***INSERTION SORT***

########## WITHOUT COMPARISON ############

#include <iostream>

using namespace std;

// Function to perform Insertion Sort

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1], that are greater than key,

// to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

// Function to print an array

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

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

cout << arr[i] << " ";

}

cout << endl;

}

int main() {

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

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

cout << "Original array: ";

printArray(arr, n);

insertionSort(arr, n);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

\*\*\*\*\*\*\*\*\*Q-1.Write a program to sort the elements of an array using Insertion Sort (The program

should report the number of comparisons).\*\*\*\*\*\*\*\*\*\*

#include <iostream>

using namespace std;

void insertionSort(int arr[], int n, int &comparisons) {

comparisons = 0;

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1], that are greater than key,

// to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

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

j--;

comparisons++;

}

// If the above while condition fails, it means one more comparison was made

if (j >= 0) {

comparisons++;

}

arr[j + 1] = key;

}

}

int main() {

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

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

int comparisons;

insertionSort(arr, n, comparisons);

cout << "Sorted array: ";

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

cout << arr[i] << " ";

}

cout << endl;

cout << "Number of comparisons: " << comparisons << endl;

return 0;

}

***MERGE SORT***

############ WITHOUT COMPARISON ###########

#include <iostream>

using namespace std;

// Function to merge two halves of an array

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

int n1 = mid - left + 1;

int n2 = right - mid;

// Temporary arrays

int\* L = new int[n1];

int\* R = new int[n2];

// Copy data to temporary arrays L[] and R[]

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];

// Merge the temporary arrays back into arr[left..right]

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++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

delete[] L;

delete[] R;

}

// Function to implement merge sort

void mergeSort(int arr[], int left, int right) {

if (left < right) {

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

// Sort first and second halves

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

// Function to print an array

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

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

cout << arr[i] << " ";

cout << endl;

}

int main() {

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

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

cout << "Original array: " << endl;

printArray(arr, n);

mergeSort(arr, 0, n - 1);

cout << "Sorted array: " << endl;

printArray(arr, n);

return 0;

}

\*\*\*\*\*\*\*\*\*Q-1.Write a program to sort the elements of an array using Merge Sort (The program

should report the number of comparisons).\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

using namespace std;

int comparisonCount = 0;

// Function to merge two subarrays of arr[]

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

int n1 = mid - left + 1;

int n2 = right - mid;

// Create temporary arrays

int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]

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];

// Merge the temporary arrays back into arr[left..right]

int i = 0;

int j = 0;

int k = left;

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

comparisonCount++;

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Function to implement merge sort

void mergeSort(int arr[], int left, int right) {

if (left < right) {

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

// Sort first and second halves

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

// Function to print an array

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

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

cout << arr[i] << " ";

cout << endl;

}

int main() {

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

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

cout << "Original array: \n";

printArray(arr, arr\_size);

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

cout << "Sorted array: \n";

printArray(arr, arr\_size);

cout << "Number of comparisons: " << comparisonCount << endl;

return 0;

}

***QUICK SORT***

############# WITHOUT COMPARISON ###############

#include <iostream>

// Function to swap two elements

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

// Function to partition the array

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

int pivot = arr[high]; // Pivot element

int i = low - 1; // Index of smaller element

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++; // Increment index of smaller element

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

}

}

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

return (i + 1);

}

// Function to implement quick sort

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

if (low < high) {

int pi = partition(arr, low, high); // Partitioning index

// Recursively sort elements before

// partition and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to print an array

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

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

std::cout << arr[i] << " ";

std::cout << std::endl;

}

int main() {

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

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

std::cout << "Original array: \n";

printArray(arr, n);

quickSort(arr, 0, n - 1);

std::cout << "Sorted array: \n";

printArray(arr, n);

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*Q-1.Write a program to sort the elements of an array using Quick Sort (The program

should report the number of comparisons).\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

using namespace std;

// Function to swap two elements

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

// Function to partition the array and count comparisons

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

int pivot = arr[high]; // Pivot element

int i = low - 1; // Index of smaller element

for (int j = low; j <= high - 1; j++) {

comparisons++;

if (arr[j] < pivot) {

i++; // Increment index of smaller element

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

}

}

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

return (i + 1);

}

// Function to implement quick sort and count comparisons

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

if (low < high) {

int pi = partition(arr, low, high, comparisons); // Partitioning index

// Recursively sort elements before partition and after partition

quickSort(arr, low, pi - 1, comparisons);

quickSort(arr, pi + 1, high, comparisons);

}

}

// Function to print an array

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

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

cout << arr[i] << " ";

cout << endl;

}

int main() {

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

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

int comparisons = 0;

cout << "Original array: \n";

printArray(arr, n);

quickSort(arr, 0, n - 1, comparisons);

cout << "Sorted array: \n";

printArray(arr, n);

cout << "Number of comparisons: " << comparisons << endl;

return 0;

}

***HEAP SORT***

########## WITHOUT COMPARISON ################

#include <iostream>

using namespace std;

void heapify(int arr[], int N, int i)

{

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < N && arr[l] > arr[largest])

largest = l;

if (r < N && arr[r] > arr[largest])

largest = r;

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, N, largest);

}

}

void heapSort(int arr[], int N)

{

for (int i = N / 2 - 1; i >= 0; i--)

heapify(arr, N, i);

for (int i = N - 1; i > 0; i--) {

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

void printArray(int arr[], int N)

{

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

cout << arr[i] << " ";

cout << "\n";

}

int main()

{

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

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

cout << "Given array is \n";

printArray(arr, N);

heapSort(arr, N);

cout << "Sorted array is \n";

printArray(arr, N);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*Q-1. Write a program to sort the elements of an array using Heap Sort (The program

should report the number of comparisons).\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

using namespace std;

class HeapSort {

private:

int comparisons;

void heapify(int arr[], int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left = 2\*i + 1

int right = 2 \* i + 2; // right = 2\*i + 2

// Compare left child with root

if (left < n) {

comparisons++;

if (arr[left] > arr[largest])

largest = left;

}

// Compare right child with largest so far

if (right < n) {

comparisons++;

if (arr[right] > arr[largest])

largest = right;

}

// Swap and continue heapifying if root is not largest

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

void buildHeap(int arr[], int n) {

// Build a maxheap

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

}

public:

HeapSort() : comparisons(0) {}

void sort(int arr[], int n) {

buildHeap(arr, n);

// One by one extract an element from heap

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

// Move current root to end

swap(arr[0], arr[i]);

// Call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

int getComparisons() const {

return comparisons;

}

};

// Function to print an array

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

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

cout << arr[i] << " ";

cout << endl;

}

int main() {

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

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

cout << "Original array: " << endl;

printArray(arr, n);

HeapSort heapSort;

heapSort.sort(arr, n);

cout << "Sorted array: " << endl;

printArray(arr, n);

cout << "Number of comparisons: " << heapSort.getComparisons() << endl;

return 0;

}

***COUNT SORT***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Q-1. Write a program to sort the elements of an array using Count Sort.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

#include <algorithm> // For std::max

using namespace std;

void countSort(vector<int>& arr) {

if (arr.empty()) return;

// Find the maximum element in the array

int maxElement = \*max\_element(arr.begin(), arr.end());

int minElement = \*min\_element(arr.begin(), arr.end());

int range = maxElement - minElement + 1;

// Create a count array to store the count of each unique object

vector<int> count(range, 0);

// Store the count of each element

for (int num : arr) {

count[num - minElement]++;

}

// Modify the count array by adding the previous counts (cumulative count)

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

count[i] += count[i - 1];

}

// Output array to store sorted elements

vector<int> output(arr.size());

// Build the output array

for (int i = arr.size() - 1; i >= 0; i--) {

output[count[arr[i] - minElement] - 1] = arr[i];

count[arr[i] - minElement]--;

}

// Copy the sorted elements back to the original array

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

arr[i] = output[i];

}

}

void printArray(const vector<int>& arr) {

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

int main() {

vector<int> arr = {4, 2, 2, 8, 3, 3, 1, 7, 5, 0, 9, 6};

cout << "Original array: ";

printArray(arr);

countSort(arr);

cout << "Sorted array: ";

printArray(arr);

return 0;

}

***BREADTH – FS***

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

void bfs(vector<vector<int> >& adjList, int startNode,

vector<bool>& visited)

{

queue<int> q;

visited[startNode] = true;

q.push(startNode);

while (!q.empty()) {

int currentNode = q.front();

q.pop();

cout << currentNode << " ";

for (int neighbor : adjList[currentNode]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

void addEdge(vector<vector<int> >& adjList, int u, int v)

{

adjList[u].push\_back(v);

}

int main()

{

int vertices = 5;

vector<vector<int> > adjList(vertices);

addEdge(adjList, 0, 1);

addEdge(adjList, 0, 2);

addEdge(adjList, 1, 3);

addEdge(adjList, 1, 4);

addEdge(adjList, 2, 4);

vector<bool> visited(vertices, false);

cout << "Breadth First Traversal starting from vertex "

"0: ";

bfs(adjList, 0, visited);

return 0;

}

***DEPTH – FS***

#include <bits/stdc++.h>

using namespace std;

class Graph {

public:

map<int, bool> visited;

map<int, list<int> > adj;

void addEdge(int v, int w);

void DFS(int v);

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

}

void Graph::DFS(int v)

{

visited[v] = true;

cout << v << " ";

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

int main()

{

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal"

" (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

***PRIM***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Q-1. Write a program to determine a minimum spanning tree of a graph using the Prim’s algorithm. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

#include <climits>

#include <queue>

using namespace std;

typedef pair<int, int> Edge; // (weight, vertex)

// Function to find the MST using Prim's algorithm

void primMST(vector<vector<Edge>>& graph) {

int V = graph.size();

vector<int> key(V, INT\_MAX); // Key values used to pick minimum weight edge in cut

vector<int> parent(V, -1); // Array to store constructed MST

vector<bool> inMST(V, false); // To represent set of vertices not yet included in MST

// Priority queue to pick minimum weight edge

priority\_queue<Edge, vector<Edge>, greater<Edge>> pq;

// Starting with the first vertex (vertex 0)

key[0] = 0;

pq.push({0, 0}); // (key, vertex)

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

if (inMST[u])

continue;

inMST[u] = true; // Include vertex u in MST

// Update key values of the adjacent vertices of the picked vertex

for (auto& [weight, v] : graph[u]) {

if (!inMST[v] && weight < key[v]) {

key[v] = weight;

pq.push({key[v], v});

parent[v] = u;

}

}

}

// Print the constructed MST

cout << "Edge \tWeight\n";

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

cout << parent[i] << " - " << i << "\t" << key[i] << "\n";

}

int main() {

int V = 5; // Number of vertices in the graph

vector<vector<Edge>> graph(V);

// Graph edges (example graph)

graph[0].push\_back({2, 1});

graph[0].push\_back({3, 3});

graph[1].push\_back({2, 0});

graph[1].push\_back({3, 2});

graph[1].push\_back({4, 3});

graph[2].push\_back({3, 1});

graph[2].push\_back({1, 3});

graph[2].push\_back({6, 4});

graph[3].push\_back({3, 0});

graph[3].push\_back({4, 1});

graph[3].push\_back({1, 2});

graph[3].push\_back({5, 4});

graph[4].push\_back({6, 2});

graph[4].push\_back({5, 3});

primMST(graph);

return 0;

}

***STRASSEN***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Q-1. Write a program to multiply two matrices using the Strassen’s algorithm for matrix multiplication. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

using namespace std;

typedef vector<vector<int>> Matrix;

Matrix add(const Matrix& A, const Matrix& B) {

int n = A.size();

Matrix C(n, vector<int>(n));

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

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

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

return C;

}

Matrix subtract(const Matrix& A, const Matrix& B) {

int n = A.size();

Matrix C(n, vector<int>(n));

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

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

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

return C;

}

Matrix strassen(const Matrix& A, const Matrix& B) {

int n = A.size();

if (n == 1) {

Matrix C(1, vector<int>(1));

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

return C;

}

int k = n / 2;

Matrix A11(k, vector<int>(k)), A12(k, vector<int>(k)), A21(k, vector<int>(k)), A22(k, vector<int>(k));

Matrix B11(k, vector<int>(k)), B12(k, vector<int>(k)), B21(k, vector<int>(k)), B22(k, vector<int>(k));

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

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

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

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

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

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

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

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

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

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

}

Matrix M1 = strassen(add(A11, A22), add(B11, B22));

Matrix M2 = strassen(add(A21, A22), B11);

Matrix M3 = strassen(A11, subtract(B12, B22));

Matrix M4 = strassen(A22, subtract(B21, B11));

Matrix M5 = strassen(add(A11, A12), B22);

Matrix M6 = strassen(subtract(A21, A11), add(B11, B12));

Matrix M7 = strassen(subtract(A12, A22), add(B21, B22));

Matrix C11 = add(subtract(add(M1, M4), M5), M7);

Matrix C12 = add(M3, M5);

Matrix C21 = add(M2, M4);

Matrix C22 = add(subtract(add(M1, M3), M2), M6);

Matrix C(n, vector<int>(n));

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

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

C[i][j] = C11[i][j];

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

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

C[i + k][j + k] = C22[i][j];

}

return C;

}

void printMatrix(const Matrix& matrix) {

for (const auto& row : matrix) {

for (int val : row) {

cout << val << " ";

}

cout << endl;

}

}

int main() {

int n = 4; // Example size (must be a power of 2)

Matrix A = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12},

{13, 14, 15, 16}

};

Matrix B = {

{17, 18, 19, 20},

{21, 22, 23, 24},

{25, 26, 27, 28},

{29, 30, 31, 32}

};

Matrix C = strassen(A, B);

cout << "Result matrix: " << endl;

printMatrix(C);

return 0;

}

***0-1 KNAPSACK***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Q-1. Write a program to solve the 0-1 knapsack problem. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to solve the 0-1 Knapsack problem

int knapsack(int W, const vector<int>& weights, const vector<int>& values) {

int n = weights.size();

vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

// Build the dp array

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

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

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

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

} else {

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

}

}

}

// The maximum value that can be obtained with the given weight limit

return dp[n][W];

}

int main() {

int W = 50; // Maximum weight the knapsack can carry

vector<int> weights = {10, 20, 30}; // Weights of the items

vector<int> values = {60, 100, 120}; // Values of the items

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

cout << "Maximum value in Knapsack: " << maxValue << endl;

return 0;

}