**Knapsack using hybrid of PSO and Genetic Algorithm**

**Project Based Learning**

**M.Tech (CSE) 2022-24**

# 

# JAYPEE INSTITUE OF INFORMATION TECHNOLOGY

# SECTOR-62, NOIDA. (U.P)

**Submitted by- Subject Coordinator  
Sushrut Khajuria (MTEG220017) Dr. Parul Agarwal   
Prankur Rana (MTEG220034)**

**Section-I**

**INTRODUCTION**

0-1 knapsack problems (KPs) is a typical NP-hard problem in combinatorial optimization problem. Genetic Algorithm is one of the evolutionary algorithms that use techniques which are inspired from Darwin’s Theory. And due to the simplicity and convergence speed of the problem we have also combined Particle Swarm Optimization (PSO). In this project, we address the 0-1Knapsack issue using hybrid PSO and Genetic Algorithms which combines the strengths of PSO and Genetic Algorithm (GA). The goal is to maximise the usefulness of the items in a knapsack without filling it to capacity. Approaches like dynamic programming, backtracking, branch and bound, etc. are not very helpful for solving the Knapsack problem because it is an NP problem. When it comes to finding solutions to issues like the Knapsack problem, which is typically regarded as computationally infeasible, Genetic Algorithms and PSO clearly outperforms all other approaches.

**MOTIVATION**

The reason we opted for this project is because Genetic Algorithm and PSO are the heuristic search and optimization techniques that in reality mimic the process that we can observe in nature. As the Algorithm has been used in increasing numbers for optimization problems these days and is also used in business, research and development and other engineering disciplines.

**ABOUT THE PROJECT**

This project outlines a study that employed Particle Swarm Optimization (PSO) Genetic Algorithms (GAs) to resolve the 0-1 Knapsack Problem (KP). An example of a combinatorial optimization issue is The Knapsack Problem, which aims to maximise the benefit of items in a knapsack without filling it to capacity.

**Section-II**

**BACKGROUND**

**Knapsack Problem**

The Knapsack Problem is a combinatorial optimization Problem. Given a set of items with a weight and value, it seeks to select a number of items to be placed in a Knapsack of fixed capacity such that total weight of the items is less or equal to the capacity but its value is as large as possible. The problem often arises in resource allocation with financial constraints.

**Genetic Algorithm**

Genetic Algorithms are computer algorithms that search for good solutions to a problem from among a large number of possible solutions. They were proposed and developed in the 1960s by John Holland, his students, and his colleagues at the University of Michigan. The principles of natural evolution, such as selection by characteristics such as fitness, reproduction, and mutation, served as the basis for these computational paradigms. These mechanics are excellently suited to address a wide range of real-world issues, including computational issues, in numerous disciplines. Optimization, automatic programming, machine learning, economics, immune systems, population genetics, and social systems are some areas in which GAs are used.

**Basic elements of Genetic Algorithm**

Most GAs methods are based on the following elements, populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring.

**Chromosomes**

The chromosomes in GAs represent the space of candidate solutions. Possible chromosomes encodings are binary, permutation, value, and tree encodings. For the Knapsack problem, we use binary encoding, where every chromosome is a string of bits, 0 or 1

**Fitness Function**

GAs require a fitness function which allocates a score to each chromosome in the current population. Thus, it can calculate how well the solutions are coded and how well they solve the problem.

**Selection**

The selection process is based on fitness. Chromosomes that are evaluated with higher values (fitter) will most likely be selected to reproduce, whereas, those with low values will be discarded. The fittest chromosomes may be selected several times, however, the number of chromosomes selected to reproduce is equal to the population size, therefore, keeping the size constant for every generation. This phase has an element of randomness just like the survival of organisms in nature. The most used selection methods, are roulette-wheel, rank selection, steady-state selection, and some others.

**Crossover**

Crossover is the process of combining the bits of one chromosome with those of another. This is to create an offspring for the next generation that inherits traits of both parents. Crossover randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring.

**Mutation**

Mutation is performed after crossover to prevent falling all solutions in the population into a local optimum of solved problem. Mutation changes the new offspring by flipping bits from 1 to 0 or from 0 to 1.

