EXPERIMENT-1

AIM:

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

THEORY:

Insertion sort is a simple sorting technique based on the method by which people arrange a hand of playing cards. The algorithm builds the final sorted array one element at a time. It divides the list into two parts — a sorted part and an unsorted part. Initially, the first element is considered sorted, and the rest are unsorted. Each new element from the unsorted part is picked and inserted into its correct position in the sorted part by shifting larger elements one position to the right.

This process is repeated until all the elements are placed in their correct order, resulting in a completely sorted array.
Insertion sort works efficiently for small data sets and partially sorted arrays because the number of comparisons and shifts required is minimal in such cases. However, it becomes inefficient for large lists because every insertion may require multiple element shifts.
The time complexity of insertion sort depends on the initial order of the elements:

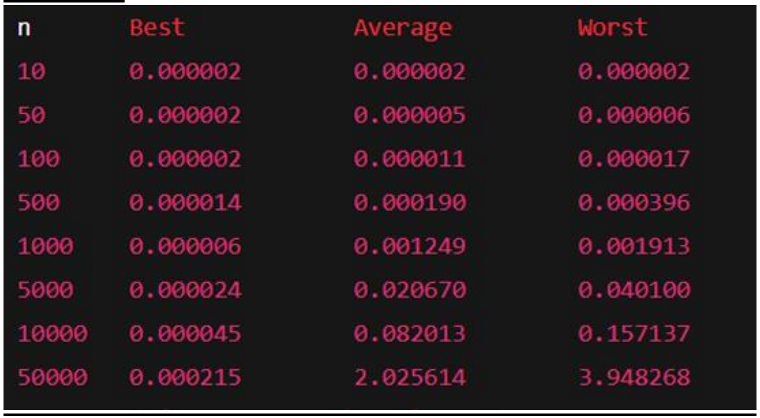
- Best Case (Already Sorted): O(n)
- Average Case: O(n^2)
- Worst Case (Reverse Order): O(n^2)

The space complexity is O(1) since it is an in-place sorting algorithm requiring no extra storage apart from the input array itself.

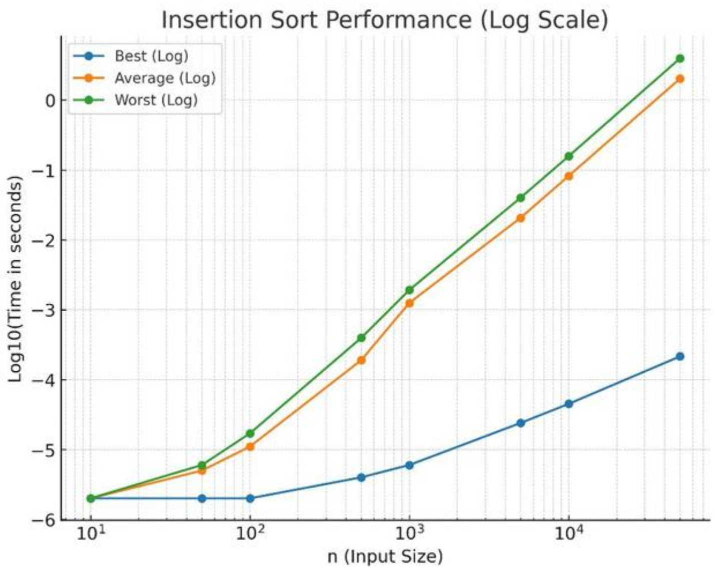
PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
void insertion_sort(int A[], int n) {
    int i, j, key;
    for (j = 1; j < n; j++) {
        key = A[j];
        i = j - 1;
        while (i >= 0 && A[i] > key) {
            A[i + 1] = A[i];
            i = i - 1;
        }
        A[i + 1] = key;
    }
}
void fill_random(int A[], int n, int lower, int upper) {
    for (int i = 0; i < n; i++)
        A[i] = (rand() % (upper - lower + 1)) + lower;
}
void fill_sorted(int A[], int n) {
    for (int i = 0; i < n; i++)
        A[i] = i;
}


void fill_reverse(int A[], int n) {
    for (int i = 0; i < n; i++)
        A[i] = n - i;
}

double get_time_taken(int A[], int n) {
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    insertion_sort(A, n);

    clock_gettime(CLOCK_MONOTONIC, &end);
    return (end.tv_sec - start.tv_sec) +
           (end.tv_nsec - start.tv_nsec) / 1e9;
}
double safe_log(double x) {
    return (x <= 0.0) ? 0.0 : log(x);
}

int main() {
    srand(time(NULL));
    int sizes[] = {10, 50, 100, 500, 1000, 5000, 10000, 50000,100000,500000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    FILE *fp = fopen("insertion_benchmark.csv", "w");

    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "n,Best,Average,Worst,Best (Log),Average (Log),Worst (Log)\n");

    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        int *A = (int *)malloc(n * sizeof(int));
        double best, avg, worst;

        // best case
        fill_sorted(A, n);
        best = get_time_taken(A, n);

        // average case
        fill_random(A, n, 1, 1000);
        avg = get_time_taken(A, n);

        // worst case
        fill_reverse(A, n);
        worst = get_time_taken(A, n);
        double log_best = safe_log(best);
        double log_avg = safe_log(avg);
        double log_worst = safe_log(worst);

        fprintf(fp, "%d,%f,%f,%f,%f,%f,%f\n", n, best, avg, worst, log_best, log_avg, log_worst);

        free(A);
        printf("n = %d done.\n", n);
    }
    fclose(fp);
    printf("Data saved to insertion_benchmark.csv\n");
    return 0;
}
```

OUTPUT:



```
n        Best        Average     Worst
10       0.000002    0.000002    0.000002
50       0.000002    0.000005    0.000006
100      0.000002    0.000011    0.000017
500      0.000014    0.000190    0.000396
1000     0.000006    0.001249    0.001913
5000     0.000024    0.020670    0.040100
10000    0.000045    0.082013    0.157137
50000    0.000215    2.025614    3.948268
```



Complexities

n	Best	Average	Worst	Best (Log)	Average (Log)	Worst (Log)
10	0.000002	0.000002	0.000002	-5.69897	-5.69897	-5.69897
50	0.000002	0.000005	0.000006	-5.69897	-5.30103	-5.22185
100	0.000002	0.000011	0.000017	-5.69897	-4.95861	-4.76955
500	0.000004	0.00019	0.000396	-5.39794	-3.72125	-3.40231
1000	0.000006	0.001249	0.001913	-5.22185	-2.90344	-2.71829
5000	0.000024	0.02067	0.0401	-4.61979	-1.68466	-1.39686
10000	0.000045	0.082013	0.157137	-4.34679	-1.08612	-0.80372
50000	0.000215	2.025614	3.948268	-3.66756	0.306557	0.596407

LEARNING OUTCOMES:

EXPERIMENT-2

AIM:

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

THEORY:

Merge Sort is a highly efficient, stable, comparison-based sorting algorithm that follows the Divide and Conquer approach. The main idea is to divide the given array into smaller subarrays until each subarray has only one element, and then repeatedly merge these subarrays to produce new sorted subarrays until there is only one sorted array remaining.

In the divide phase, the list is recursively split into two halves until single-element lists are obtained. In the conquer phase, these smaller lists are merged together in a sorted manner. The merging process involves comparing elements from each half and combining them into a new list in ascending order.

This algorithm ensures that the merging process is always linear relative to the size of the combined lists, while the number of levels of division is logarithmic. Hence, the efficiency of Merge Sort remains consistent across best, average, and worst cases.

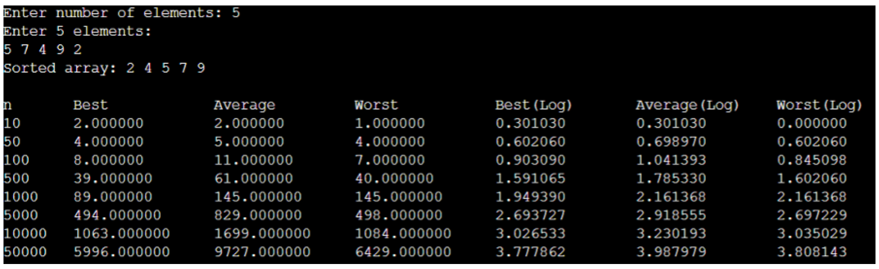
It performs well on large datasets and guarantees a time complexity of O(n log n) in all cases. However, Merge Sort requires additional memory space proportional to the size of the input array, making its space complexity O(n). Despite this, its predictable performance and stability make it a preferred choice for sorting large datasets and linked lists.

PROGRAM:

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

size_t space_used = 0;  // global counter for extra memory allocated

// Merge two halves
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    // Allocate temp arrays (this counts as extra space)
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));
    space_used += (n1 + n2) * sizeof(int);

    // Copy data
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
```

```
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

    free(L);
    free(R);
}

// Merge Sort recursive
void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

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

// Generate sorted array (Best Case)
void generate_sorted(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
}

// Generate reverse sorted array (Worst Case)
void generate_reverse(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = n - i;
    }
}

int main() {
    int sizes[] = {10, 50, 100, 500, 1000, 5000, 10000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    FILE *fp = fopen("merge_sort_times.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // CSV header
    fprintf(fp, "n,best_time(µs),avg_time(µs),worst_time(µs),best_log,avg_log,worst_log,space_bytes\n");

    srand(time(NULL));

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

        clock_t start, end;
        double best_time, avg_time, worst_time;

        // ---- BEST CASE ----
        generate_sorted(arr, n);
        space_used = 0;
        start = clock();
        merge_sort(arr, 0, n - 1);
        end = clock();
        best_time = ((double)(end - start)) / CLOCKS_PER_SEC * 1e6;  // µs
        size_t space_best = space_used;

        // ---- AVERAGE CASE ----
        generate_random(arr, n);
        space_used = 0;
        start = clock();
        merge_sort(arr, 0, n - 1);
        end = clock();
        avg_time = ((double)(end - start)) / CLOCKS_PER_SEC * 1e6;  // µs
        size_t space_avg = space_used;

        // ---- WORST CASE ----
        generate_reverse(arr, n);
        space_used = 0;
        start = clock();
        merge_sort(arr, 0, n - 1);
        end = clock();
        worst_time = ((double)(end - start)) / CLOCKS_PER_SEC * 1e6;  // µs
        size_t space_worst = space_used;
        // For space complexity, report maximum observed
        size_t space_final = (space_best > space_avg) ? space_best : space_avg;
        if (space_worst > space_final) space_final = space_worst;

        // Print to console
        printf("n=%d | Best=%f µs | Avg=%f µs | Worst=%f µs | Space=%zu bytes\n",
               n, best_time, avg_time, worst_time, space_final);

        // Write to CSV
        fprintf(fp, "%d,%f,%f,%f,%f,%f,%f,%zu\n",
                n,
                best_time,
                avg_time,
                worst_time,
                log(best_time + 1e-9),
                log(avg_time + 1e-9),
                log(worst_time + 1e-9),
                space_final);

        free(arr);
    }

    fclose(fp);
    printf("\nCSV file 'merge_sort_times.csv' has been saved.\n");

    return 0;
}
```

OUTPUT:

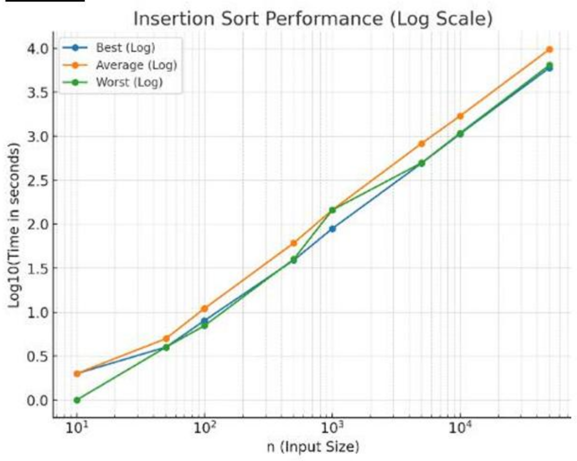


```
Enter number of elements: 5
Enter 5 elements:
5 7 4 9 2
Sorted array: 2 4 5 7 9

n       Best            Average         Worst           Best(Log)       Average(Log)    Worst(Log)
10      2.000000        2.000000        1.000000        0.301030        0.301030        0.000000
50      4.000000        5.000000        4.000000        0.602060        0.698970        0.602060
100     8.000000        11.000000       7.000000        0.903090        1.041393        0.845098
500     39.000000       61.000000       40.000000       1.591065        1.785330        1.602060
1000    89.000000       145.000000      145.000000      1.949390        2.161368        2.161368
5000    494.000000      829.000000      498.000000      2.693727        2.918555        2.697229
10000   1063.000000     1699.000000     1084.000000     3.026533        3.230193        3.035029
50000   5996.000000     9727.000000     6429.000000     3.777862        3.987979        3.808143
```

Complexities

n	Best	Average	Worst	Best(Log)	Average(Log)	Worst(Log
10	2.000000	2.000000	1.000000	0.301030	0.301030	0.000000
50	4.000000	5.000000	4.000000	0.602060	0.698970	0.602060
100	8.000000	11.000000	7.000000	0.903090	1.041393	0.845098
500	39.000000	61.000000	40.000000	1.591065	1.785330	1.602060
1000	89.000000	145.000000	145.000000	1.949390	2.161368	2.161368
5000	494.000000	829.000000	498.000000	2.693727	2.918555	2.697229
10000	1063.000000	1699.000000	1084.000000	3.026533	3.230193	3.035029
50000	5996.000000	9727.000000	6429.000000	3.777862	3.987979	3.808143

GRAPH:



LEARNING OUTCOMES:

EXPERIMENT-3

AIM:

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

THEORY:

Quick sort is a divide and conquer sorting algorithm that works by repeatedly partitioning the array around a special element called the pivot. The basic idea is to rearrange the elements of the array such that all elements less than the pivot come before it and all elements greater than the pivot come after it. After this partitioning step, the pivot is in its correct sorted position. The same process is then recursively applied to the left and right subarrays formed around the pivot.

The power of quick sort lies in this efficient partitioning. Instead of merging (as in merge sort), it reorders the elements in-place, which means it generally requires very little extra memory. The performance of quick sort depends on how well the pivot divides the array:

If the pivot splits the array into two nearly equal halves at each step, the height of the recursion tree is about $\log_2 n$, and each level processes n elements during partitioning. This gives a time complexity of O(n log n).

If the pivot is consistently chosen poorly (for example, always the smallest or largest element in an already sorted or reverse-sorted array), the array is divided into one subarray of size $n-1$ and one of size 1 at each step. This leads to a degenerate recursion tree of height n, resulting in $O(n^2)$ time.

Thus, the time complexity of quick sort for different cases is:

Best Case (balanced partitions): O(n log n)

Average Case: O(n log n)

Worst Case (highly unbalanced partitions): $O(n^2)$

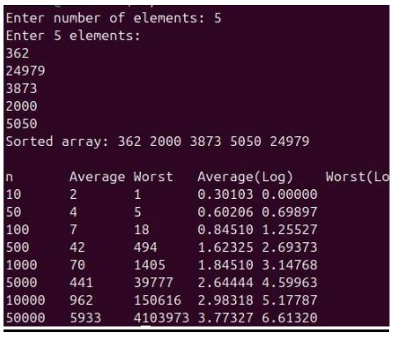
In its standard in-place implementation, quick sort requires only a small amount of extra space for the recursive calls, so its space complexity is typically O(log n) due to the recursion stack (for well-balanced partitions). Because of its good average performance and in-place behavior, quick sort is widely used in practice for sorting large datasets.

PROGRAM:

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

// Swap two elements
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
// Partition (general, pivot is at end)
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);
```

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

// Partition with random pivot
int partition_random(int arr[], int low, int high) {
    int random = low + rand() % (high - low + 1);
    swap(&arr[random], &arr[high]); // move random pivot to end
    return partition(arr, low, high);
}

// QuickSort with last element as pivot
void quickSort_last(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort_last(arr, low, pi - 1);
        quickSort_last(arr, pi + 1, high);
    }
}

// QuickSort with random pivot
void quickSort_random(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition_random(arr, low, high);
        quickSort_random(arr, low, pi - 1);
        quickSort_random(arr, pi + 1, high);
    }
}

// Generate array
void generateArray(int arr[], int n, int type) {
    if (type == 0) {
        // Sorted
        for (int i = 0; i < n; i++) arr[i] = i;
    } else if (type == 1) {
        // Random
        for (int i = 0; i < n; i++) arr[i] = rand() % n;
    } else {
        // Reverse sorted
        for (int i = 0; i < n; i++) arr[i] = n - i;
    }
}

int main() {
    FILE *fp;
    fp = fopen("quicksort_results.csv", "w");
    if (!fp) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "n,best,average,worst,best_log,average_log,worst_log\n");

    srand(time(NULL));
    int sizes[] = {100, 500, 1000, 5000, 10000, 20000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        int *arr = (int*)malloc(n * sizeof(int));
        clock_t start, end;
        double best, avg, worst;

        // Best case: sorted input + random pivot
        generateArray(arr, n, 0);
        start = clock();
        quickSort_random(arr, 0, n - 1);
        end = clock();
        best = ((double)(end - start) / CLOCKS_PER_SEC) * 1e6;

        // Average case: random input + random pivot
        generateArray(arr, n, 1);
        start = clock();
        quickSort_random(arr, 0, n - 1);
        end = clock();
        avg = ((double)(end - start) / CLOCKS_PER_SEC) * 1e6;

        // Worst case: sorted input + last pivot
        generateArray(arr, n, 0);
        start = clock();
        quickSort_last(arr, 0, n - 1);
        end = clock();
        worst = ((double)(end - start) / CLOCKS_PER_SEC) * 1e6;

        // Logs
        double best_log = log(best);
        double avg_log = log(avg);
        double worst_log = log(worst);

        // Write to CSV
        fprintf(fp, "%d,%.2f,%.2f,%.2f,%.5f,%.5f,%.5f\n",
                n, best, avg, worst, best_log, avg_log, worst_log);

        free(arr);
    }

    fclose(fp);
    printf("Results stored in quicksort_results.csv\n");
    return 0;
}
```

OUTPUT:

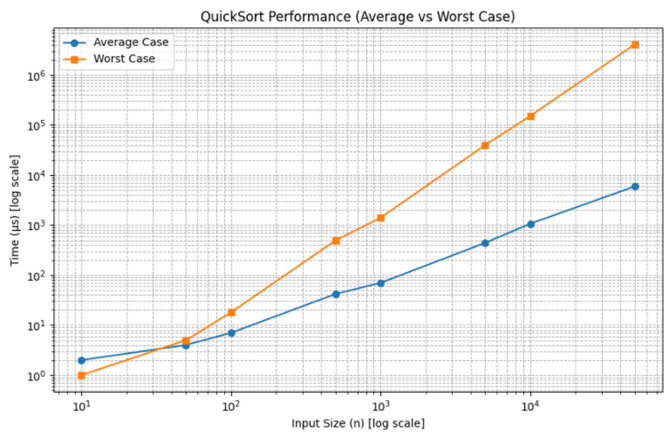


```
Enter number of elements: 5
Enter 5 elements:
362
24979
3873
2000
5050
Sorted array: 362 2000 3873 5050 24979

n       Average Worst   Average(Log)    Worst(Lo
10      2       1       0.30103 0.00000
50      4       5       0.60206 0.69897
100     7       18      0.84510 1.25527
500     42      494     1.62325 2.69373
1000    70      1405    1.84510 3.14768
5000    441     39777   2.64444 4.59963
10000   962     150616  2.98318 5.17787
50000   5933    4103973 3.77327 6.61320
```

Complexities

n	Average (µs)	Worst (µs)	Average (Log10)	Worst (Log10)
10	2	1	0.30103	0
50	4	5	0.60206	0.69897
100	7	18	0.8451	1.25527
500	42	494	1.62325	2.69373
1000	70	1405	1.8451	3.14613
5000	441	39777	2.64444	4.59987
10000	1066	152066	3.02716	5.18178

GRAPH:



LEARNING OUTCOMES:

EXPERIMENT-4

AIM:

Write a c program to implement knapsack problem using greedy method.

THEORY:

The Knapsack Problem is a classical optimization problem in which a set of items is given, each with a specific weight and value, and the objective is to determine the combination of items that maximizes the total value without exceeding a given weight capacity of the knapsack.

The Greedy Method provides an efficient approach to solving the Fractional Knapsack Problem, where items can be divided into smaller fractions. The idea is to select items based on the highest value-to-weight ratio (value/weight) first, ensuring the maximum value is achieved for the limited capacity.

In this method, items are sorted in descending order of their value-to-weight ratio. Starting from the highest ratio, items are added completely to the knapsack as long as there is remaining capacity. When the next item cannot be added in full due to limited capacity, a fraction of it is taken to fill the remaining space.

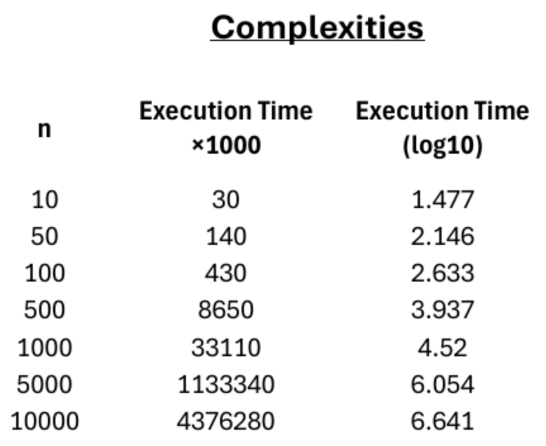
This greedy approach ensures that the solution obtained is optimal for the fractional version of the problem because at every step, the choice that appears to be the best (highest value per unit weight) leads to the most efficient use of the knapsack capacity.

The time complexity of the algorithm is O(n log n), mainly due to the sorting step of items based on their value-to-weight ratio. The space complexity is O(n), required to store the weights, values, and ratios of the items

PROGRAM:

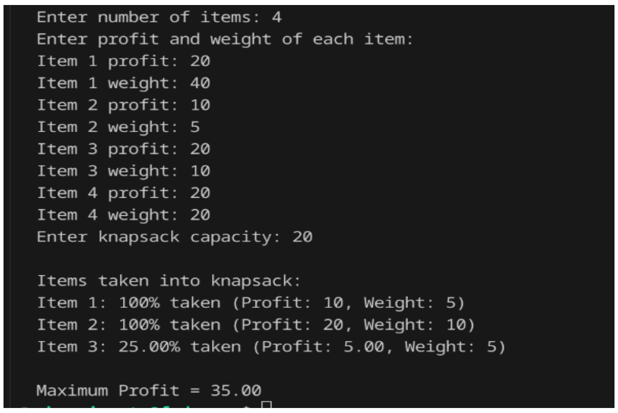
```
#include <stdio.h>
struct Item {
    int weight;
    int profit;
    float ratio;
};

void sortItems(struct Item items[], int n) {
    // Sort by ratio in descending order
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (items[i].ratio < items[j].ratio) {
                struct Item temp = items[i];
                items[i] = items[j];
                items[j] = temp;
            }
        }
    }
}

void knapsack(struct Item items[], int n, int capacity) {
    float totalProfit = 0.0;
    int remainingCapacity = capacity;

    printf("\nItems taken into knapsack:\n");
    for (int i = 0; i < n; i++) {
        if (items[i].weight <= remainingCapacity) {
            // take full item
            remainingCapacity -= items[i].weight;
            totalProfit += items[i].profit;
            printf("Item %d: 100%% taken (Profit: %d, Weight: %d)\n",
                   i + 1, items[i].profit, items[i].weight);
        } else {
            // take fractional part
            float fraction = (float) remainingCapacity / items[i].weight;
            totalProfit += items[i].profit * fraction;
            printf("Item %d: %.2f%% taken (Profit: %.2f, Weight: %d)\n",
                   i + 1, fraction * 100, items[i].profit * fraction, remainingCapacity);
            break; // knapsack is full
        }
    }

    printf("\nMaximum Profit = %.2f\n", totalProfit);
}

int main() {
    int n, capacity;

    printf("Enter number of items: ");
    scanf("%d", &n);

    struct Item items[n];

    printf("Enter profit and weight of each item:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d profit: ", i + 1);
        scanf("%d", &items[i].profit);
        printf("Item %d weight: ", i + 1);
        scanf("%d", &items[i].weight);
        items[i].ratio = (float)items[i].profit / items[i].weight;
    }

    printf("Enter knapsack capacity: ");
    scanf("%d", &capacity);
```

OUTCOME:



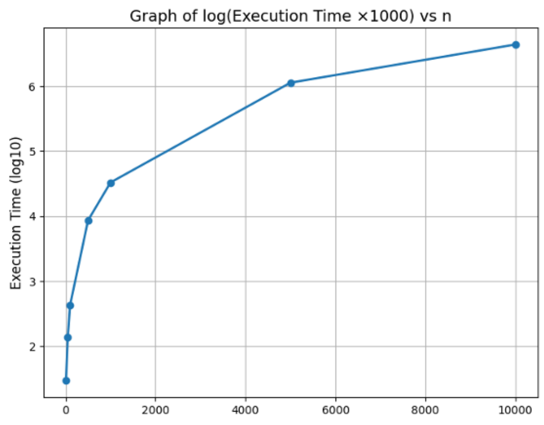
Complexities

n	Execution Time ×1000	Execution Time (log10)
10	30	1.477
50	140	2.146
100	430	2.633
500	8650	3.937
1000	33110	4.52
5000	1133340	6.054
10000	4376280	6.641



```
Enter number of items: 4
Enter profit and weight of each item:
Item 1 profit: 20
Item 1 weight: 40
Item 2 profit: 10
Item 2 weight: 5
Item 3 profit: 20
Item 3 weight: 10
Item 4 profit: 20
Item 4 weight: 20
Enter knapsack capacity: 20

Items taken into knapsack:
Item 1: 100% taken (Profit: 10, Weight: 5)
Item 2: 100% taken (Profit: 20, Weight: 10)
Item 3: 25.00% taken (Profit: 5.00, Weight: 5)

Maximum Profit = 35.00
```

GRAPH:



LEARNING OUTCOMES:

EXPERIMENT-5

AIM:

Program to implement job sequencing with deadlines using greedy method.

THEORY:

The Job Sequencing with Deadlines problem is a classic application of the greedy method in scheduling and optimization.
We are given a set of n jobs, where each job has:

- a deadline (the latest time slot by which it must be completed), and
- a profit (earned if the job is completed on or before its deadline).

Each job takes exactly one unit of time, and only one job can be processed at a time. The objective is to schedule the jobs in such a way that total profit is maximized, while ensuring that no job exceeds its deadline.
The greedy strategy used is:

1. Sort all jobs in decreasing order of profit.
   Jobs that give higher profit are given higher priority.
2. Maintain a sequence (array of time slots) representing the schedule.
   The size of this array is equal to the maximum deadline among all jobs.
3. For each job in the sorted list:
   - Try to place it in the latest free time slot on or before its deadline.
   - If such a slot is available, schedule the job there.
   - If no slot is free before its deadline, the job is skipped.

This approach ensures that:

- High profit jobs are considered first.
- Each job is scheduled as late as possible within its allowed time, leaving earlier slots free for other jobs.

Because we are always making the locally optimal choice (choosing the highest profit job and placing it in the best possible slot), this greedy algorithm results in an optimal solution for the job sequencing with deadlines problem.

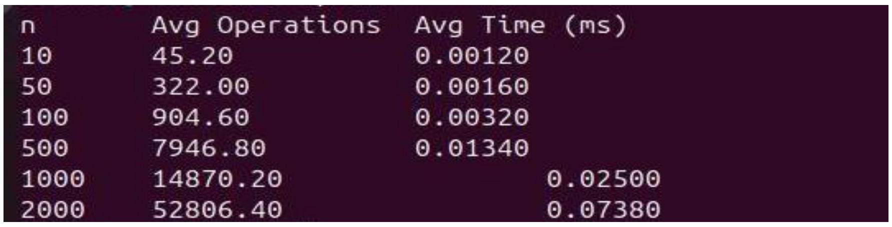
- The time complexity is generally O(n log n + n·d), where:
  - O(n log n) comes from sorting the jobs by profit.
  - O(n·d) comes from trying to place each job in a suitable slot (where d is the maximum deadline, and often d ≤ n, making it close to $O(n^2)$ in the basic implementation).
- The space complexity is O(d) for the time-slot array used to store the schedule.

PROGRAM:

```
#include <stdio.h> #include <stdlib.h> #include <time.h> struct Job {
int id;
int deadline; int profit;
};
int compare(const void *a, const void *b) { struct Job jobA = (struct Job)a;
struct Job jobB = (struct Job)b; return jobB->profit - jobA->profit;
}
int jobSequencing(struct Job jobs[], int n) { int operations = 0;
int maxDeadline = 0; for (int i = 0; i < n; i++) {
operations++;
if (jobs[i].deadline > maxDeadline) { maxDeadline = jobs[i].deadline;
}
}
int slot = (int)malloc((maxDeadline + 1) * sizeof(int)); for (int i = 0; i <= maxDeadline; i++) {
```

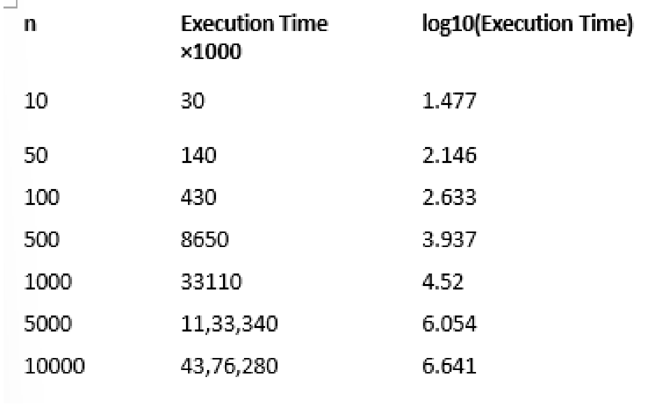
```
operations++; slot[i] = -1;
}
int totalProfit = 0;
for (int i = 0; i < n; i++) { operations++;
for (int j = jobs[i].deadline; j > 0; j--) { operations++;
if (slot[j] == -1) { slot[j] = i;
totalProfit += jobs[i].profit; break;
}
}
}
free(slot);
return operations;
}
int main() {

srand(time(0));
int sizes[] = {10, 50, 100, 500, 1000, 2000};
int numSizes = sizeof(sizes) / sizeof(sizes[0]); printf("n\tAvg Operations\tAvg Time (ms)\n"); for (int s = 0; s <
numSizes; s++) {
int n = sizes[s]; int repeats = 5;
long long totalOps = 0; double totalTime = 0.0;
for (int r = 0; r < repeats; r++) {
struct Job jobs = (struct Job)malloc(n * sizeof(struct Job)); for (int i = 0; i < n; i++) {
jobs[i].id = i + 1;
jobs[i].deadline = rand() % n + 1; jobs[i].profit = rand() % 1000 + 1;
}
qsort(jobs, n, sizeof(struct Job), compare); clock_t start = clock();
int ops = jobSequencing(jobs, n); clock_t end = clock();
totalOps += ops;
totalTime += ((double)(end - start) / CLOCKS_PER_SEC) * 1000; free(jobs);
}
printf("%d\t%.2f\t\t%.5f\n", n, (double)totalOps / repeats, totalTime / repeats);
}
return 0;
}
```

OUTCOMES:

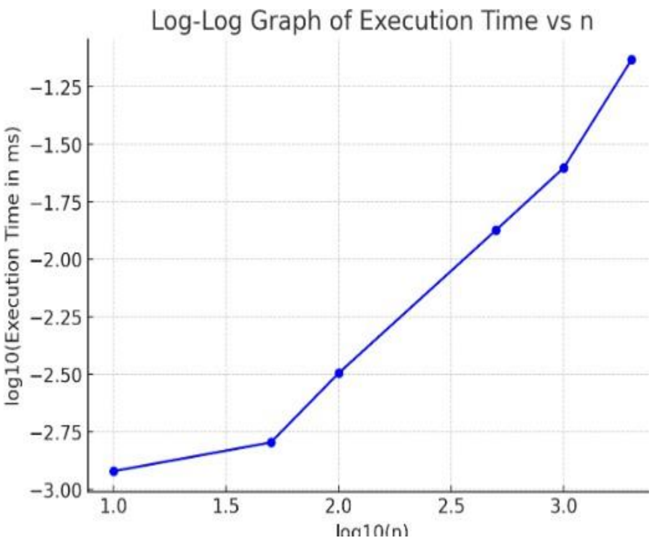


```
n       Avg Operations   Avg Time (ms)
10      45.20            0.00120
50      322.00           0.00160
100     904.60           0.00320
500     7946.80          0.01340
1000    14870.20                 0.02500
2000    52806.40                 0.07380
```

Table:



n	Execution Time ×1000	log10(Execution Time)
10	30	1.477
50	140	2.146
100	430	2.633
500	8650	3.937
1000	33110	4.52
5000	11,33,340	6.054
10000	43,76,280	6.641



LEARNING OUTCOME:

EXPERIMENT-6

AIM:

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

THEORY:

A Minimum Cost Spanning Tree (MST) is a subset of edges in a connected, weighted, undirected graph that connects all the vertices with the minimum possible total edge weight and without forming any cycles.

Prim's Algorithm is a greedy algorithm used to find the MST of a graph. It starts with a single vertex and repeatedly adds the smallest edge that connects a vertex in the tree to a vertex outside it. This process continues until all vertices are included in the tree.

The algorithm always makes the locally optimal choice of selecting the smallest weight edge available at each step, which results in a globally optimal spanning tree.

At the start, one vertex is chosen arbitrarily as part of the MST. Then, from all the edges that connect the tree to vertices not yet included, the edge with the minimum weight is selected. The vertex at the other end of this edge is added to the tree, and the process is repeated until all vertices are included.

Prim's Algorithm guarantees an MST because it always adds the smallest possible edge that connects a new vertex to the already formed tree, ensuring minimal total cost at every stage.

Time Complexity:

Using adjacency matrix: $O(V^2)$

Using adjacency list and priority queue: O(E log V)

Space Complexity:

O(V + E) for storing the graph and auxiliary data structures.

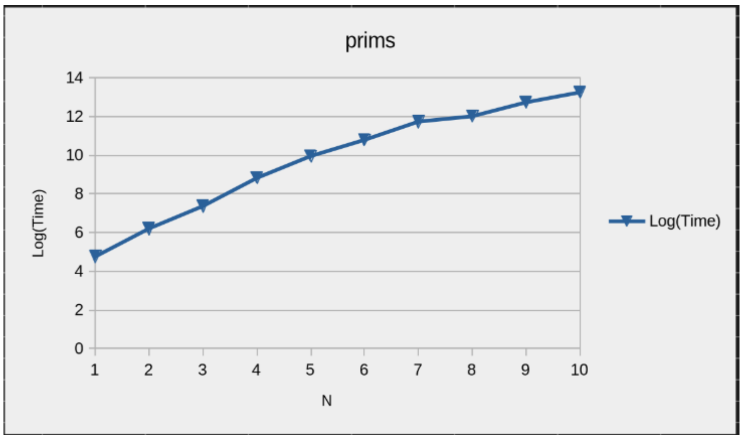
Prim's Algorithm is widely used in network design, such as laying cables, road maps, or electrical circuits, where the goal is to connect all points at the lowest possible cost.

PROGRAM:

```
#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;
  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;}}
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);
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    int cost = prims(graph, n);
    clock_gettime(CLOCK_MONOTONIC, &end);
    free_graph(graph, n);
    return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
}
double safe_log(double x) {
    return (x <= 0.0) ? 0.0 : log(x);
}
int main() {
    srand(time(NULL));
    int sizes[] = {10, 20, 30, 50, 75, 100, 150, 200, 250, 300};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    FILE *fp = fopen("prims_benchmark.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "n,Time(sec*1e7),Log(Time)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        double avg = get_time_taken(n) * 10000000;
        double log_avg = safe_log(avg);

        fprintf(fp, "%d,%f,%f\n", n, avg, log_avg);
        printf("n = %d done.\n", n);
    }
    fclose(fp);
    printf("Benchmark data saved to prims_benchmark.csv\n");
    return 0;
}
```

OUTPUT:

n	Time(sec*1e7)	Log(Time)
10	117.48	4.77
20	501.85	6.22
30	1598.09	7.38
50	6816.22	8.83
75	21128.09	9.96
100	48842.71	10.8
150	125764.26	11.74
200	165827.66	12.02
250	340568.88	12.74
300	570156.51	13.25



LEARNING OUTCOMES:

EXPERIMENT-7

AIM:

Write a program to find minimum cost spanning tree using Kruskal's Algorithm

THEORY:

The Minimum Cost Spanning Tree (MST) of a connected, weighted, undirected graph is a subset of its edges that connects all the vertices, contains no cycles, and has the minimum possible total edge weight.
Kruskal's Algorithm is a greedy algorithm used to find such an MST. Instead of growing a tree from a starting vertex (like Prim's), Kruskal's Algorithm works by selecting edges in order of increasing weight and adding them to the spanning tree as long as they do not form a cycle.
The process begins by sorting all edges of the graph in non-decreasing order of their weights. Starting from the smallest edge, each edge is considered one by one. For each edge, the algorithm checks whether the two vertices it connects belong to the same component (i.e., already connected through previously selected edges).

- If they are in different components, the edge is added to the MST, effectively merging the two components.
- If they are in the same component, adding that edge would create a cycle, so it is skipped.

To efficiently check whether two vertices belong to the same component, Kruskal's Algorithm commonly uses the Disjoint Set Union (DSU) or Union–Find data structure. This structure supports two main operations:

- Find: Determine which set (component) a particular vertex belongs to.
- Union: Merge two different sets into one when an edge connects them.

This process continues until exactly V − 1 edges have been added to the MST, where V is the number of vertices in the graph. The resulting set of edges forms a minimum cost spanning tree.
The time complexity of Kruskal's Algorithm is dominated by the sorting of edges and the Union–Find operations:

- Sorting E edges takes O(E log E) time.
- Union–Find operations over all edges take approximately O(E α(V)), where α(V) is the inverse Ackermann function, which grows very slowly and is treated as almost constant.

Thus, the overall time complexity is usually written as O(E log E) or O(E log V) (since E log E is roughly E log V in connected graphs). The space complexity is O(V + E) to store the graph, along with additional O(V) space for the Union–Find data structure.

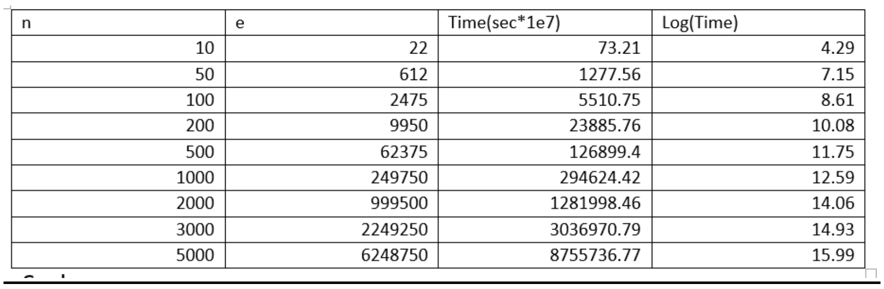
PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.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]);
```

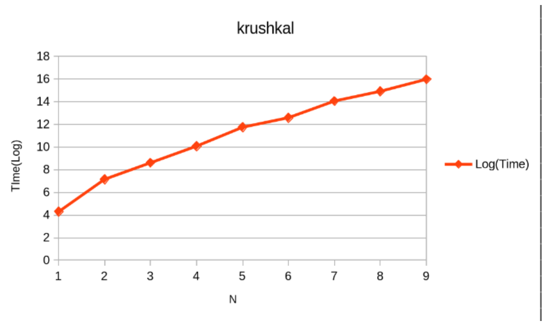
```
}
void union_sets(int a, int b) {
   a = find_set(a);
   b = find_set(b);
   if (a != b) {
      if (rank_arr[a] < rank_arr[b]) {
         int temp = a;
         a = b;
         b = temp;
      }
      parent[b] = a;
      if (rank_arr[a] == rank_arr[b])
         rank_arr[a]++;}}
int compare(const 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;
   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);
   struct timespec start, end;
   clock_gettime(CLOCK_MONOTONIC, &start);
```

```
    int cost = kruskals(edges, e, n);
    clock_gettime(CLOCK_MONOTONIC, &end);
    free(edges);
    return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
}
double safe_log(double x) {
    return (x <= 0.0) ? 0.0 : log(x);}
int main() {
    srand(time(NULL));
    int sizes[] = {10, 50, 100, 200, 500, 1000, 2000, 3000, 5000};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    FILE *fp = fopen("kruskals_benchmark.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "n,e,Time(sec*1e7),Log(Time)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        int e = n * (n - 1) / 4;
        double avg = get_time_taken(n) * 10000000;
        double log_avg = safe_log(avg);
        fprintf(fp, "%d,%d,%f,%f\n", n, e, avg, log_avg);
        printf("n = %d, e ~ %d done.\n", n, e);
    }
    fclose(fp);
    printf("Benchmark data saved to kruskals_benchmark.csv\n");
    return 0;
}
```

OUTCOME:



n	e	Time(sec*1e7)	Log(Time)
10	22	73.21	4.29
50	612	1277.56	7.15
100	2475	5510.75	8.61
200	9950	23885.76	10.08
500	62375	126899.4	11.75
1000	249750	294624.42	12.59
2000	999500	1281998.46	14.06
3000	2249250	3036970.79	14.93
5000	6248750	8755736.77	15.99



LEARNING OUTCOME:

EXPERIMENT-8

AIM:

Implement 0/1 Knapsack problem using dynamic programming.

THEORY:

The 0/1 Knapsack Problem is a classic optimization problem where we are given:

- a set of n items,
- each item i has a weight w_i and a value v_i,
- and a knapsack with maximum capacity W.

The goal is to maximize the total value of items placed in the knapsack without exceeding the capacity W. In the 0/1 version, each item can be either taken (1) or not taken (0) — no fractions are allowed.
Using dynamic programming, the problem is solved by building a table that stores the best possible value for smaller subproblems and then using those results to build the solution for the full problem.
We define a DP table K[i][w] as the maximum value that can be obtained using the first i items and a knapsack capacity w. The solution is built using the following recurrence:

- If the weight of the i-th item is greater than the current capacity w, we cannot include this item:
  K[i][w] = K[i−1][w]
- Otherwise, we choose the better of two options:
  1. Exclude the item: value = K[i−1][w]
  2. Include the item: value = v_i + K[i−1][w − w_i]

So the recurrence is:
K[i][w] = max(K[i−1][w], v_i + K[i−1][w − w_i])
The table is filled row by row (or item by item), starting from base cases where either i = 0 (no items) or w = 0 (zero capacity), for which the maximum value is 0.
Once the table is completely filled, the entry K[n][W] gives the maximum value achievable for n items and capacity W. Optionally, by tracing back through the table, we can also determine which items are included in the optimal solution.
The time complexity of this dynamic programming approach is O(nW), since we compute values for all n items and all capacities from 0 to W. The space complexity is also O(nW) for the full table, which can be optimized to O(W) using a 1D array if only the maximum value (and not the exact selected items) is required.

PROGRAM:

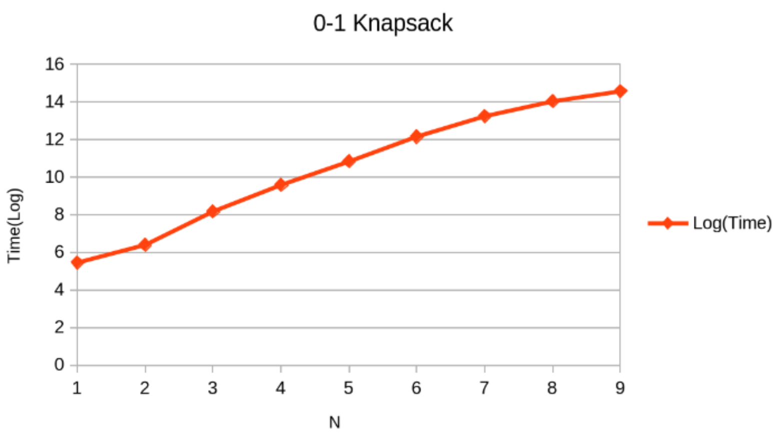
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
int max(int a, int b) {
  return (a > b) ? a : b;
}
int knapsack(int *val, int *wt, int n, int capacity) {
  int **dp = (int **)malloc((n + 1) * sizeof(int *));
  for (int i = 0; i <= n; i++) {
    dp[i] = (int *)malloc((capacity + 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;
        wt[i] = (rand() % 50) + 1;
    }
}
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);
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    int max_val = knapsack(val, wt, n, capacity);
    clock_gettime(CLOCK_MONOTONIC, &end);
    free(val);
    free(wt);
    return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
}
double safe_log(double x) {
    return (x <= 0.0) ? 0.0 : log(x);
}

int main() {
    srand(time(NULL));
    int sizes[] = {10, 20, 50, 100, 200, 500, 1000, 1500, 2000};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    FILE *fp = fopen("knapsack_benchmark.csv", "w");

    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "n,capacity,Time(sec*1e7),Log(Time)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        int capacity = n * 10;
        double avg = get_time_taken(n, capacity) * 10000000;
        double log_avg = safe_log(avg);
        fprintf(fp, "%d,%d,%f,%f\n", n, capacity, avg, log_avg);
        printf("n = %d, capacity = %d done.\n", n, capacity);
```

```
}
fclose(fp);
printf("Benchmark data saved to knapsack_benchmark.csv\n");
return 0;
```

OUTPUT:

n	capacity	Time(sec*1e7)	Log(Time)
10	100	231.08	5.44
20	200	595.87	6.39
50	500	3541.25	8.17
100	1000	14511.2	9.58
200	2000	50568.71	10.83
500	5000	190863.66	12.16
1000	10000	564232.58	13.24
1500	15000	1245050.92	14.03
2000	20000	2127479.1	14.57



LEARNING OUTCOME:

EXPERIMENT-9

AIM:

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

THEORY:

The All-Pairs Shortest Path (APSP) problem aims to find the shortest path distances between every pair of vertices in a weighted graph. The graph may be directed or undirected, and edge weights can be positive, zero, or negative, but there must be no negative weight cycle reachable in the graph.
The Floyd–Warshall Algorithm is a dynamic programming method to solve the APSP problem. Instead of computing shortest paths from each vertex separately, it systematically updates a distance matrix by considering all vertices as possible intermediate nodes in paths.
We start with a matrix D, where:

- D[i][j] = weight of the edge from i to j if it exists,
- D[i][j] = 0 if i = j,
- D[i][j] = ∞ (a very large value) if there is no direct edge from i to j.

The key idea is to check, for every pair of vertices (i, j), whether going through an intermediate vertex k gives a shorter path than the current known distance. This means:

- D_k(i, j) is the shortest distance from i to j using only vertices {1…k} as intermediates.
- We either keep the old distance D_{k-1}(i, j) (not using k),
  or improve it by going via k using D_{k-1}(i, k) + D_{k-1}(k, j).

By iterating k from 1 to n, the algorithm gradually allows more intermediate vertices in paths. After all iterations, D(i, j) contains the length of the shortest path from vertex i to vertex j.
The algorithm can also be extended to detect negative weight cycles:

- If after completion, any D[i][i] < 0, it indicates a negative cycle reachable from vertex i.

Time Complexity:

- The algorithm uses three nested loops over all vertices, so its time complexity is $O(n^3)$, where n is the number of vertices.

Space Complexity:

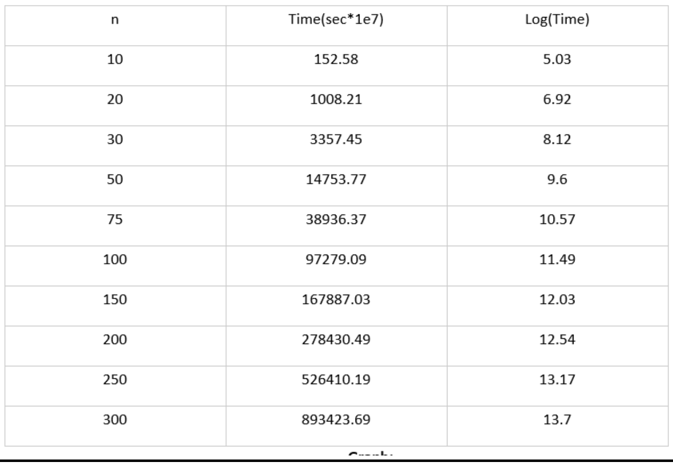
- It requires a distance matrix of size n × n, so the space complexity is $O(n^2)$.

Because it is simple and works well for dense graphs and small to medium-sized graphs, and can handle negative edge weights, Floyd–Warshall is widely used in routing, graph analysis, and transitive closure computations.

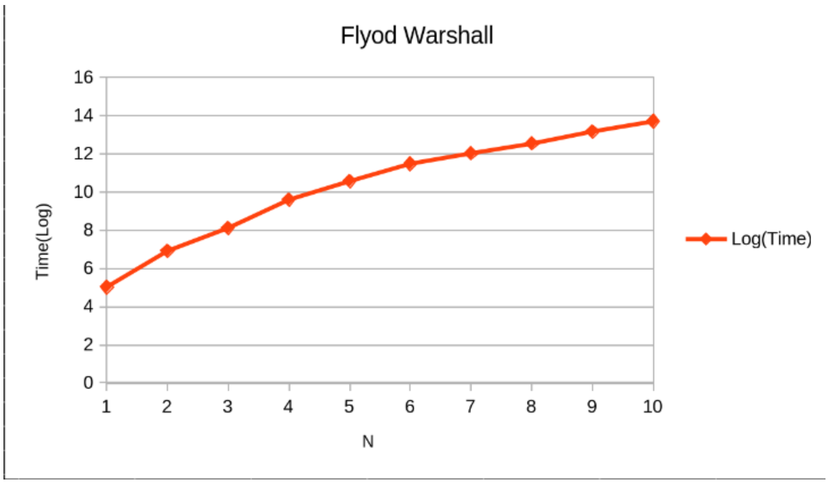
PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <limits.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);
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    floydWarshall(graph, n);
    clock_gettime(CLOCK_MONOTONIC, &end);
    free_graph(graph, n);
    return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
}
double safe_log(double x) {
    return (x <= 0.0) ? 0.0 : log(x);
}
int main() {
    srand(time(NULL));
    int sizes[] = {10, 20, 30, 50, 75, 100, 150, 200, 250, 300};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    FILE *fp = fopen("floydwarshall_benchmark.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "n,Time(sec*1e7),Log(Time)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        double avg = get_time_taken(n) * 10000000;
        double log_avg = safe_log(avg);
        fprintf(fp, "%d,%f,%f\n", n, avg, log_avg);
        printf("n = %d done.\n", n);
    }
    fclose(fp);
    printf("Benchmark data saved to floydwarshall_benchmark.csv\n");
    return 0;
}
```

OUTPUT:



n	Time(sec*1e7)	Log(Time)
10	152.58	5.03
20	1008.21	6.92
30	3357.45	8.12
50	14753.77	9.6
75	38936.37	10.57
100	97279.09	11.49
150	167887.03	12.03
200	278430.49	12.54
250	526410.19	13.17
300	893423.69	13.7



LEARNING OUTCOMES:

EXPERIMENT-10

AIM:

Program to implement 8-queens problem using backtrack method.

THEORY:

The 8-Queens Problem is a classic example of using the backtracking method in algorithm design. The problem is to place eight queens on a standard 8×8 chessboard so that no two queens threaten each other. This means that no two queens can be placed in the same row, column, or diagonal.

The solution is obtained by placing one queen in each row and checking whether it is safe to place the next queen in a given column. A position is considered safe if no other queen already placed can attack it. If a safe position is found, the algorithm proceeds to place the next queen in the next row. If no safe position exists in a row, the algorithm backtracks — it removes the previously placed queen and tries the next possible position in that earlier row.

This trial-and-error process continues until either all eight queens are placed successfully (indicating a valid solution) or all possibilities are exhausted. The backtracking method ensures that the algorithm explores all potential configurations efficiently by eliminating invalid arrangements early.

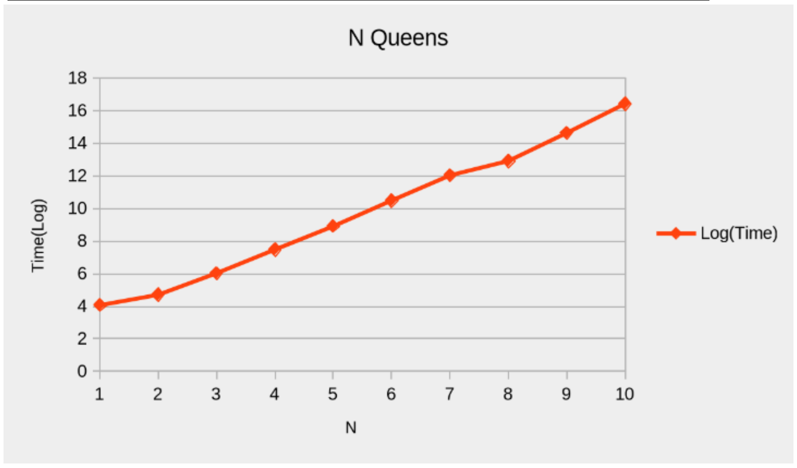
The 8-Queens Problem demonstrates the power of backtracking in solving combinatorial problems, where decisions are made sequentially and wrong decisions are undone to explore alternative paths. The time complexity in the worst case is approximately O(N!), where N is the number of queens, as each queen can be placed in any column of a row before pruning invalid choices.

PROGRAM:

```c
#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) {
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    int solutions = n_queens(n);
   clock_gettime(CLOCK_MONOTONIC, &end);
   return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;}
double safe_log(double x) {
    return (x <= 0.0) ? 0.0 : log(x);}
int main() {
    int sizes[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    FILE *fp = fopen("nqueens_benchmark.csv", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "n,solutions,Time(sec*1e7),Log(Time)\n");
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        double avg = get_time_taken(n) * 10000000;
        double log_avg = safe_log(avg);
        fprintf(fp, "%d,%d,%f,%f\n", n, solution_count, avg, log_avg);
        printf("n = %d, solutions = %d done.\n", n, solution_count);    }
    fclose(fp);
    printf("Benchmark data saved to nqueens_benchmark.csv\n");
    return 0;
}
```

OUTPUT:

n	solutions	Time(sec*1e7)
4	2	58.51
5	10	110.86
6	4	416.44
7	40	1781.6
8	92	7450.26
9	352	36147.46
10	724	169260.16
11	2680	410038.67
12	14200	2293010.9
13	73712	13749789.54



LEARNING OUTCOMES: