Daa lab file

1. **Implement the insertion inside iterative and recursive Binary search tree and compare their performance.**

Bst – iterative

Code :-

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

struct Node\* newNode = createNode(data);

if (root == NULL) {

return newNode;

}

struct Node\* current = root;

struct Node\* parent = NULL;

while (current != NULL) {

parent = current;

if (data < current->data) {

current = current->left;

} else if (data > current->data) {

current = current->right;

} else {

free(newNode);

return root;

}

}

if (data < parent->data) {

parent->left = newNode;

} else {

parent->right = newNode;

}

return root;

}

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("In-order traversal of the BST: ");

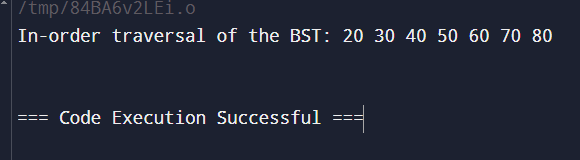
inorderTraversal(root);

printf("\n");

return 0;

}

Output :-



Bst – recursive

Code :-

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

// If the tree is empty, return a new node

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else if (data > root->data) {

root->right = insert(root->right, data);

}

return root;

}

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("In-order traversal of the BST: ");

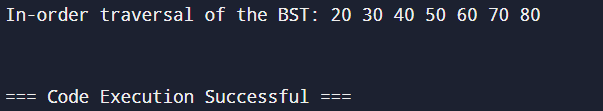
inorderTraversal(root);

printf("\n");

return 0;

}

Output :-



Comparison :-

Generally, both iterative and recursive insertion have similar time complexities of O(log n) in a balanced BST, but the recursive method can lead to stack overflow for a large number of insertions due to function call stack limitations.

1. **Implement divide and conquer based merge sort and quick sort algorithms and compare their performance for the same set of elements.**

Merge sort :-

Code :-

#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {

int i, j, k;

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

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

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

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

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

i = 0;

j = 0;

k = left;

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 left, int right) {

if (left < right) {

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

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void printArray(int arr[], int size) {

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

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

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

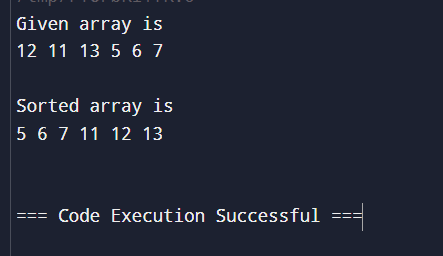
printf("\nSorted array is \n");

printArray(arr, arr\_size);

return 0;

}

Output :-



Quick sort:-

Code :-

#include <stdio.h>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

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

if (low < high) {

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

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void printArray(int arr[], int size) {

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

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

}

printf("\n");

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

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

printf("Original array: \n");

printArray(arr, n);

quickSort(arr, 0, n - 1);

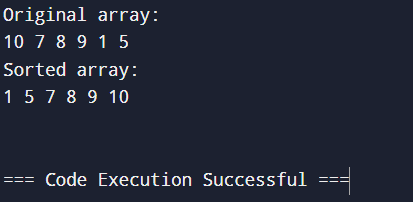
printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output:-



Comparison :-

Merge Sort: Has a time complexity of O(n log n) and guarantees stability, but it requires extra space proportional to the array size for merging.

Quick Sort: Has an average time complexity of O(n log n), but it can degrade to O(n2) if the pivot elements are poorly chosen. It’s generally faster in practice due to its in-place nature and low constant factors, but it is not stable.

1. **Compare the performance of Strassen method of matrix multiplication with traditional way of matrix multiplication.**

Strassen method of matrix multiplication :-

Code :-

#include <stdio.h>

#include <stdlib.h>

#define MAX 64

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 newSize = size / 2;

int temp1[MAX][MAX], temp2[MAX][MAX];

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 M1[MAX][MAX], M2[MAX][MAX], M3[MAX][MAX], M4[MAX][MAX], M5[MAX][MAX], M6[MAX][MAX], M7[MAX][MAX];

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

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

A11[i][j] = A[i][j];

A12[i][j] = A[i][j + newSize];

A21[i][j] = A[i + newSize][j];

A22[i][j] = A[i + newSize][j + newSize];

B11[i][j] = B[i][j];

B12[i][j] = B[i][j + newSize];

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

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

}

}

add(A11, A22, temp1, newSize);

add(B11, B22, temp2, newSize);

strassen(temp1, temp2, M1, newSize);

add(A21, A22, temp1, newSize);

strassen(temp1, B11, M2, newSize);

subtract(B12, B22, temp2, newSize);

strassen(A11, temp2, M3, newSize);

subtract(B21, B11, temp2, newSize);

strassen(A22, temp2, M4, newSize);

add(A11, A12, temp1, newSize);

strassen(temp1, B22, M5, newSize);

subtract(A21, A11, temp1, newSize);

add(B11, B12, temp2, newSize);

strassen(temp1, temp2, M6, newSize);

subtract(A12, A22, temp1, newSize);

add(B21, B22, temp2, newSize);

strassen(temp1, temp2, M7, newSize);

add(M1, M4, temp1, newSize);

subtract(temp1, M5, temp2, newSize);

add(temp2, M7, C, newSize);

add(M3, M5, C + newSize, newSize);

add(M2, M4, C + newSize \* MAX, newSize);

subtract(M1, M2, temp1, newSize);

add(temp1, M3, temp2, newSize);

add(temp2, M6, C + newSize \* MAX + newSize, newSize);

}

void display(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 size;

printf("Enter the size of matrices (must be a power of 2): ");

scanf("%d", &size);

int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];

printf("Enter matrix A:\n");

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

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

scanf("%d", &A[i][j]);

printf("Enter matrix B:\n");

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

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

scanf("%d", &B[i][j]);

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

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

C[i][j] = 0;

strassen(A, B, C, size);

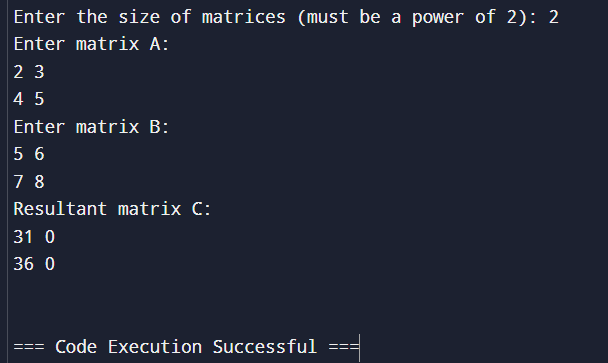
printf("Resultant matrix C:\n");

display(C, size);

return 0;

}

Output :-



Traditional way matrix multiplication :-

Code :-

#include <stdio.h>

#define ROW1 2

#define COL1 3

#define ROW2 3

#define COL2 2

void multiplyMatrices(int firstMatrix[ROW1][COL1], int secondMatrix[ROW2][COL2], int result[ROW1][COL2]) {

for (int i = 0; i < ROW1; i++) {

for (int j = 0; j < COL2; j++) {

result[i][j] = 0;

}

}

for (int i = 0; i < ROW1; i++) {

for (int j = 0; j < COL2; j++) {

for (int k = 0; k < COL1; k++) {

result[i][j] += firstMatrix[i][k] \* secondMatrix[k][j];

}

}

}

}

void displayMatrix(int matrix[ROW1][COL2]) {

for (int i = 0; i < ROW1; i++) {

for (int j = 0; j < COL2; j++) {

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

}

printf("\n");

}

}

int main() {

int firstMatrix[ROW1][COL1] = {

{1, 2, 3},

{4, 5, 6}

};

int secondMatrix[ROW2][COL2] = {

{7, 8},

{9, 10},

{11, 12}

};

int result[ROW1][COL2];

multiplyMatrices(firstMatrix, secondMatrix, result);

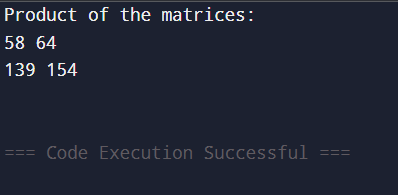
printf("Product of the matrices:\n");

displayMatrix(result);

return 0;

}

Output :-



Comparison :-

Traditional Matrix Multiplication: Runs in O(N3) time, making it slower for large matrices.

Strassen’s Matrix Multiplication: Runs in approximately O(N2.81) time, which generally performs better than the traditional approach for large matrices but requires additional memory and may have larger constant overheads, making it less efficient for small matrices.

1. **Implement the activity selection problem to get a clear understanding of greedy approach.**

Code:-

#include <stdio.h>

#include <stdlib.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 findMaxDeadline(struct Job jobs[], int n) {

int max = jobs[0].deadline;

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

if (jobs[i].deadline > max)

max = jobs[i].deadline;

}

return max;

}

void jobScheduling(struct Job jobs[], int n) {

qsort(jobs, n, sizeof(struct Job), compare);

int max\_deadline = findMaxDeadline(jobs, n);

int result[max\_deadline + 1];

for (int i = 0; i <= max\_deadline; i++)

result[i] = -1;

int jobCount = 0;

int totalProfit = 0;

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

for (int j = jobs[i].deadline; j > 0; j--) {

if (result[j] == -1) {

result[j] = jobs[i].id;

jobCount++;

totalProfit += jobs[i].profit;

break;

}

}

}

printf("Job sequence that maximizes profit:\n");

for (int i = 1; i <= max\_deadline; i++) {

if (result[i] != -1)

printf("Job %d\n", result[i]);

}

printf("Total number of jobs done: %d\n", jobCount);

printf("Total profit: %d\n", totalProfit);

}

int main() {

int n;

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

scanf("%d", &n);

struct Job jobs[n];

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

printf("Enter deadline and profit for job %d: ", i + 1);

scanf("%d %d", &jobs[i].deadline, &jobs[i].profit);

jobs[i].id = i + 1;

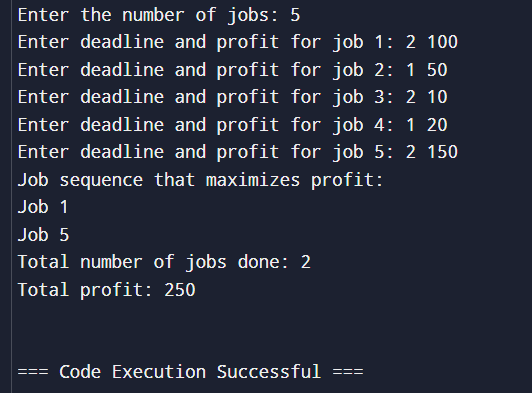
}

jobScheduling(jobs, n);

return 0;

}

Output :-



1. **Get a detailed insight of dynamic programming approach by the implementation of Matrix Chain Multiplication problem and see the impact of parenthesis positioning on time requirements for matrix multiplication.**

Code :-

#include <stdio.h>

#include <limits.h>

int matrixChainMultiplication(int p[], int n) {

int m[n][n];

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

m[i][i] = 0;

}

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

for (int i = 1; i < n - L + 1; i++) {

int j = i + L - 1;

m[i][j] = INT\_MAX;

for (int k = i; k <= j - 1; k++) {

int q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];

if (q < m[i][j]) {

m[i][j] = q;

}

}

}

}

return m[1][n - 1];

}

int main() {

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

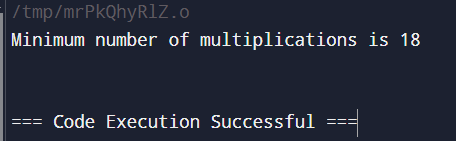
int n = sizeof(p) / sizeof(p[0]);

printf("Minimum number of multiplications is %d\n", matrixChainMultiplication(p, n));

return 0;

}

Output :-



1. **Compare the performance of Dijkstra and Bellman ford algorithm for the single source shortest path problem.**

Dijkstra :-

Code :-

#include <stdio.h>

#include <limits.h>

#define V 9

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

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == 0 && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

return min\_index;

}

void printSolution(int dist[]) {

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

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

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

}

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

int dist[V];

int sptSet[V];

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

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

dist[src] = 0;

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

int u = minDistance(dist, sptSet);

sptSet[u] = 1;

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

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

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

}

}

}

printSolution(dist);

}

int main() {

int graph[V][V] = {

{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

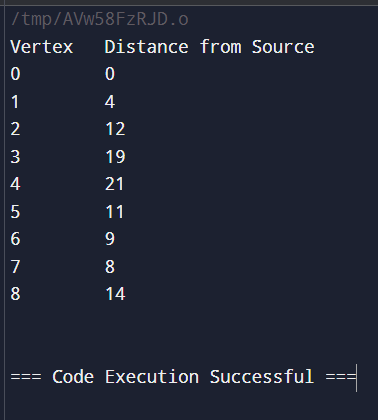
};

dijkstra(graph, 0);

return 0;

}

Output :-



Bellman Ford :-

Code :-

#include <stdio.h>

#include <limits.h>

#include <stdlib.h>

struct Edge {

int src, dest, weight;

};

struct Graph {

int V, E;

struct Edge\* edge;

};

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;

}

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

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

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

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

}

void BellmanFord(struct Graph\* graph, int src) {

int V = graph->V;

int E = graph->E;

int dist[V];

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

dist[i] = INT\_MAX;

dist[src] = 0;

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

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

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

}

}

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

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

printf("Graph contains negative weight cycle\n");

return;

}

}

printSolution(dist, V);

}

int main() {

int V = 5;

int E = 8;

struct Graph\* graph = createGraph(V, E);

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

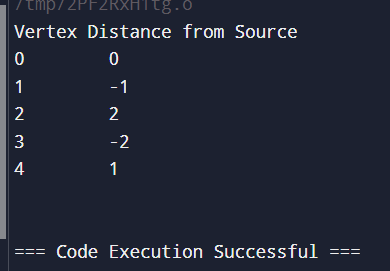
graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;

}

Output :-



Comparison :-

Dijkstra's Algorithm is typically faster, especially on dense graphs with many edges and non-negative weights, as it is optimized by using a priority queue.

Bellman-Ford Algorithm has a higher time complexity, so it tends to be slower, especially on large graphs. However, it can handle negative weights and detect negative-weight cycles, which Dijkstra’s algorithm cannot.

1. **Through 0/1 Knapsack problem, analyze the greedy and dynamic programming approach for the same dataset.**

0/1 knapsack problem by greedy approach :-

Code :-

#include <stdio.h>

#include <stdlib.h>

struct Item {

int weight;

int value;

};

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

struct Item \*item1 = (struct Item \*)a;

struct Item \*item2 = (struct Item \*)b;

double ratio1 = (double)item1->value / item1->weight;

double ratio2 = (double)item2->value / item2->weight;

return (ratio1 > ratio2) ? -1 : (ratio1 < ratio2);

}

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

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

double maxValue = 0.0;

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

if (capacity >= items[i].weight) {

capacity -= items[i].weight;

maxValue += items[i].value;

} else {

maxValue += items[i].value \* ((double)capacity / items[i].weight);

break;

}

}

return maxValue;

}

int main() {

struct Item items[] = { {10, 60}, {20, 100}, {30, 120} };

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

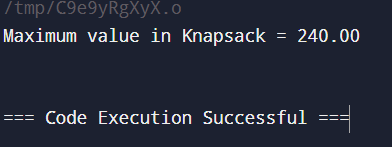
int capacity = 50;

printf("Maximum value in Knapsack = %.2f\n", fractionalKnapsack(items, n, capacity));

return 0;

}

Output :-



0/1 knapsack problem dynamic programming :-

Code :-

#include <stdio.h>

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

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

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

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

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

dp[i][w] = 0;

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

dp[i][w] = (values[i - 1] + dp[i - 1][w - weights[i - 1]]) > dp[i - 1][w]

? (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][W];

}

int main() {

int values[] = {60, 100, 120};

int weights[] = {10, 20, 30};

int W = 50;

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

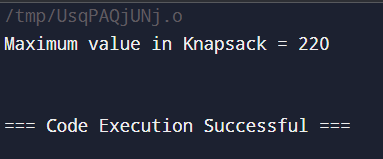
int maxValue = knapsack(W, weights, values, n);

printf("Maximum value in Knapsack = %d\n", maxValue);

return 0;

}

Output :-



1. **Implement the sum of subset.**

Code :-

#include <stdio.h>

#include <stdbool.h>

#define MAX 100

void printSubset(int subset[], int size) {

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

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

}

printf("\n");

}

void subsetSum(int set[], int subset[], int n, int subsetSize, int total, int targetSum) {

// If the subset's sum matches the target, print it

if (total == targetSum) {

printSubset(subset, subsetSize);

return;

}

if (n == 0 || total > targetSum) {

return;

}

subset[subsetSize] = set[0];

subsetSum(set + 1, subset, n - 1, subsetSize + 1, total + set[0], targetSum);

subsetSum(set + 1, subset, n - 1, subsetSize, total, targetSum);

}

int main() {

int set[] = {3, 34, 4, 12, 5, 2};

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

int targetSum = 9;

int subset[MAX];

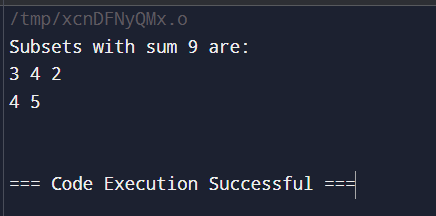
printf("Subsets with sum %d are:\n", targetSum);

subsetSum(set, subset, n, 0, 0, targetSum);

return 0;

}

Output :-



1. **Compare the Backtracking and Branch & Bound Approach by the implementation of 0/1 Knapsack problem. Also compare the performance with dynamic programming approach.**

0/1 knapsack problem using backtracking :-

Code :-

#include <stdio.h>

int knapsackBacktracking(int W, int n, int weights[], int values[], int currentWeight, int currentValue) {

if (n == 0) {

return currentValue;

}

if (currentWeight > W) {

return 0;

}

int exclude = knapsackBacktracking(W, n - 1, weights, values, currentWeight, currentValue);

int include = knapsackBacktracking(W, n - 1, weights, values, currentWeight + weights[n - 1], currentValue + values[n - 1]);

return (include > exclude) ? include : exclude;

}

int main() {

int weights[] = {10, 20, 30};

int values[] = {60, 100, 120};

int W = 50; // Capacity of the knapsack

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

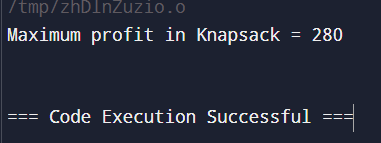
int maxProfit = knapsackBacktracking(W, n, weights, values, 0, 0);

printf("Maximum profit in Knapsack = %d\n", maxProfit);

return 0;

}

Output :-



0/1 knapsack problem using branch & bound :-

Code :-

#include <stdio.h>

#include <stdlib.h>

struct Item {

int weight;

int value;

};

int level;

int profit;

int weight;

float bound;

};

float bound(struct Node u, int n, int W, struct Item items[]) {

if (u.weight >= W) return 0;

float result = u.profit;

int totalWeight = u.weight;

int j = u.level + 1;

while (j < n && totalWeight + items[j].weight <= W) {

totalWeight += items[j].weight;

result += items[j].value;

j++;

}

if (j < n) {

result += (W - totalWeight) \* (float)items[j].value / items[j].weight;

}

return result;

}

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

struct Item \*item1 = (struct Item \*)a;

struct Item \*item2 = (struct Item \*)b;

double r1 = (double)item1->value / item1->weight;

double r2 = (double)item2->value / item2->weight;

return r2 - r1 > 0 ? 1 : -1;

}

int knapsackBranchAndBound(int W, struct Item items[], int n) {

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

struct Node u, v;

int maxProfit = 0;

u.level = -1;

u.profit = 0;

u.weight = 0;

u.bound = bound(u, n, W, items);

struct Node queue[1000];

int front = 0, rear = 0;

queue[rear++] = u;

while (front != rear) {

u = queue[front++];

if (u.level == n - 1) continue;

v.level = u.level + 1;

v.weight = u.weight + items[v.level].weight;

v.profit = u.profit + items[v.level].value;

if (v.weight <= W && v.profit > maxProfit) {

maxProfit = v.profit;

}

v.bound = bound(v, n, W, items);

if (v.bound > maxProfit) {

queue[rear++] = v;

}

v.weight = u.weight;

v.profit = u.profit;

v.bound = bound(v, n, W, items);

if (v.bound > maxProfit) {

queue[rear++] = v;

}

}

return maxProfit;

}

int main() {

struct Item items[] = {{10, 60}, {20, 100}, {30, 120}};

int W = 50;

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

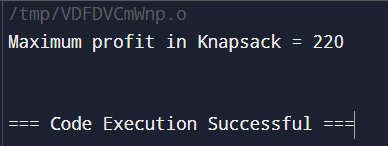
int maxProfit = knapsackBranchAndBound(W, items, n);

printf("Maximum profit in Knapsack = %d\n", maxProfit);

return 0;

}

Output :-



Comparison :-

Backtracking: Exponential in worst-case, O(2n) because it explores all combinations.

Branch and Bound: Prunes some branches, making it more efficient than backtracking, but worst-case remains exponential.

Dynamic Programming: Polynomial O(n⋅W) where W is the knapsack capacity, as it fills up a table of size n×W.

1. **Compare the performance of Rabin-Karp, Knuth-Morris-Pratt and naive stringmatching algorithms.**

Rabin-krap :-

Code :-

#include <stdio.h>

#include <string.h>

#define d 256

#define q 101

void rabinKarpSearch(char \*text, char \*pattern) {

int m = strlen(pattern);

int n = strlen(text);

int i, j;

int patternHash = 0;

int textHash = 0;

int h = 1;

for (i = 0; i < m - 1; i++) {

h = (h \* d) % q;

}

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

patternHash = (d \* patternHash + pattern[i]) % q;

textHash = (d \* textHash + text[i]) % q;

}

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

if (patternHash == textHash) {

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

if (text[i + j] != pattern[j]) {

break;

}

}

if (j == m) {

printf("Pattern found at index %d\n", i);

}

}

if (i < n - m) {

textHash = (d \* (textHash - text[i] \* h) + text[i + m]) % q;

if (textHash < 0) {

textHash = (textHash + q);

}

}

}

}

int main() {

char text[] = "ABCCBAABCABC";

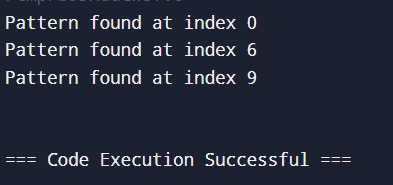
char pattern[] = "ABC";

rabinKarpSearch(text, pattern);

return 0;

}

Output :-



Knuth-Morris-Pratt :-

Code :-

#include <stdio.h>

#include <string.h>

void computeLPSArray(char \*pattern, int m, int \*lps) {

int length = 0;

lps[0] = 0;

int i = 1;

while (i < m) {

if (pattern[i] == pattern[length]) {

length++;

lps[i] = length;

i++;

} else {

if (length != 0) {

length = lps[length - 1];

} else {

lps[i] = 0;

i++;

}

}

}

}

void KMPSearch(char \*text, char \*pattern) {

int n = strlen(text);

int m = strlen(pattern);

int lps[m];

computeLPSArray(pattern, m, lps);

int i = 0;

int j = 0;

while (i < n) {

if (pattern[j] == text[i]) {

i++;

j++;

}

if (j == m) {

printf("Pattern found at index %d\n", i - j);

j = lps[j - 1];

}

else if (i < n && pattern[j] != text[i]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

}

int main() {

char text[] = "ABABDABACDABABCABAB";

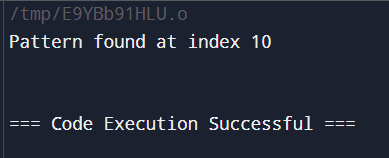
char pattern[] = "ABABCABAB";

KMPSearch(text, pattern);

return 0;

}

Output :-



Naive stringmatching :-

Code :-

#include <stdio.h>

#include <string.h>

void naiveStringMatching(char \*text, char \*pattern) {

int n = strlen(text);

int m = strlen(pattern);

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

int j = 0;

while (j < m && text[i + j] == pattern[j]) {

j++;

}

if (j == m) {

printf("Pattern found at index %d\n", i);

}

}

}

int main() {

char text[] = "ABABDABACDABABCABAB";

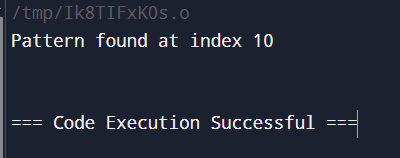
char pattern[] = "ABABCABAB";

naiveStringMatching(text, pattern);

return 0;

}

Output :-



Comparison :-

Naive: O((n−m+1)⋅m)

Rabin-Karp: O(n+m) average, O((n−m+1)⋅m) worst

Knuth-Morris-Pratt : O(n+m)

**Github link :**

https://github.com/Snehasaini16/sneha\_500126085