حسابگری زیستی؛
حل مسئله فروشنده دوره‌گرد به کمک الگوریتم ژنتیک؛
کامیار میرزاوزیری؛ 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 فروشندهٔ 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 فروشندهٔ 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 نهایتا 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):
```

```
        global 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 مرحلهٔ 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 هزینهٔ 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):
    global tsp
    if not self.__cache_is_valid:
        self.__cached_cost = 0
        for i in range(len(self.__data)):
            self.__cached_cost += 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 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.

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

```
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 هزینهٔ 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 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
```

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

    start = min(side1, side2)
    end = max(side1, side2)
    if PRINT_SLICE_INFO:
        print(start, end)

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

    second_child = Chromosome()
    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()
c2 = Chromosome()
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):
    global tsp
```

```
        result = Chromosome()
        result.__data = self.__data

        count = random.randint(0, MUTATION_DEGREE)
        for _ in range(count):
            (src, dst) = random.sample(range(tsp.dimension), 2)
            if PRINT_SLICE_INFO:
                print(src, dst)
            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()
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
```

توجه کنیم که در جهشی که پیاده‌سازی کردیم می‌تواند به تعدادی تصادفی جهش درجی رخ دهد. که در این مثال برای سادگی این تعداد را یک در نظر گرفتیم.

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 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()
```

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.

```
if improvement < IMPROVEMENT_THRESHOLD:
    stagnancy += 1
    if stagnancy == math.floor(STAGNANCY_THRESHOLD / 2):
        MUTATION_DEGREE = math.floor(MUTATION_DEGREE *
STAGNANCY_ESCAPE_DEGREE)
    if stagnancy >= math.floor(STAGNANCY_THRESHOLD / 2):
        population.escape_stagnancy(STAGNANCY_ESCAPE_PROPORTION)
    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()
```

نکته قابل توجه و خلاقانه در این قسمت این است که هنگامی که به نیمی از آستانه رکود می‌رسیم، از متدهایی برای فرار از رکود و پیش‌گیری توقف استفاده می‌کنیم. برای فرار از رکود لازم است به نوعی تنوع را در جامعه زیاد کنیم و از همگرایی زودرس جلوگیری کنیم. به این منظور اولا، حداکثر تعداد جهش‌های درجی که روی یک کروموزوم می‌تواند انجام شود را در پارامتر خاصی ضرب می‌کنیم، و همچنین درصدی از پایین جامعه را حذف و با کروموزوم‌های تصادفی جایگزین می‌کنیم تا از رکود جلوگیری شود. این راهکارهای ابتکاری با توجه به مشاهدات انجام شده توانست تأثیر به‌سزایی در اجرای الگوریتم بگذارد.

```
def escape_stagnancy(self, proportion: float):
    count = math.floor(len(self.__data) * proportion)
    self.__data[:count] = [Chromosome() for _ in range(count)]
    self.__data.sort()
```

۹ کد نهایی

در نهایت کد زیر را برای حل سؤال می‌نویسیم.

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

    if VERBOSE_LEVEL > -1:
        print(f"Iteration: {i}")
    if VERBOSE_LEVEL > 0:
        print(f"Best Answer: {population.answer()}")
    elif VERBOSE_LEVEL == 0:
        print(f"Best Answer: {population.answer().cost()}")
    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 را با آزمون و خطا به دست می‌آوریم. به نظر می‌رسد هرچه جمعیت بزرگ‌تر باشد الگوریتم پاسخ بهتری می‌دهد چرا که دیرتر همگرا می‌شود. از طرفی آستانهٔ رکود و بهبودی در حد کم کافی به نظر می‌رسد و زیاد کردن آن تفاوت چندانی در عملکرد ندارد.

برای جلوگیری از همگرایی زودرس باید احتمال جهش را بسیار بالا و سهم فرزندان در هر نسل را پایین در نظر بگیریم.

