

حسابگری زیستی؛  
حل مسئله فروشنده دوره‌گرد به کمک الگوریتم ژنتیک؛  
کامیار میرزاویری؛ ۶۱۰۳۹۶۱۵۲

---

## ۱ خواندن مسئله

پیش از هر کاری لازم است مسئله را از فایل ورودی بخوانیم و آن را پارس کنیم تا بتوانیم به تعداد و فواصل شهرها (رئوس) دسترسی داشته باشیم. برای این منظور کلاسی به نام TSP تعریف می‌کنیم که هر instance آن یک مسئله فروشنده دوره‌گرد می‌باشد و با گرفتن مسیر فایل ورودی ایجاد می‌شود.

```
class TSP:
    EDGE_WEIGHT_TYPE_EXPLICIT = 'EXPLICIT'
    EDGE_WEIGHT_TYPE_GEO = 'GEO'
    EDGE_WEIGHT_TYPE_EUC_2D = 'EUC_2D'

    def __init__(self, path: str):
        self.dimension = None
        self.edge_with_type = None
        self.distances = None

        with open(path) as f:
            lines = f.readlines()
            lines = list(map(lambda line: line.strip(), lines))
            self.__parse(lines)
```

برای جلوگیری از طولانی شدن توابع مربوط به parse را در این گزارش نمی‌آوریم اما در فایل سورس پایتون موجود است.

## ۲ 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 فروشنده دوره‌گرد به دنبال یک دور در گراف هستیم که می‌توانیم این دور را با دنباله‌ای از رئوس نشان دهیم که عضو آخر دنباله به عضو اول متصل است. برای مثال کروموزوم 1, 3, 2, دوری را نشان می‌دهد که از 1 شروع شده، به 2 می‌رود، سپس به 3 می‌رود و نهایتاً به 1 باز می‌گردد. لذا کروموزوم‌ها به صورت جایگشت‌هایی از رئوس می‌باشند. پس جایگشت‌ها را به عنوان کروموزوم انتخاب می‌کنیم. در پیاده‌سازی پایتون کلاس زیر را تعریف می‌کنیم.

```
class Chromosome:
    def __init__(self, tsp: TSP):
```

```

self.tsp = tsp
self.__data = list(range(tsp.dimension))
random.shuffle(self.__data)
self.__cached_cost = 0
self.__cache_is_valid = False

```

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

### ۳ تابع برازش

در مرحله بعد باید تابع برازش را برای مسئله تعریف کنیم یا به عبارتی دیگر بدانیم کدام کروموزوم برازندگی بیشتری دارد. به سادگی می‌توان متوجه شد که به دنبال کمینه کردن طول دور هستیم پس تابع هزینه می‌تواند طول دور باشد و تابع برازش نیز برعکس تابع هزینه باشد. به عبارت دیگر هرچه هزینه یک کروموزوم کمتر باشد آن را برازنده‌تر می‌دانیم. پس عملگر کوچک‌تر را به کلاس Chromosome اضافه می‌کنیم.

```

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

def cost(self):
    if not self.__cache_is_valid:
        self.__cached_cost = 0
        for i in range(len(self.__data)):
            self.__cached_cost += self.tsp.distances[self.__data[i - 1]][
self.__data[i]]
        self.__cache_is_valid = True
    return self.__cached_cost

```

با توجه به تعریف بالا  $c1 < c2$  به این معنی است که هزینه  $c1$  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.

```

BAYG29 = 'testcase.bayg29.tsp'
tsp = TSP(BAYG29)
c1 = Chromosome(tsp)
c2 = Chromosome(tsp)
print(c1)

```

```
print(c2)
print(c1 < c2)
```

حاصل زیر چاپ می‌شود.

```
[9, 7, 23, 3, 11, 14, 25, 4, 13, 12, 27, 17, 24, 28, 20, 19, 16, 18, 0, 5,
 26, 6, 15, 22, 8, 21, 1, 10, 2]: 4920
[19, 6, 22, 17, 25, 2, 20, 12, 15, 14, 23, 10, 11, 27, 18, 26, 24, 28, 9,
 4, 21, 1, 8, 13, 0, 16, 5, 7, 3]: 4980
False
```

همانطور که دیده می‌شود هزینه  $c1$  کمتر از  $c2$  است لذا برازنده‌تر است پس  $c1 < c2$  نادرست تعبیر می‌شود.

