حسابگری زیستی؛
حل مسئله N وزیر به کمک الگوریتم ممتیک؛
کامیار میرزاوزیری؛ ۶۱۰۳۹۶۱۵۲

---

۱ کروموزوم‌ها و جمعیت اولیه

اولین مرحله در حل یک مسئله به کمک الگوریتم ژنتیک تعریف کروموزوم یا همان نمایش یک پاسخ برای مسئله است. برای مسئله N وزیر به دنبال یک چینش از وزیرها در صفحهٔ شطرنج هستیم. اولین ایده این است که یک ماتریس $N \times N$ در نظر بگیریم و وزیرها را در این ساختار نمایش دهیم. اما با کمی تفکر به این نتیجه می‌رسیم که یک جایگشت برای نمایش کافی هست. چرا که طبق صورت مسئله در هر سطر و هر ستون دقیقا یک وزیر قرار می‌گیرد پس می‌توانیم جایگشتی به طول N از اعداد 1 تا N در نظر بگیریم که عدد iـام موجود در آن نشان‌دهندهٔ سطری باشد که وزیر حاضر در ستون iـام در آن قرار گرفته. به این ترتیب نه تنها از پیچیدگی کروموزوم‌ها کم می‌شود بلکه پاسخ‌های نامطلوب بسیاری نیز حذف می‌شوند. پس جایگشت‌ها را به عنوان کروموزوم انتخاب می‌کنیم. در پیاده‌سازی پایتون کلاس زیر را تعریف می‌کنیم.

```
class Chromosome:
    def __init__(self, data=None):
    global QUEENS

    if data is None:
        self.__data = list(range(QUEENS))
        random.shuffle(self.__data)
    else:
        self.__data = data
```

در constructor این کلاس یک جایگشت رندوم ایجاد می‌کنیم، و جمعیت اولیه را به کمک این متد ایجاد می‌کنیم.

۲ تابع برازش

در مرحلهٔ بعد باید تابع برازش را برای مسئله تعریف کنیم یا به عبارتی دیگر بدانیم کدام کروموزوم برازندگی بیشتری دارد. همانطور که در بخش قبل اشاره کردیم وزیرها در سطر یا ستون یکسان تهدید ندارند لذا تنها لازم است تهدیدهای قطری را بررسی کنیم. برای این منظور در هر یک از قطرهای اصلی و فرعی، چنانچه تعداد وزیرها بیش از یکی بود، تعداد وزیرهای اضافی را به عنوان نمره منفی برای کروموزوم در نظر می‌گیریم. با توجه به این محاسبه، مشخصهٔ cost را برای کلاس Chromosome تعریف می‌کنیم و آن را در constructor محاسبه می‌کنیم. همچنین عملگر کوچک‌تر را به کلاس اضافه می‌کنیم.

```
def __init__(self, data=None):
```

```
        ...

        self.__maindiagonals = {key: 0 for key in range(-QUEENS, QUEENS + 1)}
        self.__antidiagonals = {key: 0 for key in range(2 * QUEENS - 1)}
        self.cost = 0
        for i in range(QUEENS):
            self.__maindiagonals[i - self.__data[i]] += 1
            self.__antidiagonals[i + self.__data[i]] += 1
        diagonals = list(self.__maindiagonals.values()) + list(self.
    __antidiagonals.values())
        for diagonal in diagonals:
            if (diagonal > 0):
                self.cost += diagonal - 1

def __lt__(self, other):
    return self.cost > other.cost
```

با توجه به تعریف بالا c1 < c2 به این معنی است که هزینهٔ c1 از c2 بیشتر است یا به عبارت دیگر c2 برازنده‌تر است.

برای نمایش کروموزوم‌ها متد زیر را به کلاس Chromosome اضافه می‌کنیم.

```
def __str__(self):
    return self.__data.__str__() + ': ' + str(self.cost)
```

حال برای مثال دو کروموزوم تصادفی ایجاد می‌کنیم و آن را نمایش می‌دهیم، همچنین بررسی می‌کنیم که کدام کروموزوم برازنده‌تر است.

```
QUEENS = 4
c1 = Chromosome()
c2 = Chromosome()
print(c1)
print(c2)
print(c1 < c2)
```

حاصل زیر چاپ می‌شود.

```
[3, 0, 1, 2]: 3
[2, 0, 3, 1]: 0
True
```

همانطور که دیده می‌شود هزینهٔ c2 کمتر از c1 است لذا برازنده‌تر است پس c1 < c2 درست تعبیر می‌شود.

