Longest Common Subsequence Menu ===
1. Find LCS of two strings
2. Exit
Enter your choice: 2
Exiting program...
```

Conclusion: LCS algorithm was implemented successfully in C .

DATE –

HUFFMAN ENCODING

Aim: C program to implement Huffman encoding algorithm.

Problem Statement

The Huffman Coding problem is to construct an optimal prefix code (binary coding tree) for a given string X, based on the frequency of each distinct character in X. The objective is to minimize the total length of the encoded string by assigning shorter codes to more frequent characters.

Input:

- A string X of length n with d distinct characters

Output:

- A coding tree for X (an optimal prefix-free binary tree for the characters of X)

ALGORITHMS:

Algorithm Huffman(X)

Input: String X of length n with d distinct characters
Output: Coding tree for X

Compute the frequency f(c) of each character c of X.

Initialize a priority queue Q.

for each character c in X do

  Create a single-node binary tree T storing c.

  Insert T into Q with key f(c).

while Q.size() > 1 do

  f1 ← Q.minKey()

  T1 ← Q.removeMin()

  f2 ← Q.minKey()

  T2 ← Q.removeMin()

  Create a new binary tree T with left subtree T1 and right subtree T2.

  Insert T into Q with key f1 + f2.

return tree Q.removeMin()

Recurrence Relation

The key operation in Huffman coding is merging two smallest frequency trees d - 1 times using a min-heap (priority queue). Each extract-min and insert operation in a heap of size k takes O(log k) time.

So, the recurrence relation for the total time spent in building the Huffman tree is:

T(d)= O(d.logd)

Time Complexity

I] **Best Case:**

Time Complexity: O(d log d)

➞ Frequencies of characters are distinct and no extra heap balancing is needed.

II] **Average Case:**

Time Complexity: O(d log d)

➞ Involves:

- Inserting d nodes into the priority queue: O(d log d)
- Performing d - 1 remove and insert operations: O(d log d)

III] **Worst Case:**

Time Complexity: O(d log d)

➞ The structure of the Huffman tree might be skewed, but each heap operation is log d and done d times.

Space Complexity

I] **Best Case:**

Space Complexity: O(d)

➞ One tree node per character.

II] **Average Case:**

Space Complexity: O(d)

III] **Worst Case:**

Space Complexity: O(d)

➞ Total number of nodes in the final binary Huffman tree is 2d - 1.

PROGRAM

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>


#define MAX_TREE_HT 100

#define MAX_CHARS 256


struct MinHeapNode {

   char data;

   unsigned freq;

   struct MinHeapNode *left, *right;

   int first_occurrence;

   int tree_num;

};


struct MinHeap {

   unsigned size;

   unsigned capacity;

   struct MinHeapNode** array;

};


void printArr(int arr[], int n) {

   for (int i = 0; i < n; ++i)

      printf("%d", arr[i]);

   printf("\n");

}


struct MinHeapNode* newNode(char data, unsigned freq, int first_occurrence, int tree_num) {

   struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof (struct MinHeapNode));

   temp->left = temp->right = NULL;

   temp->data = data;

   temp->freq = freq;

   temp->first_occurrence = first_occurrence;

   temp->tree_num = tree_num;

   return temp;

}


struct MinHeap* createMinHeap(unsigned capacity) {

   struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));

   minHeap->size = 0;

   minHeap->capacity = capacity;

   minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct MinHeapNode*));

   return minHeap;

}


void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {

   struct MinHeapNode* t = *a;

   *a = *b;

   *b = t;

}


int compareNodes(struct MinHeapNode* a, struct MinHeapNode* b) {

   if (a->freq != b->freq) {

      return a->freq < b->freq;

   }


   if ((a->data != '$') != (b->data != '$')) {

      return a->data != '$';  // True if a is character node

   }


   if (a->data != '$' && b->data != '$') {

      return a->first_occurrence < b->first_occurrence;

   }


   return a->tree_num < b->tree_num;

