تعریف ژن و کروموزوم در این پروژه به اینصورت است که هر ژن برابر یک operator و یا operand است و از کنار هم قرار گرفتن این ها یک کروموزم به تعداد equation داده شده، تشکیل میشود.

تولید جمعیت اولیه در این پروژه به اینصورت است که ابتدا تعداد operator و operand هارا مشخص کرده و سپس به تعداد populationSize کروموزوم تولید میکنیم. تولید هر کروموزوم به اینصورت است که به تعداد operator و operand ها داده تصادفی تولید کرده و سپس مقدار فاینال هر کروموزوم را بدست آورده و در دیکشنری داده های هر یک را ذخیره میکنیم.

```
def makeFirstPopulation(self):
        numberOfOperands = int(self.equationLength / 2) + 1
        numberOfOperators = self.equationLength - numberOfOperands
        data = []
        for j in range(populationSize):
            randomOperands = [int(x) for x in random.choices(self.operands, k=numberOfOperands)]
            randomOperators = random.choices(self.operators, k=numberOfOperators)
            stringOps = ''
            for i in range(numberOfOperators):
                temp = str(randomOperands[i]) + randomOperators[i]
                stringOps +=  temp
            stringOps += str(randomOperands[numberOfOperands - 1])
            finalNumber = eval(stringOps)
            data.append({'stringOps': stringOps,
                         'answer': finalNumber,
                         'difference': 0})
        return data
```

پیاده سازی تابع fitness در این پروژه به اینصورت است که اختلاف هر کروموزوم را با goalNumber محاسبه کرده و آنرا در قسمت difference دیکشنری ذخیره میکنیم.

```
for i in range(populationSize):
                self.population[i]['difference'] = abs(self.goalNumber - self.population[i]['answer'])
                if self.population[i]['answer'] == self.goalNumber:
                    print(self.population[i]['stringOps'])
                    isFind = True
                    break
```

تشکیل matingPool در این پروژه به اینصورت است که ابتدا داده هارا بر اساس difference به ترتیب صعودی سورت میکنیم و سپس درصدی از داده هایی با اختلاف کمتر را جدا و ذخیره میکنیم. سپس در تابع matingPool بر اساس رتبه هر کروموزم آنرا کپی میکنیم. اینکار به این علت است که کروموزم هایی که اختلاف کمتری با جواب نهایی دارند و در واقع داده بهتری برای ما هستند تعداد بیشتری نسبت به کروموزوم های دیگر داشته باشند تا درصد انتخاب آنها در جمعیت نهایی افزایش یابد. سپس داده هارا شافل کرده و به تعداد populationSize داده هایی رندوم از آن انتخاب میکنیم.

```
carriedChromosomes = []
for i in range(int(populationSize*carryPercentage)):
    carriedChromosomes.append(self.population[i])

matingPool = self.createMatingPool()
random.shuffle(matingPool)
choosedData = random.choices(matingPool, k = populationSize - int(populationSize*carryPercentage))
```

```
def createMatingPool(self):
        matingPool = []
        for i in range(populationSize):
            for j in range(populationSize - i):
                matingPool.append(self.population[i])
        return matingPool
```

تشکیل crossoverPool به اینصورت است که ژن های دو کروموزوم را ترکیب کرده و دو فرزند از آنها تشکیل میدهیم. برای اینکار دو عضور رندوم از matingPool را انتخاب کرده، یک عدد رندوم تولید کرده و آنرا با crossoverProbability مقایسه میکنیم، اگر بیشتر بود که فرزندان کپی والدشان میشوند، در غیر اینصورت یک عدد رندوم در بازه equation تولید میشود و ژن ها از فرزندان به والدین انتقال داده میشوند.

```
def createCrossoverPool(self, matingPool):
        crossoverPool = []
        for i in range(len(matingPool)):
            if random.random() > crossoverProbability:
                crossoverPool.append(matingPool[i])
            else:
                idx0 , idx1 = random.sample(range(len(matingPool)), 2)
                rand = random.randint(0, self.equationLength - 1)
                for j in range(rand):
                    a = list(matingPool[idx0]['stringOps'])
                    b = list(matingPool[idx1]['stringOps'])
                    b[j] = matingPool[idx0]['stringOps'][j]
                    a[j] = matingPool[idx1]['stringOps'][j]
                    matingPool[idx0]['stringOps'] = "".join(a)
                    matingPool[idx1]['stringOps'] = "".join(b)
                crossoverPool.append(matingPool[i])
        return crossoverPool
```

