**THE MULTI-DIMENTIONAL KNAPSACK**

**A PROJECT REPORT**

**Group members:**

|  |  |
| --- | --- |
| **1. Shivam Sinha** | **:19BCE0591** |
| **2. Aarush Bhat** | **:19BCE0564** |
| **3. Aditya Beri** | **:19BCE0312** |
| **4. Sachin Shukla** | **:17BEC0170** |

# DATA STRUCTURES AND ALGORITHMS - CSE2003

**SENDHIL KUMAR K.S**



## ABSTRACT

**───────────────────────────────────────**



This project is taken up to show that dynamic programming is a useful technique of solving certain kind of problems. The multidimensional knapsack problem can be solved with a dynamic programming approach. When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and reuse them as necessary 

* (memorization).



* Dynamic programming has algorithm that finds solutions to sub problems and stores them in memory for later use. It is more efficient than

“bruteforce methods”, which solve the same sub problems over and over again.

## INTRODUCTION

**──────────────────────────────────────────────**

**Knapsack problem**-Given some items, pack the knapsack to get the maximumtotal value. Each item has some weight, volume and some value. Total weight that we can carry is no more than some fixed number W and volume V. So we must consider weights of items, volume of the items as well as their value. How to pack the knapsack to achieve maximum total value of packed items?

There are two versions of this problem. We will try to solve the multidimensional knapsack problem using dynamic programming where indivisible items are considered i.e. you either take an item or not.

The Multidimensional knapsack problem (MKP) has varied applications in various fields, for example economy: Consider a set of projects (variables) (*j* = 1*,* 2*,* 3*..., n*) and a set of resources (constraints) (*i* = 1*,* 2*,* 3*... m*). Each project has assigned a profit *cj* and resource consumption values *aij*. The problem is to find a subset of all projects leading to the highest possible profit and not exceeding given resource limits *bi* and the volume *vi.*

The MKP is a well-known NP-Hard combinatorial optimization problem

[73] which can be formulated as follows Maximize



Subject to the constraints

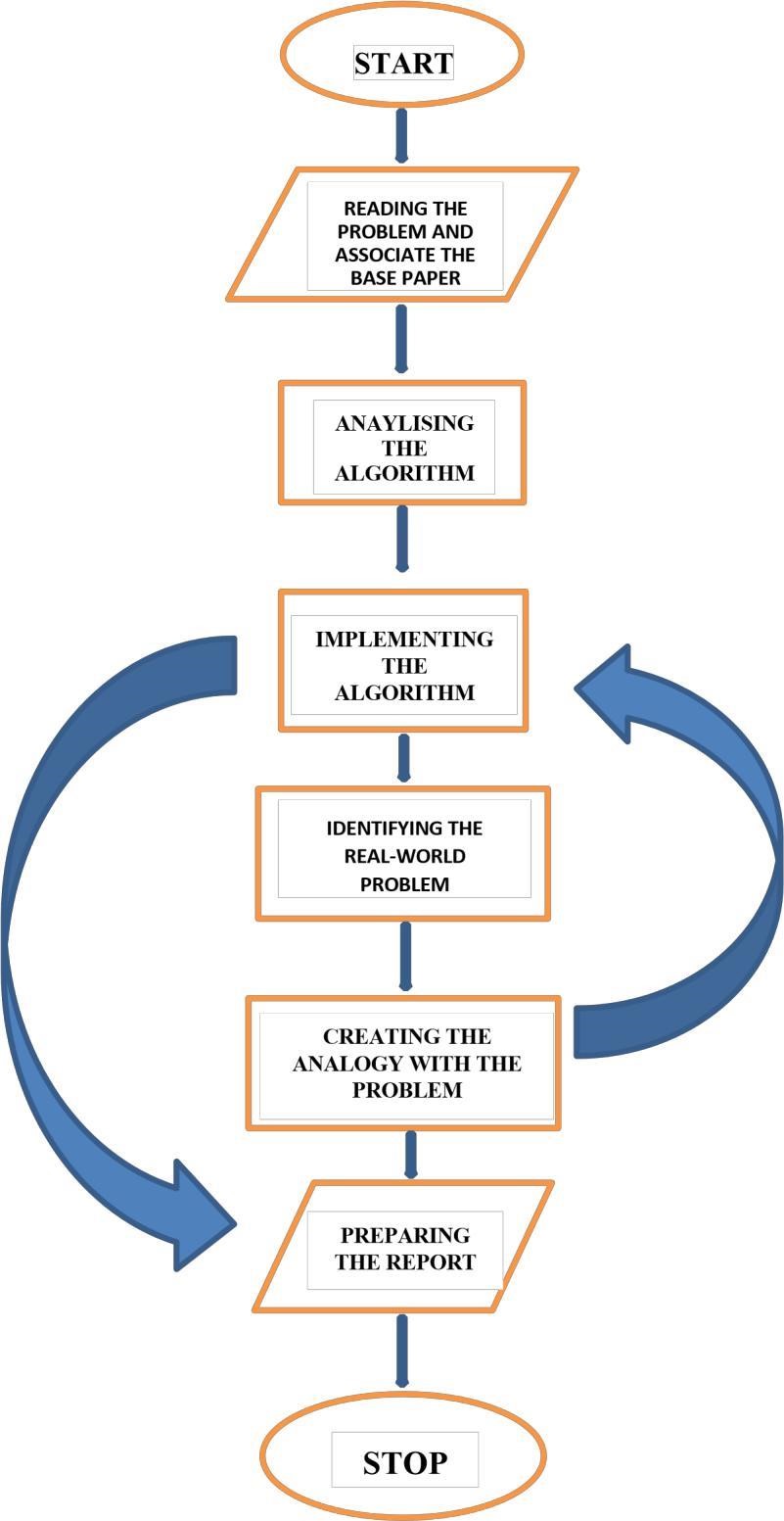
  

The objective function *Z* = *f* (*x*1*, x*2*...xn*) should be maximized subject to the constraints given by (2.2). Here in a MKP, it is necessary that *aij* are positive. This necessary condition paves way for better heuristics to obtain optimal or near optimal solutions.

The popularity of knapsack problems stems from the fact that it has attracted researchers from both camps: the theoreticians as wells as the practicians enjoy the fact that these simple structured problems can be used as sub problems to solve more complicated ones. Practicians on the other hand, enjoy the fact that these problems can model many industrial opportunities such as cutting stock, cargo loading, and processor allocation in distributed systems.

The special case of MKP with *m* = 1 is the classical knapsack problem (01KP), whose usual statement is the following. Given a knapsack of capacity b and n objects, each being associated a profit a volume occupation, one wants to select k (*k <*= *n* and k not fixed ) objects such that the total profit is maximized and the capacity of the knapsack is not exceeded. It is well known that 01KP is strongly NPHard because there are polynomial approximation algorithms to solve it. This is not the case for the general MKP. Various algorithms to obtain exact solutions of such problems were designed and well documented in literature.

### FLOWCHART



## ABOUT METHODOLOGY

**───────────────────────────────────────**

o **DYNAMIC PROGRAMMING**

1. Dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler sub problems, solving each of those sub problems just once, and storing their solutions – ideally, using a memory-based data structure. The next time the same sub problem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.

1. Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure.

1. Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved sub problems and will combine their solutions to give the best solution for the given problem.

## ALGORITHM

**───────────────────────────────────────**

Knapsack problem could be implemented using 3 different algorithms

1. **Greedy algorithm**

// Purpose: To find the maximum profit of the Knapsack using greedy technique.

// Input: M is the capacity of the Knapsack.

* 1. ItemList ← ItemList.SortDecreasingBy( ci /wi )

2. i= 0 /\* S has all items sorted by ci /wi\*/

* 1. ∀j, rj= Wj **j in each round**:

**For the processor p**

* 1. Send <rj> to S

* 1. Receive <c

I,wi> from S

* 1. rj ← rj – wi **For the source S**:

* 1. while i ≤ length(ItemList) do

* 1. Receive <rj> from all pj
  2. l = (pj, rj ).SortDecreasingBy(rj ) /\**Sort by remaining capacity\*/*
  3. for p

 j in l do if wi≤ rj then

Send item ItemList[i] to pj else 14. Send <⊥> to pj /\**Send next item \*/*

15. end if

16.

i ← i + 1

* 1. end for
  2. end while

//Output: MP, the maximum profit

1. **Dynamic programming**

// Purpose: To find the maximum profit of the Knapsack using dynamic programming. // Input: M is the capacity of the Knapsack. N is the number of objects.

* 1. Initialize the solution by assigning 0 to all xj

* 1. Intercept matrix D djj = bi/aij , if aij > 0, djj = M, a large value ; otherwise.

* 1. Identify 0 value variables( redundant): If any column has<1 entry in D, then the corresponding variable identified as a redundant variable

* 1. Dominant variable: Identify the smallest element (dominant variable) in each column of

D.

* 1. Multiply the smallest elements with the corresponding cost coefficients. If the product is Maximum in kth column, then set xk = 1 and update the objective function value

f(x1, x2,

..., xn).

* 1. Update the constraint matrix: bi = bi – aij for all i and set aik = 0 for all i 7. If aij = 0 for all i and j, then display the solution. Otherwise go to step 4. //Output: The optimal solution.

1. **Branch and bound**

// Purpose: To find the maximum profit of the Knapsack using branch and bound technique.

// Input: M is the capacity of the Knapsack. N is the number of items.

* 1. search MKP(bins, items, sumProfit)
  2. if bins==∅ or items == ∅

* 1. if sumProfit > bestProfit then bestProfit = sumProfit; return

* 1. ri = reduce(bins,items) /\* Pisinger’s R2 reduction \*/
  2. if ri = ∅

* 1. search MKP(bins, items \ ri, sumProfit)

* 1. return

* 1. upperBound = compute upper bound(items,bins)

* 1. if (sumProfit + upperBound ≤ bestProfit

* 1. return /\* upper-bound based pruning using SMKP bound \*/

* 1. if (validate upper bound(upperBound))

* 1. bestProfit = upperbound

* 1. return /\* bound-and-bound \*/

* 1. bin = choose bin(bins)

 undominatedAssignments = generate undominated(items,capacity(bin))

foreach A ∈ sort assignments(undominatedAssignments)

* 1. if not(symmetric(A))

* 1. assign A to bin
  2. search MKP(bins \ bin, items \ A, sumProfit+P j∈A pj) //Output: The optimal solution.

## Dynamic Programming Algorithm

### ─────────────────────────────────────────────────────

// WL = backpack weight limit

// SL = backpack size limit

for w = 0, 1, ..., WL // weight

for s = 0, 1, ..., SL // size

A[0, w, s] = INFINITY

A[0, 0, 0] = 0 for i = 1, 2, ..., n

// items for

w = 0, 1, ..., WL // weight

for s = 0, 1, ..., SL // size

// wi, si and ci are the weight, size and cost of the item respectively

// A[i-1, w-wi, s-si] + ci = 0 when wi > w or si > s

A[i, w, s] = min(A[i-1, w, s], A[i-1, w-wi, s-si] + ci)

## PROBLEM IMPLEMENTATION

### ─────────────────────────────────────────────────────

**PROBLEM STATEMENT:**

A traveler gets diverted and has to make an unscheduled stop in what turns out to be Shangri La. Opting to leave, he is allowed to take as much as he likes of the following items, so long as it will fit in his knapsack, and he can carry it.

He knows that he can carry no more than given 'weights' in total; and that the capacity of his knapsack of given 'cubic lengths'.

Looking just above the bar codes on the items he finds their weights and volumes. He digs out his recent copy of a financial paper and gets the value of each item.

How many of each item does he take to maximize the value of items he is carrying away with him

**C-CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**typedefstruct {**

**char \*name; double**

**value; double weight;**

**double volume;**

**} itemcontraints;**

**itemcontraints items[] = {**

**{"panacea", 3000.0, 3, 25},**

**{"ichor", 1800.0, 1, 15},**

**{"gold", 2500.0, 2, 20},**

**{"silver", 2000.0, 4, 10},**

**{"Ruby", 15000.0, 3, 20},**

**};**

**int n = sizeof (items) / sizeof (itemcontraints); int \*count; int \*best; doublebest\_value; void knapsack (inti, double value, double weight, double volume) { int j, m1, m2, m;**

**if (i == n) {**

**if (value >best\_value) { best\_value**

= **value; for (j = 0; j < n; j++) { best[j] =**

**count[j];**

**}**

**}**

**return;**

**}**

**m1 = weight / items[i].weight;**

**m2 = volume / items[i].volume; m**

**= m1 <m2 ?m1 : m2;**

**for (count[i] = m; count[i] >= 0; count[i]--) { knapsack(**

**i + 1,**

**value + count[i] \* items[i].value, weight -**

**count[i] \* items[i].weight, volume -**

**count[i] \* items[i].volume**

**);**

**}**

**}**

**int main () { count = malloc(n \* sizeof (int)); best =**

**malloc(n \* sizeof (int)); best\_value = 0; floatwt,vol;**

**printf("enter the max weight that can be carried"); scanf("%f",&wt); printf("enter the max volume of bag"); scanf("%f",&vol); knapsack(0, 0.0, wt, vol);**

