KRISHNA CHAITANYA INSTITUTE OF TECHNOLOGY AND SCIENCES

(An ISO 9001:2008 certified and AICTE Approved institution, Affliated to JNTU-Kakinada)

Devarajugattu,Peddaraveedu (M), Markapur-523320, Prakasam(D), A.P

Department of Artificial Intelligence and Machine Learning (AIML)

ALGORITHMS FOR EFFICIENT CODING LAB

(R20 Regulation, 2024-2025 ACADEMIC YEAR)

B TECH III Year II Semester

INDEX

S.No	DATE	NAME OF THE EXPERIMENT	PAGE No.	REMARKS
1		Develop a program for Binary Search with Divide and Conquer		
2		Develop a program for Merge Sort with Divide and Conquer		
3		Develop a program for Quick Sort with Divide and Conquer		
4		Develop a program for Minimum-Cost Spanning Trees (Greedy Method)		
5		Develop a program for Single Source Shortest Paths (Greedy Method)		
6		Develop a program for Optimal Binary Search Trees (Dynamic Prog.)		
7		Develop a program for Traveling Salesperson Problem (Dynamic Prog.)		
8		Develop a program for 8-Queens Problem (Backtracking)		
9		Develop a program for Graph Coloring (Backtracking)		
10		Develop a program for Hamiltonian Cycle Problem (Backtracking)		
11		Develop a program for Knapsack Problem (Backtracking)		

1. **AIM:** To implement a Binary Search algorithm using Divide and Conquer approach and measure its running time.

SOURCE CODE:

```
// Online C compiler to run C program online

#include <stdio.h>

#include <time.h>

// Function for Binary Search using Divide and Conquer

int binarySearch(int arr[], int low, int high, int target) {

    while (low <= high) {

        int mid = (low + high) / 2;

        // Check if the target is present at mid

        if (arr[mid] == target) {

            return mid;  // Return the index

        }

        // If the target is greater, ignore the left half

        if (arr[mid] < target) {

            low = mid + 1;

        }

        // If the target is smaller, ignore the right half

        else {

            high = mid - 1;

        }

    }

    return -1;  // Target not found
```

```
}
int main() {
    // Sorted array example
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 11;
    // Variables to track the running time
    clock_t start, end;
    double cpu_time_used;
    // Get the start time
    start = clock();
    int result = binarySearch(arr, 0, n - 1, target);
    // Get the end time
    end = clock();
    // Calculate time used in seconds
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    // Output result
    if (result != -1)
        printf("Target found at index %d\n", result);
    else
        printf("Target not found\n");
    // Output time taken for binary search
    printf("Binary search took %f seconds\n", cpu_time_used);
```

```
    return 0;
}
```

OUTPUT:

Target found at index 5

Binary search took 0.000002 seconds

2. **AIM:** To implement the Merge Sort algorithm using Divide and Conquer approach and measure its running time.

SOURCE CODE:

```
#include <stdio.h>

#include <time.h>

void merge(int a[], int le, int m, int ri) {

    int n1 = m - le + 1;

    int n2 = ri - m;

    int l[n1], r[n2], i, j;

    for (i = 0; i < n1; i++) {

        l[i] = a[le + i];

    }

    for (j = 0; j < n2; j++) {

        r[j] = a[m + 1 + j];

    }

    i = 0, j = 0;

    int k = le;

    while (i < n1 && j < n2) {

        if (l[i] <= r[j]) {

            a[k] = l[i];

            i++;

        } else {

            a[k] = r[j];

            j++;
```

```
        }
        k++;
    }
    while (i < n1) {
        a[k] = l[i];
        i++;
        k++;
    }
    while (j < n2) {
        a[k] = r[j];
        j++;
        k++;
    }
}
void ms(int a[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        ms(a, l, m);
        ms(a, m + 1, r);
        merge(a, l, m, r);
    }
}
```

```
void pa(int a[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}
int main() {
    int a[] = {12, 11, 13, 5, 6, 7};
    int a_size = 6;
    printf("Given array is: ");
    pa(a, a_size);
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    ms(a, 0, a_size - 1);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array is: ");
    pa(a, a_size);
    printf("Merge sort took %lf seconds to execute\n", cpu_time_used);
    return 0;
}
```

OUTPUT:

Given array is: 12 11 13 5 6 7

Sorted array is: 5 6 7 11 12 13

Merge sort took 0.000003 seconds to execute

3. **AIM:** To implement the Quick Sort algorithm using the Divide and Conquer approach and measure its running time.

SOURCE CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to swap two elements

void swap(int *a, int *b) {

    int temp = *a;

    *a = *b;

    *b = temp;

}

// Partition function

int partition(int arr[], int low, int high) {

    int pivot = arr[high]; // Choose the last element as pivot

    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;
```

```
}
// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
```

```
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Original array: ");
    printArray(arr, n);
    // Measure time
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array: ");
    printArray(arr, n);
    printf("Time taken by Quick Sort: %f seconds\n", cpu_time_used);
    return 0;
}
```

OUTPUT:

Enter the number of elements: 5

Enter the elements of the array: 3 6 8 10 1

Original array: 3 6 8 10 1

Sorted array: 1 3 6 8 10

Time taken by Quick Sort: 0.000002 seconds

4. **AIM:** To implement Prim's Algorithm using the Greedy Method to estimate the Minimum-Cost Spanning Tree (MST) and measure its running time.

SOURCE CODE:

```
#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#include <time.h>

// Function to find the vertex with minimum key value

int minKey(int key[], bool mstSet[], int V) {

    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {

        if (!mstSet[v] && key[v] < min) {

            min = key[v];

            min_index = v;

        }

    }

    return min_index;

}

// Function to print the constructed MST

void printMST(int parent[], int graph[10][10], int V) {

    printf("Edge   Weight\n");

    for (int i = 1; i < V; i++) {

        printf("%d - %d    %d\n", parent[i], i, graph[i][parent[i]]);

    }
```

```
}

// Prim's algorithm for MST

void primMST(int graph[10][10], int V) {

    int parent[V];    // Array to store constructed MST

    int key[V];       // Key values used to pick minimum weight edge

    bool mstSet[V];   // To represent set of vertices included in MST

    // Initialize all keys as INFINITE

    for (int i = 0; i < V; i++) {

        key[i] = INT_MAX;

        mstSet[i] = false;

    }

    // Always include first vertex in MST

    key[0] = 0;      // Make key 0 so that this vertex is picked as first

    parent[0] = -1;  // First node is always root of MST

    for (int count = 0; count < V - 1; count++) {

        // Pick the minimum key vertex not yet included in MST

        int u = minKey(key, mstSet, V);

        // Add the picked vertex to the MST Set

        mstSet[u] = true;

        // Update key value and parent index of adjacent vertices

        for (int v = 0; v < V; v++) {

            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

                parent[v] = u;
```

```
                key[v] = graph[u][v];
            }
        }
    }
    // Print the constructed MST
    printMST(parent, graph, V);
}
int main() {
    int V;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    int graph[10][10];
    printf("Enter the adjacency matrix (enter 0 for no edge):\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            printf("%d ", graph[i][j]);
        }
```

```
        printf("\n");
    }
    // Measure time
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    printf("\nMinimum Spanning Tree:\n");
    primMST(graph, V);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nTime taken to estimate MST: %f seconds\n", cpu_time_used);
    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 5
Enter the adjacency matrix (enter 0 for no edge):
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0
Adjacency Matrix:
0 2 0 6 0
```

2 0 3 8 5

0 3 0 0 7

6 8 0 0 9

0 5 7 9 0

Minimum Spanning Tree:

Edge   Weight

0 - 1   2

1 - 2   3

0 - 3   6

1 - 4   5

Time taken to estimate MST: 0.000002 seconds

5. **AIM:** To implement Dijkstra's Algorithm for finding the shortest path from a source vertex in a weighted graph and measure its running time.

SOURCE CODE:

```
#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#include <time.h>

#define V 5 // Number of vertices in the graph

// Function to find the vertex with minimum distance value

int minDistance(int dist[], bool sptSet[]) {

    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)

        if (!sptSet[v] && dist[v] <= min)

            min = dist[v], min_index = v;

    return min_index;

}

// Function to print the shortest path distances

void printSolution(int dist[]) {

    printf("Vertex \t Distance from Source\n");

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

        printf("%d \t %d\n", i, dist[i]);

}

// Dijkstra's algorithm for shortest path

void dijkstra(int graph[V][V], int src) {
```

```
    int dist[V];  // Output array: shortest distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is included

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

        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0; // Distance of source vertex from itself is always 0

    for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < V; v++)

            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])

                dist[v] = dist[u] + graph[u][v];

    }

    printSolution(dist);

}

int main() {

    int graph[V][V] = {

        {0, 10, 0, 30, 100},

        {10, 0, 50, 0, 0},

        {0, 50, 0, 20, 10},

        {30, 0, 20, 0, 60},

        {100, 0, 10, 60, 0}

    };

    clock_t start, end;

    double cpu_time_used;
```

```
    start = clock();

    dijkstra(graph, 0);

    end = clock();

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

    printf("Execution Time: %f seconds\n", cpu_time_used);

    return 0;

}
```

OUTPUT:

```
Vertex   Distance from Source

0        0

1        10

2        50

3        30

4        60


Execution Time: 0.000076 seconds
```

6. **AIM:** To develop a program that constructs an Optimal Binary Search Tree (BST) using Dynamic Programming and measures its execution time.

SOURCE CODE:

```
#include <stdio.h>

#include <limits.h>

#include <time.h>

// Function to compute sum of frequencies from i to j

int sum(int freq[], int i, int j) {

    int s = 0;

    for (int k = i; k <= j; k++)

        s += freq[k];

    return s;

}

// Function to construct the optimal BST

int optimalBST(int keys[], int freq[], int n) {

    int cost[n][n];

    // Initialize cost for single keys

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

        cost[i][i] = freq[i];

    // Compute cost for chains of length L

    for (int L = 2; L <= n; L++) {

        for (int i = 0; i <= n - L; i++) {

            int j = i + L - 1;

            cost[i][j] = INT_MAX;
```

```
            int sum_freq = sum(freq, i, j);
            for (int r = i; r <= j; r++) {
                int c = ((r > i) ? cost[i][r - 1] : 0) +
                        ((r < j) ? cost[r + 1][j] : 0) + sum_freq;
                if (c < cost[i][j])
                    cost[i][j] = c;
            }
        }
    }
    return cost[0][n - 1];
}
int main() {
    int keys[] = {15, 25, 35, 50};
    int freq[] = {5, 10, 20, 15};
    int n = sizeof(keys) / sizeof(keys[0]);
    clock_t start = clock();
    int cost = optimalBST(keys, freq, n);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Minimum cost of Optimal BST: %d\n", cost);
    printf("Execution time: %f seconds\n", time_taken);
    return 0;
}
```

OUTPUT:

Minimum cost of Optimal BST: 70

Execution time: 0.000002 seconds

7. **AIM:** To develop a program that solves the Traveling Salesperson Problem (TSP) using Dynamic Programming and measures its execution time.

SOURCE CODE:

```
#include <stdio.h>

#include <limits.h>

#include <time.h>

#define INF 9999

#define N 4

// Function to find the minimum cost path using recursion

int tsp(int graph[N][N], int visited[], int pos, int count, int cost, int min_cost) {

    if (count == N && graph[pos][0]) {

        return (cost + graph[pos][0] < min_cost) ? cost + graph[pos][0] : min_cost;

    }

    for (int i = 0; i < N; i++) {

        if (!visited[i] && graph[pos][i]) {

            visited[i] = 1;

            min_cost = tsp(graph, visited, i, count + 1, cost + graph[pos][i], min_cost);

            visited[i] = 0;

        }

    }

    return min_cost;

}

int main() {

    int graph[N][N] = {
```

```
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
    int visited[N] = {0};
    int min_cost;
    // Mark the starting city as visited
    visited[0] = 1;
    // Measure execution time
    clock_t start_time = clock();
    min_cost = tsp(graph, visited, 0, 1, 0, INF);
    clock_t end_time = clock();
    double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    // Display result
    printf("Minimum cost of Traveling Salesperson Problem: %d\n", min_cost);
    printf("Execution time: %f seconds\n", execution_time);
    return 0;
}
```

OUTPUT:

Minimum cost of Traveling Salesperson Problem: 80

Execution time: 0.000045 seconds

8. **AIM:** To develop a program that solves the **8-Queens Problem** using backtracking and measures its execution time.

SOURCE CODE:

```
#include <stdio.h>

#include <time.h>

#define N 8

// Function to print the board

void printSolution(int board[N][N]) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++)

            printf("%d ", board[i][j]);

        printf("\n");

    }

    printf("\n");

}

// Function to check if placing a queen at (row, col) is safe

int isSafe(int board[N][N], int row, int col) {

    int i, j;

    // Check the left side of the current row

    for (i = 0; i < col; i++)

        if (board[row][i])

            return 0;

    // Check upper diagonal on the left side

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
```

```
        if (board[i][j])
            return 0;
    // Check lower diagonal on the left side
    for (i = row, j = col; i < N && j >= 0; i++, j--)
        if (board[i][j])
            return 0;
    return 1;
}
// Recursive function to solve the N-Queens problem
int solveNQUtil(int board[N][N], int col) {
    if (col >= N)
        return 1;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1))
                return 1;
            board[i][col] = 0;  // Backtrack
        }
    }
    return 0;
}
// Function to solve N-Queens and measure execution time
```

```
void solveNQ() {
    int board[N][N] = {0};
    clock_t start = clock();
    if (solveNQUtil(board, 0)) {
        clock_t end = clock();
        printSolution(board);
        printf("Execution time: %f seconds\n", ((double)(end - start)) / CLOCKS_PER_SEC);
    } else {
        printf("Solution does not exist\n");
    }
}
// Main function
int main() {
    solveNQ();
    return 0;
}
```

OUTPUT:

```
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: 0.000013 seconds

9. **AIM:** To develop a program that solves the **Graph Coloring Problem** using backtracking and measures its execution time.

SOURCE CODE:

```
#include <stdio.h>

#include <time.h>

#define V 4  // Number of vertices in the graph

// Function to print the solution

void printSolution(int color[]) {

    printf("Solution Exists: Following are the assigned colors\n");

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

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

    printf("\n");

}

// Function to check if it's safe to assign color c to vertex v

int isSafe(int v, int graph[V][V], int color[], int c) {

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

        if (graph[v][i] && c == color[i])

            return 0;

    return 1;

}

// Backtracking function to solve graph coloring

int graphColoringUtil(int graph[V][V], int m, int color[], int v) {

    if (v == V)

        return 1;
```

```
    for (int c = 1; c <= m; c++) {
        if (isSafe(v, graph, color, c)) {
            color[v] = c;
            if (graphColoringUtil(graph, m, color, v + 1))
                return 1;
            color[v] = 0;  // Backtrack
        }
    }
    return 0;
}
// Function to solve the graph coloring problem
void graphColoring(int graph[V][V], int m) {
    int color[V] = {0}; // Initialize all vertices with 0 (uncolored)
    clock_t start = clock();
    if (!graphColoringUtil(graph, m, color, 0)) {
        printf("Solution does not exist\n");
        return;
    }
    clock_t end = clock();
    printSolution(color);
    printf("Execution time: %f seconds\n", ((double)(end - start)) / CLOCKS_PER_SEC);
}
```

```
// Main function
int main() {
    int graph[V][V] = {
        {0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0}
    };
    int m = 3;  // Number of colors
    graphColoring(graph, m);
    return 0;
}
```

OUTPUT:

Solution Exists: Following are the assigned colors

1 2 3 2

Execution time: 0.000002 seconds

10. AIM: To develop a program that finds a **Hamiltonian Cycle** using **backtracking** and measures its execution time.

SOURCE CODE:

```
#include <stdio.h>

#include <time.h>

#define V 5  // Number of vertices in the graph

// Function to print the Hamiltonian cycle

void printSolution(int path[]) {

   printf("Solution Exists: Hamiltonian Cycle is:\n");

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

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

   printf(" %d \n", path[0]);  // Print the first vertex again to complete the cycle

}

// Function to check if vertex v can be added to the Hamiltonian cycle

int isSafe(int v, int graph[V][V], int path[], int pos) {

   if (graph[path[pos - 1]][v] == 0)

      return 0;  // Check if there is an edge

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

      if (path[i] == v)

         return 0;  // Check if vertex is already included

   return 1;

}

// Recursive utility function to find a Hamiltonian cycle

int hamiltonianCycleUtil(int graph[V][V], int path[], int pos) {

   if (pos == V) {

      if (graph[path[pos - 1]][path[0]] == 1)  // Check if last vertex connects to the first

         return 1;

      else

         return 0;
```

```
    }
    for (int v = 1; v < V; v++) {
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;
            if (hamiltonianCycleUtil(graph, path, pos + 1))
                return 1;
            path[pos] = -1;  // Backtracking
        }
    }
    return 0;
}
// Function to find the Hamiltonian cycle
void hamiltonianCycle(int graph[V][V]) {
    int path[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;
    path[0] = 0;  // Start from vertex
    clock_t start = clock();
    if (!hamiltonianCycleUtil(graph, path, 1)) {
        printf("Solution does not exist\n");
        return;}
    clock_t end = clock();
    printSolution(path);
    printf("Execution time: %f seconds\n", ((double)(end - start)) / CLOCKS_PER_SEC);
}
// Main function
int main() {
    int graph[V][V] = {
        {0, 1, 0, 1, 0},
```

```
        {1, 0, 1, 1, 1},
        {0, 1, 0, 0, 1},
        {1, 1, 0, 0, 1},
        {0, 1, 1, 1, 0}
    };
    hamiltonianCycle(graph);
    return 0;
}
```

OUTPUT:

Solution Exists: Hamiltonian Cycle is:

0 1 2 4 3 0

Execution time: 0.000002 seconds

11. **AIM:** To implement the **0/1 Knapsack problem** using **Dynamic Programming** and measure its **execution time**.

SOURCE CODE:

```
#include <stdio.h>

#include <time.h>

#define MAX_ITEMS 100

#define MAX_CAPACITY 1000

// Function to find the maximum of two numbers

int max(int a, int b) {

  return (a > b) ? a : b;

}

// Function to solve the 0/1 Knapsack problem using Dynamic Programming

int knapsack(int W, int wt[], int val[], int n) {

  int dp[n + 1][W + 1];

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

    for (int w = 0; w <= W; w++) {

      if (i == 0 || w == 0)

        dp[i][w] = 0;

      else if (wt[i - 1] <= w)

        dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);

      else

        dp[i][w] = dp[i - 1][w];

    }

  }

  return dp[n][W];

}

int main() {

  int n, W;

  int val[MAX_ITEMS], wt[MAX_ITEMS];
```

```
    // Input number of items and knapsack capacity
    printf("Enter the number of items: ");
    scanf("%d", &n);
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);
    // Input values and weights of items
    printf("Enter the values of the items:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &val[i]);
    printf("Enter the weights of the items:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &wt[i]);
    // Measure execution time
    clock_t start, end;
    start = clock();
    int maxValue = knapsack(W, wt, val, n);
    end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    // Output results
    printf("The maximum value that can be obtained is: %d\n", maxValue);
    printf("Time taken to solve the Knapsack problem: %.6f seconds\n", time_taken);
    return 0;
}
```

OUTPUT:

Enter the number of items: 3

Enter the capacity of the knapsack: 50

Enter the values of the items:

60 100 120

Enter the weights of the items:

10 20 30

The maximum value that can be obtained is: 220

Time taken to solve the Knapsack problem: 0.000123 seconds