**11)**

**import** java.util.\*;

**class** DinningPhilosohper{

**private** **final** Object[]forks;

**public** DinningPhilosohper(**int** n) {

forks=**new** Object[n];

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

forks[i]=**new** Object();

}

}

**public** **void** philosopher(**int** id) {

**while**(**true**) {

think(id);

**synchronized**(forks[id]) {

**synchronized**(forks[(id+1)%forks.length]) {

eat(id);

}

}

}

}

**public** **void** think (**int** id) {

System.***out***.print("the philosopher"+id+"is thinking");

System.***out***.println();

**try** {

Thread.*sleep*((**long**) (Math.*random*()\*1000));

}**catch**(InterruptedException e) {

e.printStackTrace();

}

}

**public** **void** eat (**int** id) {

System.***out***.print("philosopher"+id+"is eating");

System.***out***.println();

**try** {

Thread.*sleep*((**long**) (Math.*random*()\*1000));

}**catch**(InterruptedException e) {

e.printStackTrace();

}

}

**public** **static** **void** main (String[]args) {

**int** philosopher=5;

DinningPhilosohper dinningphilosopher=**new** DinningPhilosohper(philosopher);

Thread[]threads=**new** Thread[philosopher];

**for**(**int** i=0;i<philosopher;i++) {

**int** id=1;

threads[i]=**new** Thread(()->dinningphilosopher.philosopher( id));

threads[i].start();

}

}

}

5)

**class** knapsack{

**public** **static** **int** knapsacksI(**int** values[],**int** weights[],**int** capacity) {

**int** n=weights.length;

**int** dp[][]=**new** **int**[n+1][capacity+1];

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

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

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

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

}**else** {

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

}

}}

**return** dp[n][capacity];

}

**public** **static** **void** main(String[]args) {

**int** values[]= {};

**int** weights[]= {};

**int** capacity=6;

**int** maxprofit=*knapsacksI*(values,weights,capacity);

System.***out***.print(maxprofit);

}

}

6)

public class NQueens {

public static void main(String[] args) {

// Compare execution times for N = 4 to N = 8

for (int n = 4; n <= 8; n++) {

System.out.println("Solving " + n + "-Queens:");

long startTime = System.currentTimeMillis();

solveNQueens(n);

long endTime = System.currentTimeMillis();

System.out.println("Time taken for " + n + "-Queens: " + (endTime - startTime) + " ms\n");

}

}

// Method to solve N-Queens problem

public static void solveNQueens(int n) {

int[][] board = new int[n][n];

if (placeQueens(board, 0)) {

printBoard(board);

} else {

System.out.println("No solution exists for " + n + "-Queens.");

}

}

// Backtracking function to place queens

private static boolean placeQueens(int[][] board, int row) {

int n = board.length;

if (row == n) {

return true; // All queens placed successfully

}

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

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

board[row][col] = 1; // Place queen

if (placeQueens(board, row + 1)) {

return true; // Recurse to place the next queen

}

board[row][col] = 0; // Backtrack

}

}

return false; // No safe position found

}

// Check if placing a queen at (row, col) is safe

private static boolean isSafe(int[][] board, int row, int col) {

int n = board.length;

// Check column

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

if (board[i][col] == 1) return false;

}

// Check upper-left diagonal

for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] == 1) return false;

}

// Check upper-right diagonal

for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {

if (board[i][j] == 1) return false;

}

return true;

}

// Print the board

private static void printBoard(int[][] board) {

for (int[] row : board) {

for (int cell : row) {

System.out.print((cell == 1 ? "Q " : ". "));

}

System.out.println();

}

System.out.println();

}

}

8)

import java.util.Arrays;

public class TSPBranchAndBound {

static final int INF = Integer.MAX\_VALUE;

public static void main(String[] args) {

// Adjacency matrix representation of the graph

int[][] graph = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

int n = graph.length;

TSPBranchAndBound tsp = new TSPBranchAndBound();

int result = tsp.solveTSP(graph, n);

System.out.println("Minimum cost: " + result);

}

// Function to solve TSP using Branch and Bound

public int solveTSP(int[][] graph, int n) {

boolean[] visited = new boolean[n];

visited[0] = true; // Start from the first city

return tspHelper(graph, visited, 0, n, 1, 0, INF);

}

// Recursive helper for Branch and Bound

private int tspHelper(int[][] graph, boolean[] visited, int currentCity, int n, int count, int cost, int bound) {

if (count == n && graph[currentCity][0] > 0) {

// Return to starting city

return Math.min(bound, cost + graph[currentCity][0]);

}

int minCost = bound;

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

if (!visited[nextCity] && graph[currentCity][nextCity] > 0) {

visited[nextCity] = true;

minCost = Math.min(

minCost,

tspHelper(graph, visited, nextCity, n, count + 1, cost + graph[currentCity][nextCity], minCost)

);

visited[nextCity] = false;

}

}

return minCost;

}

}

12)

public class SimpleMatrixMultiplication {

public static void main(String[] args) throws InterruptedException {

int n = 3; // Matrix size (n x n)

// Example matrices

int[][] matrixA = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

int[][] matrixB = {

{9, 8, 7},

{6, 5, 4},

{3, 2, 1}

};

// Sequential multiplication

long startTime = System.currentTimeMillis();

int[][] resultSequential = multiplySequential(matrixA, matrixB);

long sequentialTime = System.currentTimeMillis() - startTime;

System.out.println("Time taken (Sequential): " + sequentialTime + " ms");

printMatrix(resultSequential);

// Multithreaded multiplication

startTime = System.currentTimeMillis();

int[][] resultMultithreaded = multiplyMultithreaded(matrixA, matrixB);

long multithreadedTime = System.currentTimeMillis() - startTime;

System.out.println("Time taken (Multithreaded): " + multithreadedTime + " ms");

printMatrix(resultMultithreaded);

}

// Sequential matrix multiplication

private static int[][] multiplySequential(int[][] A, int[][] B) {

int n = A.length;

int[][] result = new int[n][n];

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

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

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

result[i][j] += A[i][k] \* B[k][j];

}

}

}

return result;

}

// Multithreaded matrix multiplication

private static int[][] multiplyMultithreaded(int[][] A, int[][] B) throws InterruptedException {

int n = A.length;

int[][] result = new int[n][n];

Thread[] threads = new Thread[n];

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

final int row = i;

threads[i] = new Thread(() -> {

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

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

result[row][j] += A[row][k] \* B[k][j];

}

}

});

threads[i].start();

}

// Wait for all threads to finish

for (Thread thread : threads) {

thread.join();

}

return result;

}

// Print a matrix

private static void printMatrix(int[][] matrix) {

for (int[] row : matrix) {

for (int value : row) {

System.out.print(value + " ");

}

System.out.println();

}

}

}

1 and 2)

import java.util.Random;

public class SortingComparison {

// Quicksort Implementation

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

if (low < high) {

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

quickSort(arr, low, pivotIndex - 1); // Sort left part

quickSort(arr, pivotIndex + 1, high); // Sort right part

}

}

private static 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, j);

}

}

swap(arr, i + 1, high);

return i + 1;

}

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

// Mergesort Implementation

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

if (left < right) {

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

mergeSort(arr, left, mid); // Sort left half

mergeSort(arr, mid + 1, right); // Sort right half

merge(arr, left, mid, right); // Merge sorted halves

}

}

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

int n1 = mid - left + 1;

int n2 = right - mid;

int[] L = new int[n1];

int[] R = new int[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++;

}

}

public static void main(String[] args) {

// Generate random data

Random rand = new Random();

int[] data = new int[500];

for (int i = 0; i < data.length; i++) {

data[i] = rand.nextInt(1000) + 1; // Random values between 1 and 1000

}

// Copy data for reuse

int[] quickSortData = data.clone();

int[] mergeSortData = data.clone();

// Measure Quicksort time

long quickStartTime = System.nanoTime();

quickSort(quickSortData, 0, quickSortData.length - 1);

long quickEndTime = System.nanoTime();

System.out.println("Quicksort Time (500 values): " + (quickEndTime - quickStartTime) / 1e6 + " ms");

// Measure Mergesort time

long mergeStartTime = System.nanoTime();

mergeSort(mergeSortData, 0, mergeSortData.length - 1);

long mergeEndTime = System.nanoTime();

System.out.println("Mergesort Time (500 values): " + (mergeEndTime - mergeStartTime) / 1e6 + " ms");

}

}