## ۴ انتخاب

برای انتخاب از ترکیب روش‌های انتخاب رتبه‌ای و رولت چندنشانگر 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, tsp: TSP, countOrData):
        if type(countOrData) == int:
            self.__data = [Chromosome(tsp) for i 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
```

## ۵ 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):
    if (type(other) != Chromosome):
        raise Exception('Cannot crossover a with a non-chromosome.')

    if (self.tsp != other.tsp):
        raise Exception('Cannot crossover a with a chromosome of another
tsp.')

    (side1, side2) = random.sample(range(self.tsp.dimension + 1), 2)

    start = min(side1, side2)
    end = max(side1, side2)
    if VERBOSE_LEVEL > 1:
        print(start, end)

    first_child = Chromosome(self.tsp)
    first_child.__data = self.__crossover(self.__data, other.__data, start,
end)

    second_child = Chromosome(self.tsp)
    second_child.__data = 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.

```

BAYG29 = 'testcase.bayg29.tsp'
tsp = TSP(BAYG29)
c1 = Chromosome(tsp)
c2 = Chromosome(tsp)
print(c1)
print(c2)
(c3, c4) = c1 * c2
print(c3)
print(c4)

```

حاصل زیر چاپ می‌شود.

```

[
    0, 8, 11, 17, 24, 7, 13, 15, 23,
    5, 9, 4, 10, 12, 27, 16, 6, 14,
    3, 21, 1, 22, 28, 20, 19, 18, 2, 25, 26
]: 4677
[
    0, 8, 3, 2, 18, 5, 28, 15, 12,
    9, 11, 21, 4, 10, 23, 27, 14, 16,
    13, 17, 6, 19, 7, 26, 1, 24, 22, 20, 25
]: 4758
9 18
[
    8, 3, 2, 18, 28, 15, 11, 21, 23,
    5, 9, 4, 10, 12, 27, 16, 6, 14,
    13, 17, 19, 7, 26, 1, 24, 22, 20, 25, 0
]: 4953
[
    8, 17, 24, 7, 13, 15, 5, 12, 6,
    9, 11, 21, 4, 10, 23, 27, 14, 16,
    3, 1, 22, 28, 20, 19, 18, 2, 25, 26, 0
]: 5093

```

## ۶ جهش

پس از تعریف crossover نوبت به تعریف جهش یا همان mutation می‌رسد. شهودا به نظر می‌رسد جهش درجی یا Insert Mutation در این مسئله بهتر از جهش جابجایی عمل کند لذا از این جهش استفاده می‌کنیم. برای پیاده‌سازی جهش از عملگر معکوس یعنی ~ استفاده می‌کنیم.

```
def __invert__(self):
    (src, dst) = random.sample(range(self.tsp.dimension), 2)
    if VERBOSE_LEVEL > 1:
        print(src, dst)

    result = Chromosome(self.tsp)
    result.__data = self.__data
    v = result.__data[src]
    result.__data = result.__data[:src] + result.__data[src + 1:]
    result.__data = result.__data[:dst] + [v] + result.__data[dst:]

    return result
```

حال برای مثال یک کروموزوم تصادفی ایجاد می‌کنیم و عمل جهش را روی آن انجام می‌دهیم. سپس نتیجه را بررسی می‌کنیم.

```
BAYG29 = 'testcase.bayg29.tsp'
tsp = TSP(BAYG29)
c = Chromosome(tsp)
print(c)
print(~c)
```

حاصل زیر چاپ می‌شود.

```
[19, 21, 24, 26, 25, 14, 15, 3, 27, 13, 17, 10, 2, 7, 23, 9, 4, 22, 0,
8, 6, 16, 12, 28, 11, 20, 5, 18, 1]: 4649
14 5
[19, 21, 24, 26, 25, 23, 14, 15, 3, 27, 13, 17, 10, 2, 7, 9, 4, 22, 0,
8, 6, 16, 12, 28, 11, 20, 5, 18, 1]: 4738
```

## ۷ 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  $c$  درصد از فرزندان برتر نسل جدید را به همراه  $c - 100$  درصد از اعضای قبلی نگه داریم و جمعیت جدید را برابر آن‌ها قرار دهیم.

