

Department of Computer Science & Engineering

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES DELHI



योग: 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:

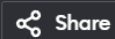
Program -1: To implement following algorithm using array as a data structure and analyse its time complexity. A) Insertion sort B) Selection sort C) Bubble sort D) Quick sort E) Heap sort F) Merge sort

A) INSERTION SORT

Theory-

Programming Code-

main.c



Run

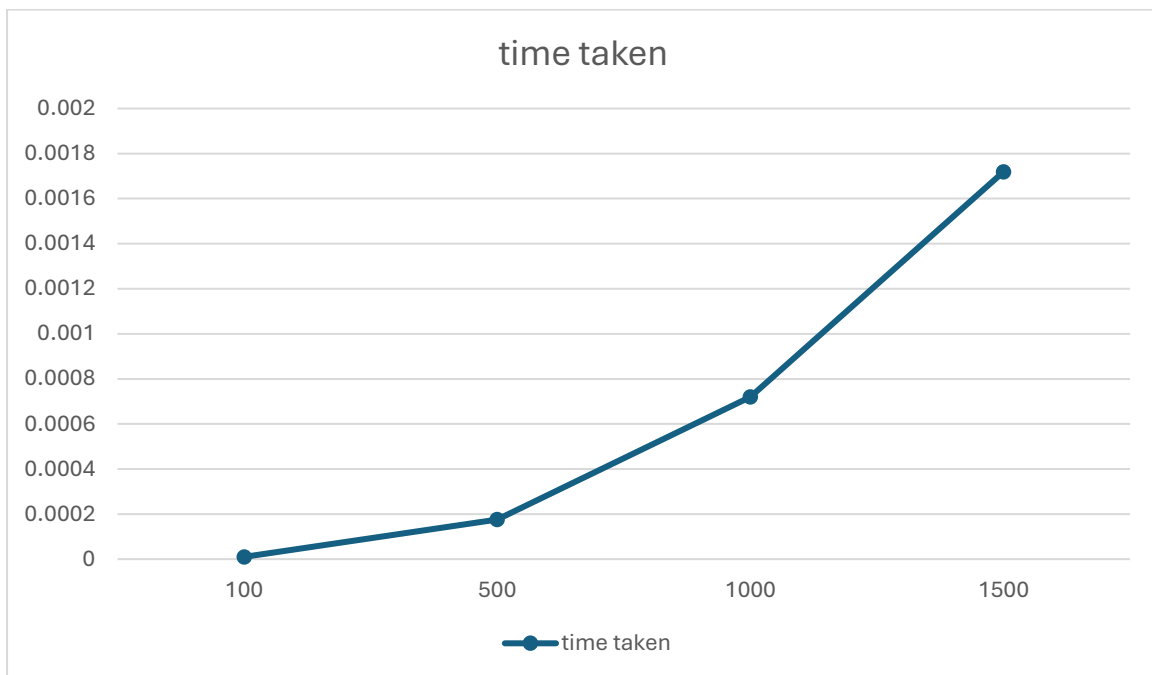
```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  // insertion sort
6  void insertionSort(int arr[], int n) {
7      for (int i = 1; i < n; i++) {
8          int key = arr[i];
9          int j = i - 1;
10         while (j >= 0 && arr[j] > key) {
11             arr[j + 1] = arr[j];
12             j = j - 1;
13         }
14         arr[j + 1] = key;
15     }
16 }
17
18 // random array
19 void generateRandomArray(int arr[], int n) {
20     for (int i = 0; i < n; i++) {
21         arr[i] = rand() % 10000;
22     }
23 }
24
25 // measure sorting time
26 void measureSortingTime(int arr[], int n) {
27     clock_t start, end;
28     start = clock();
29
30     insertionSort(arr, n);
31
32     end = clock();
33     double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
34     printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);
35 }
36
37 int main() {
38     srand(time(0));
39
40     int sizes[] = {100, 500, 1000, 1500};
41     int numSizes = sizeof(sizes) / sizeof(sizes[0]);
42
43     for (int i = 0; i < numSizes; i++) {
44         int n = sizes[i];
45         int arr[n];
46         generateRandomArray(arr, n);
47         measureSortingTime(arr, n);
48     }
49
50     return 0;
51 }
52

```

Output-

```
Output Clear  
/tmp/10wDJ6qdzT.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



Share

Run

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  // swap two elements
6  void swap(int *xp, int *yp) {
7      int temp = *xp;
8      *xp = *yp;
9      *yp = temp;
10 }
11
12 // Selection Sort
13 void selectionSort(int arr[], int n) {
14     int i, j, min_idx;
15
16     for (i = 0; i < n-1; i++) {
17         min_idx = i;
18         for (j = i+1; j < n; j++)
19             if (arr[j] < arr[min_idx])
20                 min_idx = j;
21         swap(&arr[min_idx], &arr[i]);
22     }
23 }
24
25 void printArray(int arr[], int size) {
26     int i;
27     for (i = 0; i < size; i++)
28         printf("%d ", arr[i]);
29     printf("\n");
30 }
31
32 // array of random numbers
33 void generateRandomArray(int arr[], int size) {
34     for (int i = 0; i < size; i++) {
35         arr[i] = rand() % 10000;
36     }
37 }
38
39 // measure the time taken
40 void measureTime(int arr[], int size) {
41     clock_t start, end;
42     double cpu_time_used;
43
44     start = clock();
45     selectionSort(arr, size);
46     end = clock();
47
48     cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
49
50     printf("Time taken to sort %d elements: %f seconds\n", size, cpu_time_used);
51 }
52

