حسابگری زیستی؛
حل مسئله برش تنه به کمک الگوریتم تبرید تدریجی؛
کامیار میرزاوزیری؛ ۶۱۰۳۹۶۱۵۲

---

۱ خوانش مسئله و نمایش پاسخ

ابتدا کلاس IO را پیاده‌سازی می‌کنیم که مسئله را از فایل ورودی خوانده و نتیجه را در فایل خروجی ذخیره کند.

```
class IO:
    @classmethod
    def init(cls, num):
        cls.num = num
        cls.o = open(f'output{cls.num}.stock', 'w')

    @classmethod
    def read(cls):
        with open(f'input{cls.num}.stock') as f:
            t = f.read()
        return t

    @classmethod
    def print(cls, t='', to_file=False):
        print(t)
        if to_file:
            cls.o.write(str(t) + '\n')
```

در constructor این کلاس شماره مسئله را به عنوان ورودی دریافت می‌کنیم و فایل خروجی را باز می‌کنیم. همچنین متد read مسئله ورودی را خوانده و به صورت خام برمی‌گرداند. متد print نیز اطلاعات را روی iostream چاپ می‌کند. همچنین در صورتی که پارامتر دوم این متد True باشد اطلاعات را در فایل خروجی نیز وارد می‌کند. برای اطلاعات غیرضروری مانند جزئیات هر تکرار، این پارامتر را False در نظر می‌گیریم و تنها پاسخ نهایی مسئله را در فایل ذخیره می‌کنیم.

همچنین نیاز داریم تا پس از خواندن ورودی خام، آن را parse کنیم. این کار را به کمک قابلیت regex در پایتون انجام می‌دهیم.

```
# Read length and requests from input file
NUM = 5
IO.init(NUM)
raw_input = IO.read()
length = int(re.findall(r'Stock Length:\s*(\d+)', raw_input)[0])
requests = list(map(int, re.findall(r'Requests:((\s*\d+,)*\s*(\d+)?)',
    raw_input)[0][0].split(',')))
```

۲ کلاس تنه

کلاسی برای نمایندگی تنه‌ها (کاغذ، تیرآهن، ...) در پایتون پیاده‌سازی می‌کنیم.

```
class Stock:
    def __init__(self, length=None):
        if length is not None:
            self.__free_length = length
            self.__cuts = []
    def copy(self):
        new = Stock()
        new.__free_length = self.__free_length
        new.__cuts = self.__cuts.copy()
        return new
```

متد constructor این کلاس، به عنوان تنها ورودی طول را دریافت می‌کند. در هر instance از این کلاس، لیستی از برش‌های آن را، به همراه فضای خالی باقی‌مانده نگه می‌داریم. هرچند فضای خالی از روی برش‌ها و طول کل قابل محاسبه است اما به این روش هزینه محاسباتی کم‌تری متحول می‌شویم و در هر لحظه به فضای باقی‌مانده دسترسی داریم. همچنین متد copy را برای کپی کردن یک تنه پیاده‌سازی می‌کنیم. این متد زمانی که می‌خواهیم پاسخی در همسایگی یک پاسخ ایجاد کنیم کاربردی است چرا که نمی‌خواهیم پاسخ اصلی تغییری کند اما پاسخ جدیدی مانند آن با کمی تغییر ایجاد شود پس برای حل مشکل mutable بودن instanceها در پایتون به این متد نیاز خواهیم داشت. دو متد بسیار ساده add و remove را به این کلاس اضافه می‌کنیم. توجه کنیم که در هر متد لازم است همزمان به اضافه یا کم کردن یک برش به تنه، فضای آزاد را تغییر دهیم، و همچنین هنگام اضافه کردن برش دقت کنیم که فضای آزاد کافی وجود داشته باشد.

```
def add(self, length):
    if self.__free_length >= length:
        self.__free_length -= length
        self.__cuts.append(length)
    else:
        raise Exception('insufficient free length')
def remove(self, index):
    self.__free_length += self.__cuts[index]
    self.__cuts.remove(self.__cuts[index])
```

متد ساده empty را نیز برای بررسی خالی بودن یک تنه اضافه می‌کنیم که تنه‌هایی که در طی جست‌وجو خالی می‌شوند را حذف کنیم و هنگامی که هزینه یک پاسخ را بر اساس تعداد تنه‌های آن پاسخ محاسبه می‌کنیم هزینه درستی بیابیم.

```
def empty(self):
    return not self.__cuts
```

متدهای شناخته‌شده __str__ و __repr__ را نیز برای نمایش یک تنه پیاده‌سازی می‌کنیم که در فایل سورس موجود است.