```
def __replacement(self, children):
```

```

n = len(children.__data__)
children_count = REPLACEMENT_CHILDREN_PROPORTION * n
parents_count = n - children_count
self.__data__ = children[-children_count:] + self.__data__[-parents_count:]
self.__data__.sort()

```

## ۸ شرط توقف

شرط توقف را به این صورت در نظر می‌گیریم که بعد از تعداد مشخصی نسل هر بار تغییرات هر نسل نسبت به نسل قبل از خود کمتر از مقدار مشخصی باشد. برای این منظور دو مقدار مشخص زیر را تعریف می‌کنیم.

```

IMPROVEMENT_THRESHOLD
STAGNANCY_THRESHOLD

```

شرط پایان را نیز به صورت زیر تعریف می‌کنیم.

```

improvement = population.answer() - answer
answer = population.answer()
if improvement < IMPROVEMENT_THRESHOLD:
    stagnancy += 1
    if stagnancy >= STAGNANCY_THRESHOLD:
        break
else:
    stagnancy = 0

```

توجه کنید که برای اجرا شدن کد بالا لازم است `answer()` را روی `Population` تعریف کنیم.

```

def answer(self) -> Chromosome:
    return self.__data__[-1]

```

همچنین لازم است تفاوت دو کروموزوم را بتوانیم اندازه‌گیری کنیم که این کار را با تعریف عمل تفریق روی کلاس `Chromosome` انجام می‌دهیم.

```

def __sub__(self, other):
    return other.cost() - self.cost()

```

## ۹ کد نهایی

در نهایت کد زیر را برای حل سؤال می‌نویسیم.

```

N = 500
MUTATION_PROBABILITY = .8
REPLACEMENT_CHILDREN_PROPORTION = .2

```

```

IMPROVEMENT_THRESHOLD = 10
STAGNANCY_THRESHOLD = 10

population = Population(tsp, N)
answer = population.answer()
stagnancy = 0
i = 0
while True:
    population.iterate()
    improvement = population.answer() - answer
    answer = population.answer()

    if VERBOSE_LEVEL > 0:
        print(f"Iteration: {i}")
        print(f"Best Answer: {population.answer()}")
    if VERBOSE_LEVEL > 1:
        print(f"All Answers: {population.answers()}")

    if improvement < IMPROVEMENT_THRESHOLD:
        stagnancy += 1
        if stagnancy >= STAGNANCY_THRESHOLD:
            break
    else:
        stagnancy = 0

    i += 1

if VERBOSE_LEVEL == 0:
    print(population.answer())

```

پارامترهای الگوریتم با آزمون و خطا به دست آمده‌اند. به نظر می‌رسد هرچه جمعیت بزرگ‌تر باشد الگوریتم پاسخ بهتری می‌دهد چرا که دیرتر همگرا می‌شود. از طرفی رکود و بهبودی که تعریف شده در همین حد کافی به نظر می‌رسد و زیاد کردن آن تفاوت چندانی در عملکرد ندارد. برای جلوگیری از همگرایی زودرس احتمال جهش را بسیار بالا و سهم فرزندان در هر نسل را پایین در نظر گرفتیم.

همچنین باید تابع `iterate` و توابع دیگر که برا اجرای کد فوق لازم می‌باشند را به `Population` اضافه کنیم.

```

def iterate(self):
    children = self.__crossover()
    children.__mutate()
    self.__replacement(children)

```



```

def __crossover(self):
    parents = self.__choose()
    random.shuffle(parents)
    children = []
    for i in range(0, len(parents) - 1, 2):
        children += parents[i] * parents[i + 1]
    return Population(None, children)

def __mutate(self):
    for child in self.__data:
        if random.random() < MUTATION_PROBABILITY:
            child = ~child

def answers(self) -> list:
    return list(map(lambda c: c.cost(), self.__data))

```

## ۱۰ نتایج

حال می‌توانیم کد موجود را روی داده‌ها اجرا کنیم و نتایج را بررسی کنیم.

### ۱.۱۰ bayg29

برای این مجموعه شهرها، پس از اجرای الگوریتم، و گذشت ۶۹ نسل، الگوریتم روی پاسخ 1624 ایستاد.

