ISP611: OPTIMIZATION ALGORITHMS AND APPLICATION

INDIVIDUAL ASSIGNMENT

KNAPSACK PROBLEM: MAXIMIZING CARRY-ON LUGGAGE FOR TRAVELLING USING FLIGHT

21 MAY 2021

NAME: **MOHAMAD NAQIUDDIN BIN MOHD ZAINI**

STUDENT ID: **2020957767**

CLASS: **CS2595A**

1.0 **CASE STUDY**

The knapsack problem is a combinatorial optimization problem in which it must be calculate based on the amount of each object to include in a collection such that the total weight is less than or equal to a given limit and the total value is as large as possible given a set of objects, each with a weight and a value. It gets its name from the dilemma that anyone with a fixed-size knapsack faces when they have to fill it with the most important objects. The issue often occurs in resource allocation, where decision makers must select from a collection of non-divisible projects or tasks when working under a strict budget or time constraint.

In this case study, knapsack problem is used based on the carry-on luggage that one person can bring with them on flight. An aeroplane plays a vital role in the economics and travel sectors. Usually, people take an aeroplane to travel inside and outside of a country. Furthermore, people brought luggage which usually full of clothes, food or souvenir while travelling. Sadly, there's a weight limit for carry-on luggage that one person can bring with them on the flight. As an example, it is Malaysia Airports rules and regulations that luggage allowed onto the aircraft should not exceed maximum weight of 7kg. Based on the Knapsack problem, we will maximize the weight of luggage that one person can bring on the flight with the limit that airlines have set by using Simple Genetic Algorithm.

2.0 **OBJECTIVE FUNCTION**

To maximize the weight of a luggage that allowed onto the aircraft based on the weight limit set by the airlines. The weight limit of luggage is 7kg and there are only several items that can passenger brings onto the flight. Thus, based on the restrictions of item, several necessary things were selected to be included in this case study.

This knapsack problem is a 0-1 knapsack problem, in which the number of copies of each type of object is limited to either zero or one. Given a set of n items numbered 1 through n, each with a weight Wi and a value Vi, as well as a maximum weight capacity W, solve the following problem.

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}.$$

Figure 1.0: Knapsack Problem

The number of instances of item I to include in the knapsack is represented by Xi. Informally, the aim is to maximize the total of the values of the objects in the knapsack such that the weights are less than or equal to the capacity of the knapsack.

3.0 **CHROMOSOMES REPRESENTATION**

The genes/things that were shortlisted was based on “What Can I Bring?” by United States Transportation Security Administration. There a lot of other things that can be bring on flight, but only several were selected for this case study, as them aim is only to show on understanding of Genetic Algorithm. Further investigation and development can be made in this case study if were required.

Table 1.0 Information on Genes Selected

No	**Things**	**Weight**	**Values**
	Food/Drink		
1	Chocolate	40g	1
2	Water bottle (100mL)	100g	2
3	Fruits (Orange, Apple, Banana)	400g	3
4	Bread	405g	4
5	Cooked Meat, Seafood and Vegetable	1000g	5
	Devices		
6	Camera	800g	6
7	Laptop	2000g	7
8	Headphones	15g	8
9	Printer	2700g	9
10	Desktop Computer	3170g	10
	Belongings		
11	Hair Straightener	460g	11
12	Hair Dryers	907g	12
13	Blanket	1600g	13
14	Pillow	1360g	14
	Medical		
15	Vitamins	138g	15
16	Supplements	500g	16

Table 2.0 Chromosome representation

Ch	Items																Fitness= val += val_item_present if val>86, val=0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Ch1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ch2	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	11+12+13+14+15+16=81
Ch3	1	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1	1+9+10+14+16=50
Ch4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	2+4+6+8+10+12+14+16= 72
…																	
CHn	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

PROCESS EXPLAINATION:

Fitness

The bit of 1 represent the presence of an item in the knapsack while bit 0s represent the absent of an item in the knapsack. Since knapsack problem has it limit, any fitness that goes over the limit of weight and value will be initialized as 0. This is to exclude all the overweight luggage. The highest fitness limit value can reach is 86 and if more that that the fitness value will be turn in 0.

Selection

The other process of Genetic Algorithm of knapsack problem is the same as other problem. Selection is used to select a pair of solution which will be the parents of 2 new solutions of the next generation. Solutions with the highest fitness will be most likely to be chosen.

Crossover

A single point crossover is used in this case study. It takes 2 genomes as parameters and 2 genomes as output. The genome was randomly cut and takes the first half of genome A and the other half is genome B and it was return as the first solution of genome. The second solution is consists of first half of genome B and the second half of genome A. Both of the genome must be in the same length.

Mutation

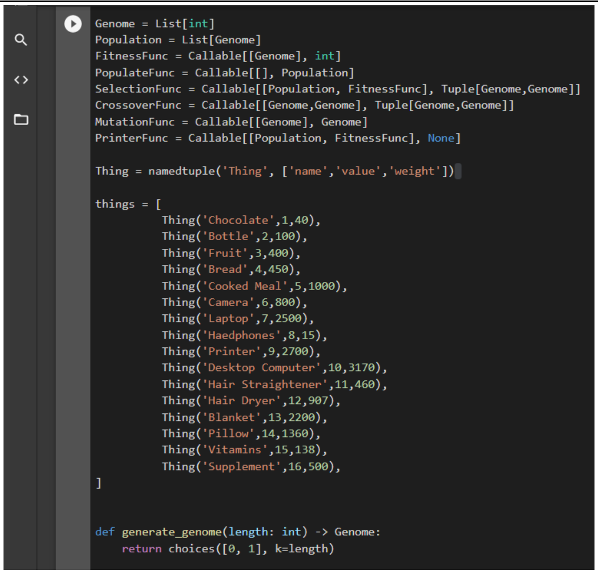
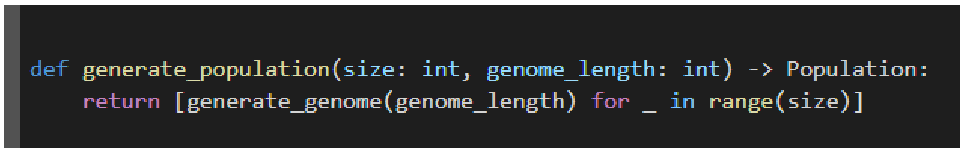
Mutation takes certain probability that changes 1 to 0 and 0 to 1 at a random position. If random index return the value higher than the probability, it is ignored. Otherwise, it is changed to absolute value of the current value minus 1.

4.0 **PROGRAMMING LANGUAGE**

The programming language that is used in this case study is Python Version 3.6.9 Google Collab as its platform.

5.0 **STEPS IN GENETIC ALGORITHM AND STEPS USED FOR CODING:**

Table 3.0 Steps in Genetic Algorithm

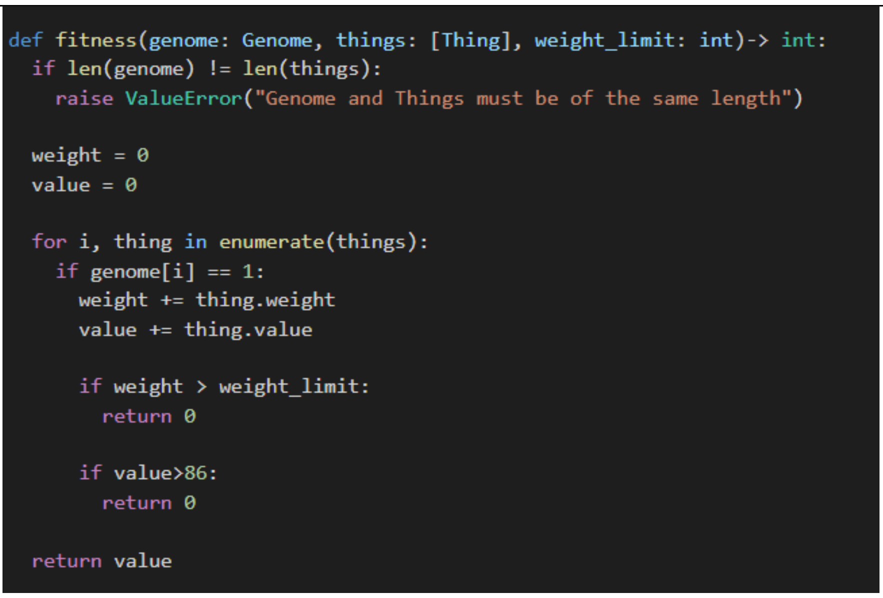
No.	**Steps**	**Coding**
INTIALIZATION		
1	The genomes/things were initialized inside a collection. The genome for 1 solution was generated by a list of 1 and 0.	
POPULATION		
2.	Population is a list of genomes. Generating population is done by calling generate_genome function multiple times until population has its own desired size.	

```python
Genome = List[int]
Population = List[Genome]
FitnessFunc = Callable[[Genome], int]
PopulateFunc = Callable[[], Population]
SelectionFunc = Callable[[Population, FitnessFunc], Tuple[Genome,Genome]]
CrossoverFunc = Callable[[Genome,Genome], Tuple[Genome,Genome]]
MutationFunc = Callable[[Genome], Genome]
PrinterFunc = Callable[[Population, FitnessFunc], None]

Thing = namedtuple('Thing', ['name','value','weight'])

things = [
          Thing('Chocolate',1,40),
          Thing('Bottle',2,100),
          Thing('Fruit',3,400),
          Thing('Bread',4,450),
          Thing('Cooked Meal',5,1000),
          Thing('Camera',6,800),
          Thing('Laptop',7,2500),
          Thing('Haedphones',8,15),
          Thing('Printer',9,2700),
          Thing('Desktop Computer',10,3170),
          Thing('Hair Straightener',11,460),
          Thing('Hair Dryer',12,907),
          Thing('Blanket',13,2200),
          Thing('Pillow',14,1360),
          Thing('Vitamins',15,138),
          Thing('Supplement',16,500),
]


def generate_genome(length: int) -> Genome:
    return choices([0, 1], k=length)
```

```python
def generate_population(size: int, genome_length: int) -> Population:
    return [generate_genome(genome_length) for _ in range(size)]
```

FITNESS FUNCTION

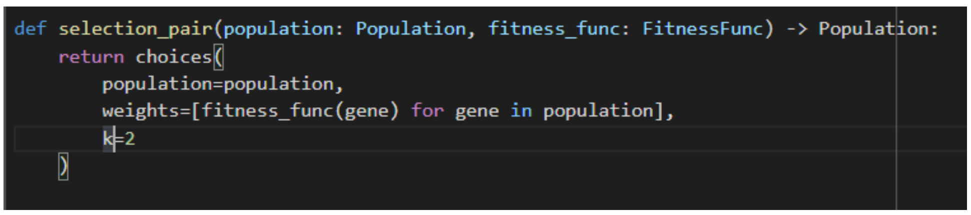
3. The fitness function will return fitness value. If fitness value is greater than 86 or weight is greater than weight limit, it will be initialized as 0.



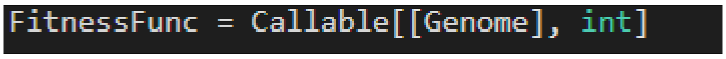
```
def fitness(genome: Genome, things: [Thing], weight_limit: int)-> int:
  if len(genome) != len(things):
    raise ValueError("Genome and Things must be of the same length")

  weight = 0
  value = 0

  for i, thing in enumerate(things):
    if genome[i] == 1:
      weight += thing.weight
      value += thing.value

      if weight > weight_limit:
        return 0

      if value>86:
        return 0

  return value
```

SELECTION FUNCTION

4. The selection pair with the highest fitness value will be the parents. With that data from fitness function is needed. Thus, a FitsnessFunc that store all the fitness data is declared and call upon in this function.



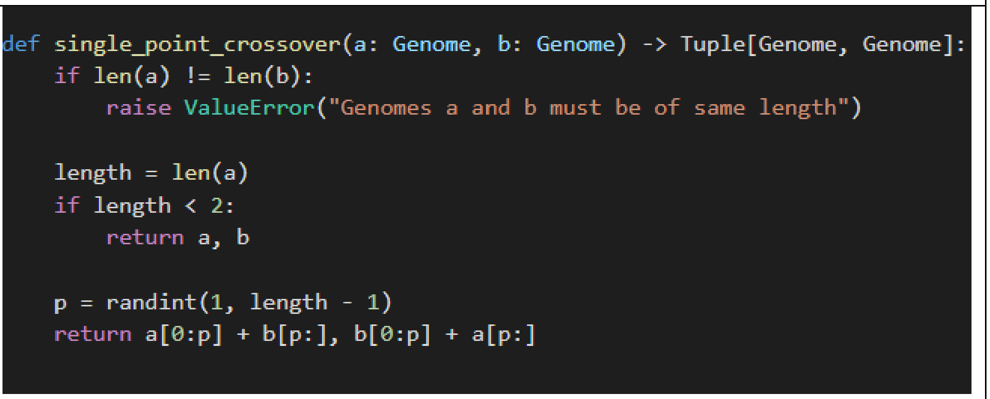
```
def selection_pair(population: Population, fitness_func: FitnessFunc) -> Population:
    return choices(
        population=population,
        weights=[fitness_func(gene) for gene in population],
        k=2
    )
```



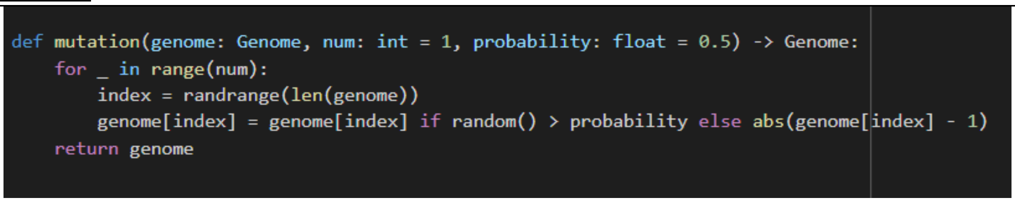
```
FitnessFunc = Callable[[Genome], int]
```

CROSSOVER FUNCTION

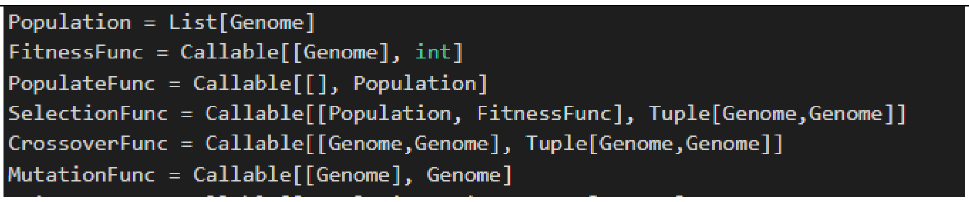
5. Two genomes were taken as parameters and it will return two output. The index is randomly opted.



```
def single_point_crossover(a: Genome, b: Genome) -> Tuple[Genome, Genome]:
    if len(a) != len(b):
        raise ValueError("Genomes a and b must be of same length")

    length = len(a)
    if length < 2:
        return a, b

    p = randint(1, length - 1)
    return a[0:p] + b[p:], b[0:p] + a[p:]
```

MUTATION FUNCTION		
6.	If random index return the value higher than the probability, it is ignored. Otherwise, it is changed to absolute value of the current value minus 1.	

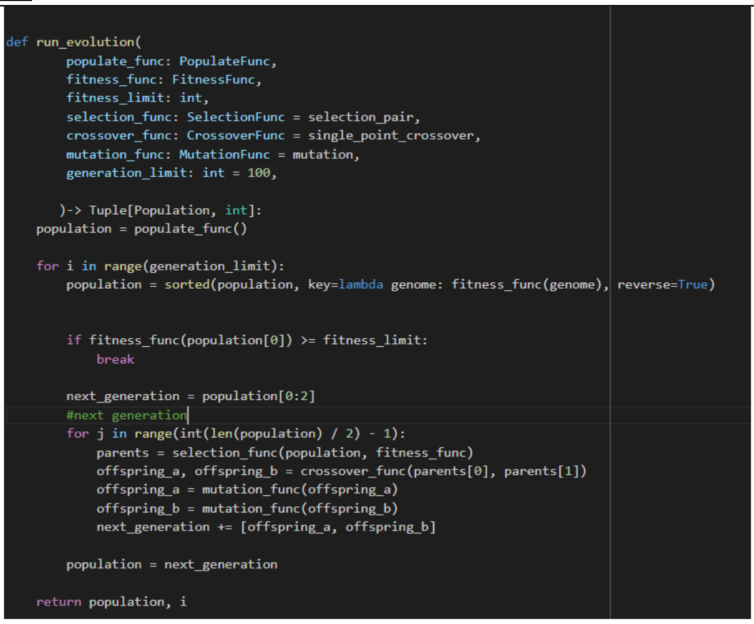
```python
def mutation(genome: Genome, num: int = 1, probability: float = 0.5) -> Genome:
    for _ in range(num):
        index = randrange(len(genome))
        genome[index] = genome[index] if random() > probability else abs(genome[index] - 1)
    return genome
```

DECLARING ALL THE FUNCTION USED AS GLOBAL		
7.	All the function that were used through out the coding is declare globally.	

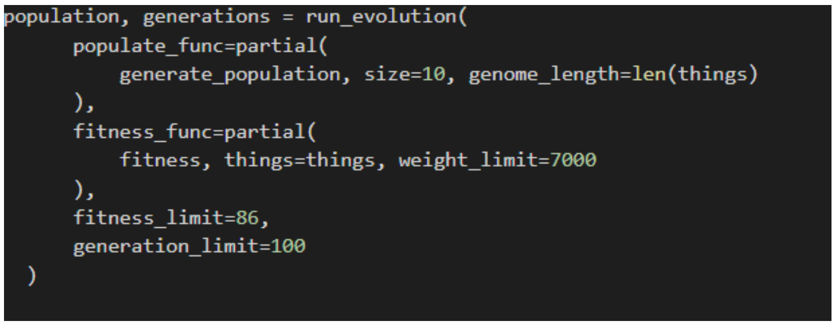
```python
Population = List[Genome]
FitnessFunc = Callable[[Genome], int]
PopulateFunc = Callable[[], Population]
SelectionFunc = Callable[[Population, FitnessFunc], Tuple[Genome,Genome]]
CrossoverFunc = Callable[[Genome,Genome], Tuple[Genome,Genome]]
MutationFunc = Callable[[Genome], Genome]
```

MAIN FUNCTION/LOOP

8. All of the function were called in this main function. Thus, the loop of genetic algorithm is placed in this function. The next generation coding were also placed here.



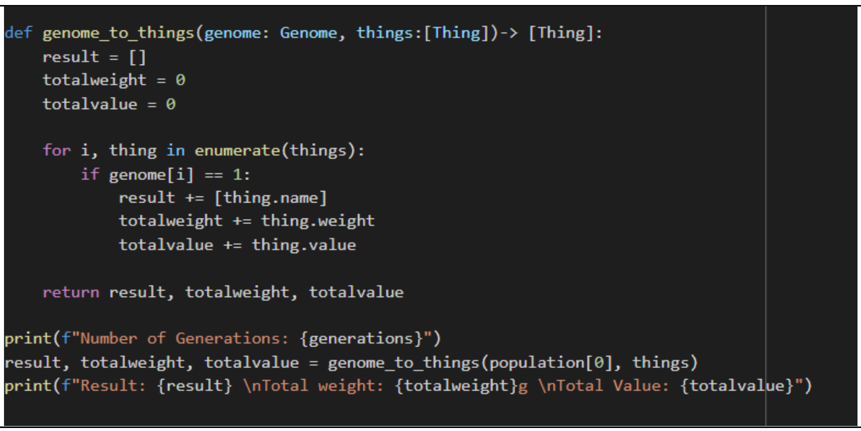
```python
def run_evolution(
        populate_func: PopulateFunc,
        fitness_func: FitnessFunc,
        fitness_limit: int,
        selection_func: SelectionFunc = selection_pair,
        crossover_func: CrossoverFunc = single_point_crossover,
        mutation_func: MutationFunc = mutation,
        generation_limit: int = 100,

       )-> Tuple[Population, int]:
    population = populate_func()

    for i in range(generation_limit):
        population = sorted(population, key=lambda genome: fitness_func(genome), reverse=True)


        if fitness_func(population[0]) >= fitness_limit:
            break

        next_generation = population[0:2]
        #next generation
        for j in range(int(len(population) / 2) - 1):
            parents = selection_func(population, fitness_func)
            offspring_a, offspring_b = crossover_func(parents[0], parents[1])
            offspring_a = mutation_func(offspring_a)
            offspring_b = mutation_func(offspring_b)
            next_generation += [offspring_a, offspring_b]

        population = next_generation

    return population, i
```



```python
population, generations = run_evolution(
        populate_func=partial(
            generate_population, size=10, genome_length=len(things)
        ),
        fitness_func=partial(
            fitness, things=things, weight_limit=7000
        ),
        fitness_limit=86,
        generation_limit=100
   )
```

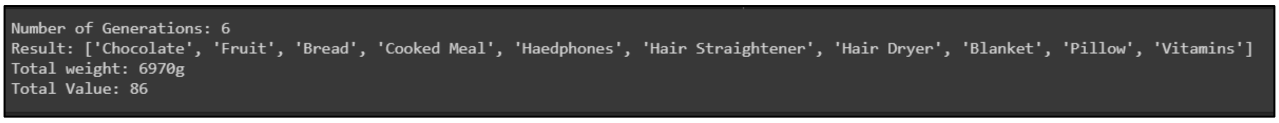
PRINT

9. A function is generate to help to print the things, totalvalue and totalweight based on the selected genomes.



```python
def genome_to_things(genome: Genome, things:[Thing])-> [Thing]:
    result = []
    totalweight = 0
    totalvalue = 0

    for i, thing in enumerate(things):
        if genome[i] == 1:
            result += [thing.name]
            totalweight += thing.weight
            totalvalue += thing.value

    return result, totalweight, totalvalue

print(f"Number of Generations: {generations}")
result, totalweight, totalvalue = genome_to_things(population[0], things)
print(f"Result: {result} \nTotal weight: {totalweight}g \nTotal Value: {totalvalue}")
```

6.0 OUTCOME AND RESULTS DISCUSSION



```
Number of Generations: 6
Result: ['Chocolate', 'Fruit', 'Bread', 'Cooked Meal', 'Haedphones', 'Hair Straightener', 'Hair Dryer', 'Blanket', 'Pillow', 'Vitamins']
Total weight: 6970g
Total Value: 86
```

The best solution of knapsack problem that a luggage bag should carry is Chocolate, Fruit, Bread, Cooked Meal, Headphones, Hair Straightener, Hair Dryer, Blanket, Pillow and Vitamins. At mutation of 0.5, it is solved in generation 6. The total weight is 6970g which is lower that the limit of 7000g. It is the maximum weight that this case study has. The total value of fitness is the highest as well with 86.

Parameter adjustment were made to see how the genetic algorithm react. The value of Population and Generation limit were remained constant:

Parameters	**Results**
Mutation: 0.5	Solved at generation 6
Mutation: 0.05	Solved in generation 32
Mutation: 0.03	Solved in generation 99

It is observed that when the mutation rate increase, it actually takes lesser generation to find the maximum fitness value.

7.0 **Suggestion for Improvement**

There are a few suggestions and room of improvement that can be made based on this case study.

1. Variation of solution

   In this case study, only consider weight and value of an item. Improvement can be made and more complexity can be added if another attribute were added which is size and robustness with a larger number of genomes. Larger genomes will help preserve heterogeneity in the population.

2. Uses of elitism

   Elitism in genetic algorithm meaning that the most fit handful of individuals are guaranteed a place in the next generation - generally without undergoing mutation. They should still be able to be selected as parents, in addition to being brought forward themselves. This is because, in this simple case study, there is solution that wasn't picked even though it is the highest.

3. Using brute force method

   Brute force will enumerate every possible packing configuration. It will choose the best solution. It is also optimality ensured. Since in this case study, it has only little amount of n, thus brute force was not included as it is very good method with large amount of n.

REFERENCE

1. *Malaysia Airlines' Hand Baggage Information – klia2.info*. (n.d.). KLIA. Retrieved May 12, 2021, from https://www.klia2.info/airlines/malaysia-airlines-hand-baggage-info/

2. *What Can I Bring? | Transportation Security Administration*. (n.d.). What Can I Bring? Retrieved May 12, 2021, from https://www.tsa.gov/travel/security-screening/whatcanibring/food

3. T. Pradhan, A. Israni and M. Sharma, "Solving the 0–1 Knapsack problem using Genetic Algorithm and Rough Set Theory," 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, 2014, pp. 1120-1125, doi: 10.1109/ICACCCT.2014.7019272.

Coding:

https://colab.research.google.com/drive/1r2sOuToQ0IaN65Xo6Oo4Pj2tHFJaAdKq?usp=sharing