۳ انتخاب

برای انتخاب از ترکیب روش‌های انتخاب رتبه‌ای و رولت چندنشانه‌گر استفاده می‌کنیم. به این صورت که ابتدا جامعه را رتبه‌بندی می‌کنیم و سپس از رولت با N اشاره‌گر استفاده می‌کنیم. برای این منظور و همچنین کاربردهایی که در ادامه می‌بینیم، کلاس Population را پیاده‌سازی می‌کنیم.

```
class Population(list):
    def __init__(self):
        if type(countOrData) == int:
            self.__data = [Chromosome() for _ in range(countOrData)]
        elif type(countOrData) == list:
            self.__data = countOrData
        else:
            raise Exception()
        self.__data.sort()

    def __choose(self):
        n = len(self.__data)
        roulette = sum([[i] * (i + 1) for i in range(n)], [])
        turning = random.randint(0, n)
        roulette = roulette[turning:] + roulette[:turning]
        pointers = range(0, len(roulette), math.ceil(len(roulette) / n))

        choices = []
        for pointer in pointers:
            choices.append(self.__data[roulette[pointer]])

        return choices
```

۴ بازترکیب

حال که تکلیف جمعیت اولیه و تابع برازش مشخص شد باید برای crossover تصمیم بگیریم. به طور شهودی به نظر می‌رسد که بازترکیب ترتیبی یا Order Recombination راه مناسبی برای ترکیب دو کروموزوم باشد. برای پیاده‌سازی بازترکیب از عملگر ضرب بین دو کروموزوم استفاده می‌کنیم.

```
def __mul__(self, other):
    global QUEENS

    (side1, side2) = random.sample(range(QUEENS + 1), 2)
    start = min(side1, side2)
    end = max(side1, side2)
    if PRINT_SLICE_INFO:
```

```
            print(start, end)
        first_child = Chromosome(self.__crossover(self.__data, other.__data,
        start, end))
        second_child = Chromosome(self.__crossover(other.__data, self.__data,
        start, end))
        return [first_child, second_child]

    @staticmethod
    def __crossover(mother_data: list, father_data: list, start: int, end: int)
        :
        dimension = len(mother_data)
        data = [None] * dimension
        data[start:end] = mother_data[start:end]
        i = end
        for v in father_data[end:] + father_data[:end]:
            if v not in data:
                if i == start:
                    i = end
                if i == dimension:
                    i = 0
                data[i] = v
                i += 1
        return data
```

حال برای مثال بین دو کروموزوم تصادفی عمل ضرب انجام می‌دهیم و دو فرزند به وجود آمده را بررسی می‌کنیم.

```
QUEENS = 8
c1 = Chromosome()
c2 = Chromosome()
print(c1)
print(c2)
(c3, c4) = c1 * c2
print(c3)
print(c4)
```

حاصل زیر چاپ می‌شود.

```
[4, 7, 1, 5, 2, 0, 3, 6]: 2
[7, 5, 0, 4, 2, 1, 3, 6]: 4
3 7
[7, 4, 1, 5, 2, 0, 3, 6]: 2
[7, 5, 0, 4, 2, 1, 3, 6]: 4
```

۵ جهش، جست‌وجوی محلی و الگوریتم ممتیک

پس از تعریف crossover نوبت به تعریف جهش یا همان mutation می‌رسد. شهودا به نظر می‌رسد جهش جابه‌جایی در این مسئله بهتر از سایر جهش‌ها عمل کند لذا از این جهش استفاده می‌کنیم. تابعی پیاده‌سازی می‌کنیم که جهش جابه‌جایی را به تعداد بار مشخصی انجام دهد.

```
def swap(self, count, should_be_better):
    global QUEENS

    result = Chromosome(self.__data)

    for _ in range(count):
        (q1, q2) = random.sample(range(QUEENS), 2)
        if PRINT_SLICE_INFO:
            print(q1, q2)
        new_cost = result.cost
        new_maindiagonals = result.__maindiagonals.copy()
        new_antidiagonals = result.__antidiagonals.copy()

        new_maindiagonals[q1 - result.__data[q1]] -= 1
        if (new_maindiagonals[q1 - result.__data[q1]] >= 1):
            new_cost -= 1
        new_maindiagonals[q2 - result.__data[q2]] -= 1
        if (new_maindiagonals[q2 - result.__data[q2]] >= 1):
            new_cost -= 1
        new_antidiagonals[q1 + result.__data[q1]] -= 1
        if (new_antidiagonals[q1 + result.__data[q1]] >= 1):
            new_cost -= 1
        new_antidiagonals[q2 + result.__data[q2]] -= 1
        if (new_antidiagonals[q2 + result.__data[q2]] >= 1):
            new_cost -= 1
        new_maindiagonals[q1 - result.__data[q2]] += 1
        if (new_maindiagonals[q1 - result.__data[q2]] > 1):
            new_cost += 1
        new_maindiagonals[q2 - result.__data[q1]] += 1
        if (new_maindiagonals[q2 - result.__data[q1]] > 1):
            new_cost += 1
        new_antidiagonals[q1 + result.__data[q2]] += 1
        if (new_antidiagonals[q1 + result.__data[q2]] > 1):
            new_cost += 1
        new_antidiagonals[q2 + result.__data[q1]] += 1
        if (new_antidiagonals[q2 + result.__data[q1]] > 1):
            new_cost += 1
```

```
        if new_cost <= result.cost or not should_be_better:
            result.__data[q1], result.__data[q2] = result.__data[q2],
    result.__data[q1]
            result.__maindiagonals = new_maindiagonals
            result.__antidiagonals = new_antidiagonals
            result.cost = new_cost

    return result
```

همچنین همانطور که دیده می‌شود پارامتر دیگری تحت عنوان should_be_better رای این تابع در نظر گرفتیم. در صورت فعال بودن این پارامتر، تابع بالا جابه‌جایی را انجام می‌دهد و دوباره هزینه را محاسبه می‌کند و تنها در صورتی که هزینه کاهش پیدا کرده باشد جابه‌جایی را اعمال می‌کند.

برای مثال یک کروموزوم تصادفی ایجاد می‌کنیم و عمل __swap(1, True) را روی آن انجام می‌دهیم. سپس نتیجه را بررسی می‌کنیم.

```
QUEENS = 8
c = Chromosome()
print(c)
print(c.__swap(1, True))
```

حاصل زیر چاپ می‌شود.

```
[3, 1, 2, 5, 7, 0, 6, 4]: 4
2 7
[3, 1, 4, 5, 7, 0, 6, 2]: 4
```

همانطور که می بینیم انجام جابه‌جایی هزینه را بیشتر می‌کرده و لذا انجام نشده.

در انجام جهش بدون توجه به این موضوع عمل می‌کنیم اما از همین متد در جست‌وجوی محلی نیز استفاده می‌کنیم و این پارامتر را فعال می‌کنیم و به این صورت الگوریتم ژنتیک تبدیل به الگوریتم ممتیک می‌شود. برای پیاده‌سازی جهش از عملگر معکوس یعنی ~ و برای پیاده‌سازی جست‌وجوی محلی حول یک کروموزوم از عملگر مثبت یعنی + استفاده می‌کنیم.

```
def __invert__(self):
    return self.__swap(random.randint(0, MUTATION_DEGREE), False)

def __pos__(self):
    return self.__swap(random.randint(LOCAL_SEARCH_DEGREE[0],
    LOCAL_SEARCH_DEGREE[1]), True)
```

۶ جایگزینی

به کمک کدهایی که در بالا نوشتیم می‌توانیم نسل بعدی را از روی نسل قبلی ایجاد کنیم. حال نیاز داریم تصمیم بگیریم که کدام کروموزوم‌ها را نگه داریم و کدام‌ها را حذف کنیم. تصمیم می‌گیریم r_0 درصد برتر از فرزندان نسل جدید، r_1 درصد از سایر فرزندان نسل جدید، r_2 درصد از سایر جامعه قبلی و $100 - r_0 - r_1 - r_2$ درصد برتر از جامعه قبلی را به عنوان جمعیت جدید در نظر بگیریم.

```
def __replacement(self, children):
    n = len(children.__data)
    best_children_count = math.floor(REPLACEMENT[0] * n)
    other_children_count = math.floor(REPLACEMENT[1] * n)
    other_parents_count = math.floor(REPLACEMENT[2] * n)
    best_parents_count = n - best_children_count - other_children_count -
   other_parents_count
    self.__data = (
        children.__data[-best_children_count:] +
        random.sample(children.__data[:(n - best_children_count)],
   other_children_count) +
        random.sample(self.__data[:(n - best_parents_count)],
   other_parents_count) +
        self.__data[-best_parents_count:]
    )
    self.__data.sort()
```

۷ شرط توقف

واضح است که توقف زمانی باید اتفاق بیافتد که هزینه برابر صفر شود، یا به عبارت دیگر هیچ دو وزیری یکدیگر را تهدید نکنند و مسئله حل شده باشد. پس متد زیر را به کلاس Chromosome اضافه می‌کنیم.

```
def solved(self):
    return self.cost == 0
```

۸ کد نهایی

در نهایت کد زیر را برای حل سؤال می‌نویسیم.

```
QUEENS = 1000
N = 40
MUTATION_DEGREE = 1
LOCAL_SEARCH_DEGREE = [150, 200]
REPLACEMENT = [.7, .05, .1]
```

```
ESCAPE_THRESHOLD_PROPORTION = .3
ESCAPE_PROPORTION = .5
population = Population(N)
i = 0
while True:
    if PRINT_ITERATION_NO:
        print(f"Iteration: {i}")
    if PRINT_ITERATION_BEST_ANSWER:
        print(f"Best Answer: {population.answer().cost}")
    if PRINT_ITERATION_BEST_ANSWER_DETAILS:
        print(population.answer())
    if PRINT_ITERATION_ALL_ANSWERS:
        print(f"All Answers: {population.answers()}")

    population.iterate()

    if population.answer().solved():
        break
    i += 1

print(population.answer())
```

۹ نمایش

کد ساده زیر را برای نمایش بصری پاسخ‌ها می‌نویسیم.

```
import sys
import json
import matplotlib.pyplot as plt
import os

BLANK = (190, 219, 57)
QUEEN = (51, 105, 30)

if (len(sys.argv) < 2):
    raise Exception('please provide a file')
path = sys.argv[1]
try:
    with open(path) as f:
        c = json.load(f)
except:
    raise Exception('invalid file')
```

```
N = len(c)
m = [[BLANK for _ in range(N)] for _ in range(N)]

for i in range(N):
    m[i][c[i]] = QUEEN


plt.imshow(m)
plt.axis('off')
plt.savefig(os.path.splitext(path)[0] + '.png')
```

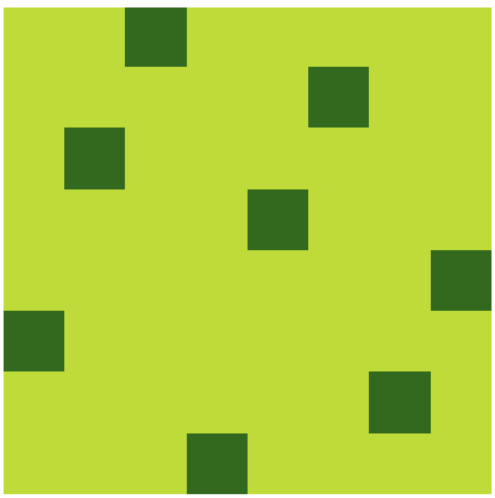
۱۰ نتایج

حال می‌توانیم کد موجود را با تعداد وزیرها متفاوت اجرا کنیم و نتایج را بررسی کنیم.

۱.۱۰ $N = 8$

اجرای الگوریتم پس از 1 نسل و گذشت 0.05 ثانیه متوقف شد.

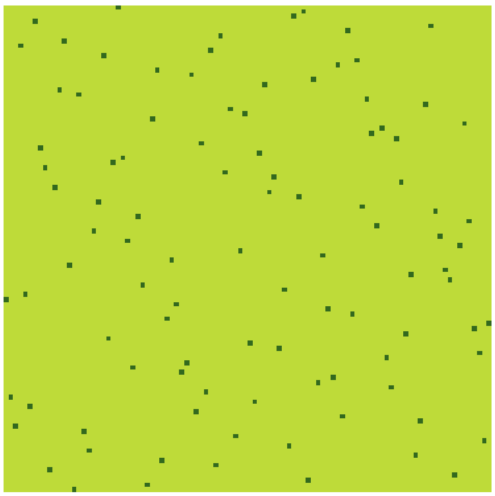
```
Answer: output.8.txt
The whole process took 0.05971717834472656
```



$\mathbf{N = 100}$ ۲.۱۰

اجرای الگوریتم پس از 27 نسل و گذشت 2 ثانیه متوقف شد.

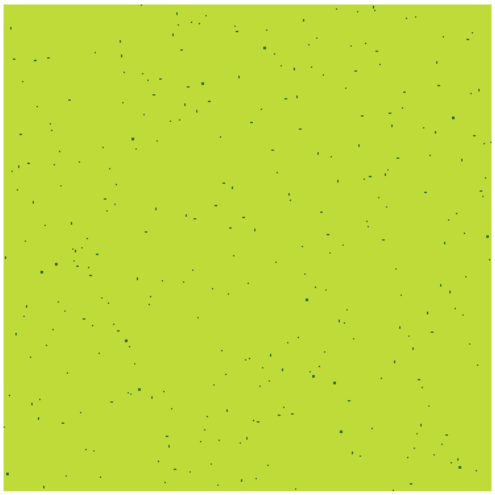
```
Answer: output.100.txt
The whole process took 2.872501850128174
```



$\mathbf{N = 300}$ ۳.۱۰

اجرای الگوریتم پس از 78 نسل و گذشت 13 ثانیه متوقف شد.

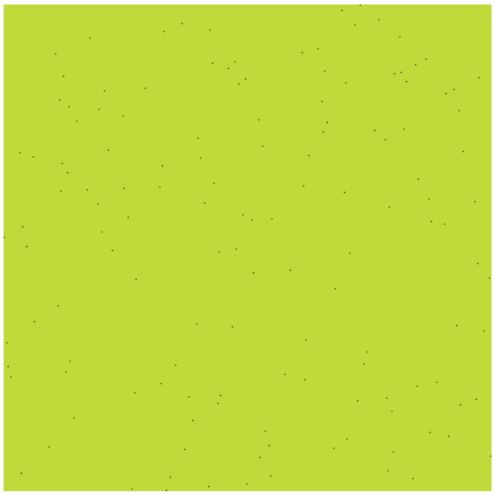
```
Answer: output.300.txt
The whole process took 13.421924352645874
```



$N = 1000$ ۴.۱۰

اجرای الگوریتم پس از 412 نسل و گذشت 3 دقیقه و 28 ثانیه متوقف شد.

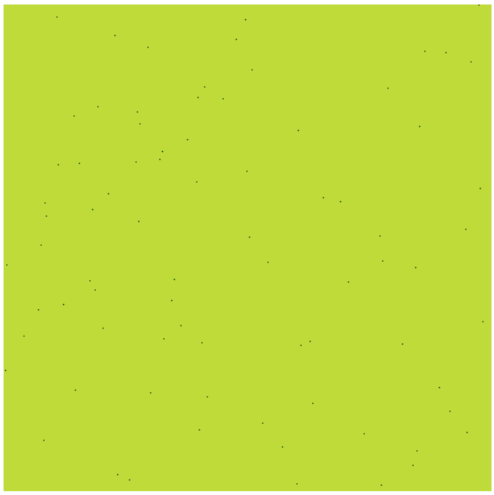
```
Answer: output.1000.txt
The whole process took 208.7185254096985
```



$N = 2000$ ۵.۱۰

اجرای الگوریتم پس از 970 نسل و گذشت 20 دقیقه و 54 ثانیه متوقف شد.

```
Answer: output.2000.txt
The whole process took 1254.3887646198273
```



$\mathbf{N = 5000}$ ۶.۱۰

با توجه به حجم مسئله جمعیت را به ناچار کوچک می‌کنیم و پارامتر N = 10 قرار می‌دهیم.
اجرای الگوریتم پس از 2298 نسل و گذشت 46 دقیقه و 14 ثانیه متوقف شد.

```
Answer: output.5000.txt
The whole process took 2774.877559185028
```