```

Iteration: 0
Best Answer: [21, 18, 19, 2, 11, 6, 24, 3, 17, 13, 26, 1, 20, 22, 23, 12,
              7, 14, 10, 15, 27, 8, 9, 0, 28, 25, 4, 5, 16]: 3617
Iteration: 1
Best Answer: [7, 24, 6, 27, 20, 22, 11, 5, 25, 28, 23, 18, 13, 12, 0, 26,
              2, 4, 8, 14, 17, 15, 19, 1, 9, 3, 21, 16, 10]: 3531
Iteration: 2
Best Answer: [2, 23, 6, 22, 27, 7, 11, 28, 8, 20, 19, 1, 4, 5, 14, 16, 13,
              10, 9, 21, 18, 26, 17, 3, 12, 0, 24, 15, 25]: 3392
Iteration: 3
Best Answer: [2, 23, 6, 22, 27, 7, 11, 28, 8, 20, 19, 1, 4, 5, 14, 16, 13,
              10, 9, 21, 18, 26, 17, 3, 12, 0, 24, 15, 25]: 3392
Iteration: 4
Best Answer: [25, 4, 11, 2, 19, 7, 20, 13, 14, 6, 18, 24, 23, 8, 27, 0, 5,
              21, 17, 16, 10, 9, 1, 15, 26, 22, 3, 12, 28]: 3328
Iteration: 5

```

Best Answer: [12, 19, 1, 9, 10, 6, 28, 27, 2, 16, 21, 13, 17, 26, 22, 7, 5, 0, 25, 4, 11, 8, 20, 3, 14, 24, 23, 15, 18]: 3115

Iteration: 6

Best Answer: [12, 19, 1, 9, 10, 6, 28, 27, 2, 16, 21, 13, 17, 26, 22, 7, 5, 0, 25, 4, 11, 8, 20, 3, 14, 24, 23, 15, 18]: 3115

Iteration: 7

Best Answer: [12, 19, 1, 9, 10, 6, 28, 27, 2, 16, 21, 13, 17, 26, 22, 7, 5, 0, 25, 4, 11, 8, 20, 3, 14, 24, 23, 15, 18]: 3115

Iteration: 8

Best Answer: [7, 0, 8, 23, 3, 14, 16, 13, 10, 9, 21, 1, 28, 25, 2, 27, 11, 5, 4, 20, 6, 22, 24, 15, 18, 12, 19, 17, 26]: 2954

Iteration: 9

Best Answer: [14, 16, 21, 1, 5, 8, 11, 27, 20, 28, 25, 2, 4, 17, 10, 13, 19, 26, 22, 7, 0, 23, 15, 12, 9, 18, 6, 24, 3]: 2600

Iteration: 10

Best Answer: [14, 16, 21, 1, 5, 8, 11, 27, 20, 28, 25, 2, 4, 17, 10, 13, 19, 26, 22, 7, 0, 23, 15, 12, 9, 18, 6, 24, 3]: 2600

Iteration: 11

Best Answer: [14, 16, 21, 1, 5, 8, 11, 27, 20, 28, 25, 2, 4, 17, 10, 13, 19, 26, 22, 7, 0, 23, 15, 12, 9, 18, 6, 24, 3]: 2600

Iteration: 12

Best Answer: [14, 16, 21, 1, 5, 8, 11, 27, 20, 28, 25, 2, 4, 17, 10, 13, 19, 26, 22, 7, 0, 23, 15, 12, 9, 18, 6, 24, 3]: 2600

Iteration: 13

Best Answer: [14, 16, 21, 1, 5, 8, 11, 27, 20, 28, 25, 2, 4, 17, 10, 13, 19, 26, 22, 7, 0, 23, 15, 12, 9, 18, 6, 24, 3]: 2600

Iteration: 14

Best Answer: [1, 20, 0, 7, 6, 10, 24, 3, 21, 16, 13, 17, 14, 15, 23, 26, 22, 18, 9, 12, 28, 2, 25, 27, 19, 4, 5, 11, 8]: 2544

Iteration: 15

Best Answer: [0, 1, 5, 8, 11, 27, 28, 25, 2, 4, 17, 10, 18, 9, 12, 3, 14, 16, 21, 13, 19, 20, 15, 7, 26, 22, 6, 24, 23]: 2507

Iteration: 16

Best Answer: [6, 14, 3, 15, 22, 23, 20, 28, 25, 2, 8, 27, 11, 5, 4, 1, 17, 21, 19, 9, 12, 0, 7, 26, 18, 10, 13, 16, 24]: 2485

Iteration: 17

Best Answer: [6, 14, 3, 15, 22, 23, 20, 28, 25, 2, 8, 27, 11, 5, 4, 1, 17, 21, 19, 9, 12, 0, 7, 26, 18, 10, 13, 16, 24]: 2485

Iteration: 18

Best Answer: [17, 10, 13, 16, 21, 18, 24, 6, 22, 14, 3, 9, 1, 23, 15, 26, 7, 27, 20, 0, 28, 25, 2, 4, 5, 11, 8, 19, 12]: 2317

Iteration: 19

Best Answer: [17, 10, 13, 16, 21, 18, 24, 6, 22, 14, 3, 9, 1, 23, 15, 26, 7, 27, 20, 0, 28, 25, 2, 4, 5, 11, 8, 19, 12]: 2317

Iteration: 20

Best Answer: [17, 10, 13, 16, 21, 18, 24, 6, 22, 14, 3, 9, 1, 23, 15, 26, 7, 27, 20, 0, 28, 25, 2, 4, 5, 11, 8, 19, 12]: 2317

Iteration: 21