**Simple Genetic Algorithm Pseudo Code**

1. Initiate the parameters
2. Initialize the population  
    a) Put values in Chromosomes

b) Create Chromosomes within search range

c) Compute fitness on population

3) Start the loop for iterations

4) Recombination

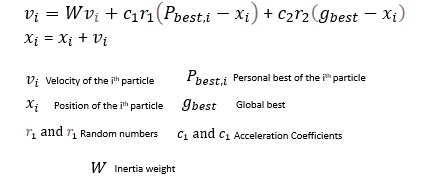
a) Selection  
 b) Crossover  
 c) Mutation

**Particle Swarm Optimization (PSO)**

The Particle Swarm Optimization is a Bio-inspired metaheuristic in flocks of birds or schools of fish. It is a type of evolutionary computation technology that was developed through the simulation of a simplified social model. PSO, which is swarm-based, causes each member of the swarm to travel to a desirable area.

However, it does not employ the evolution operator on individuals; rather, each individual is viewed as a no-volume particle (point) in the D-dimensional search space and travels at a specific speed, which can be dynamically changed in response to its own flight experience and that of other particles.

For the update of each particle using something called velocity vector which tells them how fast it will move the particle in each of the dimensions, the method for updating the speed of PSO is given by equation, and it is updating by the equation.



Steps of PSO to reach for the optimum:

**Step1:** Initialize a swarm of particles (swarm size is m), set random position and random velocity of each particle on the allowable range randomly, the position of each particle determines randomly

**Step2:** Evaluate the fitness of each particle

**Step3:** To each particle, compare the best global position undergone best p with its fitness value, if it’s better than best p, it’s the best current position best p.

**Step4:** To each particle, compare the best position undergone best g with its fitness value, if it’s better than best g , it’s the best swarm position ,the index sign of best g will be set anew.

**Step5:** Change the velocity and the position of each particle

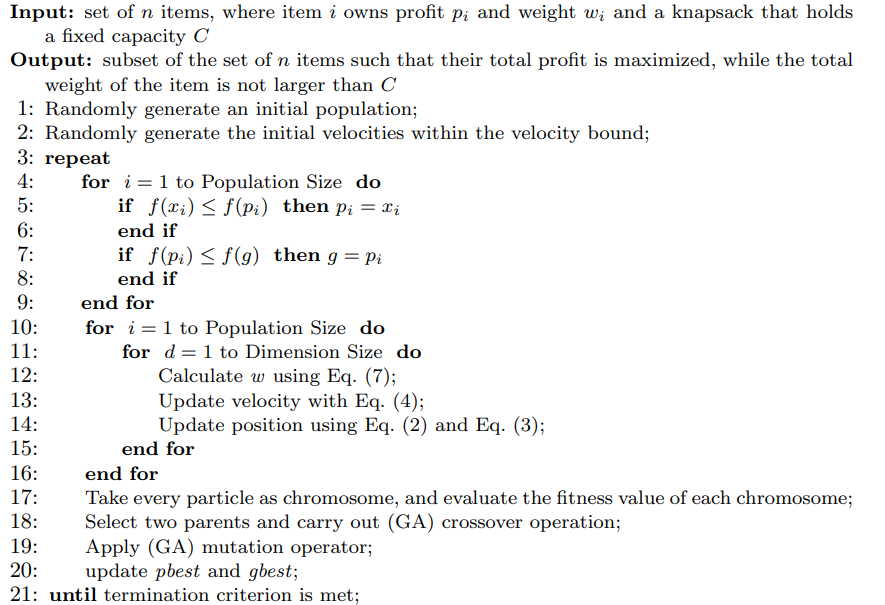
**Step6:** Check termination condition (the condition is enough good fitness value or reaching the maximum iterations, or that optimal solution changes no longer),If it meets the above condition, stop iteration; otherwise return to Step2.

**Hybrid PSO-GA Algorithm-**

The proposed hybrid PSO-GA algorithm is applied on the 0-1 knapsack problems. In this multiple solutions are generated randomly as an initial population, a set of velocities are generated randomly used to update population, and then objective function values are evaluated for each solution. After the evaluation, the population is evolved by two-steps operation, i.e. PSO operation and GA operation.

First each particle is represented as a bit string. After which the bit is updated by the above mentioned equations. Our proposed PSO-GA hybrid algorithm adopt an adaptive inertia weighted scheme for PSO. This scheme allows the search process to start first with exploration and gradually move towards exploitation by linearly increasing the inertia weight. After the position and velocity of each particle were updated, our algorithm take all of the particles as chromosomes and evaluate their fitness values again for subsequent GA operations. In GA of the presented hybrid algorithm, each operation is selected to enhance the diversity of solutions. In order to identify, inherit and protect good common genes shared by chromosomes, a crossover operator is applied to help our algorithm to converge to optima. We check if the crossover could be applied according to the crossover probability pc, and if it is, two new children are then generated from their parents. In order to reduce the bias associated with the length of the binary representation, an uniform crossover operator is applied. Firstly, a binary mask is constructed randomly, and then the children inherits the allele from their parents according to the value of the mask. For example, the first child inherits the allele from the first parent when the value of locus in mark equal to 1, and from the second parent when the value of locus in mark equal to 0. After that, a mutator operator is carried out to prevents irreversible loss of certain patterns by introducing small random changes into chromosomes. Mutation operator assures diversity in the population and prevents premature convergence. In our experiment study, we check each bit in the individual if the mutation could be applied according to the mutation probability and if it is, the value of that bit is then flipped. Finally, the pbest and gbest were updated according to the offspring’s fitness value

**Algorithm- Pseudo code of Hybrid PSO-GA Algorithm**

****

**Programming language used**

**Python-**

It is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured, object-oriented and functional programming.

The latest released version of python is 3.10.7

And we are using the same latest version of python for optimization of knapsack problem.

**The Dataset employed:**

We don’t use specific data set in our project. We just use two arrays one for the profits of the objects and the second for the corresponding weights to each object, and the knapsack has its own capacity.

**Code-**

import random

import math

# function we are attempting to optimize (minimize)

val = [35, 85, 135, 10, 25, 2, 94]

kg = [2, 3, 9, 0.5, 2, 0.1, 4]

maxKg = 25

def func1\_ev(x):

t = prof(x)

return w\_kg(x, t)

def prof(x):

total\_prof = 0

for i in range(len(x)):

total\_prof += x[i] \* val[i]

return total\_prof

def w\_kg(x, profit):

total\_kg = 0

for i in range(len(x)):

total\_kg += x[i] \* kg[i]

if total\_kg <= maxKg:

return profit - 100\*(min(0,(maxKg-total\_kg)))

elif total\_kg > maxKg:

return -profit

class Particle:

def \_\_init\_\_(self,initial):

self.position\_i=[] # particle position

self.velocity\_i=[] # particle velocity

self.pos\_best\_i=[] # best position individual

self.fit\_best\_i=-1 # best fitness individual

self.fit\_i=-1 # fitness individual

for i in range(0,num\_dimensions):

self.velocity\_i.append(random.uniform(0,1))

self.position\_i.append(initial[i])

# evaluate current fitness

def evaluate(self,costFunc):

self.fit\_i=costFunc(self.position\_i)

# check to see if the current position is an individual best

if self.fit\_i > self.fit\_best\_i:

self.pos\_best\_i=self.position\_i

self.fit\_best\_i=self.fit\_i

# update new particle velocity

def update\_velocity(self,pos\_best\_g):

w=0.99 # constant inertia weight (how much to weigh the previous velocity)

c1=1.99 # cognative constant

c2=1.99 # social constant

for i in range(0,num\_dimensions):

r1=random.random()

r2=random.random()

vel\_cognitive=c1\*r1\*(self.pos\_best\_i[i]-self.position\_i[i])

vel\_social=c2\*r2\*(pos\_best\_g[i]-self.position\_i[i])

self.velocity\_i[i]=w\*self.velocity\_i[i]+vel\_cognitive+vel\_social

# update the particle position based off new velocity updates

def update\_position(self,bounds):

for i in range(0,num\_dimensions):

self.position\_i[i]=self.position\_i[i]+self.velocity\_i[i]

if self.position\_i[i] > bounds[i][1]:

self.position\_i[i] = bounds[i][1]

elif self.position\_i[i] < bounds[i][0]:

self.position\_i[i] = bounds[i][0]

else:

self.position\_i[i] = round(self.position\_i[i])

class PSO:

def \_\_init\_\_(self,initial,bounds,num\_objects,particles,max\_iter):

global num\_dimensions

num\_dimensions=num\_objects

fit\_best\_g=-1 # best error for group

pos\_best\_g=[] # best position for group

# establish the swarm

swarm=[]

for i in range(0,particles):

swarm.append(Particle(initial))

# begin optimization loop

i=0

while i < max\_iter:

#print i,err\_best\_g

# cycle through particles in swarm and evaluate fitness

for j in range(0,particles):

swarm[j].evaluate(func1\_ev)

# determine if current particle is the best (globally)

if swarm[j].fit\_i > fit\_best\_g:

pos\_best\_g=list(swarm[j].position\_i)

fit\_best\_g=float(swarm[j].fit\_i)

# cycle through swarm and update velocities and position

for j in range(0,particles):

swarm[j].update\_velocity(pos\_best\_g)

swarm[j].update\_position(bounds)

i+=1

# print final results

print ("Initial profits: ", val)

print ("Initial weights: ", kg)

print ('FINAL: ')

print (pos\_best\_g)

sum=0

summ=0

for i in range((num\_objects)):

sum+=pos\_best\_g[i]\*val[i]

summ+=pos\_best\_g[i]\*kg[i]

print("the profit = ",sum)

print(summ)

#--- EXECUTE

initial=[] # initial starting location [x1,x2...]

for i in range(len(val)):

initial.append(0)

print(initial)

bounds=[] # input bounds [(x1\_min,x1\_max),(x2\_min,x2\_max)..

for i in range(len(val)):

bounds.append((initial[i],math.floor(maxKg/kg[i])))

print(bounds)

PSO(initial,bounds,num\_objects=len(val),particles=len(val)\*10,max\_iter=60)

import random as r

p = [35, 85, 135, 10, 25, 2, 94]

w = [2, 3, 9, 0.5, 2, 0.1, 4]

W = 25

l=len(p)

mr=0.5

n=100

fi=[]

pop=[]

g=0

def population(n):

for i in range(n):

t=[]

for j in range(l):

t.append(0) if r.random()>0.5 else t.append(1)

pop.append(t)

return pop

def fitness(pop):

fi=[0]\*len(pop)

for cr in pop:

s=0

for k in range(l):

s=s+cr[k]\*p[k]

fi[pop.index(cr)]=s

return pop,fi

def selection(pop):

newpop=[]

for cr in pop:

s=0

for k in range(l):

s=s+cr[k]\*w[k]

if(s<=9):

newpop.append(cr)

newpop,fi=fitness(newpop)

return newpop,fi

def crossover(pop):

cp=r.randint(1,6)

for i in range(int(n/2)):

p1=list(pop[r.randint(0,n-1)])

p2=list(pop[r.randint(0,n-1)])

p1[cp:],p2[cp:]=p2[cp:],p1[cp:]

pop.append(p1)

pop.append(p2)

newpop,fi=fitness(pop)

return newpop,fi

def mutation(pop):

for i in range(int(mr\*n)):

bit=r.randint(0,6)

p1=list(pop[r.randint(0,n-1)])

if(p1[bit]==1):

p1[bit]=0

else:

p1[bit]=1

pop.append(p1)

newpop,fi=fitness(pop)

return newpop,fi

pop=population(n)

pop,fi=fitness(pop)

while g<5:

#print(pop,fi)

print('selction',pop,fi)

n=len(pop)

pop,fi=crossover(pop)

print('crossover',pop,fi)

pop,fi=mutation(pop)

print('mutation',pop,fi)

pop,fi=selection(pop)

print('Generation:',g,'\n Best fitness: ',max(fi),' length: ',len(pop),'\n',fi,pop)

g+=1

q=0

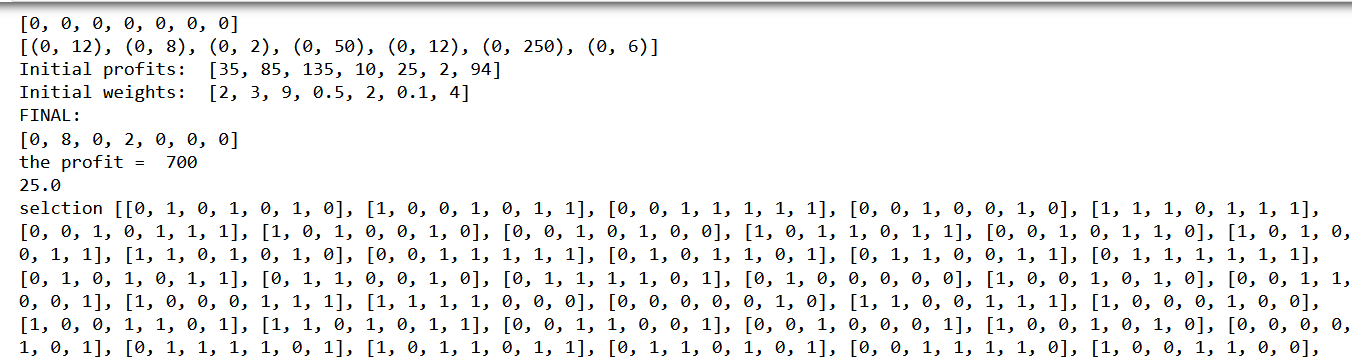
s=pop[fi.index(max(fi))]

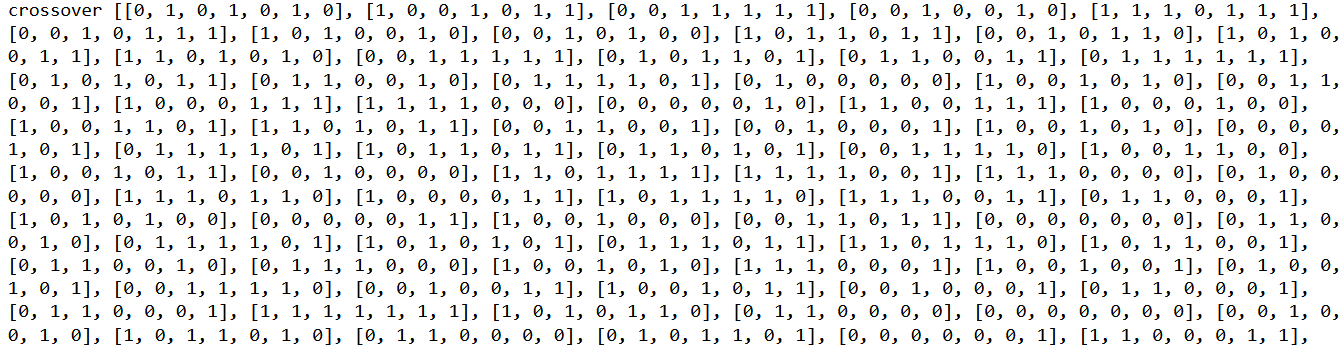
for i in range(len(s)):

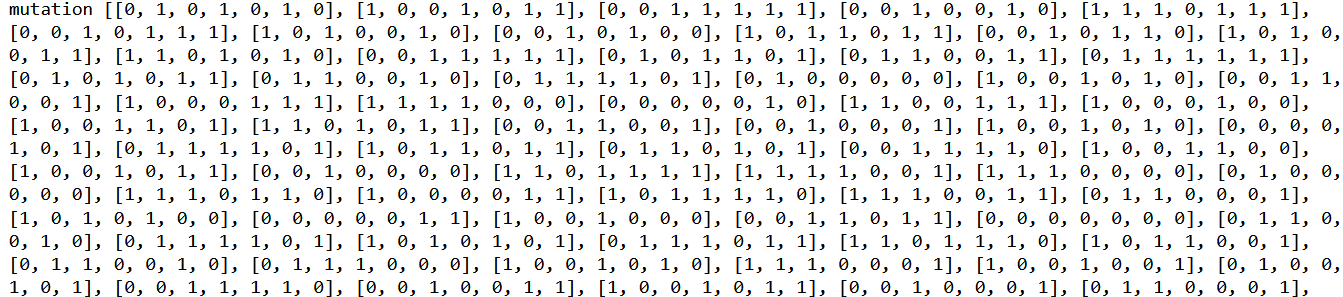
q=q+s[i]\*w[i]

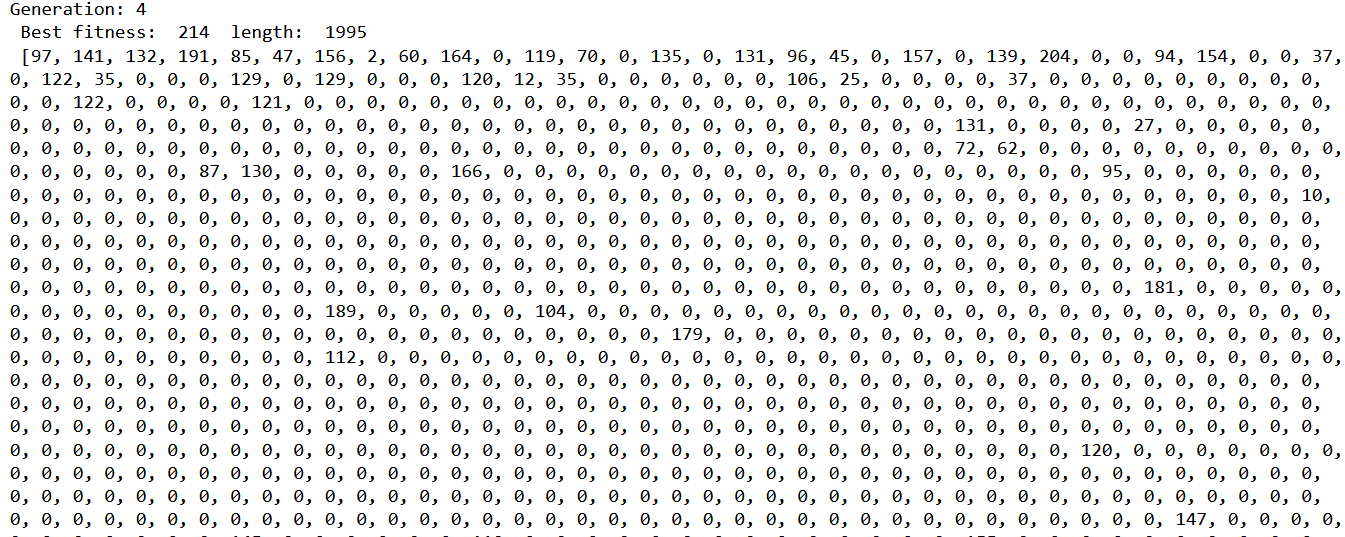
print('sol',pop[fi.index(max(fi))],'profit:',max(fi),'wt',q)

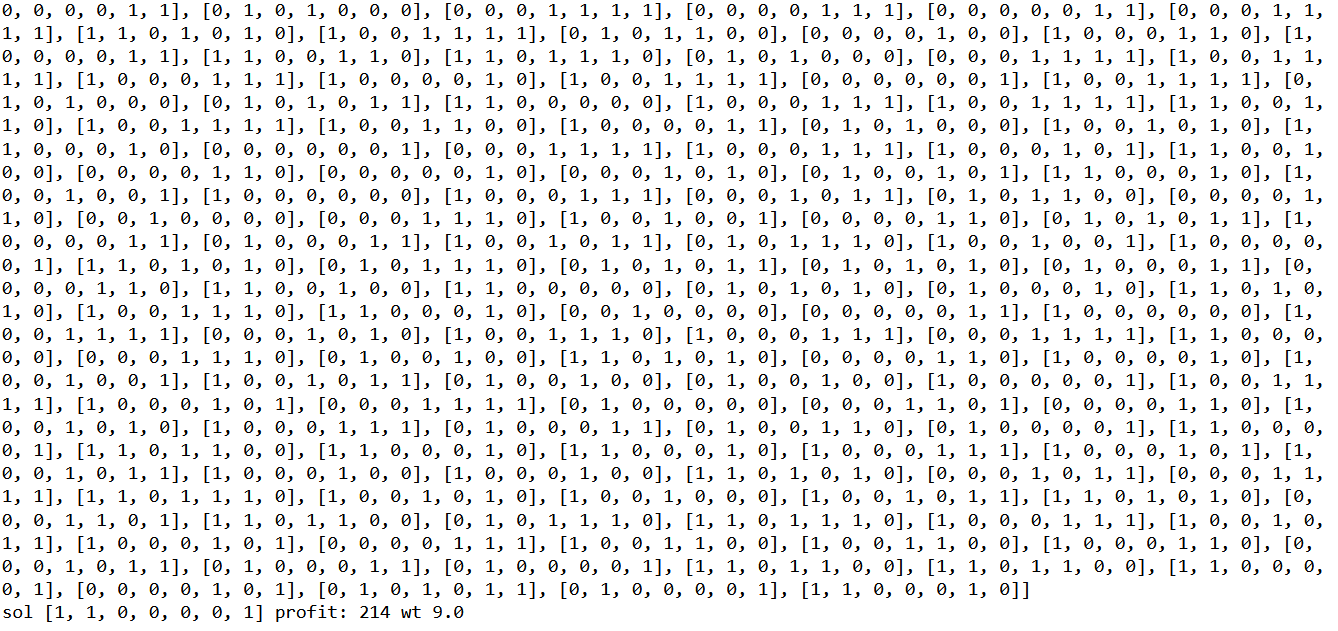
**Result-**

****

****

****





We got the optimal solution for the knapsack problem after 4 iterations. And got the best possible weight and profit of the optimal solution.