Braden Licastro
Professor Roos
CS370-Final Project Part 2

For my final project I decided to do the coding intensive one. I did the crossover and mutation ones and did the individual tests.

With the two point crossover I didn't notice too much of an improvement, but there was about a 5% improvement, as seen in the below before and after modification tests.

Before:

```
Best string in generation 101:
0001000000000100100000000001000010000000000001001000000000010
Best fitness in generation 101: 8.0
Best string so far:
0001000000000100100000000001000010000000000001001000000000010
Best fitness so far: 8.0
```

After:

```
Best string in generation 101:
0000000101000000000010001000000000000000000001000000000100000
Best fitness in generation 101: 6.0
Best string so far:
0000000101000000000010001000000000000000000001000000000100000
Best fitness so far: 6.0
```

I noticed something similar with the mutation modification. The percentage change wasn't significant, and even less with the last, but there was one, also under 5%.

Before:

```
Best string in generation 101:
1000000000000000000001000000000100100000000001000010000000000
Best fitness in generation 101: 6.0
Best string so far:
1000000000000000000001000000000100000000000001001000000000010
Best fitness so far: 6.0
```

After:

```
Best string in generation 101:
0000100000000001000000000001000000000000100000000000000000100
Best fitness in generation 101: 5.0
Best string so far:
0000000000001000001000000000010000000000000000000100000000010
Best fitness so far: 5.0
```

In all honesty I expected both of the algorithms to perform similarly in these situations. I didn't expect them to get worse, because by modifying the string in a more aggressive way the chances of you taking something that might be completely off from the expected result and making a drastic modification in the string, you have more of a chance of finding a solution faster. Before it would take

small steps toward the final solution, but with the more aggressive algorithms it will allow us to get near to the expected output more rapidly and give more time to fine tune the output.

One of the main difficulties which I didn't take into account was the function of the two number generators. I had taken into account the fact that one will not always have the larger number of the two. With this I inserted a simple check that would make sure the first generator had the lower number, and if not swap the values. This kept index out of bounds errors from happening, as noted below.

```
// Have logic issues if num2 is larger than num1. Swap them.
if (xpoint1 > xpoint2)
{
  int temp = xpoint1;
  xpoint1 = xpoint2;
  xpoint2 = temp;
}
```

Although I took that previous case into account, I completely neglected to check for a situation where the random generators put out the same numbers. This very seldom happened, but when it did neither the two-point crossover or the mutation would occur. Once I realized that this was happening and could possibly give irrational results I inserted a simple check that would keep regenerating numbers until they didn't match, then the order would be checked with the above code.

```
// Can't have double crossover if the numbers are the same.
while(xpoint1 == xpoint2)
{
  xpoint2 = rng.nextInt(LENGTH-1)+1; // random point
}
```

To code for the crossover, as seen in the code below I walked character by character through the string not modifying anything until I reached my first position. At this point I swapped the characters at their respective positions between the two strings until I reached the random stop point. From here forward I went character by character finishing the rest of the strings without crossing over the characters.

Braden Licastro
Professor Roos
CS370-Final Project Part 2

```
// Copy first portions of parents into children:
for (int i = 0; i < xpoint1; i++) {
 s1 += p1.item().charAt(i);
 s2 += p2.item().charAt(i);
}
// Switch parents, copy to second break:
for (int i = xpoint1; i < xpoint2; i++) {
 s1 += p2.item().charAt(i);
 s2 += p1.item().charAt(i);
}
// Copy remaining portions of parents into children:
for (int i = xpoint2; i < LENGTH; i++) {
 s1 += p1.item().charAt(i);
 s2 += p2.item().charAt(i);
}
```

For the mutation method I was able to reuse much of the code from the two-point crossover with a little trickery. Due to the fact that I was only modifying one string, not two as before, I was able to keep the entire second string from being modified and only use the input as a reference to generate the modified output string. As before, up until the random point where the mutation was to begin , the input string was left intact and just copied over to the output string. Once that mutation point was reached I jumped into a loop that would start copying character by character starting at the random end point and continue copying until the start point was reached. This would copy that section of the string backward, as the mutation method was supposed to do. The remainder of the string was left intact and copied to the output string to be returned.

```
// Copy first portions of parents into children:
for (int i = 0; i < rpoint1; i++) {
  result += s.charAt(i);
}
// Swap randomly selected parts:
for (int i = rpoint2; i > rpoint1; i--) {
 result += s.charAt(i);
}
// Copy remaining portions of parents into children:
for (int i = rpoint2; i < LENGTH; i++) {
 result += s.charAt(i);
}
```

After completing the code and compiling I ran 10,000 generations of the tests. Before the methods were modified I was getting fitness numbers between 25 and 30 very regularly. After changing and testing the mutation method I saw that the fitness number never left the range of 5 to 10. This was surprising to me as this is a huge difference when before on the small number of tests I was getting a very small difference between the less and more aggressive algorithms.