```

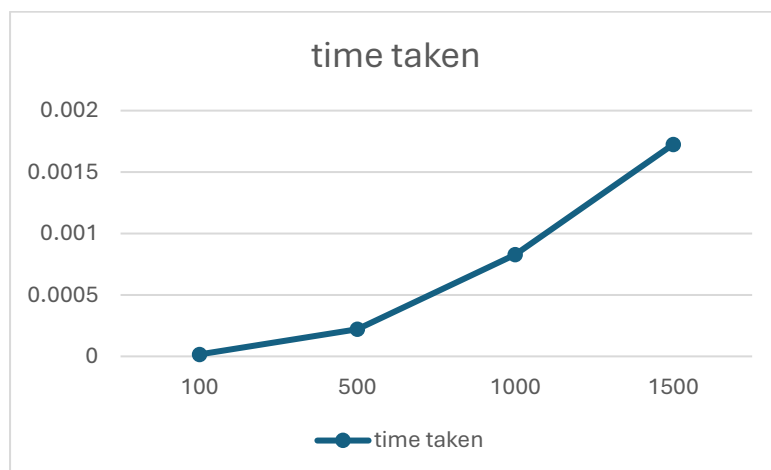
```
53 int main() {
54     srand(time(0));
55
56     int sizes[] = {100, 500, 1000, 1500};
57     int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
58
59     for (int i = 0; i < num_sizes; i++) {
60         int size = sizes[i];
61         int arr[size];
62
63         generateRandomArray(arr, size);
64
65         printf("Sorting array of size %d -\n", size);
66         measureTime(arr, size);
67     }
68
69     return 0;
70 }
```

Output-

Output Clear

```
/tmp/x6pdZ4nlwk1.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-

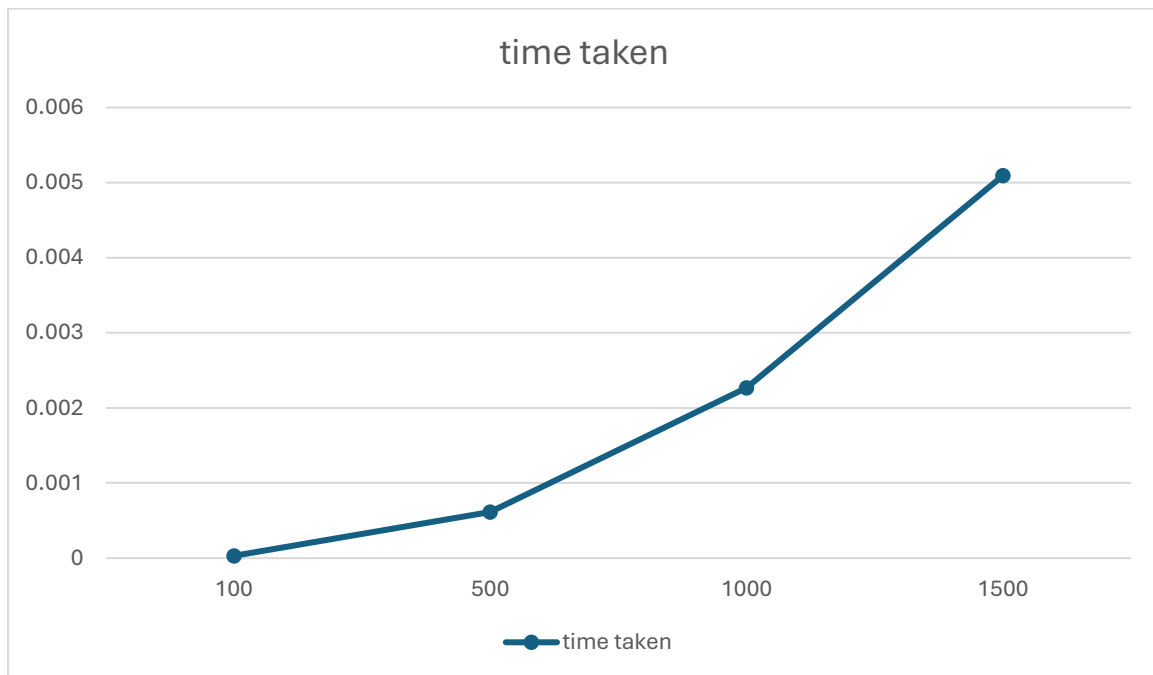
```

main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 // Bubble Sort
6 void bubbleSort(int arr[], int n) {
7     for (int i = 0; i < n-1; i++) {
8         for (int j = 0; j < n-i-1; j++) {
9             if (arr[j] > arr[j+1]) {
10                 int temp = arr[j];
11                 arr[j] = arr[j+1];
12                 arr[j+1] = temp;
13             }
14         }
15     }
16 }
17
18 // array with random elements
19 void generateRandomArray(int arr[], int n) {
20     for (int i = 0; i < n; i++) {
21         arr[i] = rand() % 10000;
22     }
23 }
24
25 void printArray(int arr[], int n) {
26     for (int i = 0; i < n; i++) {
27         printf("%d ", arr[i]);
28     }
29     printf("\n");
30 }
31
32 int main() {
33     int sizes[] = {100, 500, 1000, 1500};
34     int nSizes = sizeof(sizes)/sizeof(sizes[0]);
35
36     srand(time(0));
37
38     for (int i = 0; i < nSizes; i++) {
39         int n = sizes[i];
40         int arr[n];
41
42         generateRandomArray(arr, n);
43
44         clock_t start = clock();
45         bubbleSort(arr, n);
46         clock_t end = clock();
47
48         double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
49         printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);
50     }
51 }
52
53 return 0;
54 }
55

```

Output-

```
Output Clear  
/tmp/S8bL11ztJM.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-

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void swap(int *a, int *b) {
6      int temp = *a;
7      *a = *b;
8      *b = temp;
9  }
10
11 int partition(int arr[], int low, int high)
12     {
13         int pivot = arr[high];
14         int i = low - 1;
15
16         for (int j = low; j < high; j++) {
17             if (arr[j] <= pivot) {
18                 i++;
19                 swap(&arr[i], &arr[j]);
20             }
21         }
22         swap(&arr[i + 1], &arr[high]);
23         return i + 1;
24     }
25
26 void quickSort(int arr[], int low, int high) {
27     if (low < high) {
28         int pi = partition(arr, low, high);
29         quickSort(arr, low, pi - 1);
30         quickSort(arr, pi + 1, high);
31     }
32 }
33
34 void generateRandomArray(int arr[], int size) {
35     for (int i = 0; i < size; ++i) {
36         arr[i] = rand() % (size * 10);
37     }
38 }
39
40 void printArray(int arr[], int size) {
41     for (int i = 0; i < size; i++) {
42         printf("%d ", arr[i]);
43     }
44     printf("\n");
45 }
46
47 int main() {
48     int sizes[] = {100, 500, 1000, 1500};
49     int numSizes = sizeof(sizes) / sizeof(sizes[0]);
50
51     srand(time(NULL));
52     struct timespec start, end;
53     for (int i = 0; i < numSizes; ++i) {
54         int size = sizes[i];
55         int *arr = (int *)malloc(size * sizeof(int));
56
57         if (arr == NULL) {
58             printf("Memory allocation failed\n");
59             return 1;
60         }
61
62         generateRandomArray(arr, size);
63         clock_gettime(CLOCK_MONOTONIC, &start);
64         quickSort(arr, 0, size - 1);
65         clock_gettime(CLOCK_MONOTONIC, &end);
66
67         long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000L +
68             (end.tv_nsec - start.tv_nsec);
69         printf("Size: %d, Time taken: %ld nanoseconds\n", size, elapsed_ns);
70
71         free(arr);
72     }
73     return 0;
74 }

```

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-

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void swap(int *a, int *b) {
6      int temp = *a;
7      *a = *b;
8      *b = temp;
9  }
10 void heapify(int arr[], int n, int i) {
11     int largest = i;
12     int left = 2 * i + 1;
13     int right = 2 * i + 2;
14
15     if (left < n && arr[left] >
16         arr[largest])
17         largest = left;
18     if (right < n && arr[right] >
19         arr[largest])
20         largest = right;
21     if (largest != i) {
22         swap(&arr[i], &arr[largest]);
23         heapify(arr, n, largest);
24     }
25 }
26 void heapSort(int arr[], int n) {
27     // maxheap
28     for (int i = n / 2 - 1; i >= 0; i--)
29         heapify(arr, n, i);
30     for (int i = n - 1; i >= 0; i--) {
31         swap(&arr[0], &arr[i]);
32         heapify(arr, i, 0);
33     }
34 }
35 void generateRandomArray(int arr[], int
36     size) {
37     for (int i = 0; i < size; ++i) {
38         arr[i] = rand() % (size * 10);
39     }
40 }
41 void printArray(int arr[], int size) {
42     for (int i = 0; i < size; i++) {
43         printf("%d ", arr[i]);
44     }
45     printf("\n");
46 }
47
48 int main() {
49     int sizes[] = {100, 500, 1000, 1500};
50     int numSizes = sizeof(sizes) / sizeof
51         (sizes[0]);
52     srand(time(NULL));
53     struct timespec start, end;
54
55     for (int i = 0; i < numSizes; ++i) {
56         int size = sizes[i];
57         int *arr = (int *)malloc(size *
58             sizeof(int));
59         if (arr == NULL) {
60             printf("Memory allocation
61                 failed\n");
62             return 1;
63         }
64         generateRandomArray(arr, size);
65         clock_gettime(CLOCK_MONOTONIC,
66             &start);
67         heapSort(arr, size);
68         clock_gettime(CLOCK_MONOTONIC,
69             &end);
70         long elapsed_ns = (end.tv_sec -
71             start.tv_sec) * 1000000000L +
72             (end.tv_nsec - start.tv_nsec);
73         printf("Size: %d, Time taken: %ld
74             nanoseconds\n", size,
75                 elapsed_ns);
76         free(arr);
77     }
78     return 0;
79 }

