LAB FILE DAA

**NAME- VEDANSH RAJ**

**SAP- 500126374**

**ROLL NO – R2142232007**

**BATCH-58**

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

**performance.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int key;

struct TreeNode\* left;

struct TreeNode\* right;

};

struct TreeNode\* createNode(int key) {

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

newNode->key = key;

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

return newNode;

}

struct TreeNode\* insert\_iterative(struct TreeNode\* root, int key) {

struct TreeNode\* newNode = createNode(key);

if (root == NULL) {

return newNode;

}

struct TreeNode\* current = root;

struct TreeNode\* parent = NULL;

while (current != NULL) {

parent = current;

if (key < current->key) {

current = current->left;

} else {

current = current->right;

}

}

if (key < parent->key) {

parent->left = newNode;

} else {

parent->right = newNode;

}

return root;

}

void inorder\_traversal(struct TreeNode\* root) {

if (root != NULL) {

inorder\_traversal(root->left);

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

inorder\_traversal(root->right);

}

}

int main() {

struct TreeNode\* root = NULL;

int n, key;

printf("Enter the number of keys to insert: ");

scanf("%d", &n);

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

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

scanf("%d", &key);

root = insert\_iterative(root, key);

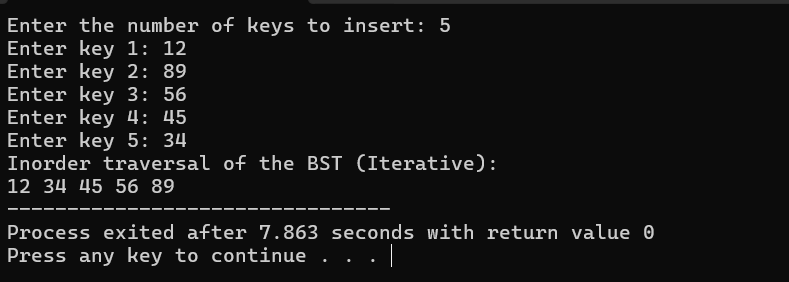
}

// Print inorder traversal of the BST

printf("Inorder traversal of the BST (Iterative):\n");

inorder\_traversal(root);

return 0;



}

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int key;

struct TreeNode\* left;

struct TreeNode\* right;

};

struct TreeNode\* createNode(int key) {

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

newNode->key = key;

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

return newNode;

}

struct TreeNode\* insert\_recursive(struct TreeNode\* root, int key) {

if (root == NULL) {

return createNode(key);

}

if (key < root->key) {

root->left = insert\_recursive(root->left, key);

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

root->right = insert\_recursive(root->right, key);

}

return root;

}

void inorder\_traversal(struct TreeNode\* root) {

if (root != NULL) {

inorder\_traversal(root->left);

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

inorder\_traversal(root->right);

}

}

int main() {

struct TreeNode\* root = NULL;

int n, key;

printf("Enter the number of keys to insert: ");

scanf("%d", &n);

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

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

scanf("%d", &key);

root = insert\_recursive(root, key);

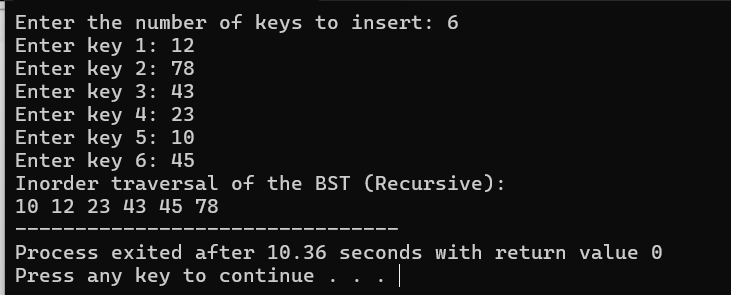
}

printf("Inorder traversal of the BST (Recursive):\n");

inorder\_traversal(root);

return 0;

}



**Iterative Insertion**:

**Time Complexity**: **O(h)**

**Recursive Insertion**:

**Time Complexity**: **O(h)**.

**Overall**:

In the best case (balanced tree): O(log n)

In the worst case (skewed tree): O(n)

**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>

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[] = {12,11,23,7,8,9,1,2};

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

printf("Given array is \n");

printArray(arr, arr\_size);

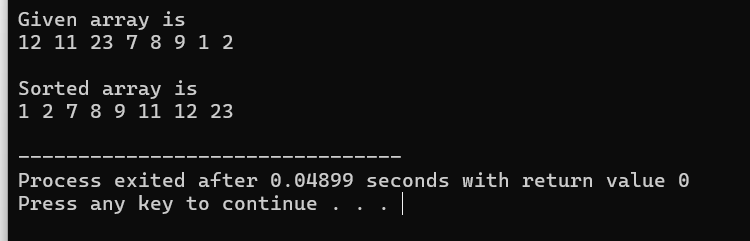
mergeSort(arr, 0, arr\_size - 1);

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

printArray(arr, arr\_size);

return 0;

}



#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 n;

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

scanf("%d", &n);

int arr[n];

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

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

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

}

quickSort(arr, 0, n - 1);

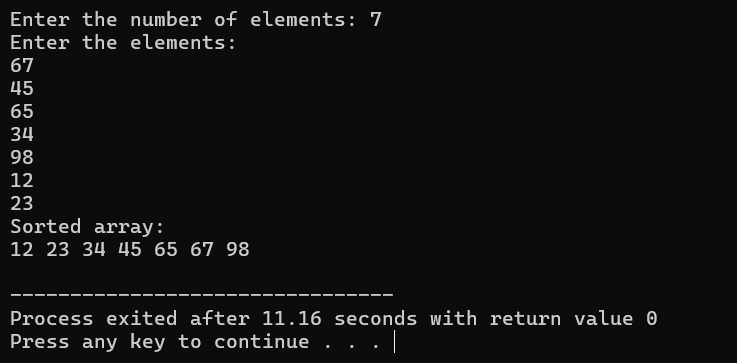
// Print the sorted array

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}



**Merge Sort**:

**Time Complexity**: **O(n log n)**.

**Quick Sort**:

**Time Complexity**:

Best case: **O(n log n)** (when the pivot divides the array into almost equal parts)

Average case: **O(n log n)**

Worst case: **O(n^2)** (when the pivot is the smallest or largest element, causing unbalanced splits)

**3-Compare the performance of Strassen method of matrix multiplication with**

**traditional way of matrix multiplication.**

#include <stdio.h>

int main() {

int z[2][2];

int i, j;

int m1, m2, m3, m4, m5, m6, m7;

int x[2][2], y[2][2];

printf("Enter the elements of the first 2x2 matrix:\n");

for(i = 0; i < 2; i++) {

for(j = 0; j < 2; j++) {

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

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

}

}

printf("Enter the elements of the second 2x2 matrix:\n");

for(i = 0; i < 2; i++) {

for(j = 0; j < 2; j++) {

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

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

}

}

printf("\nThe first matrix is: ");

for(i = 0; i < 2; i++) {

printf("\n");

for(j = 0; j < 2; j++) {

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

}

}

printf("\nThe second matrix is: ");

for(i = 0; i < 2; i++) {

printf("\n");

for(j = 0; j < 2; j++) {

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

}

}

m1 = (x[0][0] + x[1][1]) \* (y[0][0] + y[1][1]);

m2 = (x[1][0] + x[1][1]) \* y[0][0];

m3 = x[0][0] \* (y[0][1] - y[1][1]);

m4 = x[1][1] \* (y[1][0] - y[0][0]);

m5 = (x[0][0] + x[0][1]) \* y[1][1];

m6 = (x[1][0] - x[0][0]) \* (y[0][0] + y[0][1]);

m7 = (x[0][1] - x[1][1]) \* (y[1][0] + y[1][1]);

z[0][0] = m1 + m4 - m5 + m7;

z[0][1] = m3 + m5;

z[1][0] = m2 + m4;

z[1][1] = m1 - m2 + m3 + m6;

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

for(i = 0; i < 2 ; i++) {

printf("\n");

for(j = 0; j < 2; j++) {

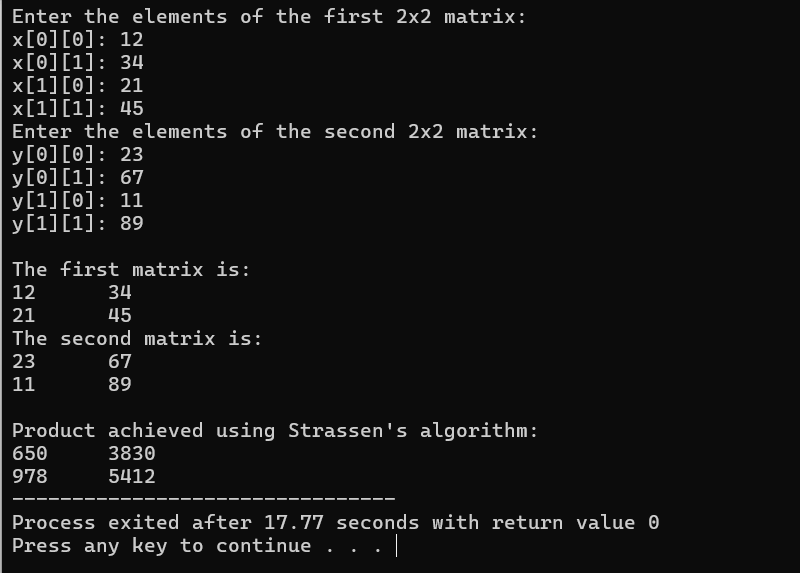
printf("%d\t", z[i][j]);

}

}

return 0;

}



**Traditional Matrix Multiplication**:

**Time Complexity**: **O(n^3)**

**Strassen's Matrix Multiplication**:

**Time Complexity**: **O(n^2.81)**

**Overall**:

**Traditional**: O(n^3)

**Strassen**: O(n^2.81) — Faster for large matrices, but requires more memory and is less stable than traditional multiplication.

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

**approach.**

#include <stdio.h>

typedef struct {

int start;

int end;

} Activity;

void sortActivities(Activity activities[], int n) {

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

for (int j = 0; j < n - i - 1; j++) {

if (activities[j].end > activities[j + 1].end) {

Activity temp = activities[j];

activities[j] = activities[j + 1];

activities[j + 1] = temp;

}

}

}

}

void activitySelection(Activity activities[], int n) {

sortActivities(activities, n);

printf("Selected Activities: \n");

int lastEnd = activities[0].end;

printf("Activity 1: Start = %d, End = %d\n", activities[0].start, activities[0].end);

int count = 1;

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

if (activities[i].start >= lastEnd) {

printf("Activity %d: Start = %d, End = %d\n", i + 1, activities[i].start, activities[i].end);

lastEnd = activities[i].end;

count++;

}

}

printf("\nTotal number of selected activities: %d\n", count);

}

int main() {

int n;

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

scanf("%d", &n);

Activity activities[n];

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

printf("Enter start and end time for activity %d: ", i + 1);

scanf("%d %d", &activities[i].start, &activities[i].end);

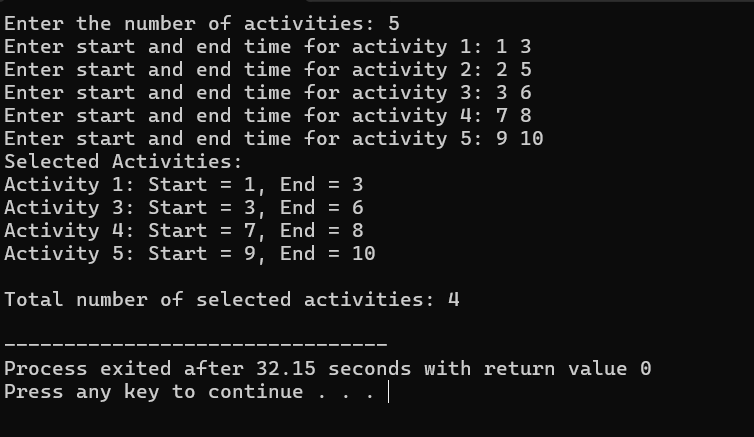
}

activitySelection(activities, n);

return 0;

}

**Time Complexity**: **O(n log n)**



**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>

int matrixChainOrder(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 n;

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

scanf("%d", &n);

int arr[n + 1];

printf("Enter the dimensions of matrices \n");

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

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

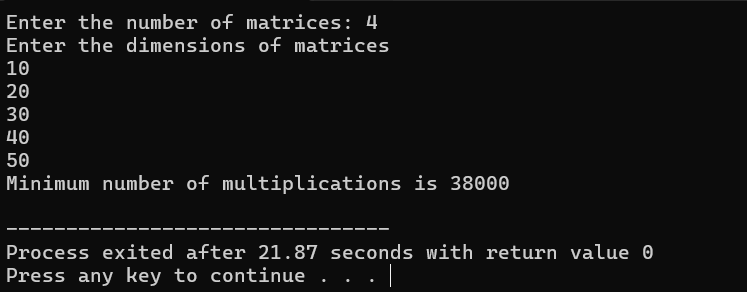
}

printf("Minimum number of multiplications is %d\n", matrixChainOrder(arr, n + 1));

return 0;

}

**Time Complexity**: **O(n^3)**



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

**source shortest path problem.**

#include<stdio.h>

#include<conio.h>

#define INFINITY

#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()

{

int G[MAX][MAX],i,j,n,u;

printf("Enter no. of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

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

printf("\nEnter the starting node:");

scanf("%d",&u);

dijkstra(G,n,u);

return 0;

}

void dijkstra(int G[MAX][MAX],int n,int startnode)

{

int cost[MAX][MAX],distance[MAX],pred[MAX];

int visited[MAX],count,mindistance,nextnode,i,j;

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

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

if(G[i][j]==0)

cost[i][j]= INFINITY;

else

cost[i][j]=G[i][j];

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

{

distance[i]=cost[startnode][i];

pred[i]=startnode;

visited[i]=0;

}

distance[startnode]=0;

visited[startnode]=1;

count=1;

while(count<n-1)

{

mindistance=INFINITY;

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

if(distance[i]<mindistance&&!visited[i])

{

mindistance=distance[i];

nextnode=i;

}

visited[nextnode]=1;

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

if(!visited[i])

if(mindistance+cost[nextnode][i]<distance[i])

{

distance[i]=mindistance+cost[nextnode][i];

pred[i]=nextnode;

}

count++;

}

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

if(i!=startnode)

{

printf("\nDistance of node%d=%d",i,distance[i]);

printf("\nPath=%d",i);

j=i;

do

{

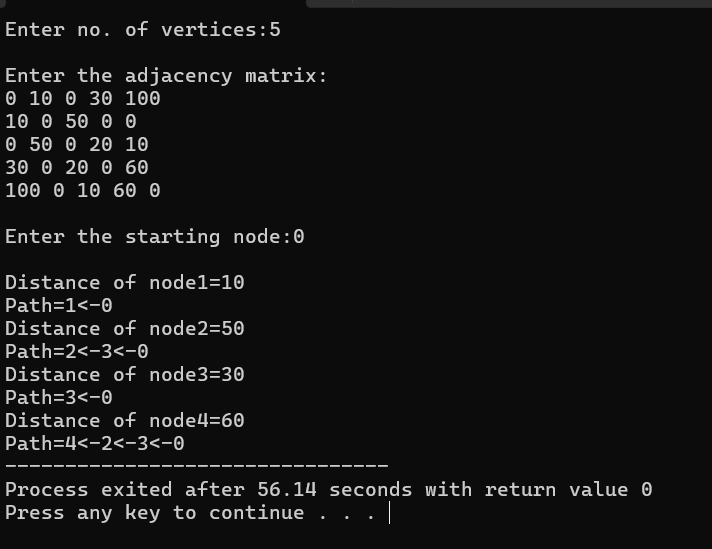
j=pred[j];

printf("<-%d",j);

}while(j!=startnode);

}

}



#include <stdio.h>

#include <limits.h>

#define V 5

#define E 8

struct Edge {

int src, dest, weight;

};

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

int dist[V];

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

dist[i] = INT\_MAX;

dist[src] = 0;

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

for (int j = 0; j < numEdges; 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;

}

}

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

int u = edges[i].src;

int v = edges[i].dest;

int weight = edges[i].weight;

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

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

}

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

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

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

}

int main() {

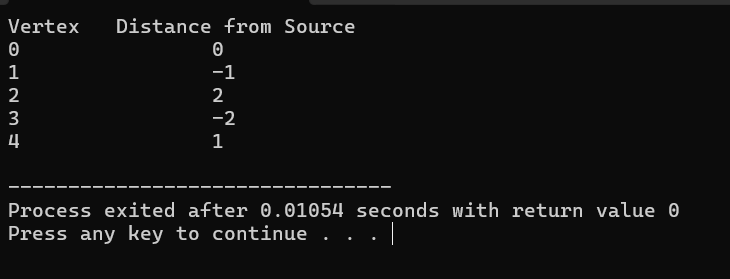
struct Edge edges[E] = { {0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2}, {1, 4, 2},

{3, 2, 5}, {3, 1, 1}, {4, 3, -3} };

bellmanFord(edges, E, 0);

return 0;

}

****

**Dijistra Time Complexity**: **O((V + E) log V)**

**Bellman-Ford Algorithm**:

**Time Complexity**: **O(V \* E)**

**Overall**:

**Dijkstra’s**: **O((V + E) log V)** (better for graphs with non-negative weights and sparse graphs).

**Bellman-Ford**: **O(V \* E)** (can handle negative weights and detect negative cycles but slower).

**7/9-Through 0/1 Knapsack problem, analyze the greedy and dynamic programming**

**approach for the same dataset. Compare the Backtracking and Branch & Bound Approach by the implementation of 0/1 Knapsack problem.**

#include <stdio.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

void knapSack(int W, int wt[], int val[], int n) {

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

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

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

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

K[i][w] = 0;

else if (wt[i - 1] <= w)

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

else

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

}

}

printf("Maximum value: %d\n", K[n][W]);

}

int main() {

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

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

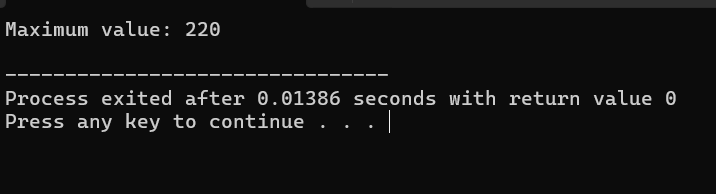
int W = 50;

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

knapSack(W, wt, val, n);

return 0;

}



#include <stdio.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

int knapSackBacktrack(int W, int wt[], int val[], int n, int idx, int currW, int currVal) {

if (idx == n || currW == W)

return currVal;

int include = 0;

if (wt[idx] <= W - currW)

include = knapSackBacktrack(W, wt, val, n, idx + 1, currW + wt[idx], currVal + val[idx]);

int exclude = knapSackBacktrack(W, wt, val, n, idx + 1, currW, currVal);

return max(include, exclude);

}

int main() {

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

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

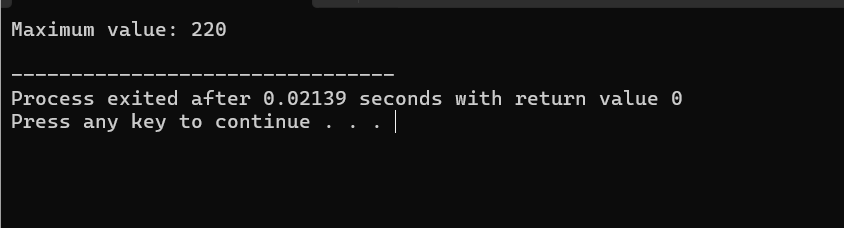
int W = 50;

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

printf("Maximum value: %d\n", knapSackBacktrack(W, wt, val, n, 0, 0, 0));

return 0;

}



#include <stdio.h>

#include <stdlib.h>

#define MAX\_ITEMS 100

struct Item {

int value;

int weight;

float ratio;

};

struct Node {

int level;

int profit;

int weight;

float bound;

};

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

float r1 = ((struct Item\*)a)->ratio;

float r2 = ((struct Item\*)b)->ratio;

if (r1 > r2) return -1;

if (r1 < r2) return 1;

return 0;

}

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

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

float result = u.profit;

int j = u.level + 1;

int totalWeight = u.weight;

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

totalWeight += items[j].weight;

result += items[j].value;

j++;

}

if (j < n)

result += (W - totalWeight) \* items[j].ratio;

return result;

}

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

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

struct Node queue[MAX\_ITEMS];

int front = -1, rear = -1;

struct Node root = {-1, 0, 0, 0.0};

queue[++rear] = root;

int maxProfit = 0;

while (front != rear) {

struct Node u = queue[++front];

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

struct Node v = u;

v.level = u.level + 1;

if (v.weight + items[v.level].weight <= W) {

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

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

if (v.profit > maxProfit)

maxProfit = v.profit;

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

if (v.bound > maxProfit)

queue[++rear] = v;

}

v = u;

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

if (v.bound > maxProfit)

queue[++rear] = v;

}

return maxProfit;

}

int main() {

struct Item items[] = {{60, 10, 6}, {100, 20, 5}, {120, 30, 4}};

int W = 50;

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

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

items[i].ratio = (float)items[i].value / items[i].weight;

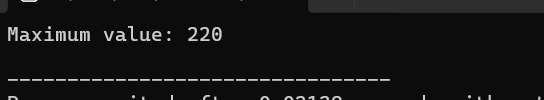
}

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

printf("Maximum profit: %d\n", maxProfit);

return 0;

}



**Time Complexity: Using Dynamic Progrramming**

**O(n \* W)** where n is the number of items and W is the knapsack capacity.

This is because we iterate over each item and each weight capacity to update the DP table.

**Space Complexity:**

**O(n \* W)** for the DP table (or **O(W)** if optimized with a single row).

**Time Complexity: Using Backtracking**

**O(2^n)** where n is the number of items.

In the worst case, backtracking explores every subset of items, leading to exponential time complexity.

While backtracking may prune some branches early (if the weight exceeds the knapsack capacity), it still explores many combinations.

**Space Complexity:**

**O(n)** for recursion stack.

**Time Complexity: Using Branch and Bound**

The time complexity can be **better than backtracking**, but in the worst case, it could still be **O(2^n)**.

The time complexity is highly dependent on how well the bound function prunes the search space. In the best case, the search tree is pruned significantly, reducing the number of nodes to explore.

**Space Complexity:**

**O(n)** for the recursive stack or the storage for active nodes in the queue.

**8.Implement the sum of subset.**

#include <stdio.h>

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

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

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

printf("\n");

}

void sumOfSubset(int set[], int subset[], int size, int target, int n, int index) {

if (target == 0) {

printSubset(subset, size);

return;

}

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

if (target - set[i] >= 0) {

subset[size] = set[i];

sumOfSubset(set, subset, size + 1, target - set[i], n, i + 1);

}

}

}

int main() {

int set[] = {10, 7, 5, 18, 12, 20, 15};

int target = 35;

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

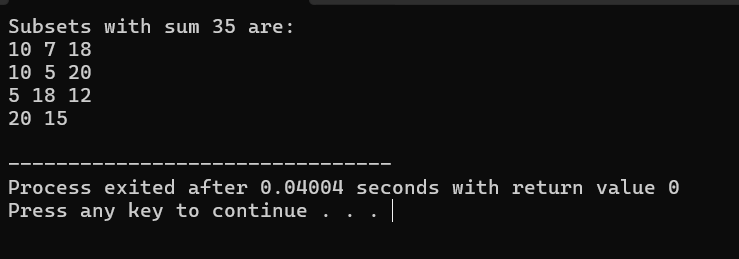
int subset[10];

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

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

return 0;

}



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

#include <stdio.h>

#include <string.h>

#define d 256

#define q 101

void rabinKarp(char\* pat, char\* txt) {

int M = strlen(pat);

int N = strlen(txt);

int p = 0, t = 0, h = 1;

for (int i = 0; i < M - 1; i++)

h = (h \* d) % q;

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

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

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

}

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

if (p == t) {

int j;

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

if (txt[i + j] != pat[j])

break;

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

}

if (i < N - M) {

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

if (t < 0)

t = (t + q);

}

}

}

int main() {

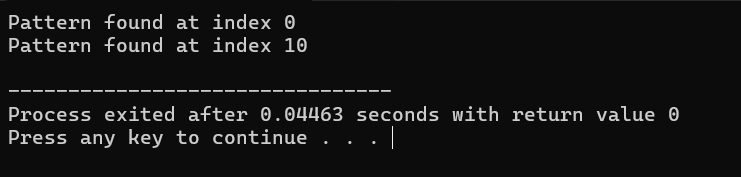
char txt[] = "GEEKS FOR GEEKS";

char pat[] = "GEEK";

rabinKarp(pat, txt);

return 0;

}



#include <stdio.h>

#include <string.h>

void computeLPSArray(char\* pat, int M, int\* lps) {

int len = 0;

lps[0] = 0;

int i = 1;

while (i < M) {

if (pat[i] == pat[len]) {

len++;

lps[i] = len;

i++;

} else {

if (len != 0) {

len = lps[len - 1];

} else {

lps[i] = 0;

i++;

}

}

}

}

void KMPSearch(char\* pat, char\* txt) {

int M = strlen(pat);

int N = strlen(txt);

int lps[M];

computeLPSArray(pat, M, lps);

int i = 0;

int j = 0;

while (i < N) {

if (pat[j] == txt[i]) {

j++;

i++;

}

if (j == M) {

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

j = lps[j - 1];

} else if (i < N && pat[j] != txt[i]) {

if (j != 0)

j = lps[j - 1];

else

i++;

}

}

}

int main() {

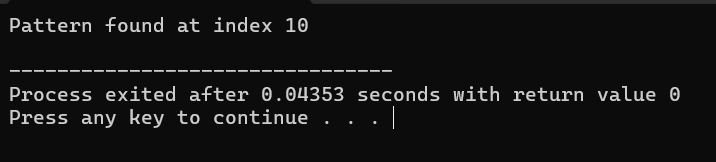
char txt[] = "ABABDABACDABABCABAB";

char pat[] = "ABABCABAB";

KMPSearch(pat, txt);

return 0;

}



#include <stdio.h>

#include <string.h>

void naiveSearch(char\* pat, char\* txt) {

int M = strlen(pat);

int N = strlen(txt);

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

int j;

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

if (txt[i + j] != pat[j])

break;

if (j == M)

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

}

}

int main() {

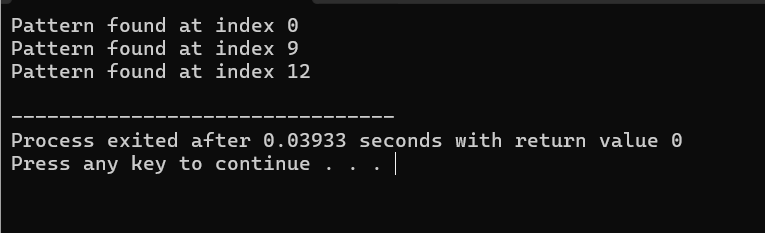
char txt[] = "AABAACAADAABAABA";

char pat[] = "AABA";

naiveSearch(pat, txt);

return 0;

}



**Time Complexity: Naïve Approach**

**Best Case:** **O(n)** if the pattern matches at the very first position (where n is the length of the text).

**Worst Case:** **O(n \* m)**.

**Average Case:** Generally **O(n \* m)**.

**Time Complexity: Rabin Karpin**

**Best Case:** **O(n + m)** when there are no hash collisions (fast match).

**Worst Case:** **O(n \* m)** if there are many hash collisions (the algorithm rechecks the pattern for every hash match).

**Average Case:** **O(n + m)** assuming hash collisions are minimal and hash functions are efficient

**Time Complexity: Knuth Morris Algorithm**

**Best Case:** **O(n)** when the pattern matches the text quickly.

**Worst Case:** **O(n + m)** where n is the length of the text and m is the length of the pattern (since the preprocessing step takes **O(m)** and the matching step takes **O(n)**).

**Average Case:** **O(n + m)**, efficient because it uses the partial match table to avoid unnecessary checks.

**GITHUB LINK**

https://github.com/vedanshraj90/Algorithms\_lab\_3rd\_sem\_500126374