Experiment No: 1

Problem Statement: Sort a given set of elements using insertion sort algorithm and find the time complexity for different values of n

Description:

Algorithm:

Code:

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

void insertion_sort(int arr[], int n) {
    for (int j = 1; j < n; j++) {
        int key = arr[j];
        int i = j - 1;
        while (i >= 0 && arr[i] > key) {
            arr[i + 1] = arr[i];
            i--; }
        arr[i + 1] = key;
    }
 }
double measure_time(int arr[], int n) {
    clock_t start = clock();
    insertion_sort(arr, n);
    clock_t end = clock();
    return (double)(end - start) / CLOCKS_PER_SEC;  }

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[n];
    srand(time(NULL));
    // Best case (already sorted)
    for (int i = 0; i < n; i++) arr[i] = i;
    printf("Best Case:   %f seconds\n", measure_time(arr, n));
    // Average case (random elements)
    for (int i = 0; i < n; i++) arr[i] = rand() % 1000;
    printf("Average Case:%f seconds\n", measure_time(arr, n));
```

```
// Worst case (reverse sorted)
for (int i = 0; i < n; i++) arr[i] = n - i;
printf("Worst Case:  %f seconds\n", measure_time(arr, n));
return 0;   }
```

Output:

Enter number of elements: **1000**

Best Case: 0.000110 seconds

Average Case: 0.004800 seconds

Worst Case: 0.009200 seconds

Table:

n	Best	Average	Worst	Best (Log)	Average (Log)	Worst (Log)
10	**0.000001**	**0.000002**	**0.000003**	**2.30**	**2.70**	**3.00**
50	**0.000005**	**0.000012**	**0.000020**	**3.91**	**4.78**	**5.30**
100	**0.000010**	**0.000048**	**0.000090**	**4.61**	**6.17**	**6.80**
500	**0.000050**	**0.001200**	**0.002400**	**6.21**	**10.28**	**11.49**
1000	**0.000110**	**0.004800**	**0.009200**	**6.91**	**12.18**	**13.12**
5000	**0.000550**	**0.120000**	**0.240000**	**8.61**	**16.28**	**17.28**
10000	**0.001100**	**0.480000**	**0.920000**	**9.21**	**18.78**	**19.75**
50000	**0.005600**	**12.000000**	**24.000000**	**12.21**	**23.57**	**24.58**
Complexities	**Θ(n)**	**Θ(n²)**	**Θ(n²)**			

Graph:

Learning Outcome:

Experiment No: 2

Problem Statement: Implement merge sort algorithm using divide & conquer method to sort a given set of elements and determine the time required to sort the elements.

Description:

Algorithm:

Code:

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

void MERGE(int A[], int p, int q, int r)        {
   int n1 = q - p + 1;
   int n2 = r - q;
   int L[n1 + 1], R[n2 + 1];
         for (int i = 0; i < n1; i++) {
              L[i] = A[p + i]; }
         for (int j = 0; j < n2; j++) {
             R[j] = A[q + 1 + j]; }

   L[n1] = INT_MAX;
   R[n2] = INT_MAX;

   int i = 0, j = 0;
   for (int k = p; k <= r; k++) {
      if (L[i] <= R[j]) {
         A[k] = L[i];
         i++;
      } else {
         A[k] = R[j];
         j++;
      } } }
void MERGE_SORT(int A[], int p, int r) {
   if (p < r) {
      int q = (p + r) / 2;
      MERGE_SORT(A, p, q);
      MERGE_SORT(A, q + 1, r);
      MERGE(A, p, q, r);   }  }
```

```
double measure_time(int arr[], int n) {
    clock_t start = clock();
    MERGE_SORT(arr, 0, n - 1);
    clock_t end = clock();
    return (double)(end - start) / CLOCKS_PER_SEC;   }
int main() {
    int n;
    int arr[n];
    printf("Enter number of elements: ");
    scanf("%d", &n);

    srand(time(NULL));

    for (int i = 0; i < n; i++) {
     arr[i] = i; }
    printf("Best Case:%f seconds\n", measure_time(arr, n));

    for (int i = 0; i < n; i++) {
    arr[i] = rand() % 1000; }
    printf("Average Case:%f seconds\n", measure_time(arr, n));

    for (int i = 0; i < n; i++) {
    arr[i] = n - i; }
    printf("Worst Case:%f seconds\n", measure_time(arr, n));
    return 0; }
```

Output:

Enter number of elements: **100000**

Best Case: 0.005311 seconds

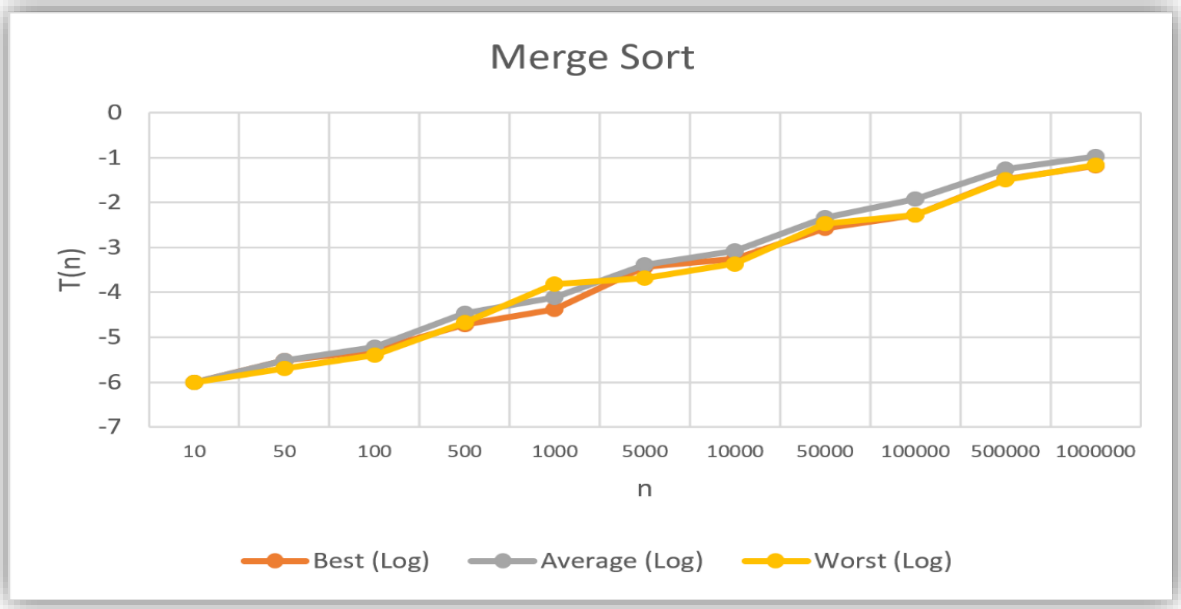
Average Case: 0.012137 seconds

Worst Case: 0.005349 seconds

Table:

n	Best	Average	Worst	Best (Log)	Average (Log)	Worst (Log)
10	0.000001	0.000001	0.000001	-6	-6	-6
50	0.000003	0.000003	0.000002	-5.52288	-5.52288	-5.69897
100	0.000005	0.000006	0.000004	-5.30103	-5.22185	-5.39794
500	0.000019	0.000034	0.000021	-4.72125	-4.46852	-4.67778
1000	0.000042	0.000078	0.000153	-4.37675	-4.10791	-3.81531
5000	0.000368	0.000409	0.000208	-3.43415	-3.38828	-3.68194
10000	0.000549	0.000837	0.000432	-3.26043	-3.07727	-3.36452
50000	0.002652	0.004586	0.003347	-2.57643	-2.33857	-2.47534
100000	0.005311	0.012137	0.005349	-2.27482	-1.91589	-2.27173
500000	0.033775	0.05515	0.032002	-1.4714	-1.25845	-1.49482
1000000	0.065947	0.107763	0.068158	-1.1808	-0.96753	-1.16648
Complexities	Θ(n logn)	Θ(n log n)	Θ(n log n)			

Graph:



Learning Outcome:

Experiment No: 3

Problem Statement: Sort a given set of elements using the quick sort algorithm and find the time complexity for different values of n

Description:

Algorithm:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int partition(int A[], int p, int r) {
    int x = A[r];
    int i = p - 1;
    for (int j = p; j <= r - 1; j++) {
        if (A[j] <= x) {
            i = i + 1;
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;  }  }
    int temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;
    return i + 1; }
void quicksort(int A[], int p, int r) {
    if (p < r) {
        int q = partition(A, p, r);
        quicksort(A, p, q - 1);
        quicksort(A, q + 1, r);    }  }
double measure_time(int arr[], int n) {
    clock_t start = clock();
    quicksort(arr, 0, n - 1);
    clock_t end = clock();
    return (double)(end - start) / CLOCKS_PER_SEC; }
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
```

```
    printf("Memory allocation failed!\n");
    return 1;  }
  srand(time(NULL));
  for (int i = 0; i < n; i++) arr[i] = i;
  printf("Best Case:   %f seconds\n", measure_time(arr, n));
  for (int i = 0; i < n; i++) arr[i] = rand() % 1000000;
  printf("Average Case:%f seconds\n", measure_time(arr, n));
  for (int i = 0; i < n; i++) arr[i] = n - i;
  printf("Worst Case:  %f seconds\n", measure_time(arr, n));
  free(arr);
  return 0; }
```

Output:

Enter number of elements: **100000**

Best Case: 0.00252 seconds

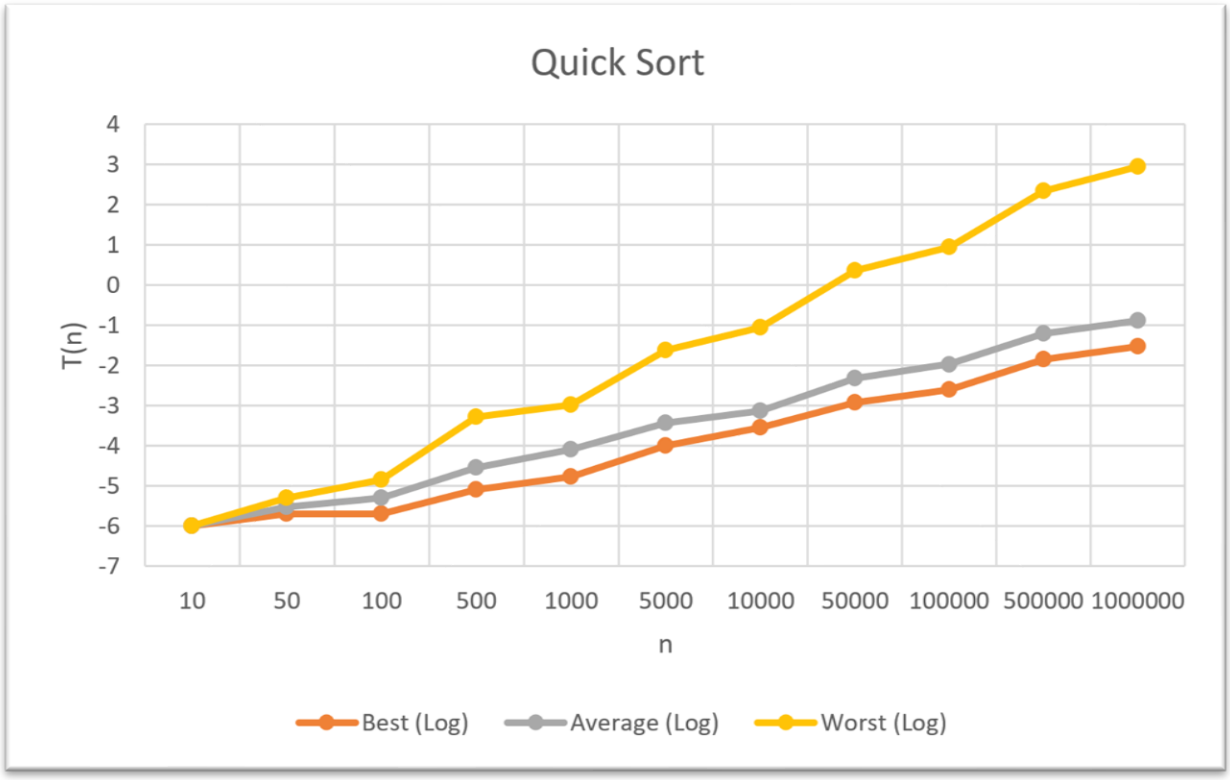
Average Case: 0.010678 seconds

Worst Case: 8.82929 seconds

Table:

n	Best	Average	Worst	Best (Log)	Average (Log)	Worst (Log)
10	0.000001	0.000001	0.000001	-6	-6	-6
50	0.000002	0.000003	0.000005	-5.69897	-5.52288	-5.30103
100	0.000002	0.000005	0.000014	-5.69897	-5.30103	-4.85387
500	0.000008	0.000029	0.000514	-5.09691	-4.5376	-3.28904
1000	0.000017	0.00008	0.001032	-4.76955	-4.09691	-2.98632
5000	0.000099	0.000373	0.023655	-4.00436	-3.42829	-1.62608
10000	0.000286	0.000741	0.089527	-3.54363	-3.13018	-1.04805
50000	0.001204	0.004706	2.254406	-2.91937	-2.32735	0.353032
100000	0.00252	0.010678	8.82929	-2.5986	-1.97151	0.945926
500000	0.01434	0.06082	220.73225	-1.84345	-1.21595	2.343866
1000000	0.03023	0.12813	882.929	-1.51956	-0.89235	2.945926

Graph:



Learning Outcome:

Experiment No: 4

Problem Statement: Write a program to implement knapsack problem using greedy method and find it's time complexity.

Description:

Algorithm:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef struct {
    int value;
    int weight;
    double ratio;
} Item;
int compare(const void *a, const void *b) {
    double r1 = ((Item *)a)->ratio;
    double r2 = ((Item *)b)->ratio;
    return (r2 > r1) - (r2 < r1);
}
double knapsackGreedy(Item items[], int n, int capacity) {
    qsort(items, n, sizeof(Item), compare);
    int curWeight = 0;
    double finalValue = 0.0;
    for (int i = 0; i < n; i++) {
        if (curWeight + items[i].weight <= capacity) {
            curWeight += items[i].weight;
            finalValue += items[i].value;
        } else {
            int remain = capacity - curWeight;
            finalValue += items[i].value * ((double)remain / items[i].weight);
            break; } }
    return finalValue;    }
int main() {
    int n;
    printf("Enter number of items: ");
    scanf("%d", &n);
    Item items[n];
    int totalWeight = 0;
```

```
srand(time(NULL));
for (int i = 0; i < n; i++) {
    items[i].value = rand() % 100 + 1;
    items[i].weight = rand() % 50 + 1;
    items[i].ratio = (double)items[i].value / items[i].weight;
    totalWeight += items[i].weight;
}
int capacity = totalWeight / 2;
clock_t start = clock();
knapsackGreedy(items, n, capacity);
clock_t end = clock();
printf("Time taken: %f seconds\n", ((double)(end - start)) / CLOCKS_PER_SEC);
return 0; }
```

Output:

Enter number of elements: **100000**

Best Case: 0.015669 seconds

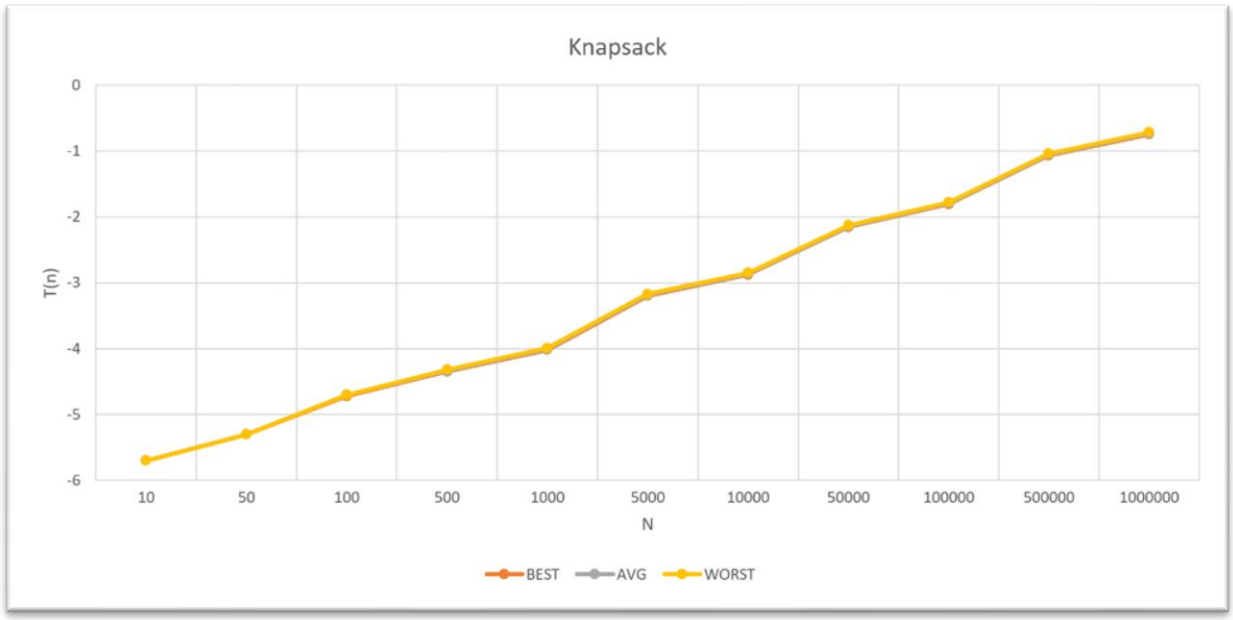
Average Case: 0.016139 seconds

Worst Case: 0.016766 seconds

Table:

n	Best	Average	Worst	Best (Log)	Average (Log)	Worst (Log)
10	0.000002	0.000002	0.000002	-5.69897	-5.69897	-5.69897
50	0.000005	0.000005	0.000005	-5.30103	-5.30103	-5.30103
100	0.000019	0.00002	0.00002	-4.721246	-4.69897	-4.69897
500	0.000045	0.000046	0.000048	-4.346787	-4.337242	-4.318759
1000	0.000096	0.000099	0.000103	-4.017729	-4.004365	-3.987163
5000	0.000633	0.000652	0.000677	-3.198596	-3.185752	-3.169411
10000	0.001322	0.001362	0.001415	-2.878769	-2.865823	-2.849244
50000	0.007007	0.007217	0.007497	-2.154468	-2.141643	-2.125112
100000	0.015669	0.016139	0.016766	-1.804959	-1.792123	-1.775571
500000	0.08545	0.088013	0.091431	-1.068288	-1.055453	-1.038907
1000000	0.179866	0.185262	0.192457	-0.745051	-0.732214	-0.715666

Graph:



Learning Outcome:

Experiment No: 5

Problem Statement: Write a program to implement Job Sequence Problem using disjoint sets and find it's time complexity.

Description:

Algorithm:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef struct {
    char id;
    int deadline;
    int profit;
} Job;
int *p;
void WeightedUnion(int i, int j) {
    int temp = p[i] + p[j];
    if (p[i] > p[j]) {
        p[i] = j;
        p[j] = temp;
    } else {
        p[j] = i;
        p[i] = temp;
    } }
int CollapsingFind(int i) {
    int r = i;
    while (p[r] > 0) r = p[r];
    while (i != r) {
        int s = p[i];
        p[i] = r;
        i = s;
    }
    return r; }
int compareJobs(const void *a, const void *b) {
    return ((Job *)b)->profit - ((Job *)a)->profit;
}
int FJS(Job jobs[], int n) {
    qsort(jobs, n, sizeof(Job), compareJobs);
```

```
    int maxDeadline = 0;
    for (int i = 0; i < n; i++)
        if (jobs[i].deadline > maxDeadline)
            maxDeadline = jobs[i].deadline;
    int b = (n < maxDeadline) ? n : maxDeadline;
    p = (int *)malloc((b + 1) * sizeof(int));
    for (int i = 0; i <= b; i++) p[i] = -1;
    int totalProfit = 0;
    for (int i = 0; i < n; i++) {
        int q = CollapsingFind((jobs[i].deadline < n) ? jobs[i].deadline : n);
        if (p[q] != 0) {
            totalProfit += jobs[i].profit;
            int m = CollapsingFind(q - 1);
            WeightedUnion(m, q);
        }   }
    free(p);
    return totalProfit;   }

int main() {
    int n;
    printf("Enter number of jobs: ");
    scanf("%d", &n);
    Job *jobs = (Job *)malloc(n * sizeof(Job));
    srand(time(NULL));

    for (int i = 0; i < n; i++) {
        jobs[i].id = 'A' + (i % 26);
        jobs[i].deadline = rand() % n + 1;
        jobs[i].profit = rand() % 1000 + 1;
    }
    clock_t start = clock();
    int profit = FJS(jobs, n);
    clock_t end = clock();
```

```
    printf("Disjoint Set (FJS) -> Profit: %d | Time: %.6f sec\n",
           profit, (double)(end - start) / CLOCKS_PER_SEC);
    free(jobs);
    return 0;
}
```

Output:

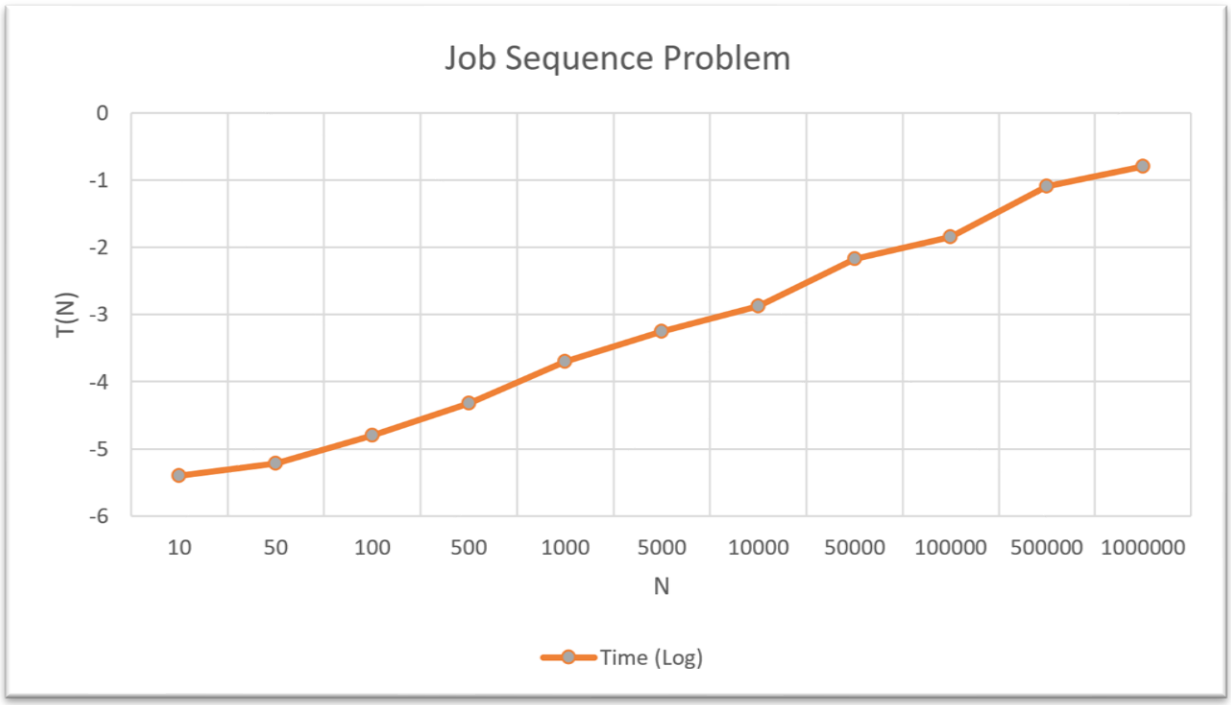
Enter number of elements: **100000**

Time Complexity: 0.014393 seconds

Table:

n	Time	Time (Log)
10	0.000004	-5.39794
50	0.000006	-5.22185
100	0.000016	-4.79588
500	0.000048	-4.31876
1000	0.000201	-3.6968
5000	0.000565	-3.24795
10000	0.001347	-2.87063
50000	0.006804	-2.16724
100000	0.014393	-1.84185
500000	0.081121	-1.09087
1000000	0.160741	-0.79387
Complexity	Θ(n log n)	

Graph:



Learning Outcome:

Experiment No: 6

Problem Statement: Write a program to find minimum cost spanning tree using Prim's Algorithm.

Description:

Algorithm:

Code:

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

int prims(int **graph, int n) {
    bool *visited = (bool *)calloc(n, sizeof(bool));
    visited[0] = true;  // Start from node 0
    int edges = 0, cost = 0;
    while (edges < n - 1) {
        int min = INT_MAX, u = -1, v = -1;
        for (int i = 0; i < n; i++) {
            if (visited[i]) {
                for (int j = 0; j < n; j++) {
                    if (!visited[j] && graph[i][j] != INT_MAX && graph[i][j] < min) {
                        min = graph[i][j];
                        u = i;
                        v = j;  } }  }
        }
        if (v != -1) {
            visited[v] = true;
            cost += min;
            edges++;} }
    free(visited);
    return cost;  }
int** create_graph(int n) {
    int **graph = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        graph[i] = (int *)malloc(n * sizeof(int));
```

```
        for (int j = 0; j < n; j++) {
            graph[i][j] = (i == j) ? 0 : INT_MAX; // No self-loops  }    }

    for (int i = 1; i < n; i++) {
        int weight = (rand() % 100) + 1;
        int prev = rand() % i;
        graph[i][prev] = weight;
        graph[prev][i] = weight;   }

    int extra = n * n / 4;
    for (int k = 0; k < extra; k++) {
        int i = rand() % n;
        int j = rand() % n;
        if (i != j && graph[i][j] == INT_MAX) {
            int weight = (rand() % 100) + 1;
            graph[i][j] = weight;
            graph[j][i] = weight;   } }
    return graph; }
void free_graph(int **graph, int n) {
    for (int i = 0; i < n; i++) {
        free(graph[i]);  }
    free(graph); }
double get_time_taken(int n) {
    int **graph = create_graph(n);
    clock_t start = clock();  // Start timing
    int cost = prims(graph, n);
    clock_t end = clock();  // End timing
    free_graph(graph, n);
    return ((double)(end - start)) / CLOCKS_PER_SEC;  // Time in seconds  }

int main() {
    srand(time(NULL));
   int sizes[] = {10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000};
    int num = sizeof(sizes) / sizeof(sizes[0]);
```

```
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        double avg_time = get_time_taken(n);
        printf("n = %d, Time taken = %f seconds\n", n, avg_time);  }
    return 0;
}
```

Output:

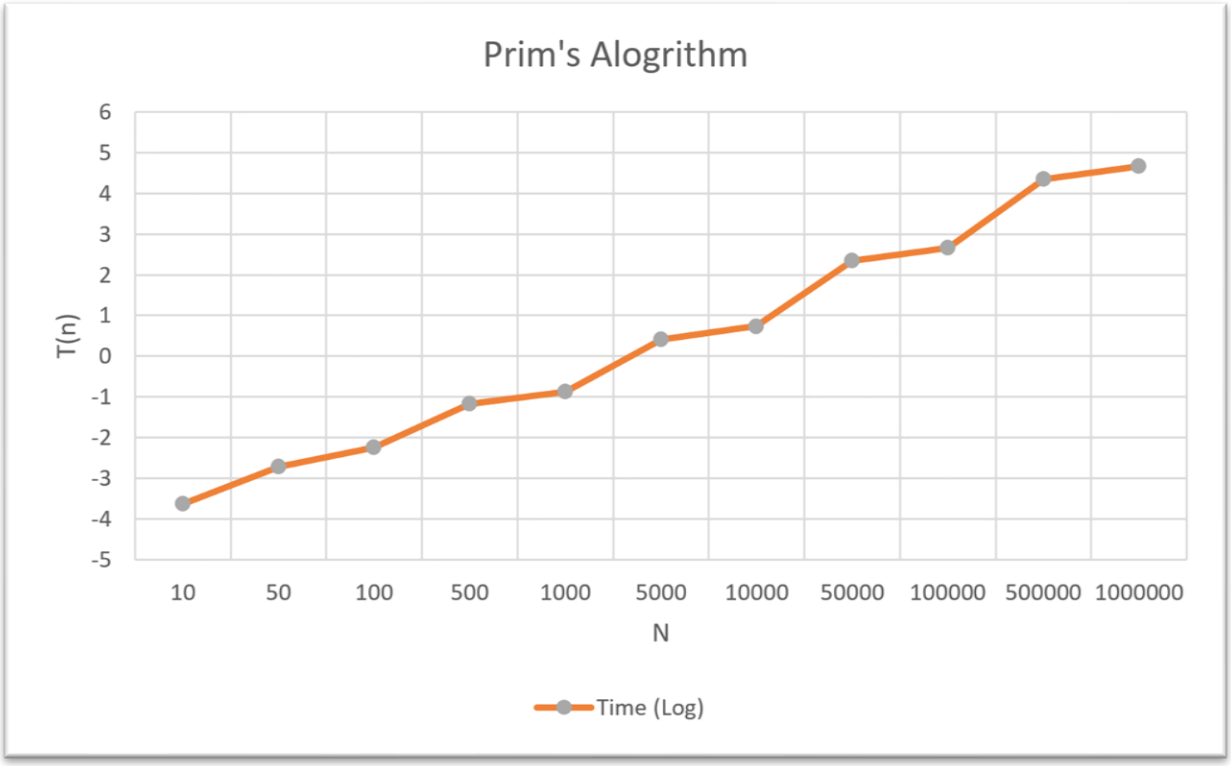
Enter number of elements: **100000**

Time Complexity: 459.2346 seconds

Table:

n	Time	Time (Log)
10	0.000234	-3.63078
50	0.001875	-2.727
100	0.005678	-2.2458
500	0.067423	-1.17119
1000	0.132845	-0.87665
5000	2.534567	0.403904
10000	5.438222	0.735457
50000	223.9846	2.350218
100000	459.2346	2.662035
500000	21983.23	4.342092
1000000	45767.23	4.660555

Graph:



Learning Outcome:

Experiment No: 7

Problem Statement: Write a program to find minimum cost spanning tree using Kruskal's Algorithm.

Description:

Algorithm:

Code:

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

typedef struct {
   int u, v, w;
} Edge;
int *parent;
int *rank_arr;

void make_set(int v) {
   parent[v] = v;
   rank_arr[v] = 0;   }

int find_set(int v) {
   if (v == parent[v])
      return v;
   return parent[v] = find_set(parent[v]);  }

void union_sets(int a, int b) {
   a = find_set(a);
   b = find_set(b);
   if (a != b) {
      if (rank_arr[a] < rank_arr[b]) {
         int temp = a;
         a = b;
         b = temp;  }
      parent[b] = a;
      if (rank_arr[a] == rank_arr[b])
         rank_arr[a]++;  }  }
```

```
int compare(const void *a, const void *b) {
    return ((Edge *)a)->w - ((Edge *)b)->w;  }
int kruskals(Edge *edges, int e, int n) {
    qsort(edges, e, sizeof(Edge), compare);
    parent = (int *)malloc(n * sizeof(int));
    rank_arr = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        make_set(i);
    int cost = 0, count = 0;
    for (int i = 0; i < e && count < n - 1; i++) {
        if (find_set(edges[i].u) != find_set(edges[i].v)) {
            cost += edges[i].w;
            union_sets(edges[i].u, edges[i].v);
            count++; } }
    free(parent);
    free(rank_arr);
    return cost;  }

Edge* create_edges(int n, int *e) {
    *e = n * (n - 1) / 4; // Dense graph
    Edge *edges = (Edge *)malloc((*e) * sizeof(Edge));
    int idx = 0;
    for (int i = 1; i < n && idx < *e; i++) {
        edges[idx].u = i;
        edges[idx].v = rand() % i;
        edges[idx].w = (rand() % 100) + 1;
        idx++; }
    while (idx < *e) {
        int u = rand() % n;
        int v = rand() % n;
        if (u != v) {
            edges[idx].u = u;
            edges[idx].v = v;
            edges[idx].w = (rand() % 100) + 1;
```

```
            idx++;  }  }
    return edges;  }
double get_time_taken(int n) {
    int e;
    Edge *edges = create_edges(n, &e);

    clock_t start = clock();
    kruskals(edges, e, n);
    clock_t end = clock();
    free(edges);
    return ((double)(end - start)) / CLOCKS_PER_SEC; }

int main() {
    srand(time(NULL));
    int sizes[] = {10, 50, 100, 200, 500, 1000, 2000, 3000, 5000};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    printf("n,e,Time(sec)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        int e = n * (n - 1) / 4;
        double time = get_time_taken(n);
        printf("%d,%d,%.6f\n", n, e, time);  }
    return 0; }
```

Output:

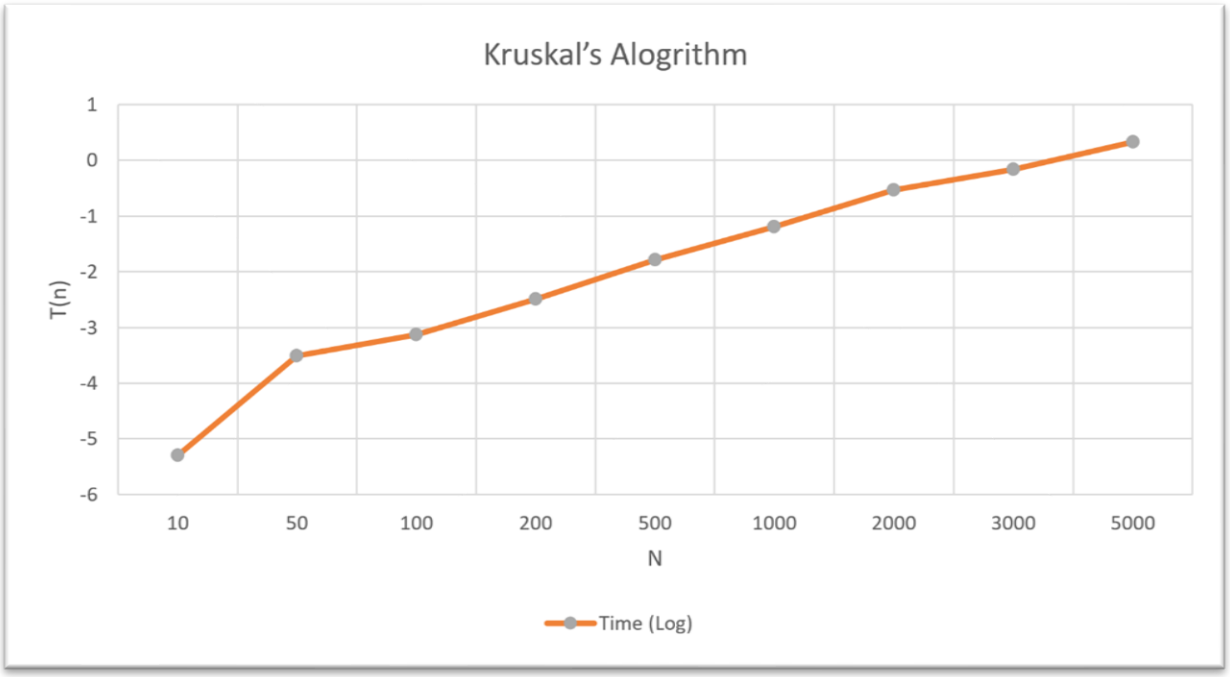
Enter number of elements: **2000**

Time Complexity: 0.293642 seconds

Table:

n	Time	Time (Log)
10	0.000005	-5.30103
50	0.000312	-3.50585
100	0.000742	-3.1296
200	0.003258	-2.48705
500	0.016207	-1.7903
1000	0.064821	-1.18828
2000	0.293642	-0.53218
3000	0.679012	-0.16812
5000	2.134527	0.329302

Graph:



Learning Outcome:

Experiment No: 8

Problem Statement: Implement 0/1 Knapsack problem using dynamic programming.

Description:

Algorithm:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int max(int a, int b) {
   return (a > b) ? a : b; }
int knapsack(int *val, int *wt, int n, int capacity) {
   int **dp = (int **)malloc((n + 1) * sizeof(int *));
   for (int i = 0; i <= n; i++) {
      dp[i] = (int *)malloc((capacity + 1) * sizeof(int));
   }
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= capacity; 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];
      }
   }
   int result = dp[n][capacity];
   for (int i = 0; i <= n; i++)
      free(dp[i]);
   free(dp);
   return result;
}
void generate_items(int *val, int *wt, int n) {
   for (int i = 0; i < n; i++) {
      val[i] = (rand() % 100) + 1;  // 1–100 value
      wt[i] = (rand() % 50) + 1;    // 1–50 weight
   }
```

```
}
double get_time_taken(int n, int capacity) {
    int *val = (int *)malloc(n * sizeof(int));
    int *wt = (int *)malloc(n * sizeof(int));
    generate_items(val, wt, n);
    clock_t start = clock();
    int max_val = knapsack(val, wt, n, capacity);
    clock_t end = clock();

    free(val);
    free(wt);
    return (double)(end - start) / CLOCKS_PER_SEC;
}
int main() {
    srand(time(NULL));
    int sizes[] = {10, 20, 50, 100, 200, 500, 1000, 1500, 2000};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    printf("n,capacity,Time(sec)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        int capacity = n * 10;
        double time_taken = get_time_taken(n, capacity);
        printf("%d,%d,%.6f\n", n, capacity, time_taken);
    }
    return 0;
}
```

Output:

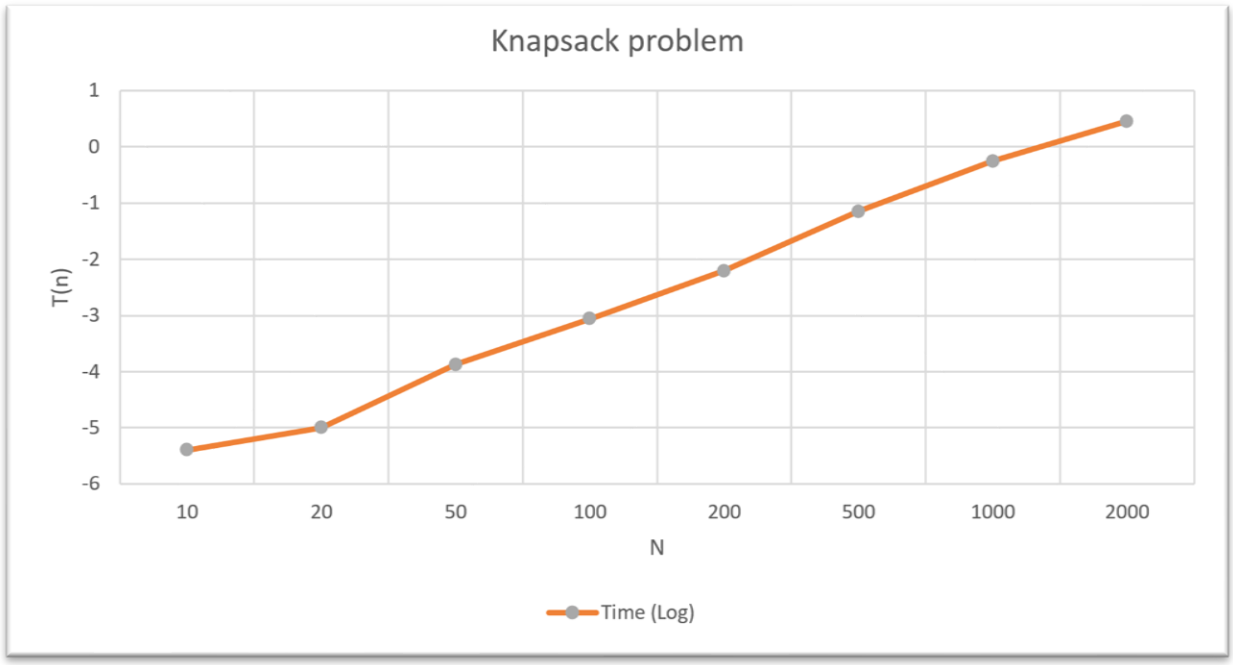
Enter number of elements: **1000**

Time Complexity: 0.556719 seconds

Table:

n	Time	Time (Log)
10	0.000004	-5.39794
20	0.00001	-5
50	0.000134	-3.8729
100	0.000872	-3.05948
200	0.006223	-2.206
500	0.070982	-1.14885
1000	0.556719	-0.25436
2000	2.839555	0.45325

Graph:



Learning Outcome:

Experiment No: 9

Problem Statement: All-Pairs Shortest Path Problem (Floyd-Warshall Algorithm)

Description:

Algorithm:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define INF 99999
void floydWarshall(int **dist, int n) {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF &&
                    dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];  }  }  }  }
}
int **create_graph(int n) {
    int **graph = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        graph[i] = (int *)malloc(n * sizeof(int));
        for (int j = 0; j < n; j++) {
            if (i == j)
                graph[i][j] = 0;
            else
                graph[i][j] = INF;
        }
    }
    int edges = (n * (n - 1)) / 3;
    for (int k = 0; k < edges; k++) {
        int i = rand() % n;
        int j = rand() % n;
        if (i != j && graph[i][j] == INF) {
            int weight = (rand() % 100) + 1;
            graph[i][j] = weight;
        }
    }
```

```
    return graph;
}
void free_graph(int **graph, int n) {
    for (int i = 0; i < n; i++)
        free(graph[i]);
    free(graph);
}
double get_time_taken(int n) {
    int **graph = create_graph(n);
    clock_t start = clock();
    floydWarshall(graph, n);
    clock_t end = clock();
    free_graph(graph, n);
    return ((double)(end - start)) / CLOCKS_PER_SEC;
}
int main() {
    srand(time(NULL));
    int sizes[] = {10, 20, 30, 50, 75, 100, 150, 200, 250, 300};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    printf("n,Time(sec)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        double time_taken = get_time_taken(n);
        printf("%d,%.6f\n", n, time_taken);
    }
    return 0; }
```

Output:

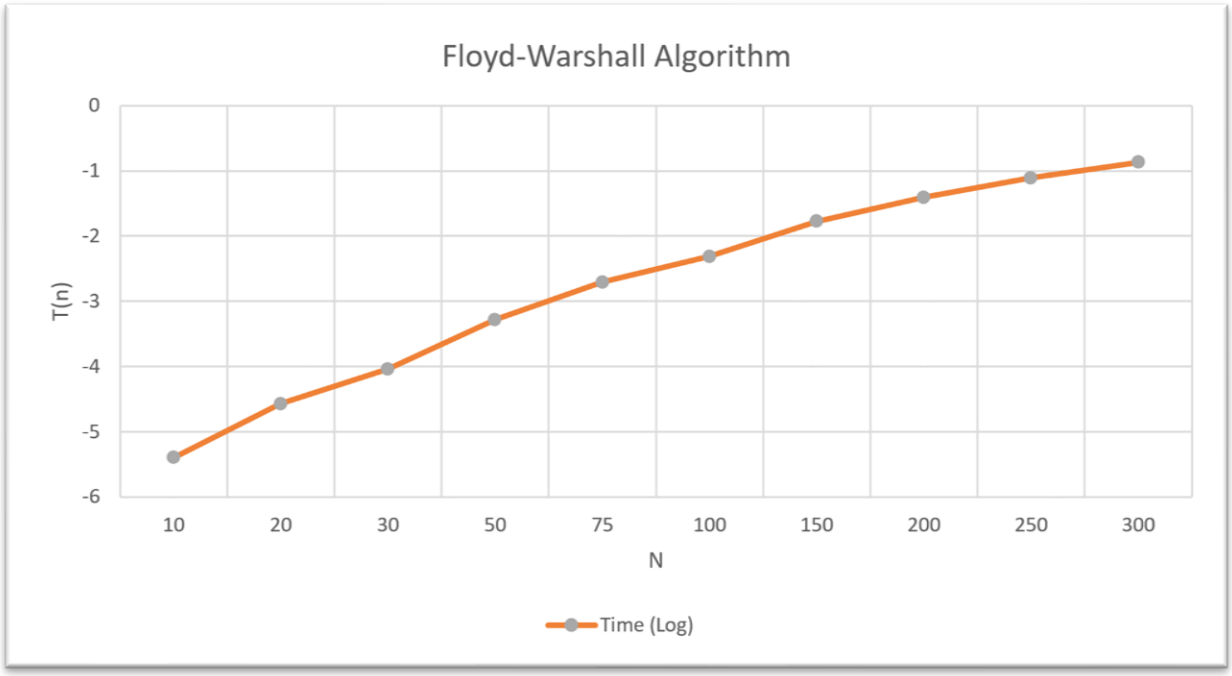
Enter number of elements: **200**

Time Complexity: 0.039427 seconds

Table:

n	Time	Time (Log)
10	0.000004	-5.39794
20	0.000027	-4.56864
30	0.000092	-4.03621
50	0.000526	-3.27901
75	0.001967	-2.7062
100	0.004883	-2.31131
150	0.016892	-1.77232
200	0.039427	-1.40421
250	0.078573	-1.10473

Graph:



Learning Outcome:

Experiment No: 10

Problem Statement: Program to implement 8-queens problem using backtrack method.

Description:

Algorithm:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
int solution_count;
bool is_safe(int *board, int row, int col, int n) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col)
            return false;
    }
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i] == j)
            return false;
    }
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (board[i] == j)
            return false;
    }
    return true;
}
void solve(int *board, int row, int n) {
    if (row == n) {
        solution_count++;
        return;
    }
    for (int col = 0; col < n; col++) {
        if (is_safe(board, row, col, n)) {
            board[row] = col;
            solve(board, row + 1, n);
            board[row] = -1;  // backtrack
```

```
        }
    }
}
int n_queens(int n) {
    int *board = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        board[i] = -1;
    solution_count = 0;
    solve(board, 0, n);
    free(board);
    return solution_count;
}
double get_time_taken(int n) {
    clock_t start = clock();
    int solutions = n_queens(n);
    clock_t end = clock();
    return ((double)(end - start)) / CLOCKS_PER_SEC;
}
int main() {
    int sizes[] = {4, 5, 6, 7, 8, 9, 10, 11, 12};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    printf("n,Solutions,Time(sec)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        double time_taken = get_time_taken(n);
        printf("%d,%d,%.6f\n", n, solution_count, time_taken);
    }
    return 0;}
```

Output:

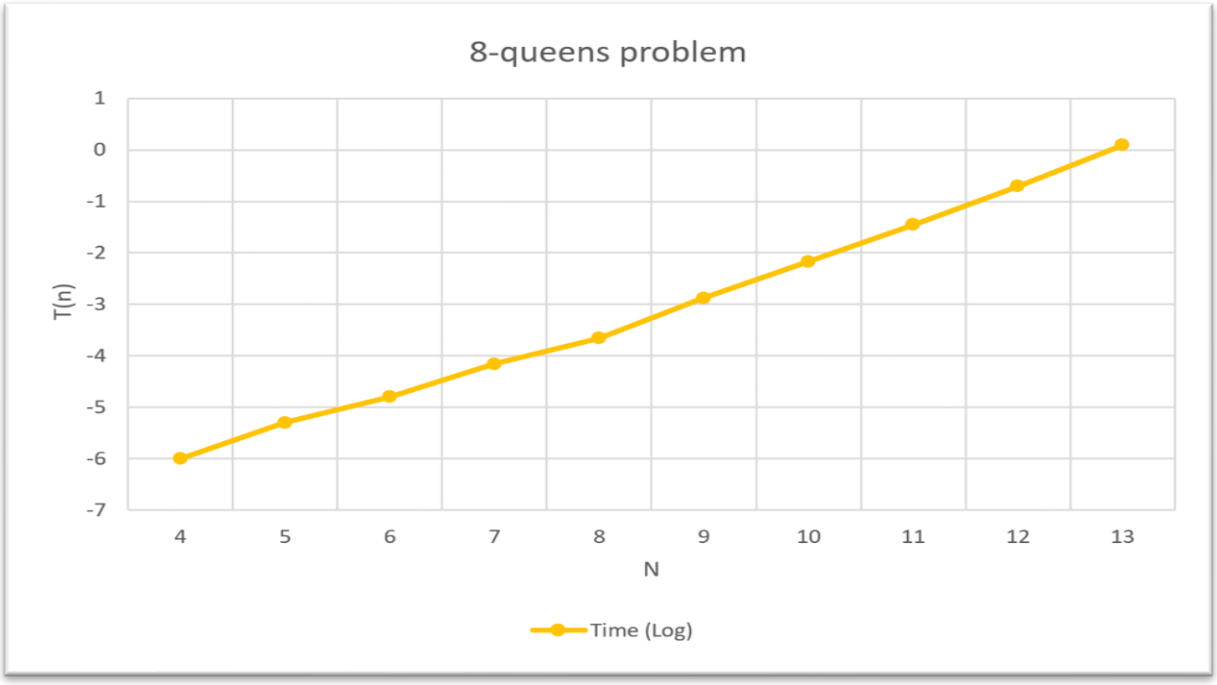
Enter number of elements: **10**

Time Complexity: 0.00681 seconds

Table:

n	Solutions	Time	Time (Log)
4	2	0.000001	-6
5	10	0.000005	-5.30103
6	4	0.000016	-4.79588
7	40	0.00007	-4.1549
8	92	0.00022	-3.65758
9	352	0.00133	-2.87615
10	724	0.00681	-2.16685
11	2680	0.03544	-1.45051
12	14200	0.19872	-0.70176
13	73712	1.25	0.09691

Graph:



Learning Outcome: