DESIGN ANALYSIS AND ALGORITHM   
LAB FILE

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

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right; };

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

printf("Memory allocation failed\n"); }

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right); }

}

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

if (root == NULL) {

return createNode(value); }

if (value < root->data) {

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

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

root->right = insert(root->right, value); }

return root;

}

int main(){

struct Node\* root = NULL;

int choice, value,n;

printf("Enter the number of values you want to insert: ");

scanf("%d",&n);

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

printf("Enter value%d: ",i+1);

scanf("%d", &value);

root = insert(root, value);

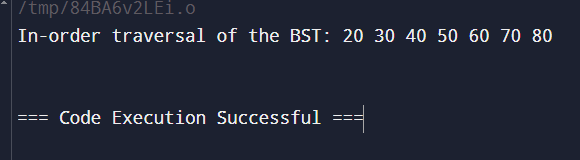
}

inorder(root);

printf("\n");

}

OUTPUT:



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

#include<stdio.h>

int main(){

int n1, n2;

printf("Enter length of arr1: ");

scanf("%d", &n1);

printf("Enter length of arr2: ");

scanf("%d", &n2);

int arr1[n1], arr2[n2];

printf("Enter the elements of arr1: \n");

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

scanf("%d", &arr1[i]);

}

printf("Enter the elements of arr2: \n");

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

scanf("%d", &arr2[j]);

}

printf("arr1:\n");

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

printf("%d ", arr1[i]);

}

printf("\narr2:\n");

for(int i=0; i<n2;i++){

printf("%d ", arr2[i]);

}

int n = n1+n2;

int mergedarr[n];

int i=0, j=0, k=0;

while(i<n1 && j<n2){

if(arr1[i]<arr2[j]){

mergedarr[k] = arr1[i];

i++;

} else {

mergedarr[k] = arr2[j];

j++;

}

k++;

}

while(i<n1){

mergedarr[k] = arr1[i];

i++;

k++;

}

while(j<n2){

mergedarr[k] = arr2[j];

j++;

k++;

}

printf("\nMerged array:\n");

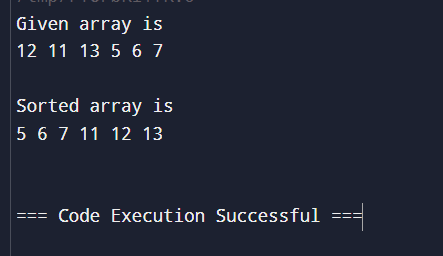
for(int j=0; j<n;j++){

printf("%d ", mergedarr[j]);

}

return 0;

OUTPUT:



QUICKSORT:  
  
#include <stdio.h>

void quickSort(int arr[], int l, int h) {

if (l < h) {

int pivot = arr[h];

int i = l;

for (int j = l; j < h; j++) {

if (arr[j] < pivot) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

i++;

}

}

int temp = arr[i];

arr[i] = arr[h];

arr[h] = temp;

quickSort(arr, l, i - 1);

quickSort(arr, i + 1, h);

}

}

int main() {

int n;

printf("Enter length of array: ");

scanf("%d", &n);

int arr[n];

printf("Enter the elements of array: \n");

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

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

}

quickSort(arr, 0, n - 1);

printf("Sorted array:\n");

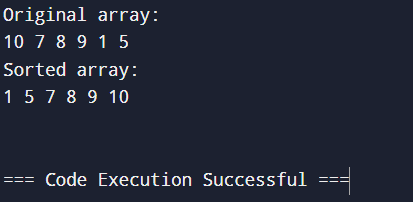
for (int i = 0; i < n; i++) {

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

}

printf("\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.  
  
**3. Compare the performance of Strassen method of matrix multiplication with traditional way of matrix multiplication.**

#include <stdio.h>

void split(int n, int matrix[n][n], int a[n/2][n/2], int b[n/2][n/2], int c[n/2][n/2], int d[n/2][n/2]) {

int mid = n / 2;

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

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

a[i][j] = matrix[i][j];

b[i][j] = matrix[i][j + mid];

c[i][j] = matrix[i + mid][j];

d[i][j] = matrix[i + mid][j + mid];

}

}

}

void add(int n, int A[n][n], int B[n][n], int C[n][n]) {

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

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

C[i][j] = A[i][j] + B[i][j];

}

void subtract(int n, int A[n][n], int B[n][n], int C[n][n]) {

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

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

C[i][j] = A[i][j] - B[i][j];

}

void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {

if (n == 1) {

C[0][0] = A[0][0] \* B[0][0];

return;

}

int mid = n / 2;

int a[mid][mid], b[mid][mid], c[mid][mid], d[mid][mid];

int e[mid][mid], f[mid][mid], g[mid][mid], h[mid][mid];

int m1[mid][mid], m2[mid][mid], m3[mid][mid], m4[mid][mid], m5[mid][mid], m6[mid][mid], m7[mid][mid];

int temp1[mid][mid], temp2[mid][mid];

split(n, A, a, b, c, d);

split(n, B, e, f, g, h);

add(mid, a, d, temp1);

add(mid, e, h, temp2);

strassen(mid, temp1, temp2, m1);

add(mid, c, d, temp1);

strassen(mid, temp1, e, m2);

subtract(mid, f, h, temp2);

strassen(mid, a, temp2, m3);

subtract(mid, g, e, temp2);

strassen(mid, d, temp2, m4);

add(mid, a, b, temp1);

strassen(mid, temp1, h, m5);

subtract(mid, c, a, temp1);

add(mid, e, f, temp2);

strassen(mid, temp1, temp2, m6);

subtract(mid, b, d, temp1);

add(mid, g, h, temp2);

strassen(mid, temp1, temp2, m7);

add(mid, m1, m4, temp1);

subtract(mid, temp1, m5, temp1);

add(mid, temp1, m7, C[0]); // C11

add(mid, m3, m5, C[0] + mid); // C12

add(mid, m2, m4, C[mid]); // C21

add(mid, m1, m3, temp1);

subtract(mid, temp1, m2, temp1);

add(mid, temp1, m6, C[mid] + mid); // C22

}

int main() {

int n = 4;

int A[4][4] = {

{12, 34, 5, 2},

{22, 10, 1, 0},

{1, 2, 3, 4},

{0, 1, 4, 5}

};

int B[4][4] = {

{3, 4, 2, 1},

{2, 1, 1, 0},

{0, 1, 1, 1},

{1, 0, 2, 3}

};

int C[4][4] = {0};

strassen(n, A, B, C);

printf("Product achieved using Strassen's algorithm:\n");

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

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

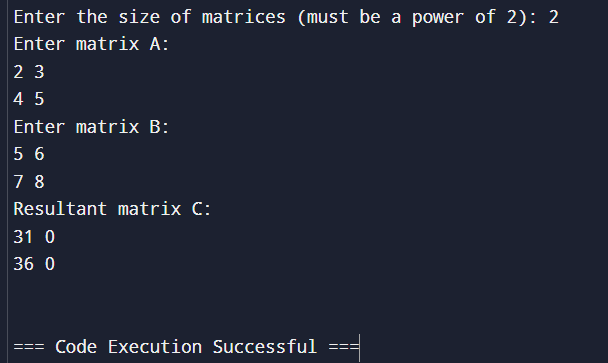
printf("%d\t", C[i][j]);

}

printf("\n");

}

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}

};

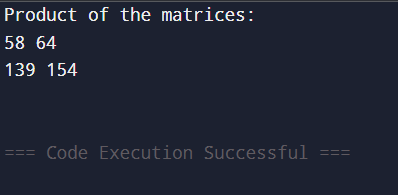
int result[ROW1][COL2];

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.

**4. Implement the activity selection problem to get a clear understanding of greedy approach.**  
#include <stdio.h>

#include <stdlib.h>

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

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

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

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

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

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

int 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) {

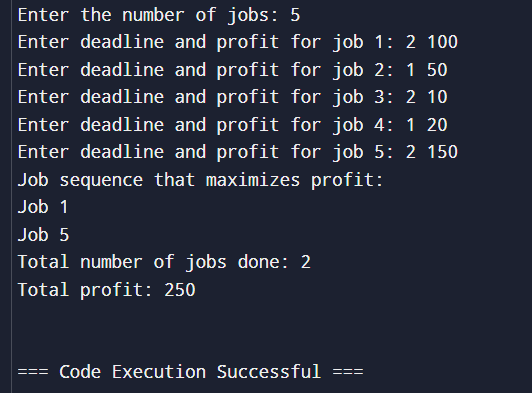
int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}  
  
OUTPUT:  


**5. 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.**#include <stdio.h>

#include <limits.h>

// Function to perform Matrix Chain Multiplication using Dynamic Programming

void matrixChainOrder(int p[], int n) {

int m[n][n]; // DP table to store the minimum multiplication cost

int s[n][n]; // Table to store the split points for optimal parenthesization

// Initialize the main diagonal to 0, since single matrices need no multiplication

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

m[i][i] = 0;

// Fill the table in a bottom-up manner

for (int len = 2; len < n; len++) { // len is the chain length

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

int j = i + len - 1;

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

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

// Cost = cost of splitting at k + cost of multiplying the results

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

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

m[i][j] = cost;

s[i][j] = k; // Store the split point

}

}

}

}

// Output the minimum cost

printf("Minimum number of multiplications is %d\n", m[1][n - 1]);

// Function to print optimal parenthesis placement

void printOptimalParenthesis(int s[][n], int i, int j) {

if (i == j)

printf("A%d", i);

else {

printf("(");

printOptimalParenthesis(s, i, s[i][j]);

printOptimalParenthesis(s, s[i][j] + 1, j);

printf(")");

}

}

printf("Optimal parenthesis arrangement: ");

printOptimalParenthesis(s, 1, n - 1);

printf("\n");

**}**

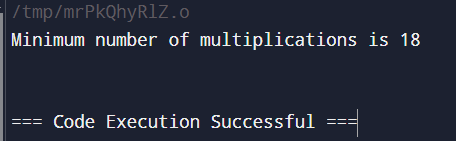
**int main() {**

**int p[] = {40, 20, 30, 10, 30}; // Matrix dimensions**

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

**matrixChainOrder(p, n);**

**return 0;**

**}** **6. Compare the performance of Dijkstra and Bellman ford algorithm for the single**

**source shortest path problem.  
  
Dijsktra algo:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 5 // Number of vertices in the graph

// Function to find the vertex with minimum distance value

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

int min = INT\_MAX, min\_index;

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

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

min = dist[v], min\_index = v;

return min\_index;

}

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

int dist[V]; // dist[i] will hold the shortest distance from src to i

int visited[V]; // visited[i] is 1 if vertex i is processed

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

dist[i] = INT\_MAX, visited[i] = 0;

dist[src] = 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];

}

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

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

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

}  
  
**Bellam ford algo:**  
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

struct Edge {

int src, dest, weight;

};

void bellmanFord(struct Edge edges[], int V, int E, int src) {

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 = edges[j].src;

int v = edges[j].dest;

int weight = edges[j].weight;

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

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

}

}

// Check for negative-weight cycles

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

int u = edges[j].src;

int v = edges[j].dest;

int weight = edges[j].weight;

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

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

return;

}

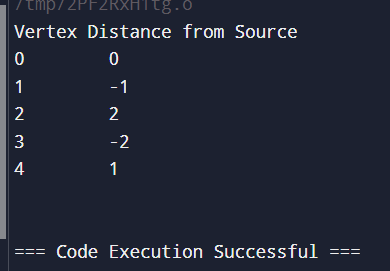
}

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

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

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

}

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.

7.**Through 0/1 Knapsack problem, analyze the greedy and dynamic programming approach for the same dataset.**#include <stdio.h>

#include <stdlib.h>

// Structure for items

typedef struct {

int value;

int weight;

double ratio; // value-to-weight ratio

} Item;

// Comparator function for sorting items by value-to-weight ratio (descending)

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

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

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

if (item2->ratio > item1->ratio) return 1;

if (item2->ratio < item1->ratio) return -1;

return 0;

}

// Greedy Approach

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

Item items[n];

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

items[i].value = values[i];

items[i].weight = weights[i];

items[i].ratio = (double)values[i] / weights[i];

}

qsort(items, n, sizeof(Item), compare); // Sort items by ratio

int totalValue = 0, remainingCapacity = capacity;

printf("Greedy Approach Selected Items:\n");

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

if (items[i].weight <= remainingCapacity) {

printf("Weight: %d, Value: %d\n", items[i].weight, items[i].value);

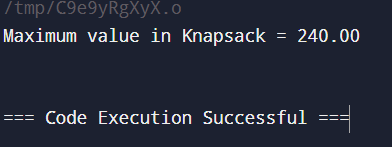
totalValue += items[i].value;

remainingCapacity -= items[i].weight;

}

}

printf("Total Value: %d\n", totalValue);

}  
  
OUTPUT:-  


// Dynamic Programming Approach

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

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

// Initialize DP table

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

else

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

}

}

// Backtrack to find the selected items

int w = capacity;

printf("Dynamic Programming Selected Items:\n");

for (int i = n; i > 0 && w > 0; i--) {

if (dp[i][w] != dp[i - 1][w]) {

printf("Weight: %d, Value: %d\n", weights[i - 1], values[i - 1]);

w -= weights[i - 1];

}

}

printf("Total Value: %d\n", dp[n][capacity]);

}

// Main Function

int main() {

int values[] = {6, 5, 8, 10};

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

int capacity = 8;

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

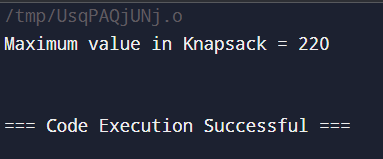
printf("Greedy Approach:\n");

knapsack\_greedy(values, weights, n, capacity);

printf("\nDynamic Programming Approach:\n");

knapsack\_dp(values, weights, n, capacity);

return 0;

}  
  
OUTPUT:  
  
  
  
**8. Implement the sum of subset**#include <stdio.h>

#define MAX 100

void sumOfSubsets(int set[], int subset[], int n, int target, int subset\_size, int sum, int index) {

// If the subset sum equals the target, print the subset

if (sum == target) {

printf("Subset: ");

for (int i = 0; i < subset\_size; i++) {

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

}

printf("\n");

return;

}

// If the sum exceeds the target or no elements are left to explore, stop

if (sum > target || index >= n) {

return;

}

// Include the current element in the subset

subset[subset\_size] = set[index];

sumOfSubsets(set, subset, n, target, subset\_size + 1, sum + set[index], index + 1);

// Exclude the current element and move to the next

sumOfSubsets(set, subset, n, target, subset\_size, sum, index + 1);

}

int main() {

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

int target = 9;

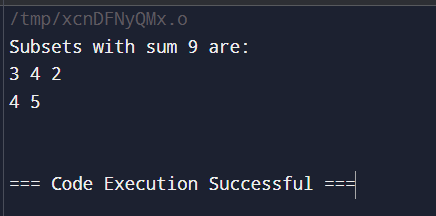
int n = sizeof(set) / sizeof(set[0]);

int subset[MAX]; // Temporary array to store subsets

printf("Subsets with sum equal to %d:\n", target);

sumOfSubsets(set, subset, n, target, 0, 0, 0);

return 0;

}  
  
  
  
**Compare the Backtracking and Branch & Bound Approach by the implementation**

**of 0/1 Knapsack problem. Also compare the performance with dynamic**

**programming approach.**#include <stdio.h>

int max\_value = 0; // To store the maximum value

void knapsack\_backtracking(int n, int weights[], int values[], int capacity, int index, int current\_value, int current\_weight) {

if (current\_weight > capacity) {

return; // If weight exceeds capacity, stop exploring

}

if (current\_value > max\_value) {

max\_value = current\_value; // Update max value

}

if (index >= n) {

return; // All items processed

}

// Include the current item

knapsack\_backtracking(n, weights, values, capacity, index + 1, current\_value + values[index], current\_weight + weights[index]);

// Exclude the current item

knapsack\_backtracking(n, weights, values, capacity, index + 1, current\_value, current\_weight);

}

int main() {

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

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

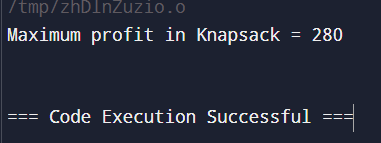
int capacity = 50;

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

knapsack\_backtracking(n, weights, values, capacity, 0, 0, 0);

printf("Maximum value (Backtracking): %d\n", max\_value);

return 0;

}  
OUTPUT:-  
  
  
**0/1 knapsack problem using branch & bound :-**

#include <stdio.h>

// Structure to represent a node in the decision tree

typedef struct {

int level, value, weight;

float bound;

} Node;

// Function to calculate the upper bound on the maximum value

float calculate\_bound(int level, int weight, int value, int n, int capacity, int weights[], int values[]) {

if (weight >= capacity) return 0;

float bound = value;

int total\_weight = weight;

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

if (total\_weight + weights[i] <= capacity) {

total\_weight += weights[i];

bound += values[i];

} else {

bound += (float)(capacity - total\_weight) \* values[i] / weights[i];

break;

}

}

return bound;

}

void knapsack\_branch\_and\_bound(int weights[], int values[], int n, int capacity) {

Node current, next;

int max\_value = 0;

// Initialize root node

current.level = -1;

current.value = 0;

current.weight = 0;

current.bound = calculate\_bound(0, 0, 0, n, capacity, weights, values);

Node queue[100]; // Queue for BFS

int front = 0, rear = 0;

queue[rear++] = current;

while (front < rear) {

current = queue[front++];

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

// Include the next item

next.level = current.level + 1;

next.weight = current.weight + weights[next.level];

next.value = current.value + values[next.level];

if (next.weight <= capacity && next.value > max\_value) {

max\_value = next.value;

}

next.bound = calculate\_bound(next.level + 1, next.weight, next.value, n, capacity, weights, values);

if (next.bound > max\_value) {

queue[rear++] = next;

}

// Exclude the next item

next.weight = current.weight;

next.value = current.value;

next.bound = calculate\_bound(next.level + 1, next.weight, next.value, n, capacity, weights, values);

if (next.bound > max\_value) {

queue[rear++] = next;

}

}

printf("Maximum value (Branch and Bound): %d\n", max\_value);

}

int main() {

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

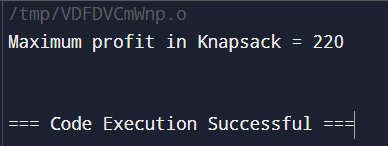
int weights[] = {10, 20, 30};

int capacity = 50;

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

knapsack\_branch\_and\_bound(weights, values, n, capacity);

return 0;

}  
  
  
  
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.

**10. Compare the performance of Rabin-Karp, Knuth-Morris-Pratt and naive stringmatching algorithms.  
  
RABIN KARP:**#include <stdio.h>

#include <string.h>

#define d 256 // Number of characters in the input alphabet

void rabinKarp(char \*text, char \*pattern, int q) {

int n = strlen(text);

int m = strlen(pattern);

int h = 1; // Value of d^(m-1)

int p = 0; // Hash value for pattern

int t = 0; // Hash value for text

int i, j;

// Precompute h = d^(m-1) % q

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

h = (h \* d) % q;

// Calculate initial hash values for pattern and first window of text

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

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

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

}

printf("Rabin-Karp Algorithm:\n");

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

// Check hash values

if (p == t) {

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

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

break;

}

if (j == m) {

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

}

}

// Calculate hash for the next window

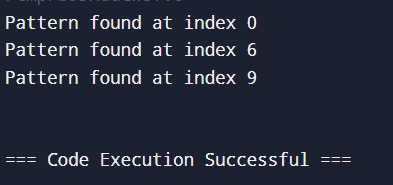
if (i < n - m) {

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

if (t < 0) t += q;

}

}

}   
  
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";

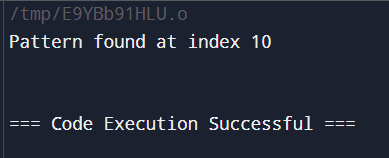
char pattern[] = "ABABCABAB";

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";

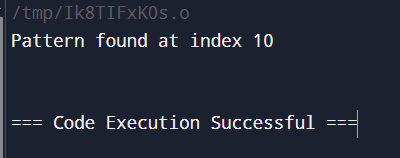
char pattern[] = "ABABCABAB";

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/AmulyaDang/Algorithms\_3rd\_sem\_500125610