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

**{**

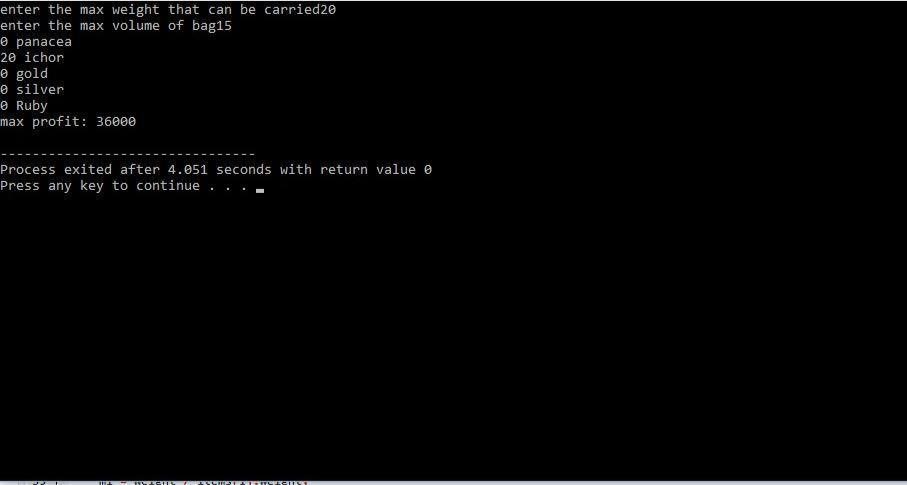
**printf("%d %s\n", best[i], items[i].name);**

**}**

**printf("best value: %.0f\n", best\_value); free(count); free(best); return 0;**

**}**

**OUTPUT**



## REAL WORLD APPLICATIONS

**1. Agriculture problem**

**Analogy with knapsack problem**

|  |  |
| --- | --- |
| Knapsack problem | Agriculture problem |
|
| **Z=maximise the profit** | Z=maximise the yield |
| **Ci=value** | Ci= yield (km/hectare) |
| **Bi=weight** | Bi= production(in million tonnes) |
| **Vi= volume** | Vi= area of the field(in million hectares) |

**Problem:**

**#include<stdio.h>**

**#include<stdlib.h> typedef**

**struct {**

**char \*name; double**

**yield; double**

**production; double area;**

**} itemcontraints;**

### itemcontraints items[] =

**{**

**{"rice", 1808, 72.3, 40.0},**

**{"wheat", 2577 , 68.8, 26.7},**

**{"maize", 1637, 9.3, 5.7},**

**{"bajra", 779, 7.5, 9.6},**

**{"Coarse cereals", 1071,25.7, 24.0},**

**};**

**int n = sizeof (items) / sizeof**

**(****itemcontraints); int \*count; int \*best;**

**double best\_yield; void knapsack (int i, double yield, double production,**

**double area) { int j, m1, m2, m;**

**if (i == n) { if (yield**

**>best\_yield) {**

**best\_yield = yield; for (j**

= **0; j < n; j++) { best[j]**

= **count[j];**

**}**

**}**

**return;**

**}**

**m1 = production / items[i].production;**

**m2 = area / items[i].area;** **m = m1**

**<m2 ?m1 : m2;**

**for (count[i] = m; count[i] >= 0; count[i]--) { knapsack(**

**i + 1,**

**yield + count[i] \* items[i].yield, production -**

**count[i] \* items[i].production, area -**

**count[i] \* items[i].area**

**);**

**}**

**}**

**int main () { count = malloc(n**

**\* sizeof (int)); best = malloc(n \***

**sizeof (int)); best\_yield = 0;**

**float pr,ar; printf("enter the**

**max production ");**

**scanf("%f",&pr);**

**printf("enter the max area of**

**the field required ");**

**scanf("%f",&ar);**

**knapsack(0, 0.0, pr, ar);**

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

**{**

**printf("%d %s\n", best[i], items[i].name);**

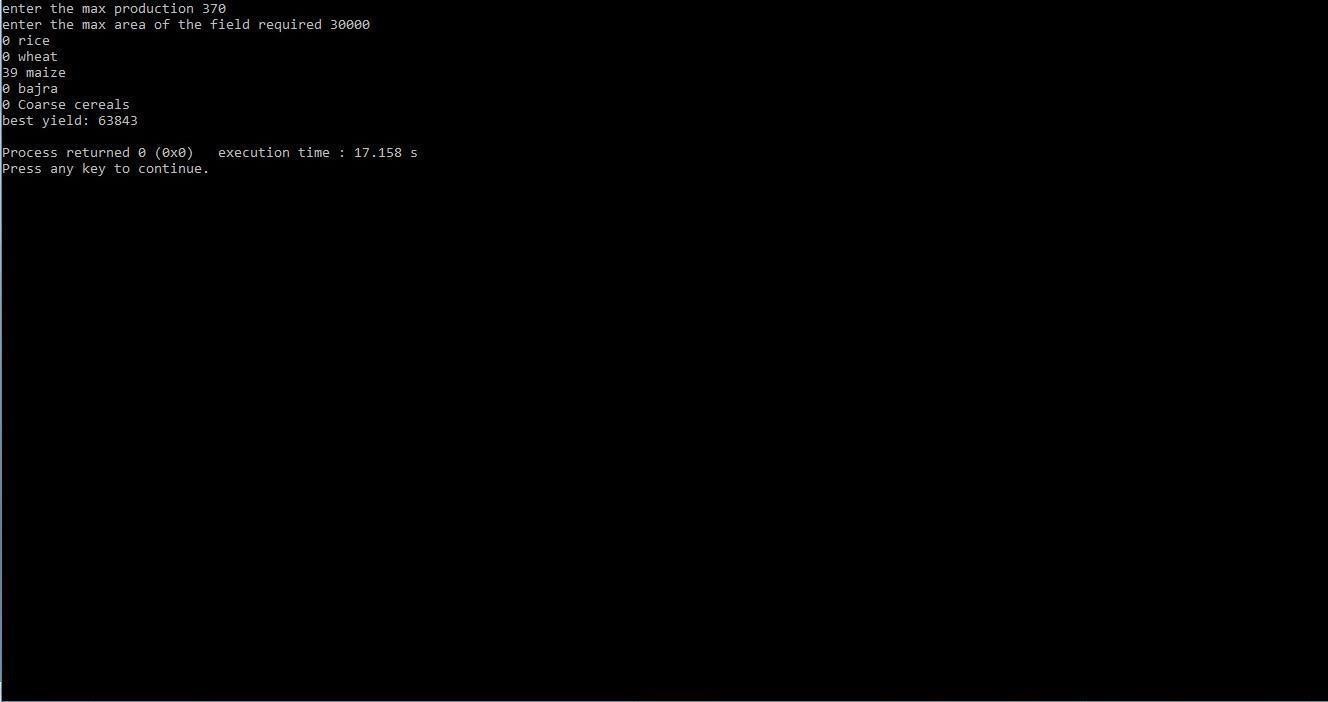
**}**

**printf("best yield: %.0f\n", best\_yield); free(count);**

**free(best); return 0;**

**}**

**OUTPUT:**



**2. Diet problem**

**Analogy with knapsack problem**

|  |  |
| --- | --- |
| Knapsack problem | Diet problem |
| **Z=maximise the profit** | Z=maximise the calorie intake |
| **Ci=value** | Ci=calories in %DV(daily values) |
| **Bi=weight** | Bi= carbohydrates in %DV |
| **Vi= volume** | Vi= dietary fiber in %DV |

**CODE:**

**#include<stdio.h>**

**#include<stdlib.h>**

**typedef struct { char**

**\*name; double calorie; double**

**carbohydrates; double** **fibers; }**

**itemcontraints;**

### itemcontraints items[] = { {"asparagus", 25.0, 1, 8},

**{"carrot", 30.0 , 2, 8},**

**{"sweet potato", 100, 8, 16},**

**{"mushrooms", 20, 1, 4},**

**{"onion", 45, 4, 12},**

**};**

**int n = sizeof (items) / sizeof** **(itemcontraints); int \*count; int \*best;**

**double best\_calorie;**

### void knapsack (int i, double calorie, double carbohydrates, double fibers)

**{ int j, m1, m2, m; if (i == n) { if (calorie >best\_calorie) { best\_calorie =** **calorie; for (j = 0; j < n; j++) {** **best[j] = count[j];**

**}**

**}**

**return;**

**}**

**m1 = carbohydrates / items[i].carbohydrates;**

**m2 = fibers / items[i].fibers; m = m1 <m2 ?m1 : m2; for (count[i] =**

**m; count[i] >= 0; count[i]--) {**

**knapsack( i**

**+ 1,**

**calorie + count[i] \* items[i].calorie, carbohydrates - count[i] \* items[i].carbohydrates,**

**fibers - count[i] \* items[i].fibers**

**);**

**} } int main () { count = malloc(n**

\* **sizeof (int)); best = malloc(n \* sizeof (int)); best\_calorie = 0; float carbo,fb;**

**printf("carbohydrates**

**content: "); scanf("%f",&carbo); printf("dietary fibers content :"); scanf("%f",&fb); knapsack(0, 0.0,**

**carbo, fb);**

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

**printf("%d %s\n", best[i], items[i].name);**

**}**

**printf("maximum calories: %.0f\n", best\_calorie);** **free(count); free(best);** **return 0;**

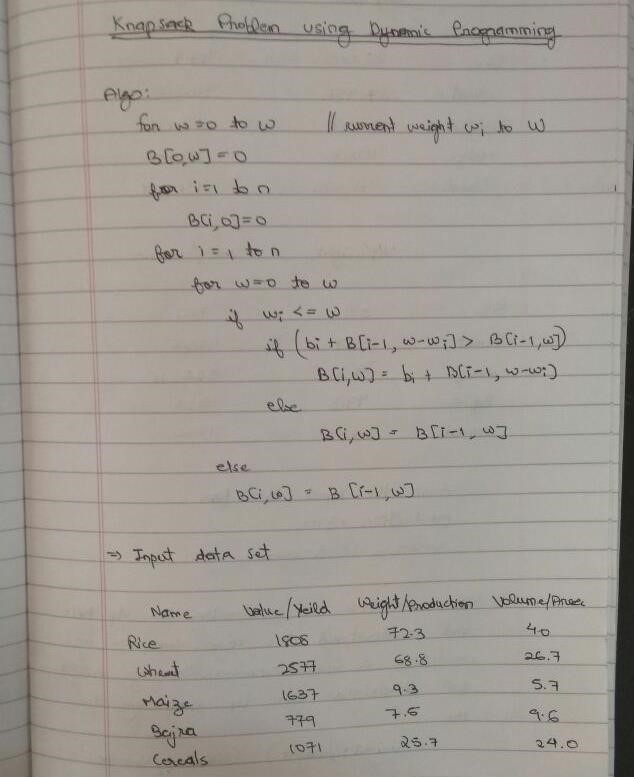
**}**

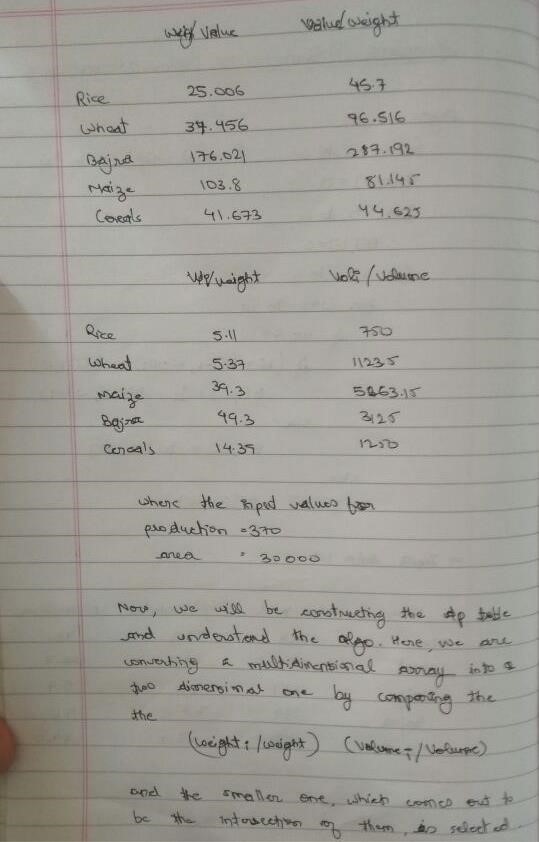
**OUTPUT:**

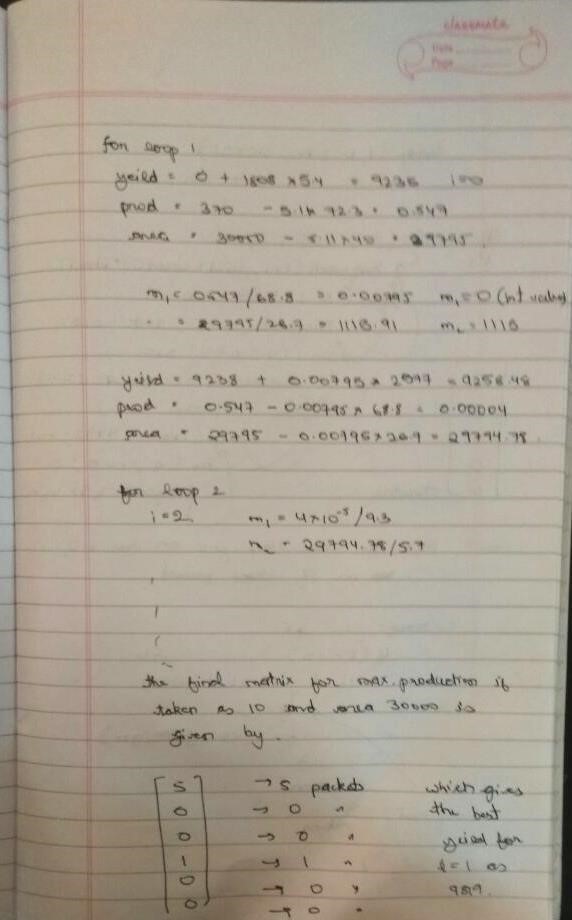


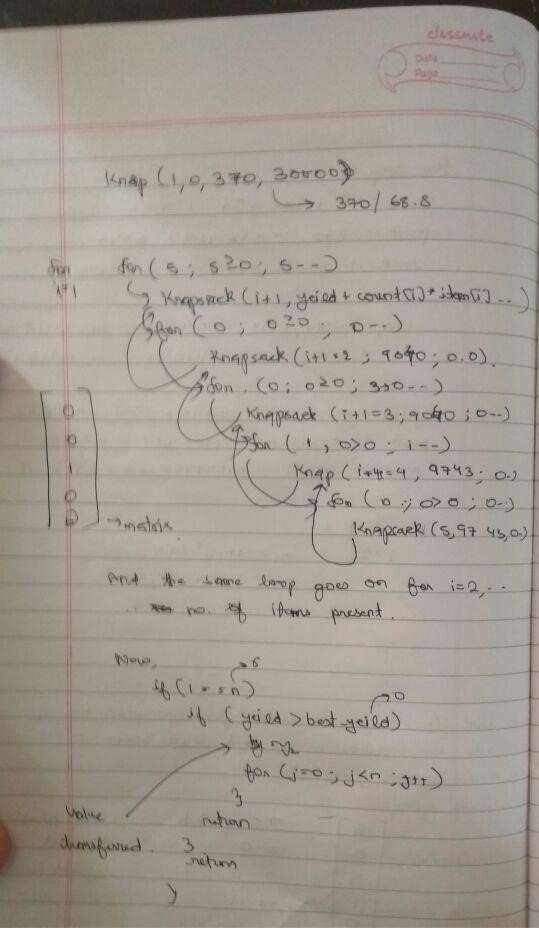
## EXECUTION OF PROGRAM(DRY RUN)

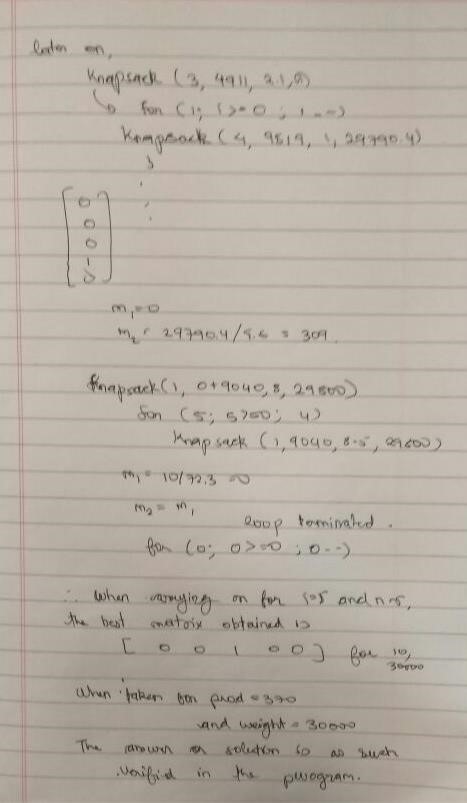
**──────────────────────────────────────────────**











## ANALYSIS

**──────────────────────────────────────────────**

Aspects such as the capacity and the number of items of the knapsack play a vital role in the computation of the number of basic operations and the total memory consumed by the algorithms used. Hence, the analysis of the above algorithms have been made by varying the number of inputs and the capacity of the knapsack.

**1. No. of computations**

The analysis of the number of computations is done by generating the number of basic operations made by the algorithms by varying the number of items, and using random values for the profit, volume and weight of each item included. The capacity of the knapsack is kept constant at each case. The results obtained are tabulated and are presented below:

Varying the number of Items and having fixed Capacity = 10

Table 1

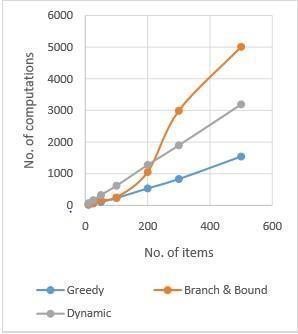
**No. of items**  **Greedy**  **Branch & bound**  **Dynamic**

|  |  |  |  |
| --- | --- | --- | --- |
| **10** | 22 | 28 | 73 |
| **25** | 65 | 77 | 168 |
| **50** | 110 | 168 | 329 |
| **100** | 235 | 253 | 623 |
| **200** | 538 | 1057 | 1279 |
| **300** | 835 | 2993 | 1899 |
| **500** | 1545 | 5012 | 3192 |

Table. 1 suggests that the number of computations of the three techniques increase

with different rates with the increase in the number of items.

The graph of Table. 1 is plotted below, with the No. of items on the X-axis and the No. of computations on the Y-axis:



Graph 1

The above graph shows that the Branch & Bound technique has a non-linear rate of increase in the No. of computations. For small capacities, Dynamic Programming technique has the better efficiency, in terms of the number of basic operations.

Varying the number of Items and having fixed Capacity = 100

Table. 2

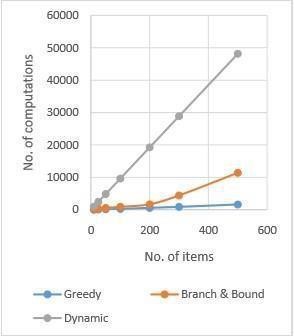
**No. of items**  **Greedy**  **Branch & bound**  **Dynamic**

|  |  |  |  |
| --- | --- | --- | --- |
| **10** | 22 | 78 | 973 |
| **25** | 82 | 342 | 2418 |
| **50** | 134 | 576 | 4829 |
| **100** | 266 | 913 | 9623 |
| **200** | 582 | 1687 | 19279 |
| **300** | 888 | 4425 | 28899 |
| **500** | 1614 | 11417 | 48192 |

Table.2 suggests that the number of computations of the Dynamic Programming technique increases with a high rate with increase in the number of items, for higher capacities, but it remains constant in the Greedy, Branch & Bound techniques.

This demonstrates the time complexity of the latter two techniques, which are independent of the capacity of the knapsack.

The graph of Table. 2 is plotted below, with the No. of items on the X-axis and the No. of computations on the Y-axis:



Graph. 2 3.2

**2. Memory required**

The analysis of the memory required for the algorithms is made by varying the total capacity of the knapsack, and the total number of items available. The total memory consumption of each of the algorithms is computed in various cases, and are presented below:

Varying the number of Items and having fixed Capacity = 10

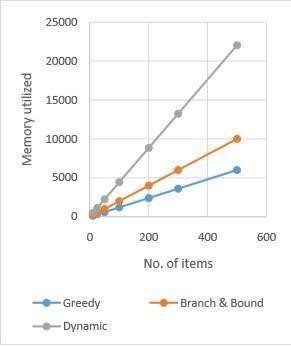
Table 3

***No. of items* Greedy**  **Branch & bound**  **Dynamic**

|  |  |  |  |
| --- | --- | --- | --- |
| *10*  *25*  *50*  *100*  *200*  *300*  *500* | 120 | 200 | 480 |
| 300 | 500 | 1140 |
| 600 | 1000 | 2240 |
| 1200 | 2000 | 4440 |
| 2400 | 4000 | 8840 |
| 3600 | 6000 | 13240 |
| 6000 | 10000 | 22040 |

Table. 3 shows that the Dynamic Programming technique has the highest memory requirement. This illustrates the working of this technique, using memoization, the process of storing solutions to the sub-problems instead of recomputing them.

The graph of Table.3is plotted below, with the No. of items on the X-axis and the Memory utilized by the algorithm on the Yaxis:



Graph. 3 The rate of increase of the memory utilization with the increase in the number of items is linear for all the three techniques, as shown in Graph 3.Varying the number of Items and having fixed Capacity = 100

Table 4

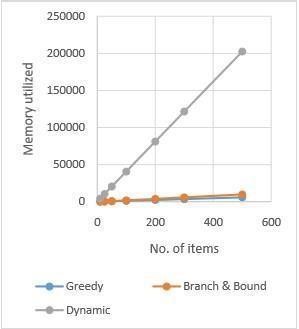
***No. of items***  **Greedy**  **Branch & bound**  **Dynamic**

***10***  **120**  **200**  **4400**

|  |  |  |  |
| --- | --- | --- | --- |
| ***25***  ***50*** | 300 | 500 | 10500 |
| 600 | 1000 | 20600 |
| ***100***  ***200*** | 1200 | 2000 | 40800 |
| 2400 | 4000 | 81200 |
| ***300*** | 3600 | 6000 | 121600 |
| ***500*** | 6000 | 10000 | 202400 |

Table 4 consists of the memory utilization of the algorithms for a high capacity (= 100). The table above shows that the memory utilized by the Dynamic programming technique is very high compared to the other algorithms at high capacities. This proves that this technique is inefficient when the capacity of the knapsack is high.

The graph of Table.4is plotted below, with the No. of items on the X-axis and the Memory utilized by the algorithm on the Yaxis:



Graph 4

## CONCLUSION

**───────────────────────────────────────**

The most efficient technique is the Greedy Algorithm, but it is inappropriate under certain conditions since it does not result in the optimal solution.

The Dynamic programming technique proves to be very efficient in terms of number of computations for lesser capacities, but as the capacity of the knapsack increases, this technique proves to be inefficient. The memory utilized by this technique is also the highest among the three approaches considered.

Thus, the most efficient approach for the Knapsack Problem is the Recursive Branch and Bound technique. It is simple and is easy to apply, and can be applied to solve the knapsack problem under all the circumstances.

For future work, genetic algorithms could be applied for the given problem, and a comparative analysis of the performance of the original algorithms and the modified algorithms could be implemented.