In this project, you’ll develop a genetic algorithm and use it to solve some progressively more complicated problems. In part 1a, we’ll focus on the basic implementation of the algorithm and the different selection mechanisms. In part 1b, we’ll apply the code developed in part 1a to some actual problems of interest and think more carefully about encoding a problem.

Warning! There is going to be a fairly significant amount of coding and experimentation in this project. I’ll provide you with descriptions of what to build and how to go about it, but you’re going to need to do the implementation yourself. This may take you a fair amount of work and time, so start early. Don’t expect to start this the night before it’s due and complete it.

What to turn in
You will need to turn in three things for this project. All of them will be due on Friday, October 10.

1. A hard copy of your source code.
2. A soft copy of your code, which should be placed in your submit directory:

   /home/submit/cs662/<your name>

   (you should make sure this directory exists before October 10, so as to avoid extra stress.)

3. A document containing graphs/discussion for each of the experiments you’re asked to conduct for this project. This should be professional-looking (i.e. not handwritten) and well-organized. Do not leave this until the last minute.

A note on graphing GA performance.
In several places in this assignment, I ask you to generate a graph of how your algorithm performs over time. For this, I’d recommend placing iterations on the x axis, and fitness on the y axis.

But what should you graph? (If this is the first time you’ve read this assignment, this may not make sense to you. Read the rest of the assignment and come back to this.) You could plot the best individual in a population, the worst, or the average.

I’d recommend plotting all three. Use a line to plot the average fitness of your population, and error bars to indicate the smallest and largest fitnesses at a particular time.

Part 1(a) - building a basic genetic algorithm.
Step 1
To start with, we'll build the basic components of a genetic algorithm in Python.

We'll start by building a Genome class. This will represent one bitstring, or individual, in a population. It should have three instance variables: a bitstring, a fitness, and a relative fitness.

It should have the following methods:

__init__(self, bitstring) :
- the constructor. This method should set up the chromosome's bitstring to be equal to the one provided.

mutate(self, prob) :
with probability prob, this method should flip one bit, chosen at random, in the bitstring.

evaluate(self, objectivefn) :
this method should apply the function objectivefn, provided as an argument, to the chromosome's bitstring, and set the fitness appropriately.

crossover(self, otherChrom, position) :
this method should perform single-point crossover with the other chromosome (otherChrom) being passed in at the position provided as an argument.

Once you've built a Chromosome, test it to make sure that all the methods work as expected.

Step 2
Now we're ready do build a simple genetic algorithm. this will also be a Python class, called GA. A GA should have the following instance variables: population, populationSize, inputLength (the length of the bitstrings), objective (the function to be optimized), fractionRetained (the fraction that will survive until the next generation), selectionMethod, and mutationRate.

It should have the following methods:

__init__(self):
- set up default values for instance variables.

makeInitialPopulation(self):
- fill the list population with randomly generated chromosomes. You will probably want to use the randint() function here.

selectChromosome(self) :
This method will be responsible for selecting a chromosome, either to be a parent or to survive on its own. To begin, implement proportional fitness, or roulette selection, where the probability of choosing a particular chromosome is equal to: (fitness / total fitness of all chromosomes). It should return the chromosome selected.

\[ \text{makeChildren (self)} : \]

For the moment, this function should choose two children, have them crossover with each other at a random point, and return a list containing the two new chromosomes.

\[ \text{runGA(self, iterations)} : \]

now you’re ready to write the main method. Generate an initial population, and for the appropriate number of generations, select a new population with proportional fitness selection combined with crossover, apply mutation, and repeat.

Reasonable values are: population size = 100, mutation rate = 5%.

Start by running your GA on the all-1s problem. This one is very easy, and so it’ll be a good function to work out the bugs. The fitness of a bitstring is simply the number of 1s it has. So, 01111 has a fitness of 4, and 00010 a fitness of 1. Try this with bitstrings of length 5,10, and 20.

Include a graph showing the average fitness of your population over time. How long does the GA take to converge? Is there a relationship between convergence time and string length?

**Step 3**

Now let’s make things a bit more interesting. First, we’ll consider an alternative selection method, called tournament selection. Rather than computing every chromosome’s relative fitness, we randomly choose two chromosomes and compare them. The high fitness wins, and gets to reproduce. To perform crossover, we conduct two tournaments and crossover the two winners.

Add tournament selection to the selectChromosome method. Include a graph showing the relative performance of tournament selection and roulette selection on the all-1s problem. Who does better?

One problem with our GA implementation is that it’s possible for good solutions to get lost - since they only survive until the next generation via crossover, performance can be slower than expected. One way around this is **elitism**; keeping the best \( n \) solutions from the previous generation. Elitism can be done deterministically (rank chromosomes by fitness and keep the top \( n \)) or probabilistically (by using the same roulette selection you implemented before.) This is an example of the exploration-exploitation problem - retaining more previous solutions reduces exploration, but allows more exploitation of past learning.

Implement probabilistic elitism in your GA, using roulette selection to retain \( N\% \) of the previous population. The remaining 100-N\% will be generated using
The crossover, as before. Graph your algorithm’s performance for \( N = 5, N = 20, N = 50, N = 75 \) on the all-1s problem.

**Step 4**

Now let’s try the GA on a slightly harder problem. Consider the N-queens problem described in Russell and Norvig. Recall that they solve this by assuming that each queen will be placed in one column. The problem is to determine the row for each queen. Since there are \( N \) rows and \( N \) columns, it requires \( \log_2 N \) bits to encode the row for each queen. (We’ll just use powers of two, to make things easier.)

