**Group A**

**Practical 1 DAA**

Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

**Without Recursion**

#include <iostream>

using namespace std;

int main() {

int n1=0,n2=1,n3,i,number;

cout<<"Enter the number of elements: ";

cin>>number;

cout<<n1<<" "<<n2<<" "; //printing 0 and 1

for(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed

{

n3=n1+n2;

cout<<n3<<" ";

n1=n2;

n2=n3;

}

return 0;

}

**With using Recursion:**

#include <iostream>

using namespace std;

int main() {

int n1=0,n2=1,n3,i,number;

cout<<"Enter the number of elements: ";

cin>>number;

cout<<n1<<" "<<n2<<" "; //printing 0 and 1

for(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed

{

n3=n1+n2;

cout<<n3<<" ";

n1=n2;

n2=n3;

}

return 0;

}

**Practical 2 DAA:**

#include <algorithm>

#include <iostream>

using namespace std;

// A structure to represent a job

struct Job {

char id; // Job Id

int dead; // Deadline of job

int profit; // Profit if job is over before or on

// deadline

};

// Comparator function for sorting jobs

bool comparison(Job a, Job b)

{

return (a.profit > b.profit);

}

// Returns maximum profit from jobs

void printJobScheduling(Job arr[], int n)

{

// Sort all jobs according to decreasing order of profit

sort(arr, arr + n, comparison);

int result[n]; // To store result (Sequence of jobs)

bool slot[n]; // To keep track of free time slots

// Initialize all slots to be free

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

slot[i] = false;

// Iterate through all given jobs

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

// Find a free slot for this job (Note that we start

// from the last possible slot)

for (int j = min(n, arr[i].dead) - 1; j >= 0; j--) {

// Free slot found

if (slot[j] == false) {

result[j] = i; // Add this job to result

slot[j] = true; // Make this slot occupied

break;

}

}

}

// Print the result

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

if (slot[i])

cout << arr[result[i]].id << " ";

}

// Driver's code

int main()

{

Job arr[] = { { 'a', 2, 100 },

{ 'b', 1, 19 },

{ 'c', 2, 27 },

{ 'd', 1, 25 },

{ 'e', 3, 15 } };

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

cout << "Following is maximum profit sequence of jobs "

"\n";

// Function call

printJobScheduling(arr, n);

return 0;

}

**Practical NO 3 DAA:**

**Write a program to solve a fractional Knapsack problem using a greedy method.**

#include <bits/stdc++.h>

using namespace std;

// Structure for an item which stores weight and

// corresponding value of Item

struct Item {

int profit, weight;

// Constructor

Item(int profit, int weight)

{

this->profit = profit;

this->weight = weight;

}

};

// Comparison function to sort Item

// according to profit/weight ratio

static bool cmp(struct Item a, struct Item b)

{

double r1 = (double)a.profit / (double)a.weight;

double r2 = (double)b.profit / (double)b.weight;

return r1 > r2;

}

// Main greedy function to solve problem

double fractionalKnapsack(int W, struct Item arr[], int N)

{

// Sorting Item on basis of ratio

sort(arr, arr + N, cmp);

double finalvalue = 0.0;

// Looping through all items

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

// If adding Item won't overflow,

// add it completely

if (arr[i].weight <= W) {

W -= arr[i].weight;

finalvalue += arr[i].profit;

}

// If we can't add current Item,

// add fractional part of it

else {

finalvalue

+= arr[i].profit

\* ((double)W / (double)arr[i].weight);

break;

}

}

// Returning final value

return finalvalue;

}

// Driver code

int main()

{

int W = 50;

Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };

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

// Function call

cout << fractionalKnapsack(W, arr, N);

return 0;

}

**Practical no 4:**

**Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.**

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**class** Item {

**float** weight;

**int** value;

Item(**float** weight, **int** value) {

**this**.weight = weight;

**this**.value = value;

}

}

**class** Node {

**int** level, profit, bound;

**float** weight;

Node(**int** level, **int** profit, **float** weight) {

**this**.level = level;

**this**.profit = profit;

**this**.weight = weight;

}

}

**public** **class** KnapsackBranchAndBound {

**static** Comparator<Item> *itemComparator* = (a, b) -> {

**double** ratio1 = (**double**) a.value / a.weight;

**double** ratio2 = (**double**) b.value / b.weight;

// Sorting in decreasing order of value per unit weight

**return** Double.*compare*(ratio2, ratio1);

};

**static** **int** bound(Node u, **int** n, **int** W, Item[] arr) {

**if** (u.weight >= W)

**return** 0;

**int** profitBound = u.profit;

**int** j = u.level + 1;

**float** totalWeight = u.weight;

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

totalWeight += arr[j].weight;

profitBound += arr[j].value;

j++;

}

**if** (j < n)

profitBound += (**int**) ((W - totalWeight) \* arr[j].value /

arr[j].weight);

**return** profitBound;

}

**static** **int** knapsack(**int** W, Item[] arr, **int** n) {

Arrays.*sort*(arr, *itemComparator*);

PriorityQueue<Node> priorityQueue =

**new** PriorityQueue<>((a, b) -> Integer.*compare*(b.bound,

a.bound));

Node u, v;

u = **new** Node(-1, 0, 0);

priorityQueue.offer(u);

**int** maxProfit = 0;

**while** (!priorityQueue.isEmpty()) {

u = priorityQueue.poll();

**if** (u.level == -1)

v = **new** Node(0, 0, 0);

**else** **if** (u.level == n - 1)

**continue**;

**else**

v = **new** Node(u.level + 1, u.profit, u.weight);

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

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

**if** (v.weight <= W && v.profit > maxProfit)

maxProfit = v.profit;

v.bound = *bound*(v, n, W, arr);

**if** (v.bound > maxProfit)

priorityQueue.offer(v);

v = **new** Node(u.level + 1, u.profit, u.weight);

v.bound = *bound*(v, n, W, arr);

**if** (v.bound > maxProfit)

priorityQueue.offer(v);

}

**return** maxProfit;

}

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

**int** W = 10;

Item[] arr = {

**new** Item(2, 40),

**new** Item(3.14f, 50),

**new** Item(1.98f, 100),

**new** Item(5, 95),

**new** Item(3, 30)

};

**int** n = arr.length;

**int** maxProfit = *knapsack*(W, arr, n);

System.***out***.println("Maximum possible profit = " + maxProfit);

}

}

**C++ solution**

**#include <iostream>**

**using namespace std;**

**int max(int x, int y) {**

**return (x > y) ? x : y;**

**}**

**int knapSack(int W, int w[], int v[], int n) {**

**int i, wt;**

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

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

**for (wt = 0; wt <= W; wt++) {**

**if (i == 0 || wt == 0)**

**K[i][wt] = 0;**

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

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

**else**

**K[i][wt] = K[i - 1][wt];**

**}**

**}**

**return K[n][W];**

**}**

**int main() {**

**cout << "Enter the number of items in a Knapsack:";**

**int n, W;**

**cin >> n;**

**int v[n], w[n];**

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

**cout << "Enter value and weight for item " << i << ":";**

**cin >> v[i];**

**cin >> w[i];**

**}**

**cout << "Enter the capacity of knapsack";**

**cin >> W;**

**cout << knapSack(W, w, v, n);**

**return 0;**

**}**

**Practical no 5:**

Design 8-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final 8-queens matrix.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**const int N = 8;**

**bool isSafe(vector<vector<int> >& board, int row, int col)**

**{**

**for (int x = 0; x < col; x++)**

**if (board[row][x] == 1)**

**return false;**

**for (int x = row, y = col; x >= 0 && y >= 0; x--, y--)**

**if (board[x][y] == 1)**

**return false;**

**for (int x = row, y = col; x < N && y >= 0; x++, y--)**

**if (board[x][y] == 1)**

**return false;**

**return true;**

**}**

**bool solveNQueens(vector<vector<int> >& board, int col)**

**{**

**if (col == N) {**

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

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

**cout << board[i][j] << " ";**

**cout << endl;**

**}**

**cout << endl;**

**return true;**

**}**

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

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

**board[i][col] = 1;**

**if (solveNQueens(board, col + 1))**

**return true;**

**board[i][col] = 0;**

**}**

**}**

**return false;**

**}**

**int main()**

**{**

**vector<vector<int> > board(N, vector<int>(N, 0));**

**if (!solveNQueens(board, 0))**

**cout << "No solution found";**

**return 0;**

**}**