**KNAPSACK 0/1**

Project Description:

Implementation of Knapsack 0/1 problem, using dynamic method.

Members:

Ipshita Ghosh

Komal Prakash

Diksha Lath

Shruti Pawan

Kritika

Aishwarya Raj

INDEX

Introduction

Approaches

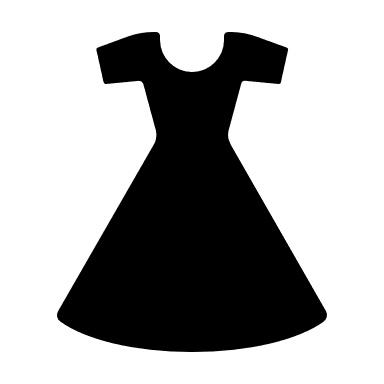
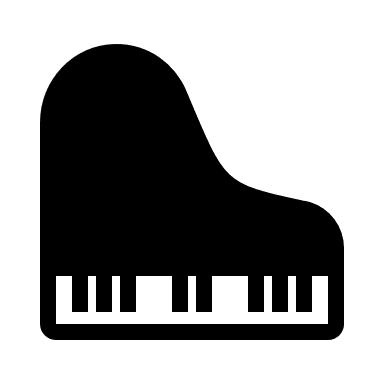
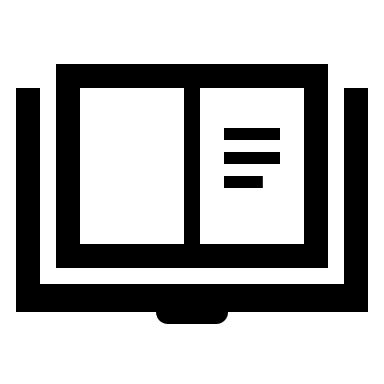
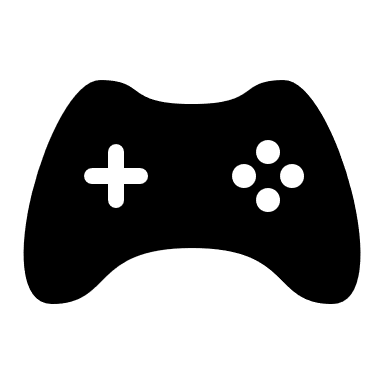
Algorithm

Code

INTRODUCTION

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).

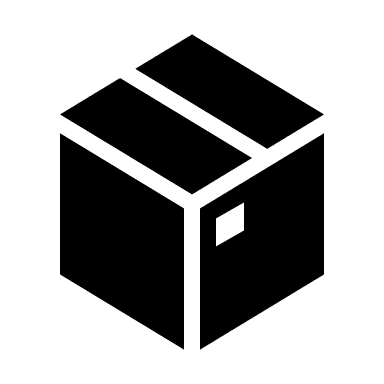
Real Life Example:-

Object 1 Object 2 Object 3 Object 4

W=4 W=10 W=7 W=6

P=3 P=9 P=5 P=8



Total Capacity = 20

Given a Box whose capacity is 20, and a number of given objects with weight and profit. The problem is to select the objects in such a way that it gives maximum profit.

As we know that the objects cannot be divided and have to be carried as a whole, i.e. either 0 or 1.

This kind of real life problems can be solved using Knapsack 0/1 problem.

APPROACHES

The knapsack 0/1 problem can be solved by two approaches

1. Recursive Approach
2. Dynamic Approach

Recursive Approach

For each item, there are two possibilities –

1. We include current item in knapSack and recurse for remaining items with decreased capacity of Knapsack. If the capacity becomes negative, do not recuse or return -INFINITY.

2. We exclude current item from knapSack and recurse for remaining items.

Finally, we return maximum value we get by including or excluding current item. The base case of the recursion would be when no items are left or capacity becomes 0.

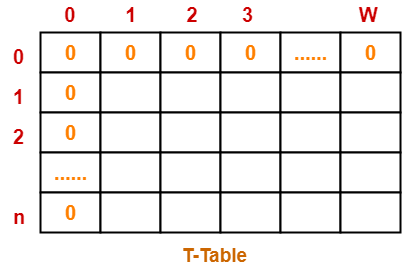
Dynamic Approach

Consider we are given-

* A knapsack of weight capacity ‘w’
* ‘n’ number of items each having some weight and value

**Step-01:**

* We draw a table say ‘T’ with (n+1) number of rows and (w+1) number of columns.
* We fill all the boxes of 0th row and 0th column with zeroes as shown-



**Step-02:**

* Start filling the table row wise top to bottom from left to right.
* We use the following formula-

**T (i , j) = max { T ( i-1 , j ) , valuei + T( i-1 , j – weighti) }**

T(i , j) = maximum value of the selected items if we can take items 1 to i and we have weight restrictions of j.

**Step-03:**

After filling the table completely, **value of the last cell** represents the maximum possible value that be put in the knapsack.

**Step-04:**

To identify the items that must be put in the knapsack to obtain the maximum profit,

* Considering the last column of the table, start scanning the entries from bottom to top.
* If an entry is encountered whose value is not same as the value which is stored in the entry immediately above it, then mark the row label of that entry.
* After scanning all the entries, the marked labels represent the items that must be put in the knapsack.

ALGORITHM

Recursive Algorithm

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

{

*// Base Case*

   if (n == 0 || W == 0)

       return 0;

*// If weight of the nth item is more than Knapsack capacity*

*// W, Then*

*// this item cannot be included in the optimal solution*

   if (wt[n-1] > W)

       return knapSack(W, wt, val, n-1);

*// Return the maximum of two cases:*

*// (1) nth item included*

*// (2) not included*

   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),

                    knapSack(W, wt, val, n-1)

                  );

}

Time Complexity

The time complexity of this Algorithm is O(2n).

This is because this Algorithm computes the same subproblem twice.

Dynamic Algorithm

*// Input:*

2 *// Values (stored in array v)*

3 *// Weights (stored in array w)*

4 *// Number of distinct items (n)*

5 *// Knapsack capacity (W)*

6 *// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.*

7

8 **for** j from 0 to W **do**:

9 m[0, j] := 0

10

11 **for** i from 1 to n **do**:

12 **for** j from 0 to W **do**:

13 **if** w[i] > j then:

14 m[i, j] := m[i-1, j]

15 **else**:

16 m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])

Time Complexity and Space Complexity

This solution will therefore run in {\displaystyle O(nW)}*O(nW)* time and {\displaystyle O(nW)}*O(nW)* space.

CODE

/\*\*

\* File\_Name : knapp.java

\* Updated : 10 April 2019

\* Contributor : Ipshita

\* Intent : To implement knapsack 0/1

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Sample Input :

\*

\* Enter the number of object : 5

\* Enter the maximum capacity of the sack : 5

\* Enter the profits of the 5 articles : 1 2 3 4 5

\* Enter the weights of the 5 articles : 1 2 3 4 5

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Sample Output :

\*

\* The object with their weight and profit:-

\*

\* objects 1 2 3 4 5

\* weight 1 2 3 4 5

\* profit 1 2 3 4 5

\*

\* In tabular form:-

\*

\* 0 0 0 0 0 0

\* 0 1 1 1 1 1

\* 0 1 2 3 3 3

\* 0 1 2 3 4 5

\* 0 1 2 3 4 5

\* 0 1 2 3 4 5

\*

\* The objects to be included are:-

\*

\* 3

\* 2

\* 1

\*

\* Maximum profit is 6

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

**package** knapsackDyn;

**import** java.util.Scanner;

**public** **class** knapp

{

//max function

**public** **static** **int** max(**int** a, **int** b)

{

**if** (a>b)

**return** a;

**else**

**return** b;

}

//Main function

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

{

**int** nObj,cap;

// nObj stores the number of Objects and cap stores the capacity of the sack

Scanner input = **new** Scanner(System.***in***);

System.***out***.println("Enter the number of object");

nObj = input.nextInt();

System.***out***.println("Enter the maximum capacity of the sack");

cap = input.nextInt();

**int** p[] = **new** **int**[nObj+1];

**int** wt[] = **new** **int**[nObj+1];

**int** k[][] = **new** **int**[nObj+1][cap+1];

**int** n = nObj, m = cap;

**int** w, i;

System.***out***.println("Enter the profits of the "+ nObj + " articles" );

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

p[i]= input.nextInt();

System.***out***.println("Enter the weights of the "+ nObj + " articles" );

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

wt[i]= input.nextInt();

//using formula

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

{

**for** (w = 0; w <= m; w++)

{

**if** (i != 0 && w != 0)

{

**if** (wt[i] <= w)

k[i][w] = *max*(p[i] + k[i - 1][w - wt[i]], k[i - 1][w]);

**else**

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

}

}

}

//formating

System.***out***.println("\nThe object with their weight and profit:-\n");

System.***out***.print("objects ");

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

System.***out***.print(i + "\t");

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

System.***out***.print("weight ");

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

System.***out***.print(wt[i] + "\t");

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

System.***out***.print("profit ");

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

System.***out***.print(p[i] + "\t");

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

//showing in tabular form

System.***out***.println("\nIn tabular form:- \n");

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

{

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

System.***out***.print(k[i][j] + "\t");

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

}

// checking objects to be included

i = n;

**int** j = cap;

**int** pr = 0;

System.***out***.println("\nThe objects to be included are:-\n");

**while**(i>0 && j>0)

{

**if**(k[i][j] == k[i-1][j])

{

i--;

}

**else**

{

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

pr = pr + p[i];

i--;

j= j-wt[i];

}

}

System.***out***.println("\nMaximum profit is " + pr);

input.close();

}

}

*The source code can be found at* github.com/ipshitag/knapsack0-1

BIBLIOGRAPHY

DyClassrooms

GeeksForGeeks

Cse.nlu.edu

Wikipedia