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:
        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.

bayg29 ۱.۱۰

پارامترها را به صورت زیر مقداردهی می‌کنیم.

```
N = 240
MUTATION_DEGREE = 9
REPLACEMENT = [.1, .4, .4]
STAGNANCY_ESCAPE_DEGREE = 2
STAGNANCY_ESCAPE_PROPORTION = 0.9
IMPROVEMENT_THRESHOLD = 1
STAGNANCY_THRESHOLD = 40
```

اجرای الگوریتم پس از 133 نسل متوقف می‌شود و پاسخ 1620 را می‌یابد.

```
Iteration: 133
Best Answer: [19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27, 0, 12, 15, 23, 7, 26,
   22, 6, 24, 18, 10, 21, 16, 13, 17, 14, 3, 9]: 1620
```

gr229 ۲.۱۰

برای پاسخ به این مسئله و مسئلهٔ بعدی که حجم بزرگ‌تری دارند لازم است الگوریتم را سریع‌تر کنیم. با یک بررسی ساده و اندازه‌گیری مراحل مختلف متوجه می‌شویم بیشترین زمان صرف crossover می‌شود که خوشبختانه به راحتی می‌توانیم آن را به صورت موازی انجام دهیم.

```
def __crossover(self):
    parents = self.__choose()
    random.shuffle(parents)

    P_COUNT = os.cpu_count()

    def pair_chunk_calculator(i, pair_chunk, rd):
        rd[i] = (sum([pair[0] * pair[1] for pair in pair_chunk], []))

    pair_chunks = numpy.array_split([[parents[i], parents[i + 1]] for i in
   range(0, len(parents) - 1, 2)], P_COUNT)
    manager = mp.Manager()
    rd = manager.dict()
    processes = [mp.Process(
        target=pair_chunk_calculator,
        args=(i, pair_chunks[i], rd)
    ) for i in range(P_COUNT)]

    for p in processes:
        p.start()
```

```
    for p in processes:
        p.join()

    return Population(sum(rd.values(), []))
```

حال می‌توانیم با سرعت بسیار بیشتری این پردازش را به صورت موازی پیش ببریم.
پارامترها را به صورت زیر مقداردهی می‌کنیم.

```
N = 240
MUTATION_DEGREE = 9
REPLACEMENT = [.1, .4, .4]
STAGNANCY_ESCAPE_DEGREE = 2
STAGNANCY_ESCAPE_PROPORTION = 0.9
IMPROVEMENT_THRESHOLD = 1
STAGNANCY_THRESHOLD = 40
```

اجرای الگوریتم پس از 1088 نسل متوقف می‌شود و پاسخ 2353 را می‌یابد.

```
Iteration: 1088
Best Answer: [127, 124, 123, 97, 114, 117, 120, 121, 118, 106, 96, 94, 95,
   98, 73, 89, 88, 87, 86, 80, 78, 76, 72, 71, 70, 69, 57, 15, 16, 11, 1,
   25, 21, 12, 0, 2, 3, 4, 5, 9, 52, 50, 223, 224, 217, 216, 226, 225, 227,
    228, 51, 58, 53, 54, 56, 82, 83, 81, 105, 111, 93, 102, 101, 100, 99,
   92, 91, 24, 19, 18, 10, 8, 6, 7, 17, 55, 22, 23, 84, 85, 79, 62, 64, 65,
    63, 61, 59, 60, 66, 67, 68, 77, 75, 74, 90, 34, 32, 33, 103, 107, 109,
   113, 128, 119, 115, 112, 125, 130, 168, 166, 167, 176, 178, 39, 38, 40,
   180, 185, 184, 182, 162, 134, 135, 136, 137, 138, 152, 153, 154, 148,
   147, 146, 145, 151, 150, 149, 204, 210, 202, 157, 200, 156, 144, 143,
   142, 140, 139, 141, 122, 116, 126, 129, 131, 110, 108, 104, 36, 35, 20,
   14, 13, 26, 27, 30, 29, 28, 31, 37, 165, 132, 133, 161, 177, 183, 46,
   44, 189, 188, 187, 193, 191, 198, 172, 171, 163, 160, 159, 155, 158,
   203, 211, 205, 206, 219, 209, 208, 207, 215, 214, 213, 212, 218, 220,
   222, 221, 201, 199, 194, 195, 186, 49, 47, 45, 41, 42, 43, 48, 190, 192,
    196, 197, 164, 175, 174, 173, 181, 179, 169, 170]: 2353
```

pr1002 ۳.۱۰

پارامترها را به صورت زیر مقداردهی می‌کنیم.

```
N = 300
MUTATION_DEGREE = 9
REPLACEMENT = [.1, .4, .4]
STAGNANCY_ESCAPE_DEGREE = 2
```

```
STAGNANCY_ESCAPE_PROPORTION = 0.9
IMPROVEMENT_THRESHOLD = 1_000
STAGNANCY_THRESHOLD = 8
```