پس از انجام crossover به تعداد populationSize – carriedchromosome عملیات mutation را روی کروموزم ها انجام میدهیم که به اینصورت است که به ازای هر کروموزوم یکی از ژن هارا تغییر میدهیم بصورتیکه عضو رندوم انتخابی را با توجه به operator و operand بودنش از بین عضو های انها انتخاب کرده و جایگزین میکنیم.

```
def mutate(self, chromosome):
    rand = random.randint(0, self.equationLength - 1)
    if rand % 2:
        randVar = random.choices(self.operators, k=1)
        a = list(chromosome['stringOps'])
        a[rand] = randVar[0]
        chromosome['stringOps'] = "".join(a)
    else:
        randVar = random.choices(self.operands, k=1)
        a = list(chromosome['stringOps'])
        a[rand] = randVar[0]
        chromosome['stringOps'] = "".join(a)
```

در نهایت کروموزم های انتخابی اولیه را به کروموزوم های بدست آمده نهایی اضافه کرده و به تعداد جمعیت اولیه کروموزم داریم.

```
for i in range(populationSize - int(populationSize*carryPercentage)):
    self.population.append(self.mutate(crossoverPool[i]))
self.population.extend(carriedChromosomes)
```

بخش سوالات

۱. **جمعیت اولیه بسیار کم یا بسیار زیاد چه مشکالتی را به وجود می آورند؟**

کم بودن جمع اولیه باعث میشود که کروموزوم هایی که تشکیل میشوند به اندازه کافی متفاوت و متنوع نباشند. زیاد بودن جمعیت اولیه نیز باعث میشود که زمان انجام عملیات ها و محاسبات مانند maitngPool، crossover و... بسیار زیاد شود و سرعت اجرا کاهش یابد.

۲. **اگر تعداد جمعیت در هر دوره افزایش یابد، چه تاثیری روی دقت و سرعت الگوریتم میگذارد؟**

انجام این کار سبب میشود که دقت الگوریتم بدلیل گستردگی و تنوع کروموزوم ها افزایش یابد اما در طرف دیگر سبب کاهش سرعت الگوریتم بدلیل زیاد شدن محاسبات شود و همچنین اگر تعداد جمعیت بسیار زیاد شود، ممکن است با مشکل کمبود حافظه مواجه شویم.

۳. **تاثیر هریک از عملیات های crossover و mutation را بیان و مقایسه کنید. آیا میتوان فقط یکی از آنها را استفاده کرد؟ چرا؟**

عملیات crossover باعث ایجاد کروموزوم فرزند از کروموزوم های والد میشود که اینکار سبب میشود فرزند ترکیبی از دو والد خود باشد که یعنی ممکن است فرزند نسبت به والد های خود ژن های بهتری داشته باشد. اما عملیات mutation به اینصورت عمل میکند که ژن های خود کروموزوم را با احتمال مشخصی تغییر میدهد. بنابراین استفاده از هر دوی این عملیات پاسخ بهتری به ما میدهد.

۴. **به نظر شما چه راهکارهایی برای سریعتر به جواب رسیدن در این مسئله خاص وجود دارد؟**

برای افزایش سرعت در مراحل اولیه میتوان داده اولیه را بصورت صحیح در جایگاه ها قرار داد، بصورتیکه operand ها و operator ها در ایندکس های درست خود قرار بگیرند. همچنین در تابع هایی نظیر crossover و mutation نیز همین ترکیب رعایت شود و ژن های والد که به فرزند داده میشوند به صورت درست در ایندکس های فرزند قرار بگیرند و در تعویض ژن ها نیز همین عمل تکرار و رعایت شود.

۵. **با وجود استفاده از این روش ها، باز هم ممکن است که کروموزوم ها پس از چند مرحله دیگر تغییر نکنند. دلیل این اتفاق و مشکلاتی که به وجود می آورد را شرح دهید. برای حل آن چه پیشنهادی می دهید.**

این اتفاق ممکن است بدلیل کاهش تنوع جمعیت رخ دهد و جمعیت در یک مینیمم محلی گیر میکند و احتمال حل مسئله کاهش میابد. برای حل این مشکل در صورت ثابت ماندن فیتنس، میتوان جمعیت جدید تولید کرد.

۶. **چه راه حلی برای تمام شدن برنامه در صورتی که مسئله جواب نداشته باشد، میدهید؟**

برای حل این مشکل میتوان یک عدد برای ماکسیمم محاسبه قرار دهیم، اگر تا زمانی که به آن عدد نرسیدیم پاسخی دریافت کردیم، یعنی مسئله پاسخ دارد، در غیراینصورت اگر به عدد ماکسیمم تعیین شده رسیدیم اما هنوز به پاسخ نرسیده بودیم، میتوانیم اعلام کنیم که مسئله جواب ندارد.