}


void minHeapify(struct MinHeap* minHeap, int idx) {

   int smallest = idx;

   int left = 2 * idx + 1;

   int right = 2 * idx + 2;


   if (left < minHeap->size && compareNodes(minHeap->array[left], minHeap->array[smallest])) {

      smallest = left;

   }


   if (right < minHeap->size && compareNodes(minHeap-
```

```
>array[right], minHeap->array[smallest])) {
        smallest = right;
    }

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

int isSizeOne(struct MinHeap* minHeap) {
    return (minHeap->size == 1);
}

struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* minHeapNode) {
    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i > 0) {
        int parent = (i - 1) / 2;
        if (compareNodes(minHeapNode, minHeap->array[parent])) {
            minHeap->array[i] = minHeap->array[parent];
            i = parent;
        } else {
            break;
        }
    }

    minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap* minHeap) {
    int n = minHeap->size - 1;
    for (int i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

int* sortByFirstOccurrence(char* input, int* char_count) {
    int len = strlen(input);
    int* char_order = (int*)malloc(MAX_CHARS * sizeof(int));

    for (int i = 0; i < MAX_CHARS; i++) {
        char_order[i] = -1;
    }

    *char_count = 0;
    for (int i = 0; i < len; i++) {
        unsigned char ch = input[i];
        if (char_order[ch] == -1) {
            char_order[ch] = i;
            (*char_count)++;
        }
    }

    return char_order;
}

struct MinHeap* createInitialHeap(char* input) {
    int len = strlen(input);
    int char_count = 0;
    int* first_occurrences = sortByFirstOccurrence(input, &char_count);

    int freq[MAX_CHARS] = {0};
    for (int i = 0; i < len; i++) {
        freq[(unsigned char)input[i]]++;
    }

    struct MinHeap* minHeap = createMinHeap(char_count);

    struct MinHeapNode** nodes = (struct MinHeapNode**)malloc(char_count * sizeof(struct MinHeapNode*));
    int node_count = 0;

    for (int i = 0; i < MAX_CHARS; i++) {
        if (freq[i] > 0) {
```

```
        nodes[node_count++] = newNode((char)i, freq[i], first_occurrences[i], 0);
      }
   }

   for (int i = 0; i < node_count - 1; i++) {
      for (int j = 0; j < node_count - i - 1; j++) {
         if (!compareNodes(nodes[j], nodes[j+1])) {
            struct MinHeapNode* temp = nodes[j];
            nodes[j] = nodes[j+1];
            nodes[j+1] = temp;
         }
      }
   }

   for (int i = 0; i < node_count; i++) {
      minHeap->array[i] = nodes[i];
   }

   minHeap->size = node_count;
   buildMinHeap(minHeap);

   free(nodes);
   free(first_occurrences);

   return minHeap;
}

struct MinHeapNode** getSortedHeapArray(struct MinHeap* minHeap) {
   struct MinHeapNode** sortedArray = (struct MinHeapNode**)malloc(minHeap->size * sizeof(struct MinHeapNode*));
   for (int i = 0; i < minHeap->size; i++) {
      sortedArray[i] = minHeap->array[i];
   }

   for (int i = 0; i < minHeap->size - 1; i++) {
      for (int j = 0; j < minHeap->size - i - 1; j++) {
         if (!compareNodes(sortedArray[j], sortedArray[j+1])) {
            struct MinHeapNode* temp = sortedArray[j];
            sortedArray[j] = sortedArray[j+1];
            sortedArray[j+1] = temp;
         }
      }
   }

   return sortedArray;
}

void printMinHeap(struct MinHeap* minHeap) {
   struct MinHeapNode** sortedArray = getSortedHeapArray(minHeap);

   for (int i = 0; i < minHeap->size; i++) {
      if (sortedArray[i]->data == '$') {
         printf("T%-2d ", sortedArray[i]->tree_num);
      } else {
         printf("%-3c ", sortedArray[i]->data);
      }
   }
   printf("\n");
   for (int i = 0; i < minHeap->size; i++) {
      printf("%-3u ", sortedArray[i]->freq);
   }
   printf("\n");

   free(sortedArray);
}

void printTree(struct MinHeapNode* root, int space) {
   if (root == NULL)
      return;

   space += 10;
   printTree(root->right, space);

   printf("\n");
   for (int i = 10; i < space; i++)
      printf(" ");

   if (root->data == '$') {
      printf("T%d(%u)\n", root->tree_num, root->freq);
   } else {
```

```
    printf("%c(%u)\n", root->data, root->freq);
  }

  printTree(root->left, space);
}

struct MinHeapNode* buildHuffmanTree(char* input) {
  struct MinHeapNode *left, *right, *top;
  struct MinHeap* minHeap = createInitialHeap(input);

  printf("\nInitial min heap with all characters:\n");
  printf("-------------------------------\n");
  printMinHeap(minHeap);

  int treeCount = 0;
  while (!isSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);
    printf("\nStep %d: Combine nodes with lowest frequencies\n", treeCount + 1);
    printf("Node 1: ");
    if (left->data == '$') {
      printf("T%d with frequency %u\n", left->tree_num, left->freq);
    } else {
      printf("'%c' with frequency %u\n", left->data, left->freq);
    }

    printf("Node 2: ");
    if (right->data == '$') {
      printf("T%d with frequency %u\n", right->tree_num, right->freq);
    } else {
      printf("'%c' with frequency %u\n", right->data, right->freq);
    }
    treeCount++;
    top = newNode('$', left->freq + right->freq, INT_MAX, treeCount);
    top->left = left;
    top->right = right;

    printf("\nTree T%d (Combined frequency: %u):\n", treeCount, top->freq);
    printTree(top, 0);

    insertMinHeap(minHeap, top);
    printf("\nCurrent Min Heap after insertion:\n");
    printf("--------------------------------\n");
    printMinHeap(minHeap);
  }
  printf("\nFinal Huffman Tree (T%d):\n", treeCount);
  struct MinHeapNode* result = extractMin(minHeap);
  printTree(result, 0);

  return result;
}

void printCodes(struct MinHeapNode* root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printCodes(root->right, arr, top + 1);
  }

  if (!(root->left) && !(root->right)) {
    printf("'%c': ", root->data);
    printArr(arr, top);
  }
}

void storeCodes(struct MinHeapNode* root, int arr[], int top, char* huffmanCode[]) {
  if (root->left) {
    arr[top] = 0;
    storeCodes(root->left, arr, top + 1, huffmanCode);
  }

  if (root->right) {
    arr[top] = 1;
    storeCodes(root->right, arr, top + 1, huffmanCode);
  }
```

```
    if (!(root->left) && !(root->right)) {
        huffmanCode[root->data] = (char*)malloc((top + 1) * sizeof(char));
        for (int i = 0; i < top; i++) {
            huffmanCode[root->data][i] = arr[i] + '0';
        }
        huffmanCode[root->data][top] = '\0';
    }
}

void compressString(char* input) {
    int len = strlen(input);
    int freq[MAX_CHARS] = {0};
    int* first_occurrences = sortByFirstOccurrence(input, &len);  // Reusing len variable
    for (int i = 0; i < strlen(input); i++) {
        freq[(unsigned char)input[i]]++;
    }
    printf("\nBuilding Huffman Tree Step by Step:\n");
    printf("==========================================\n");
    struct MinHeapNode* root = buildHuffmanTree(input);

    int arr[MAX_TREE_HT], top = 0;
    char* huffmanCode[MAX_CHARS] = {NULL};
    storeCodes(root, arr, top, huffmanCode);

    int compressedBits = 0;
    len = strlen(input);
    for (int i = 0; i < len; i++) {
        if (huffmanCode[(int)input[i]] != NULL) {
            compressedBits += strlen(huffmanCode[(int)input[i]]);
        }
    }
    printf("\nHuffman Codes:\n");
    for (int i = 0; i < MAX_CHARS; i++) {
        if (freq[i] > 0) {
            printf("'%c': %s\n", (char)i, huffmanCode[i]);
        }
    }
    for (int i = 0; i < MAX_CHARS; i++) {
        if (huffmanCode[i] != NULL) {
            free(huffmanCode[i]);
        }
    }
    free(first_occurrences);
}

int main() {
    char input[1000];
    int choice;

    do {
        printf("\n=== Huffman Coding Algorithm with Step-by-Step Tree Visualization ===\n");
        printf("1. Encode a string\n");
        printf("2. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar();
        switch (choice) {
            case 1:
                printf("Enter a string to encode: ");
                fgets(input, sizeof(input), stdin);

                size_t len = strlen(input);
                if (len > 0 && input[len-1] == '\n') {
                    input[len-1] = '\0';
                }
                if (strlen(input) > 0) {
                    compressString(input);
                } else {
                    printf("Please enter a valid string.\n");
                }
                break;
            case 2:
                printf("Exiting program...\n");
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 2);

    return 0;
}
```

OUTPUT:

```
=== Huffman Coding Algorithm with Step-by-Step Tree Visualization ===
1. Encode a string
2. Exit
Enter your choice: 1
Enter a string to encode: the cub wanted to rub on the tree branch

Building Huffman Tree Step by Step:
==================================

Initial min heap with all characters:
-------------------------------------
w   d   c   u   a   o   h   b   n   r   t   e
1   1   2   2   2   2   3   3   3   3   5   5   8

Step 1: Combine nodes with lowest frequencies
Node 1: 'w' with frequency 1
Node 2: 'd' with frequency 1

Tree T1 (Combined frequency: 2):

          d(1)

T1(2)

          w(1)

Current Min Heap after insertion:
------------------------------
c   u   a   o   T1  h   b   n   r   t   e
2   2   2   2   2   3   3   3   3   5   5   8

Step 2: Combine nodes with lowest frequencies
Node 1: 'c' with frequency 2
Node 2: 'u' with frequency 2

Tree T2 (Combined frequency: 4):

          u(2)

T2(4)

          c(2)

Current Min Heap after insertion:
------------------------------
a   o   T1  h   b   n   r   T2  t   e
2   2   2   3   3   3   3   4   5   5   8
```

```
Step 3: Combine nodes with lowest frequencies
Node 1: 'a' with frequency 2
Node 2: 'o' with frequency 2

Tree T3 (Combined frequency: 4):

          o(2)

T3(4)

          a(2)

Current Min Heap after insertion:
------------------------------
T1  h   b   n   r   T2  T3  t   e
2   3   3   3   3   4   4   5   5   8

Step 4: Combine nodes with lowest frequencies
Node 1: T1 with frequency 2
Node 2: 'h' with frequency 3

Tree T4 (Combined frequency: 5):

          h(3)

T4(5)

                    d(1)

          T1(2)

                    w(1)

Current Min Heap after insertion:
------------------------------
b   n   r   T2  T3  t   e   T4
3   3   3   4   4   5   5   5   8

Step 5: Combine nodes with lowest frequencies
Node 1: 'b' with frequency 3
Node 2: 'n' with frequency 3

Tree T5 (Combined frequency: 6):
```

```
Tree T5 (Combined frequency: 6):

          n(3)

T5(6)

          b(3)

Current Min Heap after insertion:
------------------------------
r   T2  T3  t   e   T4  T5
3   4   4   5   5   5   6   8

Step 6: Combine nodes with lowest frequencies
Node 1: 'r' with frequency 3
Node 2: T2 with frequency 4

Tree T6 (Combined frequency: 7):

                    u(2)

          T2(4)

                    c(2)

T6(7)

          r(3)

Current Min Heap after insertion:
------------------------------
T3  t   e   T4  T5  T6
4   5   5   5   6   7   8

Step 7: Combine nodes with lowest frequencies
Node 1: T3 with frequency 4
Node 2: 't' with frequency 5

Tree T7 (Combined frequency: 9):
```

```
          t(5)

T7(9)

                    o(2)

          T3(4)

                    a(2)

Current Min Heap after insertion:
------------------------------
e   T4  T5  T6      T7
5   5   6   7   8   9

Step 8: Combine nodes with lowest frequencies
Node 1: 'e' with frequency 5
Node 2: T4 with frequency 5

Tree T8 (Combined frequency: 10):

                    h(3)

          T4(5)

                              d(1)

                    T1(2)

                              w(1)

T8(10)

          e(5)

Current Min Heap after insertion:
------------------------------
T5  T6      T7  T8
6   7   8   9   10

Step 9: Combine nodes with lowest frequencies
Node 1: T5 with frequency 6
Node 2: T6 with frequency 7

Tree T9 (Combined frequency: 13):
```

```
                              u(2)

                    T2(4)

                              c(2)

          T6(7)

                    r(3)

T9(13)

                    n(3)

          T5(6)

                    b(3)

Current Min Heap after insertion:
------------------------------
    T7  T8  T9
8   9   10  13

Step 10: Combine nodes with lowest frequencies
Node 1: ' ' with frequency 8
Node 2: T7 with frequency 9

Tree T10 (Combined frequency: 17):

                    t(5)

          T7(9)

                              o(2)

                    T3(4)

                              a(2)

T10(17)

           (8)

Current Min Heap after insertion:
------------------------------
T8  T9  T10
10  13  17

Step 11: Combine nodes with lowest frequencies
Node 1: T8 with frequency 10
Node 2: T9 with frequency 13
```

```
Tree T11 (Combined frequency: 23):

                                        u(2)

                              T2(4)

                                        c(2)

                    T6(7)

                              r(3)

          T9(13)

                              n(3)

                    T5(6)

                              b(3)

T11(23)

                              h(3)

                    T4(5)

                                        d(1)

                              T1(2)

                                        w(1)

          T8(10)

                    e(5)

Current Min Heap after insertion:
------------------------------
T10 T11
17  23

Step 12: Combine nodes with lowest frequencies
Node 1: T10 with frequency 17
Node 2: T11 with frequency 23
```

```
Tree T12 (Combined frequency: 40):
                                                  u(2)
                                        T2(4)
                                                  c(2)
                              T6(7)
                                                  r(3)
                    T9(13)
                                                  n(3)
                              T5(6)
                                                  b(3)
          T11(23)
                                                  h(3)
                              T4(5)
                                                  d(1)
                                        T1(2)
                                                  w(1)
                    T8(10)
                                                  e(5)
T12(40)
                              t(5)
                    T7(9)
                                                  o(2)
                              T3(4)
                                                  a(2)
          T10(17)
                     (8)

Current Min Heap after insertion:
------------------------------
T12
40
```

```
Final Huffman Tree (T12):

                                        u(2)
                                T2(4)
                                        c(2)
                        T6(7)
                                        r(3)
                T9(13)
                                        n(3)
                        T5(6)
                                        b(3)
        T11(23)
                                        h(3)
                        T4(5)
                                        d(1)
                                T1(2)
                                        w(1)
                T8(10)
                        e(5)
T12(40)
                        t(5)
                T7(9)
                                        o(2)
                        T3(4)
                                        a(2)
        T10(17)
                 (8)

Huffman Codes:
' ': 00
'a': 0100
'b': 1100
'c': 11110
'd': 10101
'e': 100
'h': 1011
'n': 1101
'o': 0101
'r': 1110
't': 011
'u': 11111
'w': 10100

=== Huffman Coding Algorithm with Step-by-Step Tree Visualization ===
1. Encode a string
2. Exit
Enter your choice: 2
Exiting program...
```

Conclusion: Huffman encoding algorithm was implemented successfully in C .