۳ پاسخ

برای نمایندگی پاسخ‌های مسئله کلاسی به اسم Answer پیاده‌سازی می‌کنیم که مشخصا شامل لیستی از تنه‌ها می‌باشد. مقداردهی instanceهای این کلاس را به صورت تصادفی انجام می‌دهیم. به این صورت که لیست درخواست‌ها را در constructor به عنوان ورودی دریافت می‌کنیم و این لیست را بر می‌زنیم. سپس از ابتدا شروع می‌کنیم و تا جایی که ممکن است درخواست‌ها را به یک تنه اختصاص می‌دهیم و هر زمان که این کار ممکن نبود یک تنه جدید ایجاد می‌کنیم.

```
class Answer:
    def __init__(self, length=None, requests=None):
        if length is not None and requests is not None:
            self.length = length
            self.__stocks = []
            indexes = list(range(len(requests)))
            random.shuffle(indexes)
            for i in indexes:
                try:
                    self.__stocks[-1].add(requests[i])
                except:
                    self.__stocks.append(Stock(length))
                    self.__stocks[-1].add(requests[i])
    def copy(self):
        new = Answer()
        new.length = self.length
        new.__stocks = [stock.copy() for stock in self.__stocks]
        return new
```

متد copy را نیز به برای ایجاد یک پاسخ مانند پاسخ فعلی اما با قابلیت تغییر ایجاد می‌کنیم. می‌بینیم که در اینجا از متد کپی که روی Stock تعریف شده بود استفاده کردیم.

متد ساده زیر نیز برای محاسبه هزینه هر پاسخ به کار می‌رود که برابر تعداد تنه‌های پاسخ می‌باشد.

```
def stocks_count(self):
    return len(self.__stocks)
```

مشابه بخش قبل، متدهای __str__ و __repr__ را نیز برای نمایش پیاده‌سازی می‌کنیم.

```
def __str__(self):
    return 'Number of Stocks: ' + str(self.stocks_count()) + '\n' + 'Stocks
    :\n' + ''.join(map(str, self.__stocks)) + '\n'

def __repr__(self):
    return self.__str__()
```

۴ جست‌وجو در همسایگی

برای یافتن پاسخ‌های نزدیک به یک پاسخ، سعی می‌کنم یک برش تصادفی از یک تنه تصافی پاسخ انتخاب کنیم و آن را به یک تنه تصادفی دیگر منتقل کنیم. اما توجه کنیم که در این صورت هیچ‌گاه پاسخ بدتر نخواهد شد چرا که هزینه پاسخ برابر تعداد تنه‌های موجود در پاسخ است و در این صورت این تعداد هرگز بیشتر نخواهد شد لذا الگوریتم تبرید تدریجی بی‌معنی است و در واقع در حال پیاده‌سازی الگوریتم Hill Climbing می‌باشیم.

برای حل این مشکل زمانی که می‌خواهیم یک برش را به یک تنه دیگر منتقل کنیم، با احتمال کمی، یک تنه جدید می‌سازیم. اگر فرض کنیم n تنه داریم، می‌توانیم یک تنه خالی به این لیست اضافه کنیم که در این صورت احتمال انتخاب این تنه در اردر $\frac{1}{n}$ خواهد بود. پس با احتمال $\frac{1}{n}$ یک تنه جدید ایجاد می‌کنیم و قطعه را در آن تنه قرار می‌دهیم و در غیر این صورت دو تا از تنه‌های موجود را برای انجام عملیات استفاده می‌کنیم.

از آنجا که تنها یک جابجایی تغییر بسزایی ایجاد نمی‌کند تعداد این جابجایی‌ها را عدد بزرگ‌تری انتخاب می‌کنیم تا قابلیت پویش را افزایش دهیم و فضای بیشتری در اطراف را مورد بررسی قرار دهیم. برای این منظور پارامتری به اسم MUTATION_DEGREE در نظر می‌گیریم و هر بار عددی تصادفی بین ۱ و این پارامتر انتخاب می‌کنیم و آن قدر تلاش می‌کنیم یک برش را منتقل کنیم تا به این تعداد بار انتقال موفق انجام شود. (اگر تنه مقصد فضای خالی نداشته باشد برش ناموفق خواهد بود.)

پس متد mutate را برای کلاس Answer به صورت زیر تعریف می‌کنیم که پاسخی مشابه خودش خروجی می‌دهد با تعدادی جابجایی برش.

```
def mutate(self, degree):
    count = random.randint(1, degree)
    new = self.copy()
    while count > 0:
        if random.random() > 1 / self.stocks_count():
            stock_0, stock_1 = random.choices(new.__stocks, k=2)
        else:
            stock_0 = random.choice(new.__stocks)
            stock_1 = Stock(new.length)
            new.__stocks.append(stock_1)

        if stock_0.transfer_a_cut_to(stock_1):
            count -= 1
        if stock_0.empty():
            new.__stocks.remove(stock_0)
        if stock_1.empty():
            new.__stocks.remove(stock_1)
    return new
```

توجه کنیم که در نهایت اگر تنه‌هایی که در آن‌ها تغییر ایجاد کردیم خالی شدند آن‌ها را حذف می‌کنیم.

می‌بینیم که در کد بالا از متد transfer_a_cut_to بر روی کلاس Stock استفاده شده. این متد به صورت زیر پیاده‌سازی شده‌است.

```
def transfer_a_cut_to(self, other):
    cut_index = random.randrange(len(self.__cuts))
    try:
        other.add(self.__cuts[cut_index])
        self.remove(cut_index)
        return True
    except:
        return False
```

۵ فرار از رکود و بهینه‌های محلی

در این تمرین از روشی ابتکاری به اسم chaos یا آشوب استفاده کردیم تا اگر در بهینه محلی گیر کردیم بتوانیم فرار کنیم. الگوریتم مشابه تبرید تدریجی کلاسیک پیاده‌سازی شده با این تفاوت که در هر تکرار الگوریتم، چنانچه پاسخ جدید مشابه پاسخ قبلی بود معیاری به اسم stagnancy را افزایش می‌دهیم که از همین معیار برای شرط توقف نیز استفاده می‌کنیم.

اما تفاوت این الگوریتم این است که در هر تکراری که در رکود می‌گذراند معیاری به اسم chaos را به میزان پارامتری به اسم CHAOS_DEGREE افزایش می‌دهد. و این پارامتر را به عنوان ضریبی برای دما در نظر می‌گیرد. یعنی دما به صورت تدریجی کم می‌شود اما هرگاه در یک نقطه راکد شدیم، دما را به صورت تدریجی زیاد می‌کنیم (الزاما زیاد نمی‌کنیم، در بعضی موارد سرعت کم شدن کم می‌شود) به امید آن که با بالا نگه داشتن دما بتوانیم از بهینه محلی بگریزیم.

با توجه به مشاهدات، ایجاد آشوب در بعضی موارد به خصوص مسئله شماره ۵ بسیار موثر واقع شد و موفق شدیم بارها از بهینه محلی بگریزیم و نقاط بهتری پیدا کنیم. این ایده در طبیعت نیز وجود دارد که آشوب باعث بهینگی بیشتری در انتها می‌شود.

۶ پیاده‌سازی

کد پیاده‌سازی این الگوریتم به صورت زیر خواهد شد.

```
# Run the Simulated Annealing
start = time.time()
TEMPERATURE = lambda iteration, chaos: .99 ** (iteration / 100) * chaos
MUTATION_DEGREE = 5
STAGNANCY_THRESHOLD = 5000
CHAOS_DEGREE = .1

stagnancy = 0
current = Answer(length, requests)
best = current
last = current
iteration = 0
chaos = 1
while True:
    temperature = TEMPERATURE(iteration, chaos)
    IO.print(f'{iteration}\t{current.stocks_count()}\t{temperature}')
    iteration += 1
    new = current.mutate(MUTATION_DEGREE)
    delta = new.stocks_count() - current.stocks_count()
    if delta < 0 or random.random() < math.exp(-delta / temperature):
        current = new
    if last.stocks_count() == current.stocks_count():
        stagnancy += 1
        chaos += CHAOS_DEGREE
    else:
        stagnancy = 0
        chaos = 1
    if stagnancy > STAGNANCY_THRESHOLD:
        break
    last = current
    if current.stocks_count() < best.stocks_count():
        best = current

IO.print()
IO.print(best, True)
IO.print(f'Iterations: {iteration}', True)
IO.print(f'Time: {time.time() - start}', True)
IO.o.close()
```

۷ بررسی نتایج

۱.۷ مسئله اول

با اجرای الگوریتم در 65.13 ثانیه و پس از 89706 تکرار به پاسخ 51 می‌رسیم، جزئیات برش‌ها در فایل output1.stock موجود است.

۲.۷ مسئله دوم

با اجرای الگوریتم در 56.13 ثانیه و پس از 90305 تکرار به پاسخ 78 می‌رسیم، جزئیات برش‌ها در فایل output2.stock موجود است.

۳.۷ مسئله سوم

با اجرای الگوریتم در 08.4 ثانیه و پس از 83447 تکرار به پاسخ 101 می‌رسیم، جزئیات برش‌ها در فایل output3.stock موجود است.

۴.۷ مسئله چهارم

با اجرای الگوریتم در 56.10 ثانیه و پس از 84112 تکرار به پاسخ 222 می‌رسیم، جزئیات برش‌ها در فایل output4.stock موجود است.

۵.۷ مسئله پنجم

با اجرای الگوریتم در 28.491 ثانیه و پس از 135750 تکرار به پاسخ 4325 می‌رسیم، جزئیات برش‌ها در فایل output5.stock موجود است.