**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No - 1**

**AIM - Program to implement linear, binary search using recursion.**

**Linear Search**

#include <stdio.h>

int linearSearch(int arr[], int target, int index, int size) {

if (index == size) {

return -1;

}

if (arr[index] == target) {

return index;

}

return linearSearch(arr, target, index + 1, size);

}

int main() {

int arr[] = {12, 45, 67, 89, 34, 23};

int target = 89;

int size = sizeof(arr) / sizeof(arr[0]);

int result = linearSearch(arr, target, 0, size);

if (result != -1) {

printf("Element %d found at index %d.\n", target, result);

} else {

printf("Element %d not found in the array.\n", target);

}

return 0;

}

**Output**

Element 89 found at index 3.

Element 5 not found in the array.

Element 12 found at index 0.

**Binary Search**

#include <stdio.h>

int binarySearch(int arr[], int target, int left, int right) {

if (left > right) {

return -1;

}

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

return mid;

}

if (arr[mid] > target) {

return binarySearch(arr, target, left, mid - 1);

}

return binarySearch(arr, target, mid + 1, right);

}

int main() {

int arr[] = {12, 23, 34, 45, 67, 89};

int target = 45;

int size = sizeof(arr) / sizeof(arr[0]);

int result = binarySearch(arr, target, 0, size - 1);

if (result != -1) {

printf("Element %d found at index %d.\n", target, result);

} else {

printf("Element %d not found in the array.\n", target);

}

return 0;

}

**Output**

Element 45 found at index 3.

Element 12 found at index 0.

Element 95 not found in the array.

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No - 2**

**AIM - Program for Quick sort using Divide and Conquer**

#include <stdio.h>

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++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int size = sizeof(arr) / sizeof(arr[0]);

printf("Original Array: ");

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

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

}

printf("\n");

quickSort(arr, 0, size - 1);

printf("Sorted Array: ");

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

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

}

printf("\n");

return 0;

}

**Output**

Original Array: 64 25 12 22 11

Sorted Array: 11 12 22 25 64

Original Array: 90 35 74 70 24

Sorted Array: 24 35 70 74 90

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No - 3**

**AIM - Program for Merge sort using Divide and Conquer.**

#include <stdio.h>

void merge(int arr[], int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

L[i] = arr[l + i];

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

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

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++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10};

int size = sizeof(arr) / sizeof(arr[0]);

printf("Original Array: ");

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

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

}

printf("\n");

mergeSort(arr, 0, size - 1);

printf("Sorted Array: ");

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

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

}

printf("\n");

return 0;

}

**Output**

Original Array: 38 27 43 3 9 82 10

Sorted Array: 3 9 10 27 38 43 82

Original Array: 64 25 12 22 11

Sorted Array: 11 12 22 25 64

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No - 4**

**AIM - Program to implement Fractional Knapsack problem using Greedy method.**

**Greedy Method**

#include <stdio.h>

#include <stdlib.h>

struct Item {

int weight;

int value;

};

int compare(const void \*a, const void \*b) {

double ratio1 = (double)(((struct Item \*)b)->value) / ((struct Item \*)b)->weight;

double ratio2 = (double)(((struct Item \*)a)->value) / ((struct Item \*)a)->weight;

if (ratio1 < ratio2) return -1;

if (ratio1 > ratio2) return 1;

return 0;

}

void fractionalKnapsack(struct Item items[], int n, int capacity) {

qsort(items, n, sizeof(struct Item), compare);

double totalValue = 0.0;

double currentWeight = 0.0;

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

result[i] = 0.0;

}

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

if (currentWeight + items[i].weight <= capacity) {

result[i] = 1.0;

currentWeight += items[i].weight;

totalValue += items[i].value;

} else {

double remainingCapacity = capacity - currentWeight;

result[i] = remainingCapacity / items[i].weight;

totalValue += result[i] \* items[i].value;

break;

}

}

printf("Maximum total value: %.2lf\n", totalValue);

printf("Fraction of each item taken:\n");

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

printf("Item %d: %.2lf\n", i + 1, result[i]);

}

}

int main() {

int n, capacity;

printf("Enter the number of items: ");

scanf("%d", &n);

printf("Enter the capacity of the knapsack: ");

scanf("%d", &capacity);

struct Item items[n];

printf("Enter the weight and value of each item:\n");

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

scanf("%d %d", &items[i].weight, &items[i].value);

}

fractionalKnapsack(items, n, capacity);

return 0;

}

**Output**

Enter the number of items: 4

Enter the capacity of the knapsack: 10

Enter the weight and value of each item:

Item 1: 2 10

Item 2: 3 5

Item 3: 5 15

Item 4: 7 7

Maximum total value: 42.50

Fraction of each item taken:

Item 1: 1.00

Item 2: 1.00

Item 3: 1.00

Item 4: 0.71

**Dynamic Programming Method**

#include <stdio.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

int knapsack(int capacity, int weights[], int values[], int n) {

int i, w;

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

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

for (w = 0; w <= capacity; w++) {

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

dp[i][w] = 0;

} else if (weights[i - 1] <= w) {

dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);

} else {

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

}

}

}

return dp[n][capacity];

}

int main() {

int weights[] = {2, 3, 4, 5};

int values[] = {3, 4, 5, 6};

int capacity = 5;

int n = sizeof(weights) / sizeof(weights[0]);

int result = knapsack(capacity, weights, values, n);

printf("Maximum value: %d\n", result);

return 0;

}

**Output**

Maximum value: 7

**Brute Force Approach**

#include <stdio.h>

#include <stdlib.h>

struct Item {

int weight;

int value;

};

int totalValue(struct Item items[], int selected[], int n) {

int total = 0;

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

if (selected[i] == 1) {

total += items[i].value;

} else if (selected[i] == 2) {

total += (int)((double)items[i].value \* items[i].weight);

}

}

return total;

}

void bruteForceKnapsack(struct Item items[], int selected[], int n, int capacity, int currentWeight, int \*maxValue) {

if (n == 0) {

int value = totalValue(items, selected, n);

if (value > \*maxValue) {

\*maxValue = value;

}

return;

}

if (currentWeight + items[n - 1].weight <= capacity) {

selected[n - 1] = 1;

bruteForceKnapsack(items, selected, n - 1, capacity, currentWeight + items[n - 1].weight, maxValue);

selected[n - 1] = 0;

}

if (currentWeight < capacity) {

selected[n - 1] = 2;

bruteForceKnapsack(items, selected, n - 1, capacity, currentWeight + 1, maxValue);

selected[n - 1] = 0;

}

selected[n - 1] = 0;

bruteForceKnapsack(items, selected, n - 1, capacity, currentWeight, maxValue);

}

int main() {

int n, capacity;

printf("Enter the number of items: ");

scanf("%d", &n);

printf("Enter the capacity of the knapsack: ");

scanf("%d", &capacity);

struct Item items[n];

int selected[n];

printf("Enter the weight and value of each item:\n");

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

scanf("%d %d", &items[i].weight, &items[i].value);

selected[i] = 0;

}

int maxValue = 0;

bruteForceKnapsack(items, selected, n, capacity, 0, &maxValue);

printf("Maximum total value: %d\n", maxValue);

return 0;

}

**Output**

Enter the number of items: 4

Enter the capacity of the knapsack: 10

Enter the weight and value of each item:

Item 1: 2 10

Item 2: 3 5

Item 3: 5 15

Item 4: 7 7

Maximum total value: 42.50

Fraction of each item taken:

Item 1: 1.00

Item 2: 1.00

Item 3: 1.00

Item 4: 0.71

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No - 5**

**AIM - Program to implement single source shortest path.**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 6

int minDistance(int dist[], int visited[]) {

int min = INT\_MAX, min\_index;

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

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

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void printSolution(int dist[]) {

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

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

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

}

}

void dijkstra(int graph[V][V], int source) {

int dist[V];

int visited[V];

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

dist[i] = INT\_MAX;

visited[i] = 0;

}

dist[source] = 0;

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

int u = minDistance(dist, visited);

visited[u] = 1;

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

if (!visited[v] && graph[u][v] && dist[u] != INT\_MAX &&

dist[u] + graph[u][v] < dist[v]) {

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

}

}

}

printSolution(dist);

}

int main() {

int graph[V][V] = {

{0, 1, 4, 0, 0, 0},

{0, 0, 4, 2, 7, 0},

{0, 0, 0, 3, 4, 0},

{0, 0, 0, 0, 5, 0},

{0, 0, 0, 0, 0, 6},

{0, 0, 0, 0, 0, 0}

};

int source = 0;

dijkstra(graph, source);

return 0;

}

**Output**

Vertex Distance from Source

0 0

1 1

2 4

3 3

4 8

5 14

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No - 6**

**AIM - Program to implement Floyd Warshall's algorithm for solving all pairs Shortest Path problem**

#include <stdio.h>

#include <limits.h>

#define V 4

int min(int a, int b) {

return (a < b) ? a : b;

}

void printSolution(int dist[V][V]) {

printf("Shortest distances between all pairs of vertices:\n");

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

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

if (dist[i][j] == INT\_MAX) {

printf("INF\t");

} else {

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

}

}

printf("\n");

}

}

void floydWarshall(int graph[V][V]) {

int dist[V][V];

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

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

dist[i][j] = graph[i][j];

}

}

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

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

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

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX) {

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

}

}

}

}

printSolution(dist);

}

int main() {

int graph[V][V] = {

{0, 5, INT\_MAX, 10},

{INT\_MAX, 0, 3, INT\_MAX},

{INT\_MAX, INT\_MAX, 0, 1},

{INT\_MAX, INT\_MAX, INT\_MAX, 0}

};

floydWarshall(graph);

return 0;

}

**Output**

Shortest distances between all pairs of vertices:

0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No - 7**

**AIM** - **Program to implement 0/1 Knapsack problem using Dynamic programming.**

#include <stdio.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

int knapsack(int capacity, int weights[], int values[], int n) {

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

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 (weights[i - 1] <= w) {

dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);

} else {

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

}

}

}

return dp[n][capacity];

}

int main() {

int weights[] = {2, 3, 4, 5};

int values[] = {3, 4, 5, 6};

int capacity = 5;

int n = sizeof(weights) / sizeof(weights[0]);

int result = knapsack(capacity, weights, values, n);

printf("Maximum value: %d\n", result);

return 0;

}

**Output**

Maximum value: 7

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No -8**

**AIM** - **Program to implement Traveling sales person problem using Dynamic programming**.

#include <stdio.h>

#include <limits.h>

#define V 4

int min(int a, int b) {

return (a < b) ? a : b;

}

int tsp(int graph[V][V], int mask, int pos, int dp[V][1 << V]) {

if (mask == (1 << V) - 1) {

return graph[pos][0];

}

if (dp[pos][mask] != -1) {

return dp[pos][mask];

}

int ans = INT\_MAX;

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

if (!(mask & (1 << city))) {

int newAns = graph[pos][city] + tsp(graph, mask | (1 << city), city, dp);

ans = min(ans, newAns);

}

}

return dp[pos][mask] = ans;

}

int main() {

int graph[V][V] = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

int dp[V][1 << V];

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

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

dp[i][j] = -1;

}

}

int mask = 1;

int pos = 0;

int minCost = tsp(graph, mask, pos, dp);

printf("Minimum cost for TSP: %d\n", minCost);

return 0;

}

**Output**

Minimum cost for TSP: 80

graph[V][V] = {

{0, 10, 15, 20, 12},

{10, 0, 35, 25, 28},

{15, 35, 0, 30, 22},

{20, 25, 30, 0, 19},

{12, 28, 22, 19, 0}

**Output**

Minimum cost for TSP: 91

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No -9**

**AIM** - **Program to implement 8-Queens’ problem using Backtracking.**

#include <stdio.h>

#include <stdbool.h>

#define N 8

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

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

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

printf("%2d ", board[i][j]);

}

printf("\n");

}

}

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

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

if (board[row][i] == 1) {

return false;

}

}

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

if (board[i][j] == 1) {

return false;

}

}

for (int i = row, j = col; i < N && j >= 0; i++, j--) {

if (board[i][j] == 1) {

return false;

}

}

return true;

}

bool solveQueens(int board[N][N], int col) {

if (col >= N) {

return true;

}

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveQueens(board, col + 1)) {

return true;

}

board[i][col] = 0;

}

}

return false;

}

int main() {

int board[N][N] = {0};

if (solveQueens(board, 0)) {

printf("Solution:\n");

printBoard(board);

} else {

printf("No solution exists.\n");

}

return 0;

}

**Output**

**Solution:**

**1 0 0 0 0 0 0 0**

**0 0 0 0 0 0 1 0**

**0 0 0 0 1 0 0 0**

**0 0 0 0 0 0 0 1**

**0 1 0 0 0 0 0 0**

**0 0 0 1 0 0 0 0**

**0 0 0 0 0 1 0 0**

**0 0 1 0 0 0 0 0**

**Name - Ankit Ishwar Patil**

**Roll No – BT3223**

**Practical No -10**

**AIM** - **Program to implement sum of subset problem using Backtracking.**

#include <stdio.h>

#include <stdbool.h>

void printSubset(int set[], int n, bool subset[]) {

printf("Subset with the target sum: { ");

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

if (subset[i]) {

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

}

}

printf("}\n");

}

bool isSubsetSum(int set[], int n, int sum, bool subset[], int currentIndex) {

if (sum == 0) {

printSubset(set, n, subset);

return true;

}

if (currentIndex >= n || sum < 0) {

return false;

}

subset[currentIndex] = true;

if (isSubsetSum(set, n, sum - set[currentIndex], subset, currentIndex + 1)) {

return true;

}

subset[currentIndex] = false;

return isSubsetSum(set, n, sum, subset, currentIndex + 1);

}

int main() {

int set[] = {1, 3, 4, 5, 7};

int n = sizeof(set) / sizeof(set[0]);

int sum = 10;

bool subset[n];

if (!isSubsetSum(set, n, sum, subset, 0)) {

printf("No subset with the target sum found.\n");

}

return 0;

}

**Output**

Subset with the target sum: { 1 4 5 }

int set[] = {2, 4, 6, 8, 10};

int sum = 16;

**Output**

Subset with the target sum: { 2 4 10 }