Assignment 2: Genetic Algorithms Writeup

William Babincsak, Morgan Lee, Keith DeSantis

Approach to Puzzle 1:

Population Size - We found a population size of 100 to allow for appropriate diversity without adding too much computational strain.

Fitness - Fitness was calculated based on an organism's score. An organism's fitness was:

$$fitness = product(bin\ one) + sum(bin\ two) + range(bin\ three)$$

Selection - Parent selection was done using roulette selection, where the probability of a specific organism being chosen was weighted based on its fitness.

Organism's Fitness	Proportion of Total Weight	Chance of Selection
10	10/60	16.6%
20	20/60	33.3%
30	30/60	50%

The organisms with higher fitness were more likely to be selected. This was accomplished using the random.choices() function from the random Python library.

This method however requires that the sum of all fitnesses be greater than 0. To avoid the cases where a randomly generated population's summed fitness was <= 0, we normalized the weights by having each organism's weight equal:

$$weight = fitness - minimum\ fitness\ of\ population + 1$$

For example:

Given a population where the sum of fitnesses is negative, such as one with fitnesses [-10, -5, 10], the roulette selection method would result in:

Organism's Fitness	Proportion of Total Weight	Chance of Selection
-10	-10/-5	200%
-5	-5/-5	100%
10	10/-5	-200%

Which is clearly unusable nonsense as selection probabilities. By subtracting the minimum fitness of the population (-10), we normalize each weight to be at least 0, resulting in a positive sum as shown below.

Organism's Fitness	Proportion of Total Weight	Chance of Selection
-10 - (-10) = 0	0/25	0%
-5 - (-10) = 5	5/25	20%
10 - (-10) = 20	20/25	80%

This maintains the favorability of higher fitness organisms proportionally to their actual fitnesses, while also eliminating the possibility of having a negative sum of fitnesses.

The additional +1 added to the weight calculation is to handle the edge case where all members of the population's fitnesses are 0. By adding 1 to each weight we ensure that the sum of the fitnesses cannot be 0.

Initial Generation - Each member of the initial population was generated by reading in a numpy array of the initial pool, shuffling it randomly, then resizing it to a 4x10 array, with each row representing a bin.

Example with pool of 4 numbers, [3, 2, 5, 6]:

[3, 2, 5, 6].shuffle() = [6, 3, 2, 5]
[6, 3, 2, 5].resize(4, 1) = [[6], [3], [2], [5]]

Crossover - Crossover was performed by swapping a random "section" from a random bin of parent 1's for another section of equal size in a random bin of parent 2.

Example:

	Bin 1	Bin 2	Bin 3	Bin 4
Parent 1	4, 5, 8, 9	1, 23, 5, 9	0, 3, 2, 7	6, 4, 12, 9
Parent 2	-43, 4, 1, 6	9, 5, 0, 1	-24, 5, 1, 6	8, -9, 4, 1

Bin 1 is randomly selected for parent 1, and bin 3 is randomly selected for parent 2. A random section of parent 1's bin was selected, and a section of equal size was selected from parent 2's bin. In this example the section is highlighted in red and is of size 3, but the size could range from 1 to the length of a bin (10).

	Bin 1	Bin 2	Bin 3	Bin 4
Parent 1	4, 5, 8, 9	1, 23, 5, 9	0, 3, 2, 7	6, 4, 12, 9
Parent 2	-43, 4, 1, 6	9, 5, 0, 1	-24, 5, 1, 6	8, -9, 4, 1

The two sections are swapped, resulting in two new children.

	Bin 1	Bin 2	Bin 3	Bin 4
Child 1	4, -24, 5, 1	1, 23, 5, 9	0, 3, 2, 7	6, 4, 12, 9
Child 2	-43, 4, 1, 6	9, 5, 0, 1	5, 8, 9, 6	8, -9, 4, 1

This process could be repeated multiple times. After extensive testing it was determined that performing this "sectional swap" 4 times per crossover typically resulted in higher fitness outputs of the GA, with lower number of swaps not introducing enough diversity and higher number of swaps having diminishing returns or generating too much randomness to be beneficial. Therefore, 4 was chosen as the number of swaps per crossover.

Mutation - Mutation was performed by randomly choosing two different bins in an organism, then randomly picking a number from each bin, and swapping their positions.

Example:

	Bin 1	Bin 2	Bin 3	Bin 4
Organism	4, 1, 4, -6	1, 23, 5, 9	0, 3, 2, 7	6, 4, 12, 9

The randomly selected bins were Bin 1 and Bin 3. The randomly selected elements of those bins are highlighted in red.

The elements are then swapped.

	Bin 1	Bin 2	Bin 3	Bin 4
Organism	4, 1, 0, -6	1, 23, 5, 9	4, 3, 2, 7	6, 4, 12, 9

This mutation would occur rather frequently, set to 50% of the time. This is because based on tests, allowing for higher variability through more frequent mutations helped the algorithm find a similarly good organism faster (assuming elitism was enabled).

Approach to Puzzle 2:

Population Size - Similar to problem 1, we settled on a population size of 100 as a middle-ground between program performance and algorithm effectiveness.

Fitness - The fitness score used for this problem is the organism's score as defined in the assignment description:

$$fitness = 10 + height^2 - \Sigma cost \text{ for a valid tower}$$

Or:

$$fitness = 0 \text{ for an invalid tower}$$

To be a tower with nonzero score, the following rules must be followed:

1. A tower must have a door at the bottom and a lookout at the top.
2. A piece of the tower with strength n must have n or fewer pieces stacked on top of it.
3. A piece of the tower with width w must not have a piece with a width greater than w stacked on top of it.

Selection - Our selection mechanism for this problem was identical to that of problem 1, except we square the fitness values to weight the better organisms even more.

Initial Generation - Organisms are initially generated by selecting 2 pieces from the starting pool at random, then selecting a random number of the remaining pieces to fill the middle of the tower. Any remaining pieces not used by the tower are also stored as a parameter of the organism.
Example with the following tower pieces (formatted [type, width, strength, cost]):
[Door,2,4,1] [Wall,6,5,6] [Lookout,1,2,4] [Wall,3,4,2]
Randomly select top piece: [Wall,6,5,6]
Randomly select bottom piece:[Lookout 1,2,4]
Randomly generate number of middle pieces to select: 1
Select middle pieces: [Door,2,4,1]
List unused pieces: [Wall,3,4,2]

Crossover - Children are created by splitting two towers in half and then combining those four halves in a new way:

Example:
Before crossover:

Tower 1	Tower 2
[Door,2,4,1] [Wall,6,5,6] [Lookout,1,2,4] [Wall,3,4,2]	[Door,5,3,2] [Lookout,3,2,4] [Wall,5,5,5] [Door,5,3,2]

After crossover:

Child 1	Child 2
[Door,5,3,2] [Lookout,3,2,4] [Lookout,1,2,4] [Wall,3,4,2]	[Door,2,4,1] [Wall,6,5,6] [Wall,5,5,5] [Door,5,3,2]

Mutation - We set the mutation rate for part 2 to **100%**. It may sound crazy, but empirically it worked the best. A mutation consists of one of the following operations, chosen at random:

1. All pieces within the tower are shuffled.

[Wall,6,5,6] [Door,2,4,1] [Lookout,1,2,4] | [Wall,3,4,2] Becomes
[Door,2,4,1] [Lookout,1,2,4] [Wall,6,5,6] | [Wall,3,4,2]

2. One random piece of the tower is removed.

[Wall,6,5,6] [Door,2,4,1] [Lookout,1,2,4] | [Wall,3,4,2] Becomes

[Door,2,4,1] [Lookout,1,2,4]| [Wall,3,4,2] [Wall,6,5,6]

3. One random piece currently unused by the tower is added at a random location.

[Wall,6,5,6] [Door,2,4,1] [Lookout,1,2,4] | [Wall,3,4,2] Becomes
[Wall,3,4,2] [Wall,6,5,6] [Door,2,4,1] [Lookout,1,2,4] |

4. A random number of pieces that are not within the current tower configuration are swapped in.

[Wall,6,5,6] [Door,2,4,1] [Lookout,1,2,4] | [Wall,3,4,2] Becomes
[Wall,6,5,6] [Door,2,4,1] [Wall,3,4,2] | [Lookout,1,2,4]

Handling Illegal Children:

Post Process Function - All generated children (and mutated organisms) were run through an organism-specific "post-process" function. This function acted similarly for both puzzles, by comparing the organism's elements to the known initial pool, finding duplicates, and randomly replacing them with unused elements from the initial pool.

For brevity an example of post-process working on a tower from puzzle 2 will be shown:

The following tower was the result of a crossover (or a mutation):

{Lookout 2 0 2}
{Wall 3 3 1}
{Wall 3 3 1}
{Door 5 6 1}

However, the initial pool of tower pieces was:

[{Lookout 2 0 2}, {Wall 3 3 1}, {Door 5 6 1}, {Wall, 4 6 3}]

Since the tower has two instances of {Wall 3 3 1} but there is only one instance of it in the initial pool, post_process will generate a list of unused pieces from the initial pool.

unused_pieces = [{Wall, 4, 6, 3}]

It will then swap the duplicate piece for the unused piece.

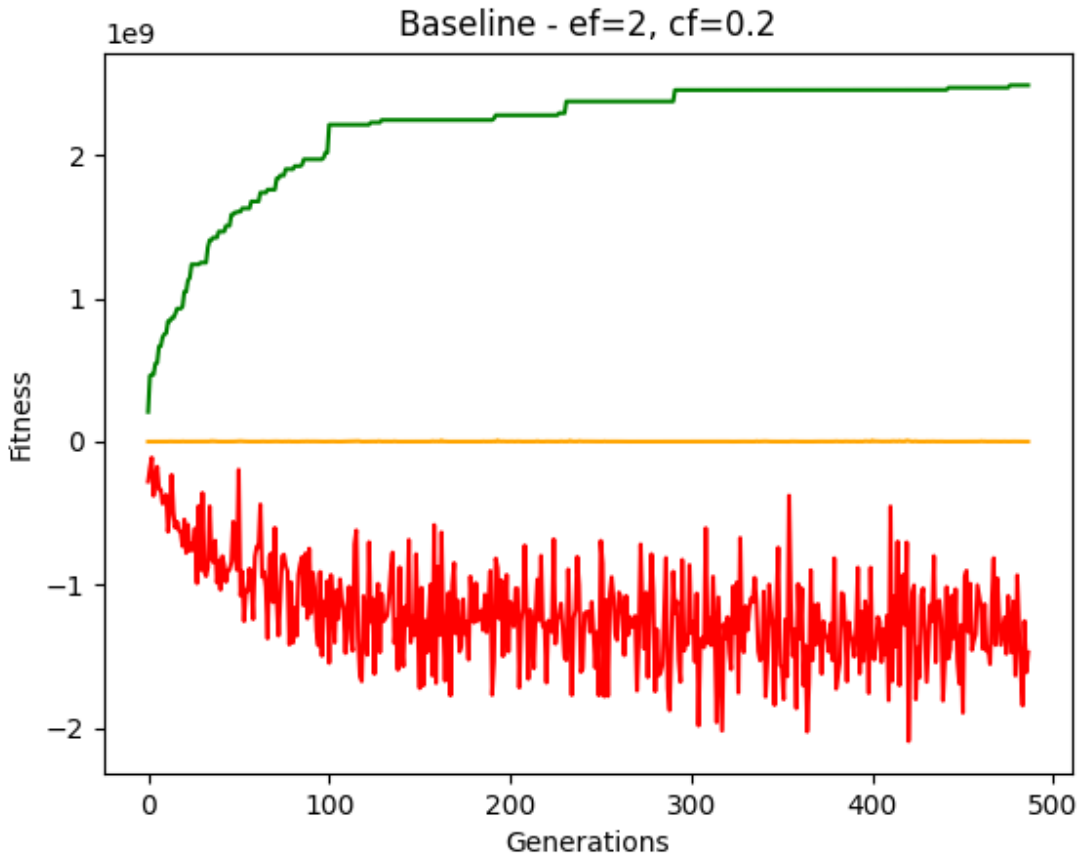
{Lookout 2 0 2}
{Wall 3 3 1}
{Wall 4 6 3}
{Door 5 6 1}

This ensures that the final organism does not have any duplicate elements that are not found in the initial pool. The logic functions similarly for puzzle 1, comparing numbers rather than tower pieces.

Graphs and Data:

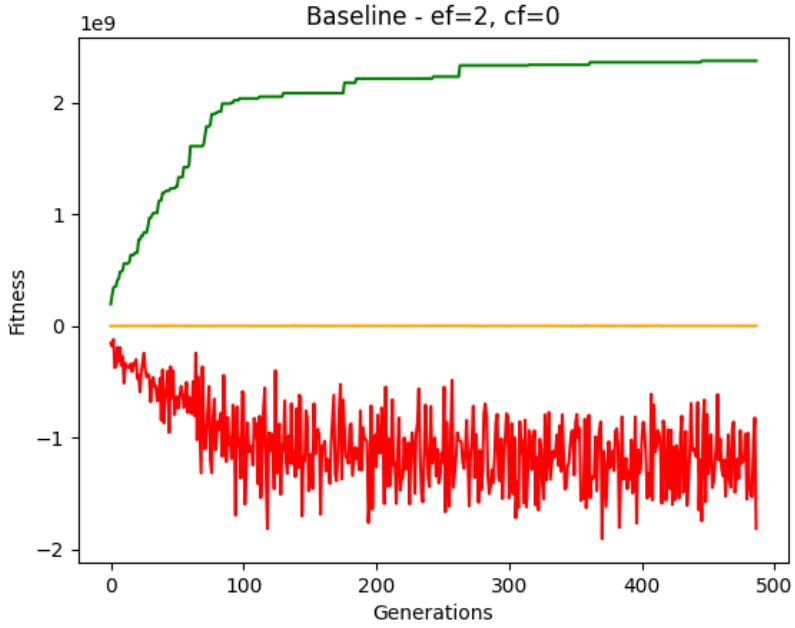
Puzzle 1:

Baseline (Elitism and Culling, 15 seconds)

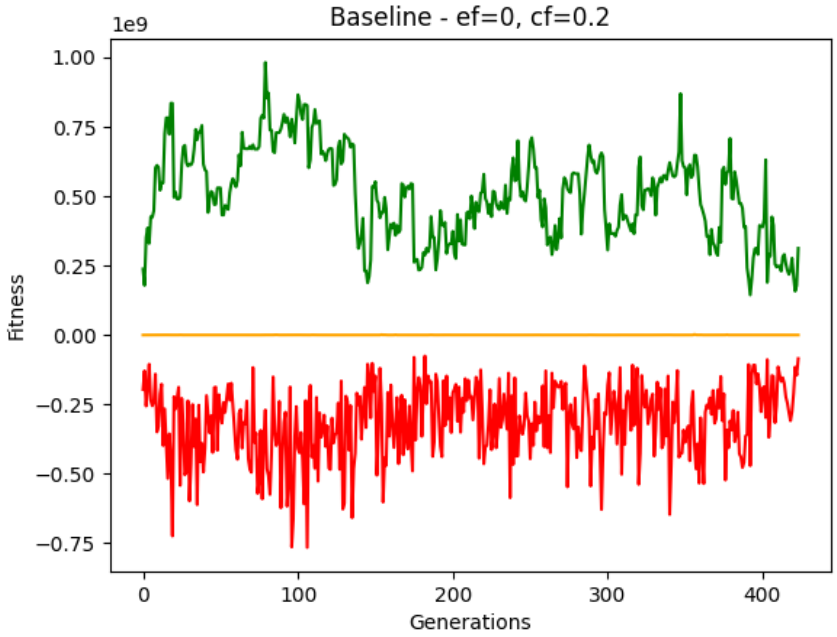


Our tuned version of the genetic algorithm for problem 1 does very well in reaching a high fitness in good time: the growth curve for the best in population is where we want it to be. The worst line goes low and stays there because it is so easy to get a very negative fitness during crossover and mutation whenever an odd number of negatives end up in bin 1. And since the more generations that have passed the higher absolute value the product is from bin 1, having it go negative only makes the worst get more negative over time.

W/Elitism Only:

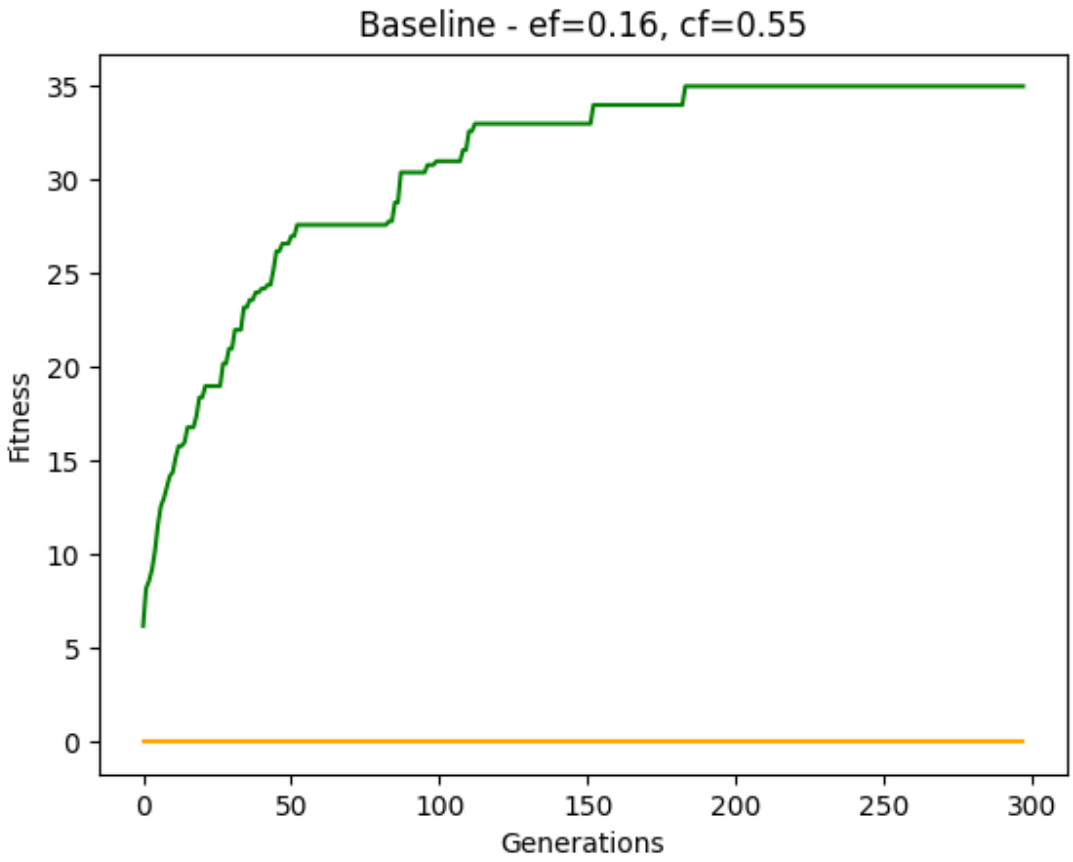


W/Culling Only:



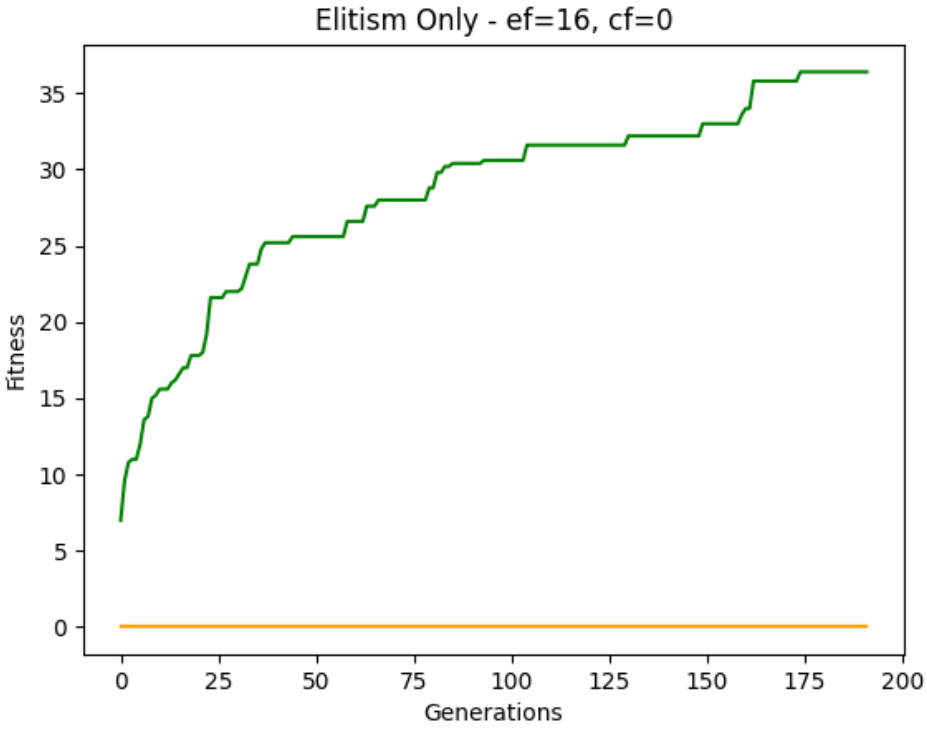
Puzzle 2:

Baseline (Elitism and Culling, 15 seconds):

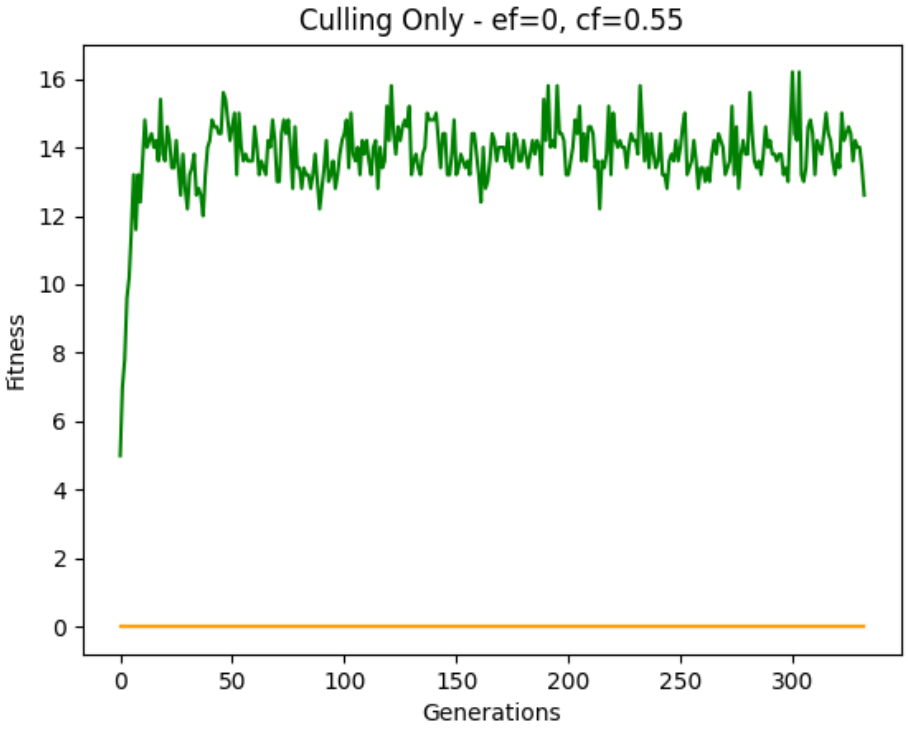


Our tuned version also does very well for problem 2. We quickly reach what we believe is or is close to our max fitness, ~35, for our sample dataset. The median and worst lines likely stay zero the entire time because there are so many possible ways to have the fitness go to zero, and thus so easy to make it happen, that useless towers are never removed completely from the population, and usually make up a majority of it.

W/Elitism Only:



W/Culling Only:



Results of experimenting with elitism and culling:

In both cases, elitism plays a big part in finding and keeping a good solution. Not only do we lose the best solutions that we find without elitism, but our best solutions we find without elitism aren't as good as the best solution we find with elitism. Elitism is key to a good genetic algorithm in both of these cases, and it didn't seem to affect the median values in either case or worst values in for problem 2. For problem 1, removing culling had a mirror effect on the worst values because there were no longer any very high positive numbers to turn very negative with a sign flip, so the average worst value was closer to zero without culling. In all cases for problem 1, the worst values approximately mirror the best values, and the median value averages to near 0. Culling, on the other hand, didn't seem to have much of an effect for either problem. The graphs with culling removed are nearly the same as the baseline graphs. For part 2, it seems to on average reach its plateau value later without culling, but reaches it nonetheless.

How Elitism and Culling were Implemented:

Our GeneticAlgorithm class has two number constructor arguments, elitism factor and culling factor, that determine elitism and culling, respectively. For both of these, if the value is less than 1, then it is treated as *percent* of the population to keep/cut, and if the value is greater than or equal to 1 it is treated as an absolute number of organisms to keep/cut.

The implementation of elitism transforms the current population into a heap, then gets references to the N most elite organisms (as determined by the elitism factor) by using the heap structure, and then adds those organisms to the next population.

The implementation of culling transforms the current population into a min heap and then pops the N worst organisms (as determined by the elitism factor) off of the heap, removing them from the population list.