```

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-

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void merge(int arr[], int left, int mid,
6             int right) {
7      int n1 = mid - left + 1;
8      int n2 = right - mid;
9
10     int *L = (int *)malloc(n1 * sizeof(int));
11     int *R = (int *)malloc(n2 * sizeof(int));
12
13     if (L == NULL || R == NULL) {
14         printf("Memory allocation failed\n");
15         exit(1);
16     }
17     // Copy data to temporary arrays
18     for (int i = 0; i < n1; i++)
19         L[i] = arr[left + i];
20     for (int j = 0; j < n2; j++)
21         R[j] = arr[mid + 1 + j];
22
23     int i = 0;
24     int j = 0;
25     int k = left;
26
27     while (i < n1 && j < n2) {
28         if (L[i] <= R[j])
29             arr[k++] = L[i++];
30         else
31             arr[k++] = R[j++];
32     }
33
34     while (i < n1)
35         arr[k++] = L[i++];
36
37     while (j < n2)
38         arr[k++] = R[j++];
39
40     free(L);
41     free(R);
42 }
43
44 void mergeSort(int arr[], int left, int right) {
45     if (left < right) {
46         int mid = left + (right - left) / 2;
47
48         mergeSort(arr, left, mid);
49         mergeSort(arr, mid + 1, right);
50         merge(arr, left, mid, right);
51     }
52 }
53
54 void generateRandomArray(int arr[], int size) {
55     for (int i = 0; i < size; ++i) {
56         arr[i] = rand() % (size * 10);
57     }
58 }
59
60 void printArray(int arr[], int size) {
61     for (int i = 0; i < size; i++) {
62         printf("%d ", arr[i]);
63     }
64     printf("\n");
65 }
66
67 int main() {
68     int sizes[] = {100, 500, 1000, 1500};
69     int numSizes = sizeof(sizes) / sizeof(sizes[0]);
70     srand(time(NULL));
71     struct timespec start, end;
72
73     for (int i = 0; i < numSizes; ++i) {
74         int size = sizes[i];
75         int *arr = (int *)malloc(size * sizeof(int));
76         if (arr == NULL) {
77             printf("Memory allocation failed\n");
78             return 1;
79         }
80
81         generateRandomArray(arr, size);
82         clock_gettime(CLOCK_MONOTONIC, &start);
83         mergeSort(arr, 0, size - 1);
84         clock_gettime(CLOCK_MONOTONIC, &end);
85
86         long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000L +
87             (end.tv_nsec - start.tv_nsec);
88         printf("Size: %d, Time taken: %ld nanoseconds\n", size, elapsed_ns);
89         free(arr);
90     }
91     return 0;
92 }

```

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-

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int linearSearch(int arr[], int size, int
    target) {
6      for (int i = 0; i < size; ++i) {
7          if (arr[i] == target) {
8              return i;
9          }
10     }
11     return -1;
12 }
13
14 void generateRandomArray(int arr[], int
    size) {
15     for (int i = 0; i < size; ++i) {
16         arr[i] = rand() % (size * 10);
17     }
18 }
19
20 int main() {
21     int sizes[] = {100, 500, 1000, 1500};
22     int numSizes = sizeof(sizes) / sizeof
        (sizes[0]);
23
24     srand(time(NULL));
25     for (int i = 0; i < numSizes; ++i) {
26         int size = sizes[i];
27         int *arr = (int *)malloc(size *
            sizeof(int));
28         if (arr == NULL) {
29             printf("Memory allocation
                failed\n");
30             return 1;
31         }
32         generateRandomArray(arr, size);
33         int target = rand() % (size * 10);
34
35         clock_t start = clock();
36         linearSearch(arr, size, target);
37         clock_t end = clock();
38
39         double time_taken = (double)(end -
            start) / CLOCKS_PER_SEC;
40         printf("Size: %d, Time taken: %f
            seconds\n", size, time_taken);
41         free(arr);
42     }
43     return 0;
44 }
45

```

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-

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int linearSearch(int arr[], int size, int
    target) {
6      for (int i = 0; i < size; ++i) {
7          if (arr[i] == target) {
8              return i;
9          }
10     }
11     return -1;
12 }
13
14 void generateRandomArray(int arr[], int
    size) {
15     for (int i = 0; i < size; ++i) {
16         arr[i] = rand() % (size * 10);
17     }
18 }
19
20 int main() {
21     int sizes[] = {100, 500, 1000, 1500};
22     int numSizes = sizeof(sizes) / sizeof
        (sizes[0]);
23
24     srand(time(NULL));
25     struct timespec start, end;
26
27     for (int i = 0; i < numSizes; ++i) {
28         int size = sizes[i];
29         int *arr = (int *)malloc(size *
            sizeof(int));
30         if (arr == NULL) {
31             printf("Memory allocation
                failed\n");
32             return 1;
33         }
34         generateRandomArray(arr, size);
35         int target = rand() % (size * 10);
36
37         clock_gettime(CLOCK_MONOTONIC,
            &start);
38         linearSearch(arr, size, target);
39         clock_gettime(CLOCK_MONOTONIC,
            &end);
40
41         long elapsed_ns = (end.tv_sec -
            start.tv_sec) * 1000000000L +
            (end.tv_nsec - start.tv_nsec);
42         printf("Size: %d, Time taken: %ld
            nanoseconds\n", size,
            elapsed_ns);
43
44         free(arr);
45     }
46
47     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
1  #include <iostream>
2  #include <queue>
3  #include <vector>
4  #include <cstdlib>
5  #include <ctime>
6  #include <chrono>
7  #include <map>
8
9  using namespace std;
10
11 struct MinHeapNode
12 {
13     char data;
14     int freq;
15     MinHeapNode *left, *right;
16
17     MinHeapNode(char data, int freq)
18     {
19         left = right = nullptr;
20         this->data = data;
21         this->freq = freq;
22     }
23 };
24
25
26 struct compare
27 {
28     bool operator()(MinHeapNode* l, MinHeapNode* r)
29     {
30         return (l->freq > r->freq);
31     }
32 };
33
34
35 void printCodes(struct MinHeapNode* root, string str, map<char, string>&
    huffmanCodes)
36 {
37     if (!root)
38         return;
39
40     if (root->data != '$')
41         huffmanCodes[root->data] = str;
42
43     printCodes(root->left, str + "0", huffmanCodes);
44     printCodes(root->right, str + "1", huffmanCodes);
45 }
46
47
48 void HuffmanCoding(char data[], int freq[], int size)
49 {
50     struct MinHeapNode *left, *right, *top;
51
52
53     priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;
54

```

```

55     for (int i = 0; i < size; i++)
56         minHeap.push(new MinHeapNode(data[i], freq[i]));
57
58
59     while (minHeap.size() != 1)
60     {
61
62         left = minHeap.top();
63         minHeap.pop();
64         right = minHeap.top();
65         minHeap.pop();
66
67
68         top = new MinHeapNode('$', left->freq + right->freq);
69
70         top->left = left;
71         top->right = right;
72
73         minHeap.push(top);
74     }
75
76
77     map<char, string> huffmanCodes;
78     printCodes(minHeap.top(), "", huffmanCodes);
79
80     cout << "\nHuffman Codes:\n";
81     for (auto pair : huffmanCodes)
82         cout << pair.first << ": " << pair.second << "\n";
83 }
84

```