اجرای الگوریتم پس از 247 نسل متوقف می‌شود و پاسخ 3996150 را می‌یابد.

```
Iteration: 247
Best Answer: [
    700, 988, 754, 660, 737, 676, 670, 790, 997, 160, 559, 583, 220, 832,
    689, 677, 427, 324, 303, 128, 345, 762, 925, 663, 640, 555, 447, 576,
    554, 713, 619, 376, 355, 44, 127, 297, 359, 386, 628, 207, 178, 144,
    244, 558, 548, 870, 864, 991, 222, 391, 104, 296, 516, 494, 174, 138,
    243, 569, 705, 749, 586, 282, 871, 973, 744, 431, 423, 993, 302, 43,
    37, 11, 570, 485, 567, 778, 928, 751, 587, 239, 245, 0, 167, 820,
    182, 199, 824, 813, 844, 858, 878, 806, 572, 286, 413, 333, 25, 14,
    597, 638, 568, 954, 903, 794, 471, 461, 265, 818, 389, 994, 52, 742,
    468, 719, 963, 946, 780, 378, 945, 936, 704, 432, 601, 743, 659, 776,
    690, 784, 992, 771, 456, 197, 484, 478, 733, 711, 933, 850, 834, 593,
    981, 615, 758, 917, 797, 594, 337, 644, 836, 912, 739, 655, 748, 48,
    354, 523, 854, 828, 1000, 606, 173, 161, 216, 15, 403, 102, 374, 368,
    649, 340, 783, 753, 283, 371, 147, 443, 487, 517, 956, 356, 65, 255,
    32, 990, 860, 941, 763, 696, 938, 915, 228, 211, 189, 157, 272, 463,
    652, 907, 213, 486, 519, 187, 241, 120, 592, 508, 261, 496, 514, 550,
    561, 268, 231, 141, 95, 135, 108, 257, 566, 35, 351, 698, 126, 452,
    880, 610, 505, 185, 346, 29, 23, 318, 294, 152, 464, 430, 861, 849,
    775, 887, 405, 177, 162, 4, 2, 46, 39, 599, 799, 675, 654, 402,
    395, 725, 947, 959, 580, 205, 208, 829, 816, 811, 489, 101, 92, 73,
    76, 133, 504, 975, 863, 625, 740, 875, 934, 709, 662, 658, 396, 290,
    59, 19, 293, 349, 38, 454, 411, 419, 407, 699, 117, 130, 473, 843,
    913, 842, 998, 506, 520, 563, 480, 299, 275, 458, 650, 636, 172, 63,
    287, 417, 380, 814, 759, 908, 596, 590, 920, 960, 931, 627, 620, 603,
    477, 886, 591, 530, 276, 312, 64, 308, 165, 212, 770, 767, 618, 490,
    564, 789, 761, 694, 716, 428, 536, 479, 158, 251, 166, 278, 450, 935,
    951, 726, 944, 787, 755, 273, 466, 70, 96, 106, 204, 540, 13, 383,
    795, 792, 129, 113, 87, 51, 201, 175, 159, 119, 651, 741, 465, 353,
    788, 680, 358, 962, 950, 924, 702, 687, 472, 442, 515, 560, 524, 551,
    873, 722, 579, 285, 89, 226, 192, 56, 394, 684, 766, 624, 315, 79,
    237, 225, 292, 323, 420, 71, 710, 961, 793, 622, 451, 331, 140, 277,
    661, 321, 614, 311, 154, 264, 898, 470, 475, 256, 503, 453, 577, 246,
    377, 634, 791, 686, 695, 674, 721, 890, 821, 807, 643, 656, 341, 425,
    41, 234, 66, 36, 888, 868, 852, 896, 892, 635, 90, 259, 926, 966,
    798, 444, 334, 968, 845, 511, 756, 57, 31, 295, 325, 557, 229, 269,
    193, 215, 217, 100, 328, 607, 728, 985, 382, 481, 549, 556, 510, 547,