So a possible potential solution for the 4-queens problem would be:

00 01 11 10 (or 0 1 3 2)

We’ll need a new fitness function for this problem. Following R & N, we’ll use the number of nonattacking pairs of queens as the fitness. In 4-queens, this has a maximum of 6, and in 8-queens, a maximum of 28.

Following is Python code for the N-queens fitness function. It is also available on the course webpage, under the project description.

```python
from math import *

### Python 2.3 allows log with an arbitrary base (not e) – if you have
### an earlier Python, this will compute log base 2.
def lg(x):
    return log(x) / log(2)

def base2Tobase10(bitstr):
    total = 0
    size = len(bitstr)
    for i in range(0, size):
        total = total + (bitstr[i] * pow(2,(size -1) - i))
    return total

### two queens threaten each other if they’re in the same row or
### on the same diagonal.
### assume row 0 is at the bottom of the board

def isThreatening(row1, col1, row2, col2):
    return (row1 == row2 or
            (abs(row2 - row1) == abs(col2 - col1)))
```

### nrows is the number of rows/columns on the chessboard
### Each queen uses \( \log_2 \) bits in bitstring, which we’ll
### assume is a list.
def nQueensFitness (bitstring, nrows) :
    queens = []
    fitness = 0
    bitsPerCol = int(lg(nrows))
    for i in range (0, nrows) :
        ### for simplicity, let’s convert rows to base 10
        queens.append(base2Tobase10(bitstring[i * bitsPerCol : (i + 1) * bitsPerCol]))

        ### for each queen, compare it to every queen greater than it.
        ### If they don’t threaten each other, add 1 to fitness.
        for i in range(0, len(queens)) :
            for j in range(i, len(queens)) :
                if (i != j) :
                    if not (isThreatening(i, queens[i], j, queens[j])) :
                        fitness = fitness + 1

    return fitness

For this problem, make any necessary changes to run your GA on this problem and try it with both 4 and 8 queens. Consider initial populations of 50, 100, and 200. Use tournament selection, retaining the top 10% of the previous generation with roulette selection. Use a mutation rate of 5%. How does the population size affect performance? How does the problem difficulty change as the board size increases?

**Step 5**

In some cases, regular crossover is not sufficient to solve our problem, because it may not generate valid solutions to our problem. For example, consider the Traveling Salesperson Problem. A solution is a permutation of the list of cities; a valid solution must visit each city exactly once. Simply crossing over two different permutations of the city list won’t work here - we’ll most likely wind up skipping some cities and visiting others twice.

To begin, we’ll need to modify the problem representation. We could encode a TSP tour as a bitstring, but for this step it will also work fine as a list of numbers representing the cities, such as [1, 3, 4, 5, 2].

Our objective function is the length of the tour. Since we want to minimize this, and our GA is coded to maximize fitness, we set fitness equal to MAXTOUR - tourlength, where MAXTOUR is any number larger than the longest tour. (the length of the longest edge times the number of edges will do.)

Now you’ll need to modify your crossover operator so that it still recombines solutions from past generations, but only produces valid tours. To do this, we’ll modify crossover as follows:

First, create a subclass of your GA class called permGA -this will allow you to reuse the main loop (the runGA method) and just change the crossover and mutation operators.
Mutation is a straightforward change - rather than flipping a bit in the bitstring, randomly swap the position of two cities in the tour. For example, 1, 4, 3, 2, 5 could mutate to 2, 4, 3, 1, 5.

We’ll implement a new version of crossover called partially matched crossover (PMX) that recombines two solutions based on position. PMX is easiest to explain by example.

Start with two strings:

A: 10 8 4 5 6 7 1 3 2 9
B: 8 7 1 2 3 10 9 5 4 6

Next, randomly choose two loci, or indices. Let’s say we picked 4 and 7.

A: 10 8 4 | 5 6 7 | 1 3 2 9
B: 8 7 1 | 2 3 10 | 9 5 4 6

PMX then exchanges the cities according to position. So, in String A, the segment 5 6 7 will be replaced with 2 3 10, and in string B, 2 3 10 will be replaced with 5 6 7. But we need to ensure that the new strings are permutations. We do this by swapping the element in A with its corresponding element in B in both places. For example, since we’re replacing 5 with 2 in String A, we need to replace 2 with 5. Similarly, 6 will swap with 3, and 7 with 10. Our new strings are:

A: 7 8 4 | 2 3 10 | 1 6 5 9
B: 8 10 1 | 5 6 7 | 9 2 4 3

In this way, we’re able to ensure that every new child is a permutation of the list of cities. This works because TSP is only concerned with the order in which cities are visited.

Following is code for generating a random TSP instance.

```python
from random import *

def makeTSP(ncities):
    matrix = [[0 for j in range(0, ncities)] for i in range(0, ncities)]
    for i in range(0, ncities):
        for j in range(i, ncities):
            if (i != j):
                matrix[i][j] = matrix[j][i] = randint(0,50)
    return matrix
```
Run your GA on random TSP problems of size 5, 10 and 20. Start with an initial population of 50. Use tournament selection, retaining the top 10% of the previous generation with roulette selection. Use a mutation rate of 5%. Include a graph showing the time to convergence as the number of cities increases. (I’d recommend showing time on the x axis and fitness on the y axis and having one line for each experiment.) You may want to generate several problems of each size and average your results.