پاسخ تمرین چهارم مهدیه نادری ۹۸۵۲۲۰۷۶

۱-

الگوریتم مورچگان:

```
Procedure AntColonyOptimization:
    Initialize necessary parameters and pheromone trials;
    while not termination do:
        Generate ant population;
        Calculate fitness values associated with each ant;
        Find best solution through selection methods;
        Update pheromone trial;
    end while
end procedure
```

مراحل:

1. همه ی مورچه‌ها در لانه خود هستند و فرومونی در محیط وجود ندارد.
2. شروع مورچه‌ها به جست‌وجو میان تمامی مسیرها در حالی که احتمال جست‌وجوی همه مسیرها با هم برابر باشد (مسیر طولانی تر ← مدت زمان بیشتری هم طول خواهد کشید.)
3. مورچه‌هایی که مسیر کوتاه را انتخاب کرده اند سریع‌تر به غذا می‌رسند و رد فرومون آن‌ها در آن مسیر قوی‌تر خواهد بود.
4. مورچه‌های بیشتری از مسیری که فرومون قوی دارد عبور می‌کنند. این امر باعث قوی‌تر شدن فرمومون در آن مسیر می‌شود. همچنین بدلیل پدیده تبخیر فرومون در مسیرهای طولانی کمتر می‌شود. در نتیجه احتمال انتخاب آن مسیر نیز کمتر می‌شود.

مثال و مراحل همگرایی:

مسئله ی زملن بندی در پردازنده های چند هسته ای را میتوان با این روش حل کرد.

فرض کنید سیستم شما دو پردازنده ی یکسان داشته و 12 عملیات T0 تا T11 داشته باشیم که زمان اجرای هر کدام متفاوت باشد. علاوه بر همه ی اینها یک گراف تقدم تاخر اجرای تسک‌ها نیز داریم که ترتیب کار ها را مشخص میکند.

چیزی که ما احتیاج داریم یک نمودار Gantt است که آغاز و پایان هر کار را در هر پردازنده به همراه بازه ی زمانی اجرایشان مشخص میکند.

برای حل این سوال ابتدا Tرا تشکیل می دهیم که یک ماتریس n*n است و بیانگر فرومون است. n تعداد پردازه های مسئله است. Tij نیز تمایل انتخاب پردازه j بعد از i است. مقادیر اولیه این ماتریس نزدیک به صفر است.

به تعداد ایپاک های مسئله کارهای زیر را انجام می دهیم:

1- یک نسل مورچه ایجاد می کنیم.
2- برای هر مورچه یک ایپاک می زنیم تا زمانی که تمامی پردازه ها وارد گراف شوند.
3- با استفاده از مقادیر T پردازه بعدی را انتخاب می‌کنیم.
4- بر روی یک State ایجاد شده که قرار گرفتیم مقدار فرومون را آپدیت می‌کنیم.
5- با رابطه تبخیر (Evaporation) مقدار T را آپدیت می‌کنیم.
6- عملیات بالا را تا زمانی تکرار می کنیم که به جواب مطلوب برسیم یا تعداد ایپاک ما تمام شود.

۲-کدهای مربوط به این سوال در فایل Q2.ipynb نوشته شده اند.

الف) ابتدا کتابخانه و ابزار لازم را فراخوانی کردیم.

```
import numpy as np
from random import randint, choices
```

سپس با توجه به صورت سوال ۳ آرایه تولید کردیم یکی برای سنگ ها، یکی برای وزن سنگ ها و یکی برای ارزش سنگ ها و یک متغیر threshold برای اینکه مجموع وزن سنگ ها از آن بیشتر نشود.

```
stones = np.array([i for i in range(1, 9)])
values = np.array([30, 10, 20, 50, 70, 15, 40, 25])
weights = np.array([2, 4, 1, 3, 5, 1, 7, 4])
threshold = 25
```

سپس تابعی نوشتیم که به کمک آن بتوانیم به صورت رندوم یک جمعیت اولیه از پاسخ ها و فیتنس آن ها تولید کنیم. پاسخ ها را به صورت اعداد باینری ۸ بیتی فرض کردیم طوریکه صفر بودن یک بیت معادل با قرار نگرفتن سنگ متناظر با آن در کوله پشتی و ۱ بودن یک بیت معادل قرار گرفتن سنگ متناظرش در کوله پشتی میباشد.

```
def random_initialization(values, weights, number_of_initialize_population, binary_length):
    population = []
    fitnesses = []
    for i in range(number_of_initialize_population):
        x = generate_random_binary(binary_length)
        population.append(x)
        fitnesses.append(fitness(x, weights, values))
    return population, fitnesses
```

تابع fitness را مطابق سوال نوشتیم. هر پاسخی که به عنوان ورودی بگیرد را بیت به بیت بررسی میکند و جمع ارزش ها و وزن های سنگ های قرار گرفته در کوله پشتی از نظر آن جواب را بدست می آورد. در حالتی که شرط کمتر از ۲۵ بودن رعایت شود ۱ را بر ارزش آن ها تقسیم میکند تا بر این اساس پاسخ ها ارزشگذاری شوند. البته این تابع برعکس عمل میکند یعنی هر چه مقدار کمی ریترن کند یعنی جواب بهتری است و هرچه پاسه به ۱ نزدیک تر باشد جواب بدتری داریم.

```
def fitness(x, weights, values, capacity=25):
    weight_sum = 0
    value_sum = 0
    for i in range(len(x)):
        if x[i] == '1':
            weight_sum += weights[i]
            value_sum += values[i]
    if weight_sum>25 or value_sum==0:
        return 1
    return 1.0/value_sum
```

توابع بعدی برای تولید پاسخ های اولیه به صورت رندوم و همچنین تولید نسل های جدید پاسخ ها از روی قبلی ها پیاده سازی شده اند.

```
import random

def generate_random_binary(n):
    result=''
    for i in range(n):
        result+=str(random.randint(0, 1))
    return result


def two_point_crossover(b1, b2):
    l = len(b1)
    p1 = random.randint(0, l)
    p2 = random.randint(0, l)
    if p2<p1:
        p1, p2 = p2,p1
    b3 = b1[0:p1] + b2[p1:p2] + b1[p2:l]
    b4 = b2[0:p1] + b1[p1:p2] + b2[p2:l]
    return b3, b4


def mutation(b, probability):
    if random.random() < probability:
        m1 = random.randint(0, len(b)-1)
        if b[m1] == '0':
            b = b[0:m1] + '1' + b[m1+1:]
        else:
            b = b[0:m1] + '0' + b[m1+1:]
    return b
```

در تابع زیر کل کد الگوریتم ژنتیک با کمک توابع اولیه ی بالا را پیاده سازی کردیم. شرط پایان الگوریتم را تعداد epoch فرض نمودیم. ابتدا جمعیت و فیتنس اولیه را به صورت رندوم بدست می آوریم. سپس در هر epoch فیتنس ها را سورت میکنیم. بعد دو نمونه از بهترین پاسخ ها را پیدا میکنیم. دو پاسخ جدید با crossover کردن آنها حساب میکنیم. دو نمونه هم با mutation. اگر این ۴ پاسخ جدید در جمعیت نبودند اضافه ی شان میکنیم. بعد از تمام شدن همه ی epoch ها بعترین پاسخ ریترن میشود.

```
def genetic_knapsack(values, weights, number_of_initialize_population, binary_length, max_population, mutation_probability, gener
    population, fitnesses = random_initialization(values, weights, number_of_initialize_population, binary_length)
    for i in range(generations):
        l = len(population)
        if l>max_population:
            population = population[l-max_population:]
            fitnesses = fitnesses[l-max_population:]

        temp = sorted(set(fitnesses))
        x_best = population[fitnesses.index(temp[0])]
        x_second_best = population[fitnesses.index(temp[1])]

        x_new_1, x_new_2 = two_point_crossover(x_best, x_second_best)

        x_new_1 = mutation(x_new_1, mutation_probability)
        x_new_2 = mutation(x_new_2, mutation_probability)

        if x_new_1 not in population:
            population.append(x_new_1)
            fitnesses.append(fitness(x_new_1, weights, values))
        if x_new_2 not in population:
            population.append(x_new_2)
            fitnesses.append(fitness(x_new_2, weights, values))
    return x_best
```

اجرای الگوریتم:

```
VALUES = [30, 10, 20, 50, 70, 15, 40, 25]
WEIGHTS = [2, 4, 1, 3, 5, 1, 7, 4]
NAMES = ["zomorod", "noqre", "yaqut", "almas", "berellian", "firuze", "aqiq", "kahroba"]

best_result = genetic_knapsack(VALUES,
                               WEIGHTS,
                               number_of_initialize_population=5,
                               binary_length=8,
                               max_population=10,
                               mutation_probability=0.1,
                               generations=1000
                              )

print("best chromosome:", best_result, " \n ------------------")
for i in range(len(best_result)):
     if best_result[i]=='1':
        print(NAMES[i])
```

```
best chromosome: 10111111
 ------------------
zomorod
yaqut
almas
berellian
firuze
aqiq
kahroba
```

طبق پاسخ بدست آمده اگر از هر نوع سنگ به جز یاقوت یکی بردارد، میتواند پول بیشتری بدست بیاورد.