**Divide and Conquer with Recursive Function**

public class DcR {

static int DAC\_Max(int a[], int index, int l)

{

int max;

if (index >= l - 2)

{

if (a[index] > a[index + 1])

return a[index];

else

return a[index + 1];

}

// Logic to find the Maximum element

// in the given array.

max = DAC\_Max(a, index + 1, l);

if (a[index] > max)

return a[index];

else

return max;

}

// Function to find the minimum no.

// in a given array.

static int DAC\_Min(int a[], int index, int l)

{

int min;

if (index >= l - 2)

{

if (a[index] < a[index + 1])

return a[index];

else

return a[index + 1];

}

// Logic to find the Minimum element

// in the given array.

min = DAC\_Min(a, index + 1, l);

if (a[index] < min)

return a[index];

else

return min;

}

// Driver Code

public static void main(String[] args)

{

// Defining the variables

int min, max;

// Initializing the array

int a[] = { 70, 250, 50, 80, 140, 12, 14 };

// Recursion - DAC\_Max function called

max = DAC\_Max(a, 0, 7);

// Recursion - DAC\_Max function called

min = DAC\_Min(a, 0, 7);

System.out.printf("The minimum number in " +

"a given array is : %d\n", min);

System.out.printf("The maximum number in " +

"a given array is : %d", max);

}

}

**OUTPUT:**

The minimum number in a given array is : 12

The maximum number in a given array is : 250BUILD SUCCESSFUL (total time: 0 seconds)

**Divide and Conquer with Non Recursive Function**

package ndc;

/\*\*

\*

\* @author Deepthi

\*/

import java.util.Arrays;

import java.util.Scanner;

import java.util.Stack;

public class NDC {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

int[] unsorted = {34, 32, 43, 12, 11, 32, 22, 21, 32};

System.out.println("Unsorted array : " + Arrays.toString(unsorted));

iterativeQsort(unsorted);

System.out.println("Sorted array : " + Arrays.toString(unsorted));

}

/\*

\* iterative implementation of quicksort sorting algorithm.

\*/

public static void iterativeQsort(int[] numbers) {

Stack stack = new Stack();

stack.push(0);

stack.push(numbers.length);

while (!stack.isEmpty()) {

int end = (int) stack.pop();

int start = (int) stack.pop();

if (end - start < 2) {

continue;

}

int p = start + ((end - start) / 2);

p = partition(numbers, p, start, end);

stack.push(p + 1);

stack.push(end);

stack.push(start);

stack.push(p);

}

}

/\*

\* Utility method to partition the array into smaller array, and

\* comparing numbers to rearrange them as per quicksort algorithm.

\*/

private static int partition(int[] input, int position, int start, int end) {

int l = start;

int h = end - 2;

int piv = input[position];

swap(input, position, end - 1);

while (l < h) {

if (input[l] < piv) {

l++;

} else if (input[h] >= piv) {

h--;

} else {

swap(input, l, h);

}

}

int idx = h;

if (input[h] < piv) {

idx++;

}

swap(input, end - 1, idx);

return idx;

}

/\*\*

\* Utility method to swap two numbers in given array

\*

\* @param arr - array on which swap will happen

\* @param i

\* @param j

\*/

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

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

**OUTPUT**

Unsorted array : [34, 32, 43, 12, 11, 32, 22, 21, 32]

Sorted array : [11, 12, 21, 22, 32, 32, 32, 34, 43]

BUILD SUCCESSFUL (total time: 0 seconds)

**STRASSEN’S MATRIX MULTIPLICATION**

public class GFG {

public int[][] multiply(int[][] A, int[][] B)

{

// Order of matrix

int n = A.length;

// Creating a 2D square matrix with size n

// n is input from the user

int[][] R = new int[n][n];

// Base case

// If there is only single element

if (n == 1)

// Returning the simple multiplication of

// two elements in matrices

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

// Matrix

else {

// Step 1: Dividing Matrix into parts

// by storing sub-parts to variables

int[][] A11 = new int[n / 2][n / 2];

int[][] A12 = new int[n / 2][n / 2];

int[][] A21 = new int[n / 2][n / 2];

int[][] A22 = new int[n / 2][n / 2];

int[][] B11 = new int[n / 2][n / 2];

int[][] B12 = new int[n / 2][n / 2];

int[][] B21 = new int[n / 2][n / 2];

int[][] B22 = new int[n / 2][n / 2];

// Step 2: Dividing matrix A into 4 halves

split(A, A11, 0, 0);

split(A, A12, 0, n / 2);

split(A, A21, n / 2, 0);

split(A, A22, n / 2, n / 2);

// Step 2: Dividing matrix B into 4 halves

split(B, B11, 0, 0);

split(B, B12, 0, n / 2);

split(B, B21, n / 2, 0);

split(B, B22, n / 2, n / 2);

// Using Formulas as described in algorithm

// M1:=(A1+A3)×(B1+B2)

int[][] M1

= multiply(add(A11, A22), add(B11, B22));

// M2:=(A2+A4)×(B3+B4)

int[][] M2 = multiply(add(A21, A22), B11);

// M3:=(A1−A4)×(B1+A4)

int[][] M3 = multiply(A11, sub(B12, B22));

// M4:=A1×(B2−B4)

int[][] M4 = multiply(A22, sub(B21, B11));

// M5:=(A3+A4)×(B1)

int[][] M5 = multiply(add(A11, A12), B22);

// M6:=(A1+A2)×(B4)

int[][] M6

= multiply(sub(A21, A11), add(B11, B12));

// M7:=A4×(B3−B1)

int[][] M7

= multiply(sub(A12, A22), add(B21, B22));

// P:=M2+M3−M6−M7

int[][] C11 = add(sub(add(M1, M4), M5), M7);

// Q:=M4+M6

int[][] C12 = add(M3, M5);

// R:=M5+M7

int[][] C21 = add(M2, M4);

// S:=M1−M3−M4−M5

int[][] C22 = add(sub(add(M1, M3), M2), M6);

// Step 3: Join 4 halves into one result matrix

join(C11, R, 0, 0);

join(C12, R, 0, n / 2);

join(C21, R, n / 2, 0);

join(C22, R, n / 2, n / 2);

}

// Step 4: Return result

return R;

}

// Method 2

// Function to subtract two matrices

public int[][] sub(int[][] A, int[][] B)

{

//

int n = A.length;

//

int[][] C = new int[n][n];

// Iterating over elements of 2D matrix

// using nested for loops

// Outer loop for rows

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

// Inner loop for columns

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

// Subtracting corresponding elements

// from matrices

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

// Returning the resultant matrix

return C;

}

// Method 3

// Function to add two matrices

public int[][] add(int[][] A, int[][] B)

{

//

int n = A.length;

// Creating a 2D square matrix

int[][] C = new int[n][n];

// Iterating over elements of 2D matrix

// using nested for loops

// Outer loop for rows

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

// Inner loop for columns

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

// Adding corresponding elements

// of matrices

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

// Returning the resultant matrix

return C;

}

// Method 4

// Function to split parent matrix

// into child matrices

public void split(int[][] P, int[][] C, int iB, int jB)

{

// Iterating over elements of 2D matrix

// using nested for loops

// Outer loop for rows

for (int i1 = 0, i2 = iB; i1 < C.length; i1++, i2++)

// Inner loop for columns

for (int j1 = 0, j2 = jB; j1 < C.length;

j1++, j2++)

C[i1][j1] = P[i2][j2];

}

// Method 5

// Function to join child matrices

// into (to) parent matrix

public void join(int[][] C, int[][] P, int iB, int jB)

{

// Iterating over elements of 2D matrix

// using nested for loops

// Outer loop for rows

for (int i1 = 0, i2 = iB; i1 < C.length; i1++, i2++)

// Inner loop for columns

for (int j1 = 0, j2 = jB; j1 < C.length;

j1++, j2++)

P[i2][j2] = C[i1][j1];

}

// Method 5

// Main driver method

public static void main(String[] args)

{

// Display message

System.out.println(

"Strassen Multiplication Algorithm Implementation For Matrix Multipication :\n");

// Create an object of Strassen class

// in he main function

GFG s = new GFG();

// Size of matrix

// Cosidering size as 4 in order to illustrate

int N = 4;

// Matrix A

// Custom input to matrix

int[][] A = { { 1, 2, 3, 4 },

{ 4, 3, 0, 1 },

{ 5, 6, 1, 1 },

{ 0, 2, 5, 6 } };

// Matrix B

// Custom input to matrix

int[][] B = { { 1, 0, 5, 1 },

{ 1, 2, 0, 2 },

{ 0, 3, 2, 3 },

{ 1, 2, 1, 2 } };

// Matrix C computations

// Matrix C calling method to get Result

int[][] C = s.multiply(A, B);

// Display message

System.out.println(

"\nProduct of matrices A and B : ");

// Iterating over elements of 2D matrix

// using nested for loops

// Outer loop for rows

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

// Inner loop for columns

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

// Printing elements of resultant matrix

// with whitespaces in between

System.out.print(C[i][j] + " ");

// New line once the all elements

// are printed for specific row

System.out.println();

}

}

}

**OUTPUT**

Strassen Multiplication Algorithm Implementation For Matrix Multipication :

Product of matrices A and B :

7 21 15 22

8 8 21 12

12 17 28 22

8 31 16 31

BUILD SUCCESSFUL (total time: 0 seconds)

**Greedy Method**

import java.util.\*;

import java.lang.\*;

import java.io.\*;

public class greedy {

public static void printMaxActivities(int s[], int f[], int n)

{

int i, j;

System.out.print("Following activities are selected : n");

// The first activity always gets selected

i = 0;

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

// Consider rest of the activities

for (j = 1; j < n; j++)

{

// If this activity has start time greater than or

// equal to the finish time of previously selected

// activity, then select it

if (s[j] >= f[i])

{

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

i = j;

}

}

}

// driver program to test above function

public static void main(String[] args)

{

int s[] = {1, 3, 0, 5, 8, 5};

int f[] = {2, 4, 6, 7, 9, 9};

int n = s.length;

printMaxActivities(s, f, n);

}

}

**OUTPUT:**

Following activities are selected : n0 1 3 4

**Dynamic programming**

public class NewClass {

static int getValue(int[] values, int rodLength) {

int[] subSolutions = new int[rodLength + 1];

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

int tmpMax = -1;

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

tmpMax = Math.max(tmpMax, values[j] + subSolutions[i - j - 1]);

subSolutions[i] = tmpMax;

}

return subSolutions[rodLength];

}

public static void main(String[] args) {

int[] values = new int[]{3, 7, 1, 3, 9};

int rodLength = values.length;

System.out.println("Max rod value: " + getValue(values, rodLength));

}

}

**OUTPUT:**

Max rod value: 17

**Shortest path problem**

import java.util.\*;

import java.lang.\*;

import java.io.\*;

public class ShortestPath {

static final int V = 9;

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

{

// Initialize min value

int min = Integer.MAX\_VALUE, min\_index = -1;

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

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

min = dist[v];

min\_index = v;

}

return min\_index;

}

// A utility function to print the constructed distance array

void printSolution(int dist[], int n)

{

System.out.println("Vertex Distance from Source");

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

System.out.println(i + " tt " + dist[i]);

}

// Function that implements Dijkstra's single source shortest path

// algorithm for a graph represented using adjacency matrix

// representation

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

{

int dist[] = new int[V]; // The output array. dist[i] will hold

// the shortest distance from src to i

// sptSet[i] will true if vertex i is included in shortest

// path tree or shortest distance from src to i is finalized

Boolean sptSet[] = new Boolean[V];

// Initialize all distances as INFINITE and stpSet[] as false

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

dist[i] = Integer.MAX\_VALUE;

sptSet[i] = false;

}

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

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

// Pick the minimum distance vertex from the set of vertices

// not yet processed. u is always equal to src in first

// iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the

// picked vertex.

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

// Update dist[v] only if is not in sptSet, there is an

// edge from u to v, and total weight of path from src to

// v through u is smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] != 0 &&

dist[u] != Integer.MAX\_VALUE && dist[u] + graph[u][v] < dist[v])

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

}

// print the constructed distance array

printSolution(dist, V);

}

// Driver method

public static void main(String[] args)

{

/\* Let us create the example graph discussed above \*/

int graph[][] = new int[][] { { 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 } };

ShortestPath t = new ShortestPath();

t.dijkstra(graph, 0);

}

}

**OUTPUT:**

Vertex Distance from Source

0 tt 0

1 tt 4

2 tt 12

3 tt 19

4 tt 21

5 tt 11

6 tt 9

7 tt 8

8 tt 14

**Travelling sales person problem**

public class TSP {

static int tsp(int[][] graph, boolean[] v,

int currPos, int n,

int count, int cost, int ans)

{

// If last node is reached and it has a link

// to the starting node i.e the source then

// keep the minimum value out of the total cost

// of traversal and "ans"

// Finally return to check for more possible values

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

{

ans = Math.min(ans, cost + graph[currPos][0]);

return ans;

}

// BACKTRACKING STEP

// Loop to traverse the adjacency list

// of currPos node and increasing the count

// by 1 and cost by graph[currPos,i] value

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

{

if (v[i] == false && graph[currPos][i] > 0)

{

// Mark as visited

v[i] = true;

ans = tsp(graph, v, i, n, count + 1,

cost + graph[currPos][i], ans);

// Mark ith node as unvisited

v[i] = false;

}

}

return ans;

}

// Driver code

public static void main(String[] args)

{

// n is the number of nodes i.e. V

int n = 4;

int[][] graph = {{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}};

// Boolean array to check if a node

// has been visited or not

boolean[] v = new boolean[n];

// Mark 0th node as visited

v[0] = true;

int ans = Integer.MAX\_VALUE;

// Find the minimum weight Hamiltonian Cycle

ans = tsp(graph, v, 0, n, 1, 0, ans);

// ans is the minimum weight Hamiltonian Cycle

System.out.println(ans);

}

}

**OUTPUT:**

80

BUILD SUCCESSFUL (total time: 0 seconds)

**Back tracking**

public class NQueenProblem {

final int N = 4;

/\* A utility function to print solution \*/

void printSolution(int board[][])

{

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

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

System.out.print(" " + board[i][j]

+ " ");

System.out.println();

}

}

/\* A utility function to check if a queen can

be placed on board[row][col]. Note that this

function is called when "col" queens are already

placeed in columns from 0 to col -1. So we need

to check only left side for attacking queens \*/

boolean isSafe(int board[][], int row, int col)

{

int i, j;

/\* Check this row on left side \*/

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

if (board[row][i] == 1)

return false;

/\* Check upper diagonal on left side \*/

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

if (board[i][j] == 1)

return false;

/\* Check lower diagonal on left side \*/

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j] == 1)

return false;

return true;

}

/\* A recursive utility function to solve N

Queen problem \*/

boolean solveNQUtil(int board[][], int col)

{

/\* base case: If all queens are placed

then return true \*/

if (col >= N)

return true;

/\* Consider this column and try placing

this queen in all rows one by one \*/

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

/\* Check if the queen can be placed on

board[i][col] \*/

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

/\* Place this queen in board[i][col] \*/

board[i][col] = 1;

/\* recur to place rest of the queens \*/

if (solveNQUtil(board, col + 1) == true)

return true;

/\* If placing queen in board[i][col]

doesn't lead to a solution then

remove queen from board[i][col] \*/

board[i][col] = 0; // BACKTRACK

}

}

/\* If the queen can not be placed in any row in

this colum col, then return false \*/

return false;

}

/\* This function solves the N Queen problem using

Backtracking. It mainly uses solveNQUtil () to

solve the problem. It returns false if queens

cannot be placed, otherwise, return true and

prints placement of queens in the form of 1s.

Please note that there may be more than one

solutions, this function prints one of the

feasible solutions.\*/

boolean solveNQ()

{

int board[][] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (solveNQUtil(board, 0) == false) {

System.out.print("Solution does not exist");

return false;

}

printSolution(board);

return true;

}

// driver program to test above function

public static void main(String args[])

{

NQueenProblem Queen = new NQueenProblem();

Queen.solveNQ();

}

}

**OUTPUT:**

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

BUILD SUCCESSFUL (total time: 0 seconds)

**Modular Arithmetic**

import java.io.\*;

public class arithmeic {

static int power(int x, int y, int p)

{

int res = 1; // Initialize result

x = x % p; // Update x if it is more than or

// equal to p

if (x == 0)

return 0; // In case x is divisible by p;

while (y > 0)

{

// If y is odd, multiply x with result

if ((y & 1) != 0)

res = (res \* x) % p;

// y must be even now

y = y >> 1; // y = y/2

x = (x \* x) % p;

}

return res;

}

// Driver Code

public static void main(String[] args)

{

int x = 2;

int y = 5;

int p = 13;

System.out.print("Power is " + power(x, y, p));

}

}

OUTPUT:

Power is 6

**Bin Packing**

import java.util.Scanner;

public class Bin\_Packing\_Algorithm {

public static void binPacking(int []a, int size, int n)

{

int binCount=1;

int s = size;

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

{

if(s - a[i] > 0)

{

s -= a[i];

continue;

}

else

{

binCount++;

s = size;

i--;

}

}

System.out.println("Number of bins required: "+binCount);

}

public static void main(String args[])

{

System.out.println("BIN - PACKING Algorithm");

System.out.println("Enter the number of items in Set: ");

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

System.out.println("Enter "+n+" items:");

int []a = new int[n];

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

a[i] = sc.nextInt();

System.out.println("Enter the bin size: ");

int size = sc.nextInt();

binPacking(a, size, n);

sc.close();

}

}

OUTPUT:

BIN - PACKING Algorithm

Enter the number of items in Set:

8

Enter 8 items:

4 5 8 3 4 5 1 6

Enter the bin size:

10

Number of bins required: 5