    876, 840, 918, 929, 785, 708, 693, 669, 720, 715, 642, 344, 859, 553,
```

```
    909, 545, 529, 267, 202, 562, 581, 439, 441, 629, 288, 367, 440, 429,
    717, 573, 571, 238, 248, 872, 964, 802, 809, 613, 865, 808, 804, 769,
    980, 972, 703, 727, 667, 250, 139, 75, 82, 317, 681, 692, 989, 322,
    67, 313, 47, 103, 348, 927, 969, 701, 948, 841, 891, 851, 855, 831,
    957, 404, 114, 109, 26, 1, 242, 145, 459, 194, 8, 284, 94, 300,
    535, 314, 513, 819, 399, 360, 171, 179, 565, 902, 921, 846, 866, 877,
    916, 835, 280, 500, 538, 434, 899, 882, 543, 526, 236, 203, 270, 93,
    457, 879, 731, 874, 955, 952, 847, 901, 848, 839, 532, 137, 316, 600,
    774, 949, 786, 352, 83, 77, 9, 111, 326, 488, 582, 379, 61, 69,
    291, 330, 22, 97, 773, 1001, 539, 781, 923, 884, 528, 900, 668, 967,
    986, 723, 653, 697, 803, 678, 747, 772, 919, 626, 156, 118, 262, 546,
    768, 970, 664, 339, 281, 27, 632, 996, 437, 460, 306, 7, 602, 416,
    397, 393, 357, 33, 525, 151, 146, 426, 28, 301, 143, 375, 630, 148,
    304, 319, 289, 332, 134, 385, 49, 34, 80, 729, 685, 932, 800, 534,
    448, 263, 122, 412, 415, 329, 170, 125, 110, 683, 976, 738, 688, 665,
    735, 679, 123, 455, 574, 365, 645, 521, 531, 853, 910, 712, 971, 88,
    115, 497, 498, 209, 219, 258, 195, 235, 240, 398, 414, 168, 221, 271,
    641, 724, 779, 502, 482, 499, 132, 91, 105, 17, 42, 274, 418, 492,
    116, 252, 150, 99, 646, 507, 483, 522, 830, 232, 214, 223, 74, 469,
    249, 612, 409, 279, 16, 364, 889, 837, 826, 552, 922, 433, 764, 965,
    930, 604, 765, 897, 867, 777, 883, 584, 589, 943, 978, 760, 914, 895,
    801, 639, 939, 730, 856, 107, 343, 942, 390, 633, 436, 392, 647, 757,
    805, 823, 298, 163, 501, 218, 45, 198, 183, 200, 616, 362, 608, 384,
    363, 335, 86, 336, 691, 62, 58, 50, 462, 186, 260, 98, 81, 347,
    369, 575, 578, 493, 169, 153, 491, 937, 953, 474, 885, 881, 822, 815,
    982, 387, 810, 595, 585, 611, 648, 621, 361, 342, 940, 827, 812, 527,
    893, 906, 833, 68, 10, 12, 401, 112, 372, 637, 388, 350, 410, 18,
    979, 750, 435, 732, 734, 495, 424, 230, 210, 544, 476, 911, 512, 542,
    533, 894, 838, 825, 537, 857, 180, 320, 6, 987, 983, 605, 623, 745,
    438, 366, 84, 196, 227, 233, 266, 310, 136, 78, 55, 254, 124, 24,
    305, 671, 672, 999, 974, 746, 657, 307, 406, 408, 752, 682, 673, 617,
    707, 588, 467, 191, 20, 53, 54, 381, 714, 706, 373, 631, 188, 155,
    164, 176, 181, 40, 72, 247, 862, 736, 224, 309, 5, 400, 609, 422,
    509, 541, 782, 904, 905, 518, 598, 817, 869, 30, 449, 85, 184, 149,
    131, 121, 445, 796, 718, 338, 190, 206, 995, 142, 666, 984, 421, 253,
    977, 446, 958, 60, 21, 327, 3, 370
]: 3996150
```