Best Answer: [17, 10, 13, 16, 21, 14, 3, 9, 19, 1, 7, 0, 5, 8, 4, 28, 2, 25, 20, 11, 27, 26, 22, 6, 23, 24, 18, 15, 12]: 2225

Iteration: 22

Best Answer: [17, 10, 13, 16, 21, 14, 3, 9, 19, 1, 7, 0, 5, 8, 4, 28, 2, 25, 20, 11, 27, 26, 22, 6, 23, 24, 18, 15, 12]: 2225

Iteration: 23

Best Answer: [12, 17, 10, 13, 16, 21, 18, 24, 6, 14, 3, 9, 19, 1, 23, 26, 22, 7, 0, 15, 20, 28, 2, 25, 8, 11, 27, 5, 4]: 2215

Iteration: 24

Best Answer: [12, 17, 10, 13, 16, 21, 18, 24, 6, 14, 3, 9, 19, 1, 23, 26, 22, 7, 0, 15, 20, 28, 2, 25, 8, 11, 27, 5, 4]: 2215

Iteration: 25

Best Answer: [12, 17, 10, 13, 16, 21, 18, 24, 6, 14, 3, 9, 19, 1, 23, 26, 22, 7, 0, 15, 20, 28, 2, 25, 8, 11, 27, 5, 4]: 2215

Iteration: 26

Best Answer: [12, 17, 10, 13, 16, 21, 18, 24, 6, 14, 3, 9, 19, 1, 23, 26, 22, 7, 0, 15, 20, 28, 2, 25, 8, 11, 27, 5, 4]: 2215

Iteration: 27

Best Answer: [23, 12, 10, 21, 16, 13, 17, 14, 7, 26, 0, 5, 8, 4, 28, 2, 25, 20, 11, 27, 22, 6, 24, 18, 15, 3, 9, 19, 1]: 2172

Iteration: 28

Best Answer: [10, 21, 16, 13, 14, 17, 1, 19, 9, 12, 15, 18, 24, 6, 22, 23, 26, 7, 27, 20, 0, 28, 25, 2, 4, 5, 11, 8, 3]: 2161

Iteration: 29

Best Answer: [10, 13, 16, 21, 14, 3, 9, 19, 1, 23, 7, 0, 5, 8, 4, 28, 2, 25, 20, 11, 27, 26, 22, 6, 24, 18, 15, 12, 17]: 2008

Iteration: 30

Best Answer: [10, 13, 16, 21, 14, 3, 9, 19, 1, 23, 7, 0, 5, 8, 4, 28, 2, 25, 20, 11, 27, 26, 22, 6, 24, 18, 15, 12, 17]: 2008

Iteration: 31

Best Answer: [10, 13, 16, 21, 14, 3, 9, 19, 1, 23, 7, 0, 5, 8, 4, 28, 2, 25, 20, 11, 27, 26, 22, 6, 24, 18, 15, 12, 17]: 2008

Iteration: 32

Best Answer: [24, 18, 14, 17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 5, 11, 27, 20, 0, 22, 26, 7, 23, 15, 6]: 1971

Iteration: 33

Best Answer: [24, 18, 14, 17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 5, 11, 27, 20, 0, 22, 26, 7, 23, 15, 6]: 1971

Iteration: 34

Best Answer: [24, 18, 14, 17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 5, 11, 27, 20, 0, 22, 26, 7, 23, 15, 6]: 1971

Iteration: 35

Best Answer: [17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 22, 26, 15, 6, 24, 18, 14]: 1911

Iteration: 36

Best Answer: [17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 22, 26, 15, 6, 24, 18, 14]: 1911

Iteration: 37

Best Answer: [17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 22, 26, 15, 6, 24, 18, 14]: 1911

Iteration: 38

Best Answer: [17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 22, 26, 15, 6, 24, 18, 14]: 1911

Iteration: 39

Best Answer: [17, 10, 16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 22, 26, 15, 6, 24, 18, 14]: 1911

Iteration: 40

Best Answer: [16, 21, 13, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 14, 17, 10]: 1873

Iteration: 41

Best Answer: [13, 16, 21, 14, 3, 9, 19, 28, 25, 2, 1, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 17, 10]: 1833

Iteration: 42

Best Answer: [13, 16, 21, 14, 3, 9, 19, 28, 25, 2, 1, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 17, 10]: 1833

Iteration: 43

Best Answer: [13, 17, 14, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 16, 21]: 1797

Iteration: 44

Best Answer: [13, 17, 14, 3, 9, 19, 12, 1, 28, 25, 2, 4, 8, 20, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 16, 21]: 1797

Iteration: 45

Best Answer: [13, 16, 21, 14, 3, 9, 19, 1, 28, 2, 25, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 17, 10]: 1786

Iteration: 46

Best Answer: [21, 16, 17, 13, 3, 9, 19, 1, 28, 2, 25, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 14, 10]: 1726

Iteration: 47

Best Answer: [21, 16, 17, 13, 3, 9, 19, 1, 28, 2, 25, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 14, 10]: 1726  
Iteration: 48  
Best Answer: [21, 16, 17, 13, 3, 9, 19, 1, 28, 2, 25, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 14, 10]: 1726  
Iteration: 49  
Best Answer: [21, 16, 17, 13, 3, 9, 19, 1, 28, 2, 25, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 14, 10]: 1726  
Iteration: 50  
Best Answer: [21, 16, 17, 13, 3, 9, 19, 1, 28, 2, 25, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 14, 10]: 1726  
Iteration: 51  
Best Answer: [21, 16, 17, 13, 3, 9, 19, 1, 28, 2, 25, 20, 4, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 18, 15, 12, 14, 10]: 1726  
Iteration: 52  
Best Answer: [3, 9, 19, 1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 21, 16, 13, 17, 14, 12]: 1671  
Iteration: 53  
Best Answer: [3, 9, 19, 1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 21, 16, 13, 17, 14, 12]: 1671  
Iteration: 54  
Best Answer: [1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 9, 12, 19]: 1656  
Iteration: 55  
Best Answer: [1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 9, 12, 19]: 1656  
Iteration: 56  
Best Answer: [1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 15, 12, 18, 10, 21, 16, 13, 17, 14, 3, 9, 19]: 1641  
Iteration: 57  
Best Answer: [1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 15, 12, 18, 10, 21, 16, 13, 17, 14, 3, 9, 19]: 1641  
Iteration: 58  
Best Answer: [1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 22, 6, 24, 15, 12, 18, 10, 21, 16, 13, 17, 14, 3, 9, 19]: 1641  
Iteration: 59  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1631  
Iteration: 60  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22, 6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1631  
Iteration: 61

Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1631  
Iteration: 62  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 5, 11, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1631  
Iteration: 63  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1624  
Iteration: 64  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1624  
Iteration: 65  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1624  
Iteration: 66  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1624  
Iteration: 67  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1624  
Iteration: 68  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1624  
Iteration: 69  
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 23, 7, 26, 15, 22,  
6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 12, 9]: 1624