```

77     map<char, string> huffmanCodes;
78     printCodes(minHeap.top(), "", huffmanCodes);
79
80     cout << "\nHuffman Codes:\n";
81     for (auto pair : huffmanCodes)
82         cout << pair.first << ": " << pair.second << "\n";
83 }
84
85 int main()
86 {
87
88     char characters[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
89     int freq[] = { 5, 9, 12, 13, 16, 45 };
90     int size = sizeof(characters) / sizeof(characters[0]);
91
92     auto start = chrono::high_resolution_clock::now();
93
94
95     HuffmanCoding(characters, freq, size);
96
97     auto end = chrono::high_resolution_clock::now();
98
99
100     chrono::duration<double, nano> duration = end - start;
101     cout << "\nTime taken: " << duration.count() << " nanoseconds" << endl;
102
103     return 0;
104 }

```

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
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <chrono>
5
6  using namespace std;
7
8
9  struct Edge
10 {
11     int src, dest, weight;
12 };
13
14
15 struct Subset
16 {
17     int parent;
18     int rank;
19 };
20
21
22 int find(Subset subsets[], int i)
23 {
24     if (subsets[i].parent != i)
25         subsets[i].parent = find(subsets, subsets[i].parent);
26     return subsets[i].parent;
27 }
28
29 void Union(Subset subsets[], int x, int y)
30 {
31     int xroot = find(subsets, x);
32     int yroot = find(subsets, y);
33
34     if (subsets[xroot].rank < subsets[yroot].rank)
35         subsets[xroot].parent = yroot;
36     else if (subsets[xroot].rank > subsets[yroot].rank)
37         subsets[yroot].parent = xroot;
38     else
39     {
40         subsets[yroot].parent = xroot;
41         subsets[xroot].rank++;
42     }
43 }
44
45
46 bool compare(Edge a, Edge b)
47 {
48     return a.weight < b.weight;
49 }
50
51
52 void KruskalMST(vector<Edge>& edges, int V)
53 {
54     vector<Edge> result;
55     int e = 0;
56     int i = 0;
57

```

```

58
59     sort(edges.begin(), edges.end(), compare);
60
61
62     Subset* subsets = new Subset[V];
63     for (int v = 0; v < V; ++v)
64     {
65         subsets[v].parent = v;
66         subsets[v].rank = 0;
67     }
68
69
70     while (e < V - 1 && i < edges.size())
71     {
72
73         Edge nextEdge = edges[i++];
74
75         int x = find(subsets, nextEdge.src);
76         int y = find(subsets, nextEdge.dest);
77
78
79         if (x != y)
80         {
81             result.push_back(nextEdge);
82             Union(subsets, x, y);
83             e++;
84         }
85     }
86
87     cout << "Edges in the Minimum Spanning Tree:\n";
88     for (auto& edge : result)
89         cout << edge.src << " -- " << edge.dest << " == " << edge.weight <<
            endl;
90
91     delete[] subsets;
92 }
93
94 int main()
95 {
96     int V = 4;
97     vector<Edge> edges = {
98         {0, 1, 10}, {0, 2, 6}, {0, 3, 5},
99         {1, 3, 15}, {2, 3, 4}
100 };
101
102     auto start = chrono::high_resolution_clock::now();
103
104
105     KruskalMST(edges, V);
106
107     auto end = chrono::high_resolution_clock::now();
108
109
110     chrono::duration<double, nano> duration = end - start;
111     cout << "Time taken: " << duration.count() << " nanoseconds" << endl;
112
113     return 0;
114 }
115

```


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:

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <time.h>
4  #include <stdlib.h>
5
6  int MatrixChainOrder(int p[], int n) {
7      int m[n][n];
8
9      for (int i = 1; i < n; i++)
10         m[i][i] = 0;
11
12     for (int L = 2; L < n; L++) {
13         for (int i = 1; i < n - L + 1; i++) {
14             int j = i + L - 1;
15             m[i][j] = INT_MAX;
16             for (int k = i; k <= j - 1; k++) {
17                 int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
18                 if (q < m[i][j])
19                     m[i][j] = q;
20             }
21         }
22     }
23     return m[1][n - 1];
24 }
25
26 void generateRandomDimensions(int *p, int n) {
27     for (int i = 0; i < n; i++) {
28         p[i] = rand() % 100 + 1;
29     }
30 }
31
32 int main() {

```

```

33     srand(time(0));
34
35     int sizes[] = {5, 10, 15, 20, 25};
36     int numTests = sizeof(sizes) / sizeof(sizes[0]);
37
38     printf("Matrix Chain Multiplication Execution Time Analysis:\n");
39     printf("Size\tMin. Multiplications\tExecution Time (ms)\n");
40
41     for (int i = 0; i < numTests; i++) {
42         int n = sizes[i];
43         int p[n + 1];
44         generateRandomDimensions(p, n + 1);
45
46         clock_t start = clock();
47         int minMultiplications = MatrixChainOrder(p, n + 1);
48         clock_t end = clock();
49
50         double time_taken = ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;
51
52         printf("%d\t\t\t\t\t%d\t\t\t\t\t%.3f\n", n, minMultiplications, time_taken);
53     }
54
55     return 0;
56 }
57

```

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:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4  #include <time.h>
5
6  #define INF INT_MAX
7  #define MAX_VERTICES 1000
8
9  typedef struct {
10     int vertex;
11     int distance;
12 } PQElement;
13
14 int minDistance(int dist[], int visited[], int n) {
15     int min = INF;
16     int minIndex = -1;
17     for (int v = 0; v < n; v++) {
18         if (!visited[v] && dist[v] < min) {
19             min = dist[v];
20             minIndex = v;
21         }
22     }
23     return minIndex;
24 }
25
26 void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int n, int source) {
27     int dist[MAX_VERTICES];
28     int visited[MAX_VERTICES] = {0};
29
30     for (int i = 0; i < n; i++) {
31         dist[i] = INF;
32     }

```

```

33     dist[source] = 0;
34
35     for (int count = 0; count < n - 1; count++) {
36         // Pick the vertex with the minimum distance that hasn't been visited
37         int u = minDistance(dist, visited, n);
38         visited[u] = 1;
39
40         for (int v = 0; v < n; v++) {
41             if (!visited[v] && graph[u][v] != INF && dist[u] != INF && dist[u] +
42                 graph[u][v] < dist[v]) {
43                 dist[v] = dist[u] + graph[u][v];
44             }
45         }
46
47         printf("Shortest distances from source vertex %d:\n", source);
48         printf("Vertex \t\t\tDistance from Source\n");
49         for (int i = 0; i < n; i++) {
50             if (dist[i] == INF) {
51                 printf("Vertex %d\t\tINF\n", i);
52             } else {
53                 printf("Vertex %d\t\t\t %d\n", i, dist[i]);
54             }
55         }
56     }
57
58     void generateGraph(int graph[MAX_VERTICES][MAX_VERTICES], int n, int maxWeight) {
59         for (int i = 0; i < n; i++) {
60             for (int j = 0; j < n; j++) {
61                 if (i != j) {
62                     graph[i][j] = rand() % maxWeight + 1;
63                 } else {
64                     graph[i][j] = INF;
65                 }
66             }
67         }
68     }
69
70     void analyzeTimeComplexity(int n) {
71         int graph[MAX_VERTICES][MAX_VERTICES];
72
73         generateGraph(graph, n, 100);
74
75         clock_t start_time = clock();
76         dijkstra(graph, n, 0);
77         clock_t end_time = clock();
78
79         double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
80         printf("Execution time for graph size %d: %f seconds\n\n", n, execution_time);

```

```

81 }
82
83 int main() {
84     int sizes[] = {10,20,30};
85     int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
86
87     for (int i = 0; i < num_sizes; i++) {
88         int n = sizes[i];
89         analyzeTimeComplexity(n);
90     }
91
92     return 0;
93 }

```

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:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4  #include <time.h>
5
6  #define INF INT_MAX
7  #define MAX_VERTICES 1000
8
9  typedef struct {
10     int u, v, weight;
11 } Edge;
12
13 void bellmanFord(Edge edges[], int V, int E, int source) {
14     int dist[V];
15
16     for (int i = 0; i < V; i++) {
17         dist[i] = INF;
18     }
19     dist[source] = 0;
20
21     for (int i = 1; i <= V - 1; i++) {
22         for (int j = 0; j < E; j++) {
23             int u = edges[j].u;
24             int v = edges[j].v;
25             int weight = edges[j].weight;
26             if (dist[u] != INF && dist[u] + weight < dist[v]) {
27                 dist[v] = dist[u] + weight;
28             }
29         }
30     }
31
32     for (int i = 0; i < E; i++) {

```

```

33     int u = edges[i].u;
34     int v = edges[i].v;
35     int weight = edges[i].weight;
36     if (dist[u] != INF && dist[u] + weight < dist[v]) {
37         printf("Graph contains negative weight cycle\n");
38         return;
39     }
40 }
41
42 printf("Shortest distances from source vertex %d:\n", source);
43 printf("Vertex \t\t\tDistance from Source\n");
44 for (int i = 0; i < V; i++) {
45     if (dist[i] == INF) {
46         printf("Vertex %d\t\t\tINF\n", i);
47     } else {
48         printf("Vertex %d\t\t\t %d\n", i, dist[i]);
49     }
50 }
51
52 }
53
54 void generateGraph(Edge edges[], int V, int E, int maxWeight) {
55     for (int i = 0; i < E; i++) {
56         int u = rand() % V;
57         int v = rand() % V;
58         while (u == v) {
59             v = rand() % V;
60         }
61         int weight = rand() % maxWeight + 1;
62         edges[i] = (Edge){u, v, weight};
63     }
64 }
65
66 void analyzeTimeComplexity(int V, int E) {
67     Edge edges[E];
68
69     generateGraph(edges, V, E, 100);
70
71     clock_t start_time = clock();
72     bellmanFord(edges, V, E, 0);
73     clock_t end_time = clock();
74
75     double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
76     printf("Execution time for graph size V=%d, E=%d: %f seconds\n\n", V, E,
77         execution_time);
78 }
79
80 int main() {
81     int sizes[][2] = {{10, 30}, {15, 200}, {20, 500}};

```

```

82     int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
83
84     for (int i = 0; i < num_sizes; i++) {
85         int V = sizes[i][0];
86         int E = sizes[i][1];
87         analyzeTimeComplexity(V, E);
88     }
89
90     return 0;
91 }
92 |

```

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:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define MAX_N 20
6
7  void printSolution(int board[MAX_N][MAX_N], int N) {
8      for (int i = 0; i < N; i++) {
9          for (int j = 0; j < N; j++) {
10             printf("%d ", board[i][j]);
11         }
12         printf("\n");
13     }
14 }
15
16 int isSafe(int board[MAX_N][MAX_N], int row, int col, int N) {
17     // Check this column on the upper side
18     for (int i = 0; i < row; i++) {
19         if (board[i][col] == 1) {
20             return 0;
21         }
22     }
23
24     // Check upper diagonal on the left side
25     for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
26         if (board[i][j] == 1) {
27             return 0;
28         }
29     }
30
31     // Check upper diagonal on the right side
32     for (int i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++) {

```



```

33-     if (board[i][j] == 1) {
34-         return 0;
35-     }
36- }
37
38-     return 1;
39- }
40
41- int solveNQueens(int board[MAX_N][MAX_N], int row, int N) {
42
43-     if (row >= N) {
44-         return 1;
45-     }
46
47-     for (int col = 0; col < N; col++) {
48-         if (isSafe(board, row, col, N)) {
49
50-             board[row][col] = 1;
51
52-             if (solveNQueens(board, row + 1, N)) {
53-                 return 1;
54-             }
55
56-             board[row][col] = 0;
57-         }
58-     }
59
60-     return 0;
61- }
62
63- void analyzeTimeComplexity(int N) {
64-     int board[MAX_N][MAX_N] = {0};
65
66-     clock_t start_time = clock();
67-     if (solveNQueens(board, 0, N)) {
68-         printf("Solution for N=%d found:\n", N);
69-         printSolution(board, N);
70-     } else {
71-         printf("No solution exists for N=%d\n", N);
72-     }
73-     clock_t end_time = clock();
74
75-     double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
76-     printf("Execution time for N=%d: %f seconds\n", N, execution_time);
77- }
78
79- int main() {
80-     int sizes[] = {4, 8, 10};
81-     int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
82
83-     for (int i = 0; i < num_sizes; i++) {
84-         int N = sizes[i];
85-         analyzeTimeComplexity(N);
86-     }
87
88-     return 0;
89- }
90

```

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:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5
6  int LCS(char* X, char* Y, int m, int n) {
7      int dp[m + 1][n + 1];
8
9      for (int i = 0; i <= m; i++) {
10         for (int j = 0; j <= n; j++) {
11             if (i == 0 || j == 0) {
12                 dp[i][j] = 0;
13             } else if (X[i - 1] == Y[j - 1]) {
14                 dp[i][j] = dp[i - 1][j - 1] + 1;
15             } else {
16                 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
17             }
18         }
19     }
20
21     return dp[m][n];
22 }
23
24 void analyzeTimeComplexity(char* X, char* Y) {
25     int m = strlen(X);
26     int n = strlen(Y);
27
28     clock_t start_time = clock();
29
30     int lcs_length = LCS(X, Y, m, n);
31

```

```

32     clock_t end_time = clock();
33
34     double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
35
36     printf("Length of LCS: %d\n", lcs_length);
37     printf("Execution time for strings of length %d and %d: %f seconds\n", m, n,
           execution_time);
38 }
39
40 int main() {
41     char* strings[] = {
42         "AGGTAB", // Length 6
43         "ABCDEFGH", // Length 8
44         "ABCDE", // Length 5
45         "AAGTCAG", // Length 7
46
47     };
48
49     int num_strings = sizeof(strings) / sizeof(strings[0]);
50
51     for (int i = 0; i < num_strings - 1; i++) {
52         for (int j = i + 1; j < num_strings; j++) {
53             printf("\nComparing string %d and string %d:\n", i+1, j+1);
54             analyzeTimeComplexity(strings[i], strings[j]);
55         }
56     }
57
58     return 0;
59 }
60

```

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 algorithm and analyse its time complexity.

Theory:

Programming Code:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5
6  // Naïve String Matching Algorithm
7  int naiveStringMatch(char* text, char* pattern) {
8      int n = strlen(text);
9      int m = strlen(pattern);
10     int count = 0;
11     for (int i = 0; i <= n - m; i++) {
12         int j = 0;
13         while (j < m && text[i + j] == pattern[j]) {
14             j++;
15         }
16         if (j == m) {
17             count++; // Found a match
18         }
19     }
20     return count;
21 }
22
23 // Rabin-Karp Algorithm
24 #define d 256
25 #define q 101
26
27 int rabinKarpStringMatch(char* text, char* pattern) {
28     int n = strlen(text);
29     int m = strlen(pattern);
30     int count = 0;
31
32     int patternHash = 0;

```



```

33     int textHash = 0;
34     int h = 1;
35
36     for (int i = 0; i < m - 1; i++) {
37         h = (h * d) % q;
38     }
39
40     // Calculate the hash value for the pattern and the first window of text
41     for (int i = 0; i < m; i++) {
42         patternHash = (d * patternHash + pattern[i]) % q;
43         textHash = (d * textHash + text[i]) % q;
44     }
45
46     // Slide the pattern over the text one by one
47     for (int i = 0; i <= n - m; i++) {
48         if (patternHash == textHash) {
49             int j = 0;
50             while (j < m && text[i + j] == pattern[j]) {
51                 j++;
52             }
53             if (j == m) {
54                 count++; // Found a match
55             }
56         }
57
58         // Calculate hash value for the next window of text
59         if (i < n - m) {
60             textHash = (d * (textHash - text[i] * h) + text[i + m]) % q;
61             if (textHash < 0) {
62                 textHash = (textHash + q);
63             }
64         }
65     }
66     return count;
67 }
68
69 // KMP Algorithm (Knuth-Morris-Pratt)
70 void computeLPSArray(char* pattern, int m, int* lps) {
71     int length = 0;
72     lps[0] = 0;
73     int i = 1;
74
75     while (i < m) {
76         if (pattern[i] == pattern[length]) {
77             length++;
78             lps[i] = length;
79             i++;
80         } else {
81             if (length != 0) {
82                 length = lps[length - 1];

```

```

83     } else {
84         lps[i] = 0;
85         i++;
86     }
87 }
88 }
89 }
90
91 int kmpStringMatch(char* text, char* pattern) {
92     int n = strlen(text);
93     int m = strlen(pattern);
94     int lps[m]; // Longest Prefix Suffix array
95     computeLPSArray(pattern, m, lps);
96
97     int i = 0;
98     int j = 0;
99     int count = 0;
100
101     while (i < n) {
102         if (pattern[j] == text[i]) {
103             j++;
104             i++;
105         }
106
107         if (j == m) {
108             count++; // Found a match
109             j = lps[j - 1];
110         } else if (i < n && pattern[j] != text[i]) {
111             if (j != 0) {
112                 j = lps[j - 1];
113             } else {
114                 i++;
115             }
116         }
117     }
118     return count;
119 }
120
121 void analyzeTimeComplexity(char* text, char* pattern) {
122     clock_t start_time, end_time;
123     double execution_time;
124
125     // Measure execution time for Naive String Matching
126     start_time = clock();
127     int naive_count = naiveStringMatch(text, pattern);
128     end_time = clock();

```

```

129     execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
130     printf("Naïve String Match - Matches: %d, Time: %f seconds\n", naive_count,
           execution_time);
131
132     // Measure execution time for Rabin-Karp Algorithm
133     start_time = clock();
134     int rabin_karp_count = rabinKarpStringMatch(text, pattern);
135     end_time = clock();
136     execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
137     printf("Rabin-Karp String Match - Matches: %d, Time: %f seconds\n",
           rabin_karp_count, execution_time);
138
139     // Measure execution time for KMP Algorithm
140     start_time = clock();
141     int kmp_count = kmpStringMatch(text, pattern);
142     end_time = clock();
143     execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
144     printf("KMP String Match - Matches: %d, Time: %f seconds\n", kmp_count,
           execution_time);
145 }
146
147 int main() {
148     char text[] = "ABABDABACDABABCABAB";
149     char pattern[] = "ABAB";
150
151     analyzeTimeComplexity(text, pattern);
152
153     return 0;
154 }
155

```

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:

```

1  #include <stdio.h>
2
3  #define N 8
4
5  void swap(int *a, int *b) {
6      if (*a > *b) {
7          int temp = *a;
8          *a = *b;
9          *b = temp;
10     }
11 }
12
13 void bitonicMerge(int arr[], int low, int cnt, int dir) {
14     if (cnt > 1) {
15         int k = cnt / 2;
16
17         for (int i = low; i < low + k; i++) {
18             swap(&arr[i], &arr[i + k]);
19         }
20
21         // Recursively merge the two halves
22         bitonicMerge(arr, low, k, dir);
23         bitonicMerge(arr, low + k, k, dir);
24     }
25 }
26
27 void bitonicSort(int arr[], int low, int cnt, int dir) {
28     if (cnt > 1) {
29         int k = cnt / 2;
30
31         bitonicSort(arr, low, k, 1);
32

```

```

33     bitonicSort(arr, low + k, k, 0);
34
35     bitonicMerge(arr, low, cnt, dir);
36 }
37 }
38
39 void printArray(int arr[], int n) {
40     for (int i = 0; i < n; i++) {
41         printf("%d ", arr[i]);
42     }
43     printf("\n");
44 }
45
46 int main() {
47     int arr[N] = { 3, 7, 8, 5, 2, 1, 6, 4 };
48
49     printf("Original Array:\n");
50     printArray(arr, N);
51
52     bitonicSort(arr, 0, N, 1); // 1 for ascending order
53
54     printf("\nSorted Array (Ascending):\n");
55     printArray(arr, N);
56
57     return 0;
58 }
59

```

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:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void swap(int *a, int *b) {
6      int temp = *a;
7      *a = *b;
8      *b = temp;
9  }
10
11 void compareAndSwap(int array[], int i, int j, int dir) {
12     if ((array[i] > array[j] && dir == 1) || (array[i] < array[j] && dir == 0)) {
13         swap(&array[i], &array[j]);
14     }
15 }
16
17 void bitonicMerge(int array[], int low, int cnt, int dir) {
18     if (cnt > 1) {
19         int k = cnt / 2;
20         for (int i = low; i < low + k; i++) {
21             compareAndSwap(array, i, i + k, dir);
22         }
23         bitonicMerge(array, low, k, dir);
24         bitonicMerge(array, low + k, k, dir);
25     }
26 }
27
28 void bitonicSort(int array[], int low, int cnt, int dir) {
29     if (cnt > 1) {
30         int k = cnt / 2;
31         bitonicSort(array, low, k, 1);
32         bitonicSort(array, low + k, k, 0);
33         bitonicMerge(array, low, cnt, dir);
34     }
35 }
36
37 void bitonicSortMain(int array[], int n) {
38     bitonicSort(array, 0, n, 1);
39 }
40
41 void measureTime(void (*sortFunc)(int[], int), int array[], int n) {
42     clock_t start_time = clock();
43     sortFunc(array, n);
44     clock_t end_time = clock();
45     double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

```

```

46     printf(" size %d: %f seconds\n", n, execution_time);
47 }
48
49 void generateRandomArray(int array[], int n) {
50     for (int i = 0; i < n; i++) {
51         array[i] = rand() % 10000;
52     }
53 }
54
55 void printArray(int array[], int n) {
56     for (int i = 0; i < n; i++) {
57         printf("%d ", array[i]);
58     }
59     printf("\n");
60 }
61
62 int main() {
63     int sizes[] = {8, 16, 32, 64, 128};
64     int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

```

```

65
66     printf("\nExecution time for graph size\n");
67     for (int i = 0; i < num_sizes; i++) {
68         int n = sizes[i];
69         int array[n];
70
71         generateRandomArray(array, n);
72
73         measureTime(bitonicSortMain, array, n);
74
75     }
76
77     return 0;
78 }
79

```


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: