EXPERIMENT NO.3

AIM: Write a program to implement fractional knapsack using Greedy Approach.

THEORY: The Fractional Knapsack Problem is an optimization problem where you have a set of items, each with a weight and a profit (or value), and a knapsack with a limited capacity. The objective is to maximize the total value that fits into the knapsack. In contrast to the 0/1 Knapsack Problem, you can take fractions of an item rather than taking an entire item or leaving it behind.

Key Points:

Greedy Strategy: In the greedy approach to solve the fractional knapsack problem, you prioritize items based on their profit-to-weight ratio (value/weight). The goal is to maximize the value gained with the least weight.

Optimality: The greedy algorithm provides an optimal solution for the fractional knapsack problem, unlike the 0/1 knapsack problem which requires dynamic programming.

Process:

1. Calculate the profit-to-weight ratio for each item.
2. Sort the items by this ratio in descending order.
3. Start by picking the item with the highest ratio, and if possible, take the whole item; otherwise, take the fraction that fits in the knapsack.
4. Continue this process until the knapsack is full.

COMPLEXITY ANALYSIS:

Time Complexity:

The time complexity of the fractional knapsack using a greedy approach is O(n log n) due to the sorting step, where n is the number of items.

CODE:

```
#include <stdio.h>

// Ansh Malik 00914811622

 struct Item { float weight;
   float profit;
};

float fractionalKnapsack(struct Item items[], int n, float
   capacity) { float currentWeight = 0.0, totalValue = 0.0;
   for (int i = 0; i < n - 1; i++) { for (int j = i + 1; j < n;
   j++) {

        float ratio1 = items[i].profit / items[i].weight; float ratio2
        = items[j].profit / items[j].weight; if (ratio1 < ratio2) { struct
           Item temp = items[i]; items[i] = items[j]; items[j] = temp;

        }
```

```
    } }
  for (int i = 0; i < n; i++) {
    if (currentWeight + items[i].weight <= capacity) {
      currentWeight += items[i].weight; totalValue
      += items[i].profit;
    } else { float remainingCapacity = capacity -
      currentWeight;
      totalValue += (items[i].profit / items[i].weight) *
      remainingCapacity; break;
    }
  }
  return totalValue;
}
int main() { int n;
  float capacity;
  printf("Enter the number of items: "); scanf("%d", &n); printf("Enter
  the capacity of the knapsack: "); scanf("%f",
  &capacity); struct Item
  items[n]; for (int i = 0; i
  < n; i++) {
    printf("Enter weight and profit of item %d: ", i + 1); scanf("%f
    %f", &items[i].weight, &items[i].profit);
  }
  float maxProfit = fractionalKnapsack(items, n, capacity); printf("The
  maximum profit is: %.2f\n", maxProfit); return 0;
}
```

OUTPUT:

```
Enter the number of items: 4
Enter the capacity of the knapsack: 4
Enter weight and profit of item 1: 1 2
Enter weight and profit of item 2: 3 4
Enter weight and profit of item 3: 5 6
Enter weight and profit of item 4: 7 8
The maximum profit is: 6.00
```

VIVA QUESTIONS:

Q1. What is the difference between the 0/1 knapsack and fractional knapsack problems?

ANS. In the 0/1 knapsack problem, you can either take the entire item or leave it, but in the fractional knapsack problem, you can take fractions of items. The greedy approach works optimally only for the fractional knapsack problem.

Q2. How does the greedy approach work for the fractional knapsack problem?

Answer: The greedy approach selects items based on their profit-to-weight ratio, prioritizing items with the highest ratio to maximize profit.

Q3. Why do we sort items based on the profit-to-weight ratio in the greedy algorithm?

ANS. Sorting by the profit-to-weight ratio ensures that we take items that provide the maximum value for the least amount of weight, leading to the optimal solution.

Q4. What is the time complexity of this algorithm?

ANS. The time complexity of the fractional knapsack problem using the greedy approach is O(n log n) due to the sorting step, where n is the number of items.

Q5. Can the greedy approach give an optimal solution for the 0/1 knapsack problem?

ANS. No, the greedy approach does not guarantee an optimal solution for the 0/1 knapsack problem. For 0/1 knapsack, dynamic programming is typically used to find the optimal solution.

EXPERIMENT NO.4

AIM: Implement Strassen's algorithm using Divide and Conquer Approach.

THEORY: Strassen's Algorithm is an efficient algorithm for matrix multiplication that uses the divide-and-conquer technique to achieve better time complexity than standard matrix multiplication.

Divide-and-Conquer Approach:

In traditional matrix multiplication, multiplying two matrices of size requires scalar multiplications.

Strassen's algorithm reduces the number of scalar multiplications to approximately , making it faster than the traditional method for large matrices.

The algorithm works by recursively dividing the matrices into smaller submatrices and multiplying them in a specific way that reduces the total number of multiplications required.

Strassen's Algorithm for Matrix Multiplication

Given two matrices and of size , the standard matrix multiplication needs 8 multiplications. Strassen's algorithm reduces this to 7 by using the following intermediary products:

1. $M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
2. $M_2 = (A_{21} + A_{22}) \times B_{11}$
3. $M_3 = A_{11} \times (B_{12} - B_{22})$
4. $M_4 = A_{22} \times (B_{21} - B_{11})$
5. $M_5 = (A_{11} + A_{12}) \times B_{22}$
6. $M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
7. $M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

The result matrix C can then be constructed from these intermediaries:

- $C_{11} = M_1 + M_4 - M_5 + M_7$
- $C_{12} = M_3 + M_5$
- $C_{21} = M_2 + M_4$
- $C_{22} = M_1 - M_2 + M_3 + M_6$

This allows for matrix multiplication with fewer scalar multiplications.

COMPLEXITY ANALYSIS:

$$T(n) = 7T(n/2) + O(n$$

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

CODE:

```
#include <stdio.h>

#include <stdlib.h>

//Ansh Malik 00914811622

#define MAX 2
void add(int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX], int
   size) { for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++) C[i][j] = A[i][j] + B[i][j];
}

void subtract(int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX], int
   size) { for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++) C[i][j] = A[i][j] - B[i][j];
}

void strassen(int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX], int
   size) { if (size == 1) {
      C[0][0] = A[0][0] * B[0][0]; return;

   }

   int new_size = size / 2;

   int      A11[MAX][MAX],      A12[MAX][MAX],      A21[MAX][MAX],
   A22[MAX][MAX];

   int      B11[MAX][MAX],      B12[MAX][MAX],      B21[MAX][MAX],
   B22[MAX][MAX];

   int      C11[MAX][MAX],      C12[MAX][MAX],      C21[MAX][MAX],
   C22[MAX][MAX];

   int       P1[MAX][MAX],       P2[MAX][MAX],       P3[MAX][MAX],
   P4[MAX][MAX],
P5[MAX][MAX], P6[MAX][MAX], P7[MAX][MAX];

   int temp1[MAX][MAX], temp2[MAX][MAX]; for (int i = 0; i < new_size;
   i++) {
      for (int j = 0; j < new_size; j++) { A11[i][j] = A[i][j];
         A12[i][j] = A[i][j + new_size];

         A21[i][j] = A[i + new_size][j];
         A22[i][j] = A[i + new_size][j + new_size]; B11[i][j] = B[i][j];
         B12[i][j] = B[i][j + new_size];

         B21[i][j] = B[i + new_size][j];

         B22[i][j] = B[i + new_size][j + new_size];

      }

   }
```

```
    add(A11, A22, temp1, new_size); add(B11, B22, temp2, new_size);
    strassen(temp1, temp2, P1, new_size); add(A21, A22, temp1, new_size);
    strassen(temp1, B11, P2, new_size); subtract(B12, B22, temp2,
    new_size); strassen(A11, temp2, P3, new_size); subtract(B21, B11,
    temp2, new_size); strassen(A22, temp2, P4, new_size); add(A11, A12,
    temp1, new_size); strassen(temp1, B22, P5, new_size); subtract(A21,
    A11, temp1, new_size); add(B11, B12, temp2, new_size);
    strassen(temp1, temp2, P6, new_size); subtract(A12, A22, temp1,
    new_size); add(B21, B22, temp2, new_size); strassen(temp1, temp2, P7,
    new_size); add(P1, P4, temp1, new_size); subtract(temp1, P5, temp2,
    new_size); add(temp2, P7, C11, new_size); add(P3, P5, C12, new_size);
    add(P2, P4, C21, new_size); add(P1, P3, temp1, new_size);
    subtract(temp1, P2, temp2, new_size); add(temp2, P6, C22, new_size);
    for (int i = 0; i < new_size; i++) { for (int j = 0; j < new_size; j++) {
            C[i][j] = C11[i][j];

            C[i][j + new_size] = C12[i][j];

            C[i + new_size][j] = C21[i][j];

            C[i + new_size][j + new_size] = C22[i][j]; } } }

void printMatrix(int matrix[MAX][MAX], int size) { for (int i = 0; i < size;
    i++) {
        for (int j = 0; j < size; j++) printf("%d ", matrix[i][j]); printf("\n");

    }

} int main()

{

    int A[MAX][MAX] = {{1, 2}, {3, 4}};

    int B[MAX][MAX] = {{5, 6}, {7, 8}}; int C[MAX][MAX]; strassen(A,
    B, C, MAX); printf("Matrix A:\n"); printMatrix(A, MAX);
    printf("\nMatrix B:\n"); printMatrix(B, MAX);
    printf("\nResultant Matrix C (A*B) using Strassen's Algorithm:\n");
    printMatrix(C, MAX); return 0;

}
```

OUTPUT:

```
Matrix A:
1 2
3 4

Matrix B:
5 6
7 8

Resultant Matrix C (A*B) using Strassen's Algorithm:
19 22
43 50
```

VIVA QUESTIONS:

Q1. What is Strassen's Algorithm?

ANS. Strassen's Algorithm is an efficient algorithm for matrix multiplication that reduces the time complexity from to approximately using divide and conquer strategy.

Q2. What is the key difference between traditional matrix multiplication and Strassen's algorithm?

ANS. The traditional algorithm involves 8 multiplications and 4 additions for 2x2 matrices, whereas Strassen's algorithm reduces it to 7 multiplications and 10 additions/subtractions.

Q3. How does divide and conquer work in Strassen's Algorithm?

ANS. The matrices are recursively divided into sub-matrices, and then Strassen's seven multiplication formulas are applied on those smaller matrices.

Q4. What are the time complexities of Strassen's algorithm and the traditional algorithm?

ANS. Strassen's algorithm has a time complexity of , while the traditional algorithm has .

Q5. Why is Strassen's Algorithm not always used in practice despite its lower time complexity?

ANS. Strassen's algorithm involves many additions and subtractions, which increase overhead and reduce practical performance for small matrices or when hardware optimizations like cache locality favor traditional algorithms.

EXPERIMENT NO.5

AIM: Implement Prim's algorithm in C programming language.

THEORY: Prim's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, weighted, and undirected graph. The MST of a graph is a subgraph that connects all the vertices together without any cycles and with the minimum possible total edge weight.

Working of Prim's Algorithm:

1. Initialization: Start with a node and add it to the MST.
2. Edge Selection: At each step, select the minimum weight edge that connects a vertex in the MST to a vertex not in the MST.
3. Repeat: Add this edge and the connected vertex to the MST. Continue until all vertices are included in the MST.
4. Termination: The algorithm stops when all vertices are included in the MST.

COMPLEXITY ANALYSIS:

Time Complexity:

The time complexity of Prim's Algorithm depends on the data structures used. If we use an adjacency matrix and a simple linear search for selecting the minimum edge, the complexity is $O(V^2)$, where V is the number of vertices. Using a priority queue (min-heap), the complexity can be reduced to O(E log V), where E is the number of edges.

CODE:

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

// Ansh Malik 00914811622

#define V 5
int minKey(int key[], int mstSet[]) { int min
   = INT_MAX, min_index;


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

      if (mstSet[v] == 0 && key[v] <
         min) min = key[v], min_index =
         v;


   return min_index;

}

void printMST(int parent[], int
   graph[V][V]) { printf("Edge
   \tWeight\n"); for (int i = 1; i
   < V; i++)
```

```
        printf("%d - %d \t%d \n", parent[i],
        i, graph[i][parent[i]]);
}
void primMST(int graph[V][V]) { int
    parent[V]; int key[V]; int
    mstSet[V]; for (int i = 0; i
    < V; i++)
        key[i] = INT_MAX, mstSet[i] = 0;
    key[0] = 0; parent[0]
    = -1;
    for (int count = 0; count < V - 1;
        count++) { int u = minKey(key,
        mstSet); mstSet[u] = 1; for
        (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v]
               == 0 && graph[u][v] <
               key[v]) parent[v] = u, key[v]
               = graph[u][v]; } }
    printMST(parent, graph);
} int main()
{
        int graph[V][V] = {
            {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}}; primMST(graph);
    return 0; }
```

OUTPUT:

```
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

VIVA QUESTIONS:

Q1. What is the primary objective of Prim's Algorithm?

ANS. The primary objective of Prim's Algorithm is to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph, ensuring the total weight of the tree is minimized.

Q2. How does Prim's Algorithm differ from Kruskal's Algorithm?

ANS. Prim's Algorithm starts from a specific vertex and grows the MST by adding the least weight edge connected to the already selected vertices. Kruskal's Algorithm, on the other hand, works by sorting all edges and adding them one by one while avoiding cycles, regardless of vertex connections.

Q3. What data structures are commonly used in Prim's Algorithm for efficiency?

ANS.A priority queue (min-heap) is often used to efficiently select the edge with the minimum weight. Additionally, arrays are used to keep track of the parent of each vertex and the minimum key values.

Q4. What is the time complexity of Prim's Algorithm using an adjacency matrix?

ANS. The time complexity of Prim's Algorithm using an adjacency matrix and a linear search for the minimum weight edge is $O(V^2)$, where V is the number of vertices.

Q5. Can Prim's Algorithm handle disconnected graphs?

ANS. No, Prim's Algorithm can only work on connected graphs. If the graph is disconnected, it cannot find a Minimum Spanning Tree, as there will be vertices that cannot be reached from others.

EXPERIMENT NO.6

AIM: Implement Kruskal's Algorithm in C programming language.

THEORY: Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) for a connected, weighted, and undirected graph. A spanning tree is a subgraph of the original graph that includes all the vertices with the minimum number of edges, and a Minimum Spanning Tree is a spanning tree with the least possible total edge weight.

The algorithm works as follows:

1. Sort all edges in non-decreasing order based on their weight.

2. Pick the smallest edge and check if including this edge forms a cycle in the MST.
If the edge forms a cycle, discard it.
If the edge doesn't form a cycle, include it in the MST.

3. Repeat the above step until the MST includes all vertices (i.e., edges for vertices).

To detect cycles, Kruskal's algorithm typically uses a disjoint-set (union-find) data structure. This allows us to efficiently determine whether two vertices belong to the same subset, meaning they are part of the same tree.

COMPLEXITY ANALYSIS:

$$O(E \log E + E\alpha(V)) = O(E \log E)$$

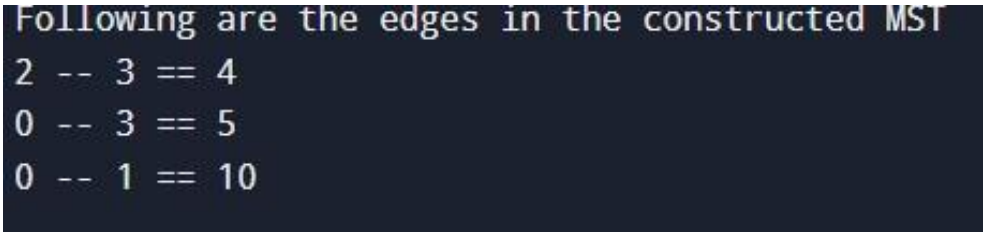
CODE:

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

// Ansh Malik 00914811622



struct Edge { int src,
   dest, weight;

};

struct Graph { int V, E; struct
   Edge* edge;

};

struct Subset { int parent; int
   rank;

};
```

```
struct Graph* createGraph(int V, int E) { struct Graph* graph = (struct Graph*)
    malloc(sizeof(struct Graph)); graph->V = V; graph->E = E;

    graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge)); return graph;
}

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

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

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

        subsets[yroot].parent = xroot; subsets[xroot].rank++;
    }

}

int compareEdges(const void* a, const void* b) { struct Edge* a1 = (struct Edge*) a; struct
    Edge* b1 = (struct Edge*) b;return a1->weight > b1->weight;

}

void KruskalMST(struct Graph* graph) { int V = graph-
    >V; struct Edge result[V]; int e = 0; int i = 0;

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compareEdges); struct Subset*
    subsets = (struct Subset*) malloc(V * sizeof(struct Subset)); for (int v = 0; v < V; ++v) {
    subsets[v].parent = v; subsets[v].rank = 0;
    }

    while (e < V - 1 && i < graph->E) {

        struct Edge next_edge = graph->edge[i++]; int x = find(subsets,
        next_edge.src); int y = find(subsets, next_edge.dest); if (x != y) { result[e++] =
        next_edge; Union(subsets, x, y);
        }

    }

    printf("Following are the edges in the constructed MST\n"); for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
    free(subsets);
}

int main() { int V = 4; int E = 5; struct Graph* graph = createGraph(V,
    E); graph->edge[0].src = 0; graph->edge[0].dest = 1; graph-
    >edge[0].weight = 10; graph->edge[1].src = 0;
```

```
    graph->edge[1].dest = 2; graph-
    >edge[1].weight = 6; graph->edge[2].src =
    0; graph->edge[2].dest = 3; graph-
    >edge[2].weight = 5; graph->edge[3].src =
    1; graph->edge[3].dest = 3; graph-
    >edge[3].weight = 15; graph->edge[4].src =
    2; graph->edge[4].dest = 3;
    graph->edge[4].weight = 4; KruskalMST(graph); free(graph->edge); free(graph);
    return 0;
}
```

OUTPUT:



VIVA QUESTIONS:

Q1. What is Kruskal's Algorithm?

ANS. Kruskal's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph. It operates by selecting the smallest edges and using a union-find data structure to ensure no cycles are formed. **Q2. What data structure is used in Kruskal's algorithm to detect cycles?**

ANS. The union-find (disjoint-set) data structure is used to detect cycles. It provides efficient methods for checking whether two vertices belong to the same set and for merging two sets.

Q3. How do you sort the edges in Kruskal's algorithm?

ANS. The edges are sorted in non-decreasing order based on their weights, as Kruskal's algorithm selects edges with the smallest weights first to construct the MST.

Q4. What is the time complexity of Kruskal's Algorithm?

ANS. The time complexity is , where is the number of edges and is the number of vertices. Sorting the edges takes , and the union-find operations take for each edge.

Q5. Why is Kruskal's Algorithm called a greedy algorithm?

ANS. It is called greedy because at each step, the algorithm picks the smallest available edge, aiming to build the MST by making the locally optimal choice at every step.