Department of Computer Science & Engineering

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES
DELHI

योगः कर्मसु कौशलम्

IN PURSUIT OF PERFECTION

DESIGN AND ANALYSIS OF ALGORITHM

LAB REPORT

(CIC-359)

FOR

BACHELOR OF TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING

5TH Semester

2024-25

Submitted to:

Submitted by:

Program -1: To implement following algorithm using array as a data structure and analyse its time complexity. A) Insertion sort B) Selection sort C) Bubble sort D) Quick sort E) Heap sort F) Merge sort

A) INSERTION SORT

Theory-

Programming Code-

main.c

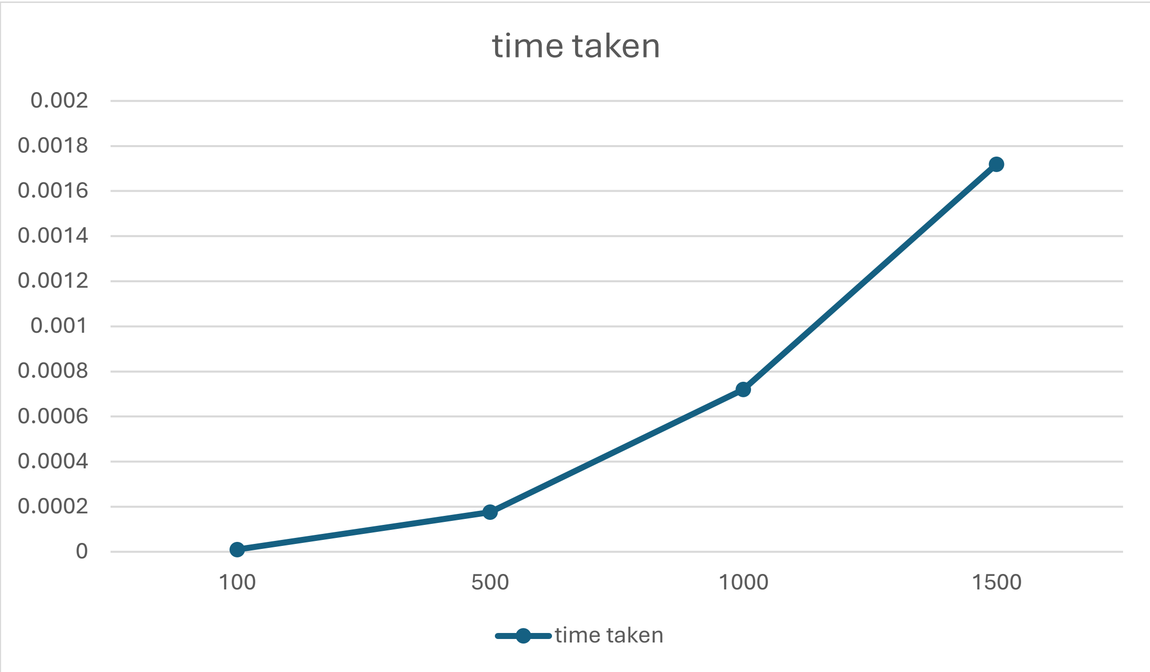
Share Run

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// insertion sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// random array
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000;
    }
}

// measure sorting time
void measureSortingTime(int arr[], int n) {
    clock_t start, end;
    start = clock();

    insertionSort(arr, n);

    end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);
}

int main() {
    srand(time(0));

    int sizes[] = {100, 500, 1000, 1500};
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int i = 0; i < numSizes; i++) {
        int n = sizes[i];
        int arr[n];
        generateRandomArray(arr, n);
        measureSortingTime(arr, n);
    }

    return 0;
}
```

Output-

```
Output                                                    Clear
/tmp/lOWDJ6qdzt.o
Time taken to sort 100 elements: 0.000010 seconds
Time taken to sort 500 elements: 0.000175 seconds
Time taken to sort 1000 elements: 0.000720 seconds
Time taken to sort 1500 elements: 0.001719 seconds


=== Code Execution Successful ===
```

Graph-



Learning Outcome-

B) SELECTION SORT

Theory-

Programming Code-

main.c

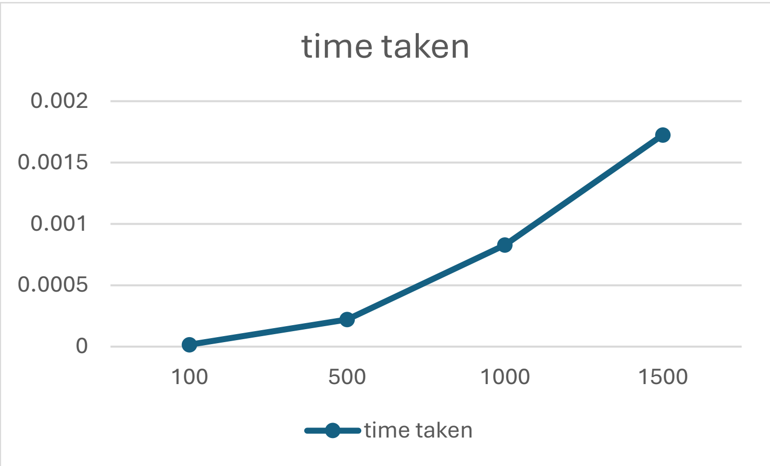
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// swap two elements
void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;

    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}

void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// array of random numbers
void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 10000;
    }
}

// measure the time taken
void measureTime(int arr[], int size) {
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    selectionSort(arr, size);
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Time taken to sort %d elements: %f seconds\n", size, cpu_time_used
        );
}

```

```c
int main() {
    srand(time(0));


    int sizes[] = {100, 500, 1000, 1500};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int i = 0; i < num_sizes; i++) {
        int size = sizes[i];
        int arr[size];

        generateRandomArray(arr, size);

        printf("Sorting array of size %d -\n", size);
        measureTime(arr, size);
    }

    return 0;
}
```

Output-

```
Output                                                    Clear
/tmp/x6pdZ4nWk1.o
Sorting array of size 100 -
Time taken to sort 100 elements: 0.000016 seconds
Sorting array of size 500 -
Time taken to sort 500 elements: 0.000220 seconds
Sorting array of size 1000 -
Time taken to sort 1000 elements: 0.000828 seconds
Sorting array of size 1500 -
Time taken to sort 1500 elements: 0.001725 seconds
```

Graph-



Learning Outcome-

C) BUBBLE SORT

Theory-

Programming Code-

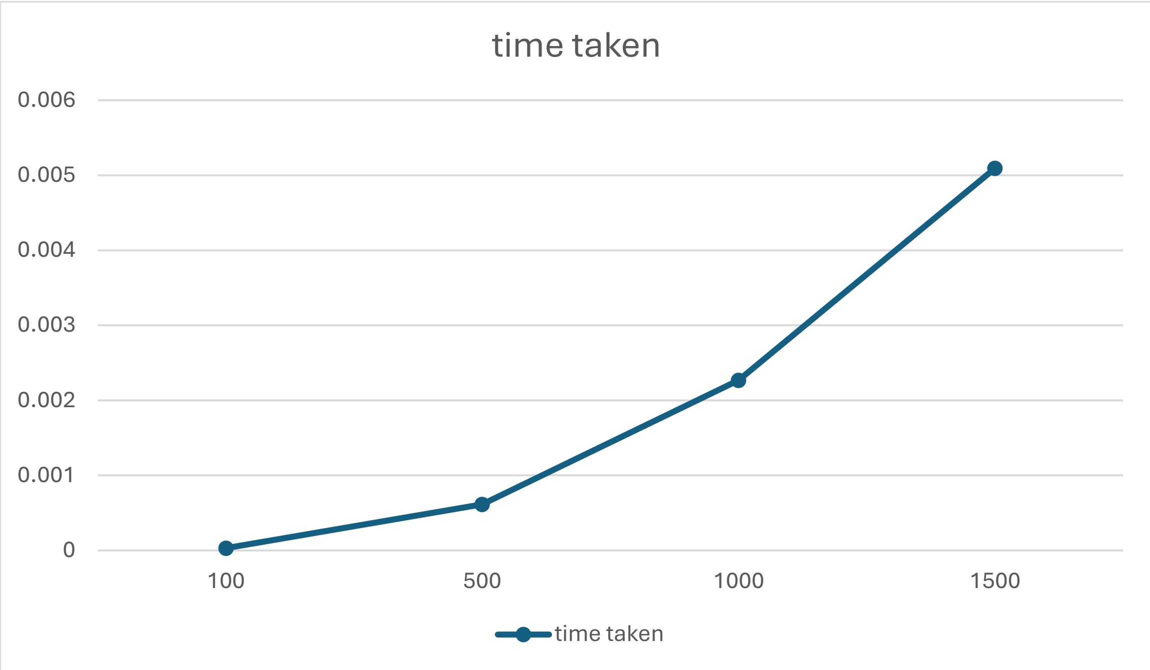
main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

// array with random elements
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int sizes[] = {100, 500, 1000, 1500};
    int nSizes = sizeof(sizes)/sizeof(sizes[0]);

    srand(time(0));

    for (int i = 0; i < nSizes; i++) {
        int n = sizes[i];
        int arr[n];

        generateRandomArray(arr, n);

        clock_t start = clock();
        bubbleSort(arr, n);
        clock_t end = clock();

        double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
        printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

    }

    return 0;
}
```

Output-

```
Output                                                     Clear
/tmp/S8bLl1ztJM.o
Time taken to sort 100 elements: 0.000031 seconds
Time taken to sort 500 elements: 0.000613 seconds
Time taken to sort 1000 elements: 0.002266 seconds
Time taken to sort 1500 elements: 0.005092 seconds


=== Code Execution Successful ===
```

Graph-



Learning Outcome-

D) QUICK SORT

Theory-

Programming Code-

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % (size * 10);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int sizes[] = {100, 500, 1000, 1500};
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    srand(time(NULL));
    struct timespec start, end;
    for (int i = 0; i < numSizes; ++i) {
        int size = sizes[i];
        int *arr = (int *)malloc(size * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed\n");
            return 1;
        }

        generateRandomArray(arr, size);
        clock_gettime(CLOCK_MONOTONIC, &start);
        quickSort(arr, 0, size - 1);
        clock_gettime(CLOCK_MONOTONIC, &end);

        long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
        printf("Size: %d, Time taken: %ld nanoseconds\n", size, elapsed_ns);

        free(arr);
    }

    return 0;
}
```

Output-

```
/tmp/eA8YHtUdtx.o
Size: 100, Time taken: 6360 nanoseconds
Size: 500, Time taken: 49500 nanoseconds
Size: 1000, Time taken: 87960 nanoseconds
Size: 1500, Time taken: 141100 nanoseconds


=== Code Execution Successful ===
```

Graph-

Learning Outcome-

E) HEAP SORT

Theory-

Programming Code-

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] >
        arr[largest])
        largest = left;
    if (right < n && arr[right] >
        arr[largest])
        largest = right;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    // maxheap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}

void generateRandomArray(int arr[], int
    size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % (size * 10);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int sizes[] = {100, 500, 1000, 1500};
    int numSizes = sizeof(sizes) / sizeof
        (sizes[0]);

    srand(time(NULL));
    struct timespec start, end;

    for (int i = 0; i < numSizes; ++i) {
        int size = sizes[i];
        int *arr = (int *)malloc(size *
            sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation
                failed\n");
            return 1;
        }

        generateRandomArray(arr, size);
        clock_gettime(CLOCK_MONOTONIC,
            &start);
        heapSort(arr, size);
        clock_gettime(CLOCK_MONOTONIC,
            &end);

        long elapsed_ns = (end.tv_sec -
            start.tv_sec) * 1000000000L +
            (end.tv_nsec - start.tv_nsec);
        printf("Size: %d, Time taken: %ld
            nanoseconds\n", size,
            elapsed_ns);
        free(arr);
    }
    return 0;
}
```

Output-

```
/tmp/wYowObEMuh.o
Size: 100, Time taken: 12211 nanoseconds
Size: 500, Time taken: 78120 nanoseconds
Size: 1000, Time taken: 196709 nanoseconds
Size: 1500, Time taken: 275149 nanoseconds


=== Code Execution Successful ===
```

Graph-

Learning Outcome-

F) MERGE SORT-

Theory-

Programming Code-

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    if (L == NULL || R == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0;
    int j = 0;
    int k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) {
        arr[k++] = L[i++];
    }
    while (j < n2) {
        arr[k++] = R[j++];
    }

    free(L);
    free(R);
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % (size * 10);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int sizes[] = {100, 500, 1000, 1500};
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);
    srand(time(NULL));
    struct timespec start, end;

    for (int i = 0; i < numSizes; ++i) {
        int size = sizes[i];
        int *arr = (int *)malloc(size * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed\n");
            return 1;
        }

        generateRandomArray(arr, size);
        clock_gettime(CLOCK_MONOTONIC, &start);
        mergeSort(arr, 0, size - 1);
        clock_gettime(CLOCK_MONOTONIC, &end);

        long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
        printf("Size: %d, Time taken: %ld nanoseconds\n", size, elapsed_ns);
        free(arr);
    }
    return 0;
}
```

Output-

```
/tmp/ziualxTp43.o
Size: 100, Time taken: 15580 nanoseconds
Size: 500, Time taken: 78540 nanoseconds
Size: 1000, Time taken: 164030 nanoseconds
Size: 1500, Time taken: 329700 nanoseconds


=== Code Execution Successful ===
```

Graph-

Learning Outcome-

Program-2: To implement Linear Search and Binary Search and analyse its time complexity.

LINEAR SEARCH

Theory-

Programming Code-

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; ++i) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % (size * 10);
    }
}

int main() {
    int sizes[] = {100, 500, 1000, 1500};
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    srand(time(NULL));
    for (int i = 0; i < numSizes; ++i) {
        int size = sizes[i];
        int *arr = (int *)malloc(size * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed\n");
            return 1;
        }
        generateRandomArray(arr, size);
        int target = rand() % (size * 10);

        clock_t start = clock();
        linearSearch(arr, size, target);
        clock_t end = clock();

        double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
        printf("Size: %d, Time taken: %f seconds\n", size, time_taken);
        free(arr);
    }
    return 0;
}
```

Output-

```
/tmp/EsJNFE9aeV.o
Size: 100, Time taken: 0.000001 seconds
Size: 500, Time taken: 0.000001 seconds
Size: 1000, Time taken: 0.000002 seconds
Size: 1500, Time taken: 0.000003 seconds


=== Code Execution Successful ===
```

Graph-

Learning Outcome-

BINARY SEARCH

Theory-

Programming Code-

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; ++i) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % (size * 10);
    }
}

int main() {
    int sizes[] = {100, 500, 1000, 1500};
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    srand(time(NULL));
    struct timespec start, end;

    for (int i = 0; i < numSizes; ++i) {
        int size = sizes[i];
        int *arr = (int *)malloc(size * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed\n");
            return 1;
        }
        generateRandomArray(arr, size);
        int target = rand() % (size * 10);

        clock_gettime(CLOCK_MONOTONIC, &start);
        linearSearch(arr, size, target);
        clock_gettime(CLOCK_MONOTONIC, &end);

        long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
        printf("Size: %d, Time taken: %ld nanoseconds\n", size, elapsed_ns);

        free(arr);
    }

    return 0;
```

Output-

```
/tmp/mMYpdxLq6l.o
Size: 100, Time taken: 300 nanoseconds
Size: 500, Time taken: 640 nanoseconds
Size: 1000, Time taken: 1170 nanoseconds
Size: 1500, Time taken: 1760 nanoseconds


=== Code Execution Successful ===
```

Graph-

Learning Outcome-

Program 3: To implement Huffman Coding and analyse its complexity.

Theory:

Programming Code:

main.cpp

```cpp
#include <iostream>
#include <queue>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <map>

using namespace std;

struct MinHeapNode
{
    char data;
    int freq;
    MinHeapNode *left, *right;

    MinHeapNode(char data, int freq)
    {
        left = right = nullptr;
        this->data = data;
        this->freq = freq;
    }
};


struct compare
{
    bool operator()(MinHeapNode* l, MinHeapNode* r)
    {
        return (l->freq > r->freq);
    }
};


void printCodes(struct MinHeapNode* root, string str, map<char, string>&
    huffmanCodes)
{
    if (!root)
        return;

    if (root->data != '$')
        huffmanCodes[root->data] = str;

    printCodes(root->left, str + "0", huffmanCodes);
    printCodes(root->right, str + "1", huffmanCodes);
}


void HuffmanCoding(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;


    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

```

```cpp
    for (int i = 0; i < size; i++)
        minHeap.push(new MinHeapNode(data[i], freq[i]));


    while (minHeap.size() != 1)
    {

        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();


        top = new MinHeapNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        minHeap.push(top);
    }


    map<char, string> huffmanCodes;
    printCodes(minHeap.top(), "", huffmanCodes);

    cout << "\nHuffman Codes:\n";
    for (auto pair : huffmanCodes)
        cout << pair.first << ": " << pair.second << "\n";
}

```

```cpp
    map<char, string> huffmanCodes;
    printCodes(minHeap.top(), "", huffmanCodes);

    cout << "\nHuffman Codes:\n";
    for (auto pair : huffmanCodes)
        cout << pair.first << ": " << pair.second << "\n";
}

int main()
{

    char characters[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };
    int size = sizeof(characters) / sizeof(characters[0]);

    auto start = chrono::high_resolution_clock::now();


    HuffmanCoding(characters, freq, size);

    auto end = chrono::high_resolution_clock::now();


    chrono::duration<double, nano> duration = end - start;
    cout << "\nTime taken: " << duration.count() << " nanoseconds" << endl;

    return 0;
}
```

Output:

```
Output                                                        Clear
/tmp/4LvD8Z4H2r.o

Huffman Codes:
a: 1100
b: 1101
c: 100
d: 101
e: 111
f: 0

Time taken: 132769 nanoseconds


=== Code Execution Successful ===
```

Learning Outcomes:

Program 4: To implement Minimum Spanning Tree and analyse its complexity.

Theory:

Programming Code;

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

using namespace std;


struct Edge
{
    int src, dest, weight;
};


struct Subset
{
    int parent;
    int rank;
};


int find(Subset subsets[], int i)
{
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Union(Subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}


bool compare(Edge a, Edge b)
{
    return a.weight < b.weight;
}


void KruskalMST(vector<Edge>& edges, int V)
{
    vector<Edge> result;
    int e = 0;
    int i = 0;

```

```cpp

    sort(edges.begin(), edges.end(), compare);


    Subset* subsets = new Subset[V];
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }


    while (e < V - 1 && i < edges.size())
    {

        Edge nextEdge = edges[i++];

        int x = find(subsets, nextEdge.src);
        int y = find(subsets, nextEdge.dest);


        if (x != y)
        {
            result.push_back(nextEdge);
            Union(subsets, x, y);
            e++;
        }
    }

    cout << "Edges in the Minimum Spanning Tree:\n";
    for (auto& edge : result)
        cout << edge.src << " -- " << edge.dest << " == " << edge.weight << endl;

    delete[] subsets;
}

int main()
{
    int V = 4;
    vector<Edge> edges = {
        {0, 1, 10}, {0, 2, 6}, {0, 3, 5},
        {1, 3, 15}, {2, 3, 4}
    };

    auto start = chrono::high_resolution_clock::now();


    KruskalMST(edges, V);

    auto end = chrono::high_resolution_clock::now();


    chrono::duration<double, nano> duration = end - start;
    cout << "Time taken: " << duration.count() << " nanoseconds" << endl;

    return 0;
}

```

Output:

```
Output                                                    Clear
/tmp/Jv1ttcyr3b.o
Edges in the Minimum Spanning Tree:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Time taken: 54400 nanoseconds


=== Code Execution Successful ===
```

Learning Outcomes:

Practical 5

To implement Matrix Multiplication and analyse its time complexity.

Theory:

Programming Code:

```
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <stdlib.h>

int MatrixChainOrder(int p[], int n) {
    int m[n][n];

    for (int i = 1; i < n; i++)
        m[i][i] = 0;

    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            m[i][j] = INT_MAX;
            for (int k = i; k <= j - 1; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
    return m[1][n - 1];
}

void generateRandomDimensions(int *p, int n) {
    for (int i = 0; i < n; i++) {
        p[i] = rand() % 100 + 1;
    }
}

int main() {
```

```
    srand(time(0));

    int sizes[] = {5, 10, 15, 20, 25};
    int numTests = sizeof(sizes) / sizeof(sizes[0]);

    printf("Matrix Chain Multiplication Execution Time Analysis:\n");
    printf("Size\tMin. Multiplications\tExecution Time (ms)\n");

    for (int i = 0; i < numTests; i++) {
        int n = sizes[i];
        int p[n + 1];
        generateRandomDimensions(p, n + 1);

        clock_t start = clock();
        int minMultiplications = MatrixChainOrder(p, n + 1);
        clock_t end = clock();

        double time_taken = ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;

        printf("%d\t\t\t\t%d\t\t\t\t%.3f\n", n, minMultiplications, time_taken);
    }

    return 0;
}
```

OUTPUT

```
Matrix Chain Multiplication Execution Time Analysis:
Size    Min. Multiplications    Execution Time (ms)
5               5488                    0.015
10              281170                  0.004
15              87198                   0.008
20              595504                  0.012
25              188046                  0.022


=== Code Execution Successful ===
```

Learning Outcome:

Practical 6

Aim: To implement Dijkstra‘s algorithm and analyse its time complexity.

Theory:

Programming Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>

#define INF INT_MAX
#define MAX_VERTICES 1000

typedef struct {
    int vertex;
    int distance;
} PQElement;

int minDistance(int dist[], int visited[], int n) {
    int min = INF;
    int minIndex = -1;
    for (int v = 0; v < n; v++) {
        if (!visited[v] && dist[v] < min) {
            min = dist[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int n, int source) {
    int dist[MAX_VERTICES];
    int visited[MAX_VERTICES] = {0};

    for (int i = 0; i < n; i++) {
        dist[i] = INF;
    }
```

```c
    dist[source] = 0;

    for (int count = 0; count < n - 1; count++) {
        // Pick the vertex with the minimum distance that hasn't been visited
        int u = minDistance(dist, visited, n);
        visited[u] = 1;

        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] != INF && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printf("Shortest distances from source vertex %d:\n", source);
    printf("Vertex \t\t\tDistance from Source\n");
    for (int i = 0; i < n; i++) {
        if (dist[i] == INF) {
            printf("Vertex %d\t\tINF\n", i);
        } else {
            printf("Vertex %d\t\t\t %d\n", i, dist[i]);
        }
    }
}

void generateGraph(int graph[MAX_VERTICES][MAX_VERTICES], int n, int maxWeight) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                graph[i][j] = rand() % maxWeight + 1;
            } else {
                graph[i][j] = INF;
            }
        }
    }
}

void analyzeTimeComplexity(int n) {
    int graph[MAX_VERTICES][MAX_VERTICES];

    generateGraph(graph, n, 100);

    clock_t start_time = clock();
    dijkstra(graph, n, 0);
    clock_t end_time = clock();

    double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Execution time for graph size %d: %f seconds\n\n\n", n, execution_time);
```

```
}

int main() {
    int sizes[] = {10,20,30};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        analyzeTimeComplexity(n);
    }

    return 0;
}
```

Output

```
Shortest distances from source vertex 0:
Vertex          Distance from Source
Vertex 0             0
Vertex 1             65
Vertex 2             28
Vertex 3             40
Vertex 4             16
Vertex 5             43
Vertex 6             36
Vertex 7             38
Vertex 8             36
Vertex 9             50
Execution time for graph size 10: 0.000163 seconds


Shortest distances from source vertex 0:
Vertex          Distance from Source
Vertex 0             0
Vertex 1             9
Vertex 2             20
Vertex 3             13
Vertex 4             11
Vertex 5             31
Vertex 6             13
Vertex 7             7
Vertex 8             18
Vertex 9             11
Vertex 10            12
Vertex 11            17
Vertex 12            26
Vertex 13            8
Vertex 14            24
Vertex 15            12
Vertex 16            2
Vertex 17            6
```

```
Vertex 18           3
Vertex 19           15
Execution time for graph size 20: 0.000130 seconds


Shortest distances from source vertex 0:
Vertex          Distance from Source
Vertex 0            0
Vertex 1            8
Vertex 2            14
Vertex 3            5
Vertex 4            15
Vertex 5            10
Vertex 6            21
Vertex 7            5
Vertex 8            11
Vertex 9            18
Vertex 10           5
Vertex 11           17
Vertex 12           1
Vertex 13           20
Vertex 14           11
Vertex 15           22
Vertex 16           14
Vertex 17           8
Vertex 18           12
Vertex 19           18
Vertex 20           22
Vertex 21           11
Vertex 22           14
Vertex 23           2
Vertex 24           16
Vertex 25           11
Vertex 26           9
Vertex 27           19
Vertex 28           16
Vertex 29           20
Execution time for graph size 30: 0.000197 seconds



=== Code Execution Successful ===
```

Learning Outcome:

Practical 7

To implement Bellman Ford algorithm and analyse its time complexity.

Theory:

Programming Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>

#define INF INT_MAX
#define MAX_VERTICES 1000

typedef struct {
    int u, v, weight;
} Edge;

void bellmanFord(Edge edges[], int V, int E, int source) {
    int dist[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INF;
    }
    dist[source] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j].u;
            int v = edges[j].v;
            int weight = edges[j].weight;
            if (dist[u] != INF && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    for (int i = 0; i < E; i++) {
```

```
        int u = edges[i].u;
        int v = edges[i].v;
        int weight = edges[i].weight;
        if (dist[u] != INF && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle\n");
            return;
        }
    }

    printf("Shortest distances from source vertex %d:\n", source);
    printf("Vertex \t\t\tDistance from Source\n");
    for (int i = 0; i < V; i++) {
        if (dist[i] == INF) {
            printf("Vertex %d\t\t\tINF\n", i);
        } else {
            printf("Vertex %d\t\t\t %d\n", i, dist[i]);
        }
    }

}

void generateGraph(Edge edges[], int V, int E, int maxWeight) {
    for (int i = 0; i < E; i++) {
        int u = rand() % V;
        int v = rand() % V;
        while (u == v) {
            v = rand() % V;
        }
        int weight = rand() % maxWeight + 1;
        edges[i] = (Edge){u, v, weight};
    }
}

void analyzeTimeComplexity(int V, int E) {
    Edge edges[E];

    generateGraph(edges, V, E, 100);

    clock_t start_time = clock();
    bellmanFord(edges, V, E, 0);
    clock_t end_time = clock();

    double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Execution time for graph size V=%d, E=%d: %f seconds\n\n\n", V, E,
        execution_time);

}

int main() {
    int sizes[][2] = {{10, 30}, {15, 200}, {20, 500}};
```

```c
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int i = 0; i < num_sizes; i++) {
        int V = sizes[i][0];
        int E = sizes[i][1];
        analyzeTimeComplexity(V, E);
    }

    return 0;
}
```

OUTPUT

```
Shortest distances from source vertex 0:
Vertex          Distance from Source
Vertex 0             0
Vertex 1             163
Vertex 2             87
Vertex 3             122
Vertex 4             157
Vertex 5             129
Vertex 6             82
Vertex 7             158
Vertex 8             183
Vertex 9             64
Execution time for graph size V=10, E=30: 0.000157 seconds


Shortest distances from source vertex 0:
Vertex          Distance from Source
Vertex 0             0
Vertex 1             46
Vertex 2             31
Vertex 3             35
Vertex 4             35
Vertex 5             15
Vertex 6             40
Vertex 7             41
Vertex 8             31
Vertex 9             32
Vertex 10            29
Vertex 11            34
Vertex 12            24
Vertex 13            57
```

```
Vertex 14           25
Execution time for graph size V=15, E=200: 0.000083 seconds


Shortest distances from source vertex 0:
Vertex          Distance from Source
Vertex 0            0
Vertex 1            3
Vertex 2            12
Vertex 3            4
Vertex 4            9
Vertex 5            16
Vertex 6            16
Vertex 7            15
Vertex 8            35
Vertex 9            13
Vertex 10           3
Vertex 11           17
Vertex 12           10
Vertex 13           18
Vertex 14           15
Vertex 15           25
Vertex 16           29
Vertex 17           21
Vertex 18           14
Vertex 19           8
Execution time for graph size V=20, E=500: 0.000135 seconds




=== Code Execution Successful ===
```

Learning Outcome:

Practical 8

Implement N Queen's problem using Back Tracking.

Theory:

Programming Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_N 20

void printSolution(int board[MAX_N][MAX_N], int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
}

int isSafe(int board[MAX_N][MAX_N], int row, int col, int N) {
    // Check this column on the upper side
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1) {
            return 0;
        }
    }

    // Check upper diagonal on the left side
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return 0;
        }
    }

    // Check upper diagonal on the right side
    for (int i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++) {
```

```c
            if (board[i][j] == 1) {
                return 0;
            }
        }

        return 1;
}

int solveNQueens(int board[MAX_N][MAX_N], int row, int N) {

    if (row >= N) {
        return 1;
    }

    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col, N)) {

            board[row][col] = 1;

            if (solveNQueens(board, row + 1, N)) {
                return 1;
            }

            board[row][col] = 0;
        }
    }

    return 0;
}

void analyzeTimeComplexity(int N) {
    int board[MAX_N][MAX_N] = {0};
```

```c

    clock_t start_time = clock();
    if (solveNQueens(board, 0, N)) {
        printf("Solution for N=%d found:\n", N);
        printSolution(board, N);
    } else {
        printf("No solution exists for N=%d\n", N);
    }
    clock_t end_time = clock();

    double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Execution time for N=%d: %f seconds\n", N, execution_time);
}

int main() {
    int sizes[] = {4, 8, 10};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int i = 0; i < num_sizes; i++) {
        int N = sizes[i];
        analyzeTimeComplexity(N);
    }

    return 0;
}

```

OUTPUT

```
Solution for N=4 found:
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
Execution time for N=4: 0.000079 seconds
Solution for N=8 found:
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
Execution time for N=8: 0.000069 seconds
Solution for N=10 found:
1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
Execution time for N=10: 0.000108 seconds


=== Code Execution Successful ===
```

Learning Outcome:

Practical 9

To implement Longest Common Subsequence problem and analyse its time complexity.

Theory:

Programming Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int LCS(char* X, char* Y, int m, int n) {
    int dp[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = (dp[i - 1][j] > dp[i][j - 1]) ? dp[i - 1][j] : dp[i][j - 1]; // If no match, take the max
            }
        }
    }

    return dp[m][n];
}

void analyzeTimeComplexity(char* X, char* Y) {
    int m = strlen(X);
    int n = strlen(Y);

    clock_t start_time = clock();

    int lcs_length = LCS(X, Y, m, n);

```

```
    clock_t end_time = clock();

    double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    printf("Length of LCS: %d\n", lcs_length);
    printf("Execution time for strings of length %d and %d: %f seconds\n", m, n,
        execution_time);
}

int main() {
    char* strings[] = {
        "AGGTAB",   // Length 6
        "ABCDEFGH", // Length 8
        "ABCDE",    // Length 5
        "AAGTCAG",  // Length 7

    };

    int num_strings = sizeof(strings) / sizeof(strings[0]);

    for (int i = 0; i < num_strings - 1; i++) {
        for (int j = i + 1; j < num_strings; j++) {
            printf("\nComparing string %d and string %d:\n", i+1, j+1);
            analyzeTimeComplexity(strings[i], strings[j]);
        }
    }

    return 0;
}

```

OUTPUT

```
Comparing string 1 and string 2:
Length of LCS: 2
Execution time for strings of length 6 and 8: 0.000002 seconds

Comparing string 1 and string 3:
Length of LCS: 2
Execution time for strings of length 6 and 5: 0.000001 seconds

Comparing string 1 and string 4:
Length of LCS: 4
Execution time for strings of length 6 and 7: 0.000002 seconds
```

```
Comparing string 2 and string 3:
Length of LCS: 5
Execution time for strings of length 8 and 5: 0.000001 seconds

Comparing string 2 and string 4:
Length of LCS: 3
Execution time for strings of length 8 and 7: 0.000002 seconds

Comparing string 3 and string 4:
Length of LCS: 2
Execution time for strings of length 5 and 7: 0.000002 seconds


=== Code Execution Successful ===
```

Learning Outcome:

Practical 10

To implement naïve String Matching algorithm, Rabin Karp algorithm and Knuth Morris Pratt algo rithm and analyse its time complexity.

Theory:

Programming Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// Naïve String Matching Algorithm
int naiveStringMatch(char* text, char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int count = 0;
    for (int i = 0; i <= n - m; i++) {
        int j = 0;
        while (j < m && text[i + j] == pattern[j]) {
            j++;
        }
        if (j == m) {
            count++;  // Found a match
        }
    }
    return count;
}

// Rabin-Karp Algorithm
#define d 256
#define q 101

int rabinKarpStringMatch(char* text, char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int count = 0;

    int patternHash = 0;
```

```
    int textHash = 0;
    int h = 1;

    for (int i = 0; i < m - 1; i++) {
        h = (h * d) % q;
    }

    // Calculate the hash value for the pattern and the first window of text
    for (int i = 0; i < m; i++) {
        patternHash = (d * patternHash + pattern[i]) % q;
        textHash = (d * textHash + text[i]) % q;
    }

    // Slide the pattern over the text one by one
    for (int i = 0; i <= n - m; i++) {
        if (patternHash == textHash) {
            int j = 0;
            while (j < m && text[i + j] == pattern[j]) {
                j++;
            }
            if (j == m) {
                count++;  // Found a match
            }
        }

        // Calculate hash value for the next window of text
        if (i < n - m) {
            textHash = (d * (textHash - text[i] * h) + text[i + m]) % q;
            if (textHash < 0) {
                textHash = (textHash + q);
            }
        }
    }
    return count;
}

// KMP Algorithm (Knuth-Morris-Pratt)
void computeLPSArray(char* pattern, int m, int* lps) {
    int length = 0;
    lps[0] = 0;
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
```

```
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int kmpStringMatch(char* text, char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int lps[m];  // Longest Prefix Suffix array
    computeLPSArray(pattern, m, lps);

    int i = 0;
    int j = 0;
    int count = 0;

    while (i < n) {
        if (pattern[j] == text[i]) {
            j++;
            i++;
        }

        if (j == m) {
            count++;  // Found a match
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return count;
}

void analyzeTimeComplexity(char* text, char* pattern) {
    clock_t start_time, end_time;
    double execution_time;

    // Measure execution time for Naïve String Matching
    start_time = clock();
    int naive_count = naiveStringMatch(text, pattern);
    end_time = clock();
```

```
    execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Naïve String Match - Matches: %d, Time: %f seconds\n", naive_count,
        execution_time);

    // Measure execution time for Rabin-Karp Algorithm
    start_time = clock();
    int rabin_karp_count = rabinKarpStringMatch(text, pattern);
    end_time = clock();
    execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Rabin-Karp String Match - Matches: %d, Time: %f seconds\n",
        rabin_karp_count, execution_time);

    // Measure execution time for KMP Algorithm
    start_time = clock();
    int kmp_count = kmpStringMatch(text, pattern);
    end_time = clock();
    execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("KMP String Match - Matches: %d, Time: %f seconds\n", kmp_count,
        execution_time);
}

int main() {
    char text[] = "ABABDABACDABABCABAB";
    char pattern[] = "ABAB";

    analyzeTimeComplexity(text, pattern);

    return 0;
}
```

OUTPUT

```
Naïve String Match - Matches: 3, Time: 0.000003 seconds
Rabin-Karp String Match - Matches: 3, Time: 0.000001 seconds
KMP String Match - Matches: 3, Time: 0.000001 seconds


=== Code Execution Successful ===
```

Learning Outcome:

Practical 11

To implement Sorting Network.

Theory:

Programming Code:

```c
#include <stdio.h>

#define N 8

void swap(int *a, int *b) {
    if (*a > *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
    }
}

void bitonicMerge(int arr[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;

        for (int i = low; i < low + k; i++) {
            swap(&arr[i], &arr[i + k]);
        }

        // Recursively merge the two halves
        bitonicMerge(arr, low, k, dir);
        bitonicMerge(arr, low + k, k, dir);
    }
}

void bitonicSort(int arr[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;

        bitonicSort(arr, low, k, 1);

```

```c
            bitonicSort(arr, low + k, k, 0);

            bitonicMerge(arr, low, cnt, dir);
        }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[N] = { 3, 7, 8, 5, 2, 1, 6, 4 };

    printf("Original Array:\n");
    printArray(arr, N);

    bitonicSort(arr, 0, N, 1);  // 1 for ascending order

    printf("\nSorted Array (Ascending):\n");
    printArray(arr, N);

    return 0;
}

```

OUTPUT

```
Original Array:
3 7 8 5 2 1 6 4

Sorted Array (Ascending):
1 2 4 6 3 7 5 8


=== Code Execution Successful ===
```

Analysis of the time complexity

Programming Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void compareAndSwap(int array[], int i, int j, int dir) {
    if ((array[i] > array[j] && dir == 1) || (array[i] < array[j] && dir == 0)) {
        swap(&array[i], &array[j]);
    }
}

void bitonicMerge(int array[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        for (int i = low; i < low + k; i++) {
            compareAndSwap(array, i, i + k, dir);
        }
        bitonicMerge(array, low, k, dir);
        bitonicMerge(array, low + k, k, dir);
    }
}

void bitonicSort(int array[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        bitonicSort(array, low, k, 1);
        bitonicSort(array, low + k, k, 0);
        bitonicMerge(array, low, cnt, dir);
    }
}

void bitonicSortMain(int array[], int n) {
    bitonicSort(array, 0, n, 1);
}

void measureTime(void (*sortFunc)(int[], int), int array[], int n) {
    clock_t start_time = clock();
    sortFunc(array, n);
    clock_t end_time = clock();
    double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
```

```c
    printf(" size %d: %f seconds\n", n, execution_time);
}

void generateRandomArray(int array[], int n) {
    for (int i = 0; i < n; i++) {
        array[i] = rand() % 10000;
    }
}

void printArray(int array[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int sizes[] = {8, 16, 32, 64, 128};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    printf("\nExecution time for graph size\n");
    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        int array[n];

        generateRandomArray(array, n);

        measureTime(bitonicSortMain, array, n);

    }

    return 0;
}
```

OUTPUT

```
Execution time for graph size
 size 8: 0.000002 seconds
 size 16: 0.000003 seconds
 size 32: 0.000006 seconds
 size 64: 0.000014 seconds
 size 128: 0.000031 seconds


=== Code Execution Successful ===
```

Learning Outcome: