Lab 4 – Co-Evolution:

Submitters: Inbar Lev Tov 316327246, Gal Gibly 315093435

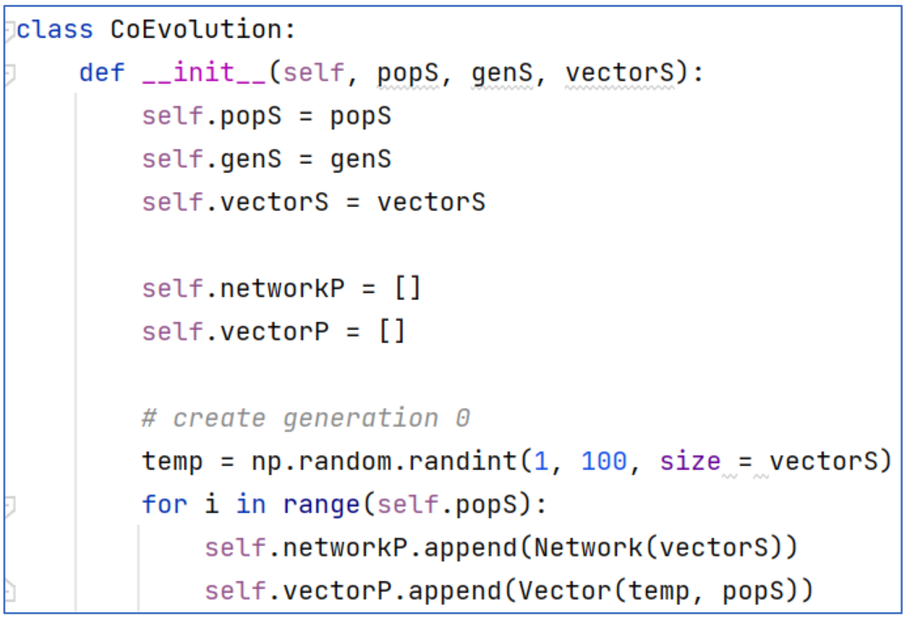
Note that the assignment was done with python version 3.11.0

Elements of coevolution:

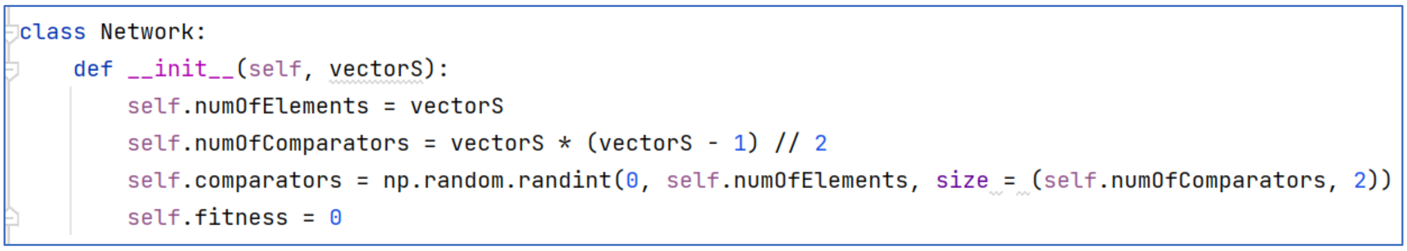
- We have 2 populations (vectors of numbers and sorting networks) that coexists in nature together.
- It's a prey and predator form of coevolution.
- Each of them creates opportunities for the other to improve:
  - The numbers vectors make the sorting networks reconsider its comparators order by being too complex to solve.
  - The sorting networks make the numbers vectors reconsider its numbers order by managing to sort too many numbers into place.
- Since each generation is reliant upon its predecessors, and we want them both to evolve at a unified rate – both of them would have the same number of generations and the same population size for each generation.

Representation & Fitness:

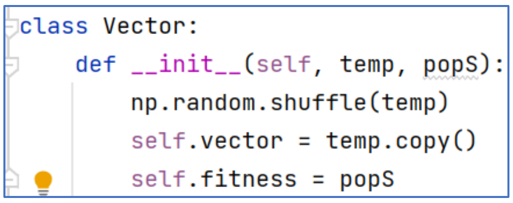
- We create the Vector class in the form of vectors with 6 / 16 random numbers (Naturals $n \in N$).
- Their fitness is measured with the number of networks they manage to "overcome" – if a network fails to sort **all** elements into order, that network fails.
- We create the Network class in the form of possible comparators: n numbers requires maximum of
$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$
comparators. We would create an array of possible pairs to compare – with the intention of reducing the number of pairs as much as possible.
- Their fitness is measured with the number of elements we managed to sort into the right place



```python
class CoEvolution:
    def __init__(self, popS, genS, vectorS):
        self.popS = popS
        self.genS = genS
        self.vectorS = vectorS

        self.networkP = []
        self.vectorP = []

        # create generation 0
        temp = np.random.randint(1, 100, size = vectorS)
        for i in range(self.popS):
            self.networkP.append(Network(vectorS))
            self.vectorP.append(Vector(temp, popS))
```



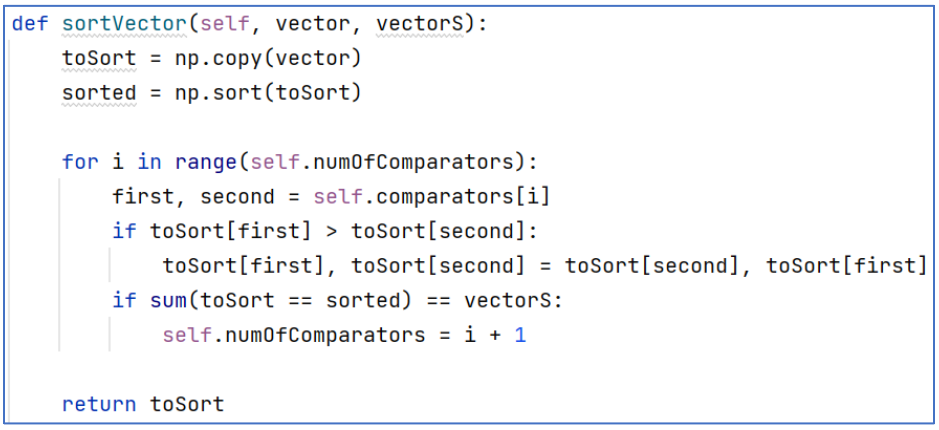
```python
class Network:
    def __init__(self, vectorS):
        self.numOfElements = vectorS
        self.numOfComparators = vectorS * (vectorS - 1) // 2
        self.comparators = np.random.randint(0, self.numOfElements, size = (self.numOfComparators, 2))
        self.fitness = 0
```



```python
class Vector:
    def __init__(self, temp, popS):
        np.random.shuffle(temp)
        self.vector = temp.copy()
        self.fitness = popS
```

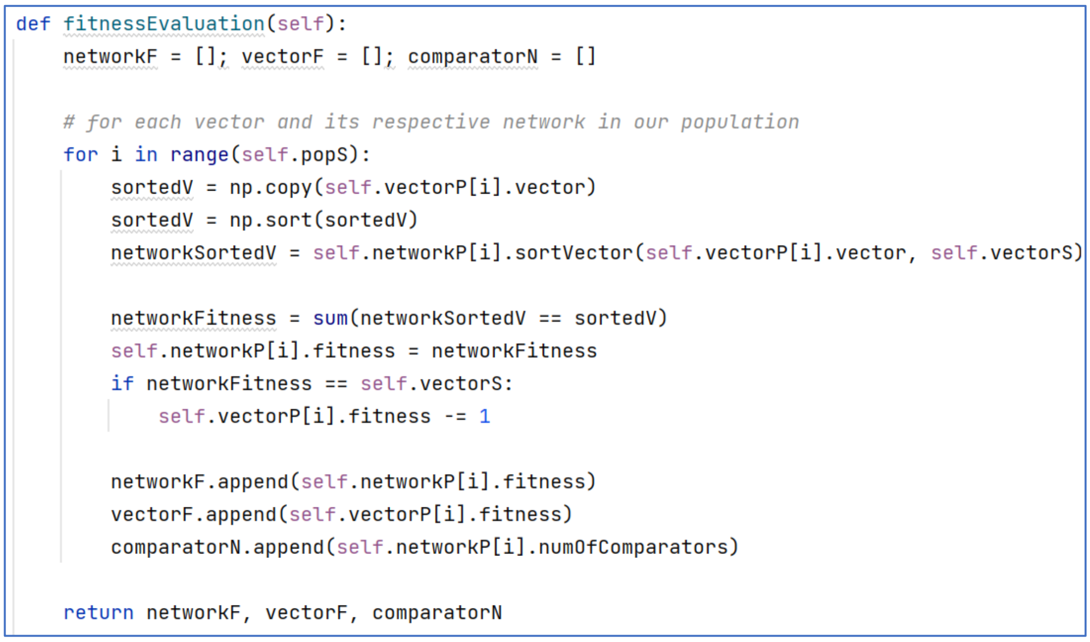
Building & Improvement Heuristics:

- As shown before, the building process is already done in the constructor of each class
  - Each vector contains a permutation of the base numbers – a good quality for future comparisons.
  - Each network is initialized with a permutation of $\frac{n^2-n}{2}$ pairs to be compared later.
- As for improvement:
  - The vector part is easy – because we simply take a different permutation when the vector fails in defeating a network.
  - Improving the network requires more work – which is what the heuristic would mainly focus on.
- For each **network** in every **generation**, our heuristic goes as follows:
  - We evaluate the sorting of the network through comparators.
  - If at some point we reach a sorted vector, we know when to stop comparing – we **improve** our network by limiting the comparators number to the one we had so far. The network is complete.



```python
def sortVector(self, vector, vectorS):
    toSort = np.copy(vector)
    sorted = np.sort(toSort)

    for i in range(self.numOfComparators):
        first, second = self.comparators[i]
        if toSort[first] > toSort[second]:
            toSort[first], toSort[second] = toSort[second], toSort[first]
        if sum(toSort == sorted) == vectorS:
            self.numOfComparators = i + 1

    return toSort
```

  - We then compare the network fitness to the truly sorted vector.
  - If they match: the network is already built with its comparators – no need of improvement.
  - If they don't match: we add 1 to the vector's fitness because the vector "defeated" the network.
  - We remember to improve the network by building a better one in the next generation (perfect networks would pass on through the generation as they are).

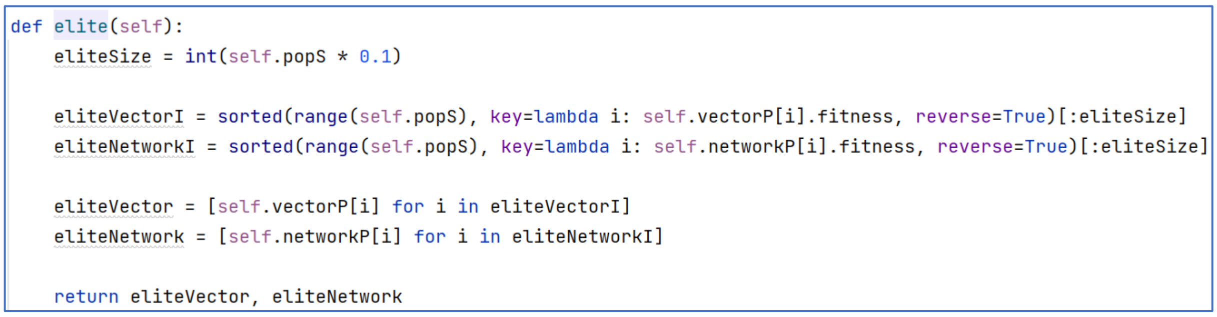


```python
def fitnessEvaluation(self):
    networkF = []; vectorF = []; comparatorN = []

    # for each vector and its respective network in our population
    for i in range(self.popS):
        sortedV = np.copy(self.vectorP[i].vector)
        sortedV = np.sort(sortedV)
        networkSortedV = self.networkP[i].sortVector(self.vectorP[i].vector, self.vectorS)

        networkFitness = sum(networkSortedV == sortedV)
        self.networkP[i].fitness = networkFitness
        if networkFitness == self.vectorS:
            self.vectorP[i].fitness -= 1

        networkF.append(self.networkP[i].fitness)
        vectorF.append(self.vectorP[i].fitness)
        comparatorN.append(self.networkP[i].numOfComparators)

    return networkF, vectorF, comparatorN
```

Elite Team, Parent Selection, Crossover and Mutation:

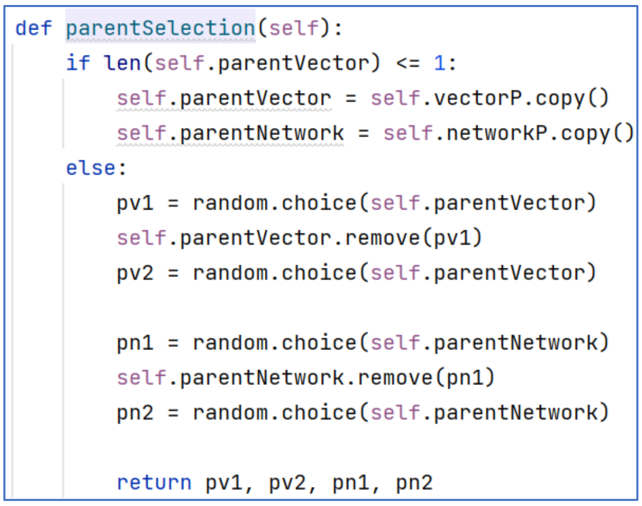
In order to fit our genetic engine to our new project, there are some elements we were able to preserve, while others had to be discarded.

- Elite Team:
  - We still had to take 10% of the best vectors and networks according to their best fitness.
  - Strong genes would pass on as they are in order to maintain their good prospect for the future.
  - Aging criteria, however, became obsolete here because we prioritize efficiency of the network and complexity of the vector over age.



```python
def elite(self):
    eliteSize = int(self.popS * 0.1)

    eliteVectorI = sorted(range(self.popS), key=lambda i: self.vectorP[i].fitness, reverse=True)[:eliteSize]
    eliteNetworkI = sorted(range(self.popS), key=lambda i: self.networkP[i].fitness, reverse=True)[:eliteSize]

    eliteVector = [self.vectorP[i] for i in eliteVectorI]
    eliteNetwork = [self.networkP[i] for i in eliteNetworkI]

    return eliteVector, eliteNetwork
```

- Parent selection:
  - The other 90% are assembled as new children from parents with good prospects.
  - We want to exercise as much diversity as possible while maintaining a sense of randomness.
  - Which is why, instead of dealing with roulette like the last genetic algorithm, we randomly choose two parents from the parents pool.
  - After being chosen, the parent is removed from the pool in order to ensure it will not be chosen again.
  - When we reach the end of the list we reinitialize it with the original list and start again until we have 2 different parents for 90% of the new children.



```python
def parentSelection(self):
    if len(self.parentVector) <= 1:
        self.parentVector = self.vectorP.copy()
        self.parentNetwork = self.networkP.copy()
    else:
        pv1 = random.choice(self.parentVector)
        self.parentVector.remove(pv1)
        pv2 = random.choice(self.parentVector)

        pn1 = random.choice(self.parentNetwork)
        self.parentNetwork.remove(pn1)
        pn2 = random.choice(self.parentNetwork)

        return pv1, pv2, pn1, pn2
```

- Crossover:
- For vectors, we take the ones who have the strongest fitness = more complex permutations.
- For networks:
  - We take 50% of the comparators from parent 1 and the other 50% from parent 2.
  - We know who the stronger parent is due to fitness, and so we know that his 50% of comparators are a better start for the comparison process for the new child.
  - We position the stronger parent's comparators before the second parent's comparators in the child to increase efficiency in the **beginning** of the process (the faster the search ends, the less comparators we need, the more efficient and strong the network is).
  - The combination of good comparators from the 1st and 2nd parent can make an even more of a lethal network than the options we had before for simple crossover.

```python
def crossover(self, pv1, pv2, pn1, pn2):
    childVector = Vector([1], 0)
    if pv1.fitness > pv2.fitness:
        childVector.vector = pv1.vector.copy()
        childVector.fitness = pv1.fitness
    else:
        childVector.vector = pv2.vector.copy()
        childVector.fitness = pv2.fitness

    childNetwork = Network(pn1.numOfElements)
    childNetwork.numOfComparators = pn1.numOfElements * (pn1.numOfElements - 1) // 2
    i = 0
    if pn1.fitness > pn2.fitness:
        childNetwork.comparators = np.where(i < pn1.numOfComparators / 2, pn1.comparators, pn2.comparators)
    else:
        childNetwork.comparators = np.where(i < pn2.numOfComparators / 2, pn2.comparators, pn2.comparators)

    return childVector, childNetwork
```

- Mutation:
- For vectors: simply shuffling the vectors – gives us a fair chance of getting a better permutation each time.
- For networks:
  - We split it into 2 parts.
  - We randomly shuffle a smaller group of the network's comparators (supposedly in the size of the vector's size).
  - Then we randomly shuffle the rest of comparators in a separate group.
  - It's a much larger group in comparison because the true number of comparators is exponentially bigger than the vector's size.
  - Then we unite them in reverse order (first the larger group, then the smaller one), and the new network is finished.
  - We do it because we discovered (during our research) that sometimes there are certain advantages with separating a huge group into smaller ones before performing the real changes we want to complete.
  - Also, putting the smaller group in the end of the list instead of in the beginning gave us relatively smaller changes in future generation's fitness. We know that because when you relegate a small group of comparators to the end of the line, and then let them go completely (using the improvement heuristic) – when you see the fitness improvements over generations you eventually realize how bad those pairs were in the first place and how good it was to delay their processing into the network by pushing them to the end of the list.

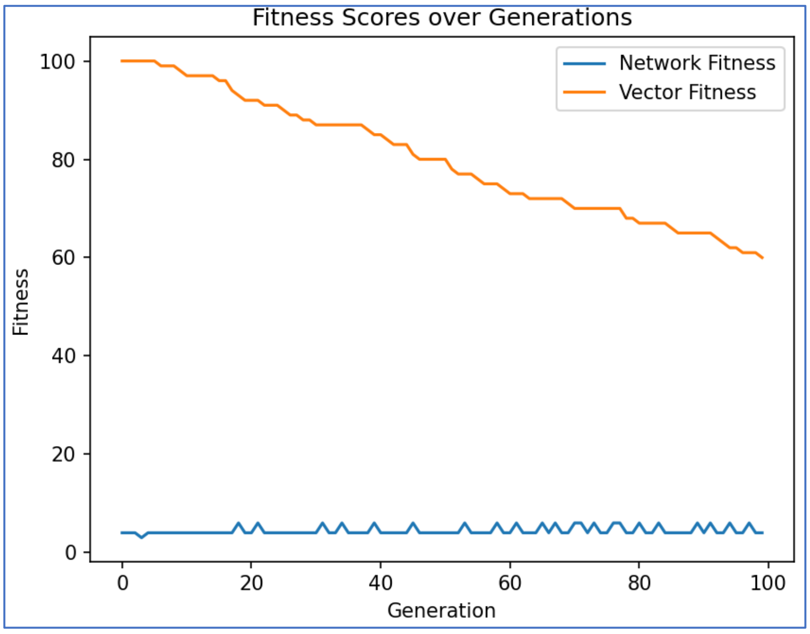
```python
def mutation(self):
    n1 = np.random.randint(0, self.numOfElements)
    n2 = np.random.randint(0, self.numOfComparators - n1)
    self.comparators[n1], self.comparators[n2] = self.comparators[n2], self.comparators[n1]
```

<u>Bitonic Networks:</u>

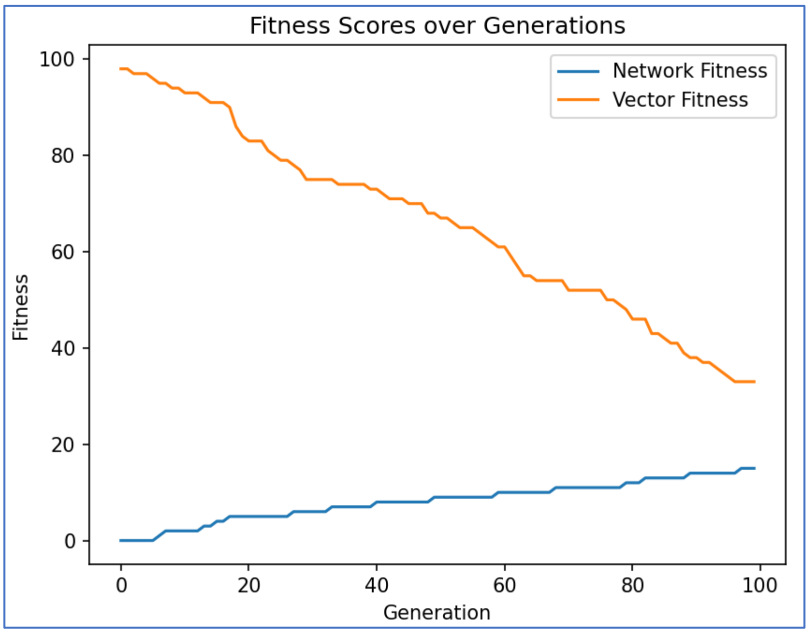
- After an extensive research we decided to forgone the use of bitonic networks.
- We felt the results achieved by normal sorting networks were sufficient enough, and that there was no great improvement by working on bit level.
- Also, we saw an extensive lag in time and efficiency resulted from transforming each network to bit level and back – and since there was no improvement in overall results, it felt redundant.

Convergence Graphs:

- Lets start by talking about the convergence graph for a vector of 6 elements:



- The good thing we can observe here is that the vector's fitness present a trend of going monotonously down. Not too fast to be considered a massive reduction in quality and power, but still going down over time.
- We can also notice that while the vector loses some of its power, its still sets a certain level of complexity that is relatively high over most of the networks. Basically we "allow" some of the networks the possibility of winning while still maintaining a standard of which network should work and which one shouldn't.
- As for the networks, we've had less success in showing a trend of some kind – but that would later improve in the vector of 16.
- What we would have wanted to see with regards to networks is a graph that goes monotonously up – improving the quality of the networks and its ability to sort numbers over time. Unfortunately, what we see here is that quality goes up and down constantly – which means we might lose some good results along the way in favor of bad ones.
- We have worked meticulously in order to correct those mistakes, and we managed to get good results in the 16, which will be shown later.
- Now lets talk about the graph of 16:



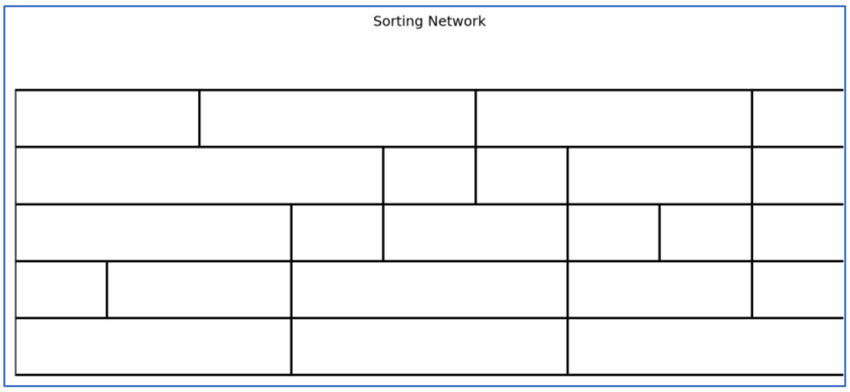
- Here we managed to correct the mistakes of the past graph while still maintaining the good stuff.
- We have both the steady climb of network quality until we reach the wanted results (sorting all 16 numbers) and the degeneration of vector over time but in a good way – the vector gives up the ability to defeat **all** networks but still maintain a certain level of complexity to most networks in its vicinity.
- Ultimately, the reason for the improvement is the algorithm itself.
- Comparators over 16 holds way more possibilities of comparison than it does in the 6. We simply showed that the algorithm indeed works – more numbers to compare, more power relegated to the networks than it does to the vector.
- That is the best example of the **tradeoff** between complexity of vectors and the power of sorting networks.

The Networks Themselves:

- Based on the calculation we did in the beginning regarding the number of possible comparators:

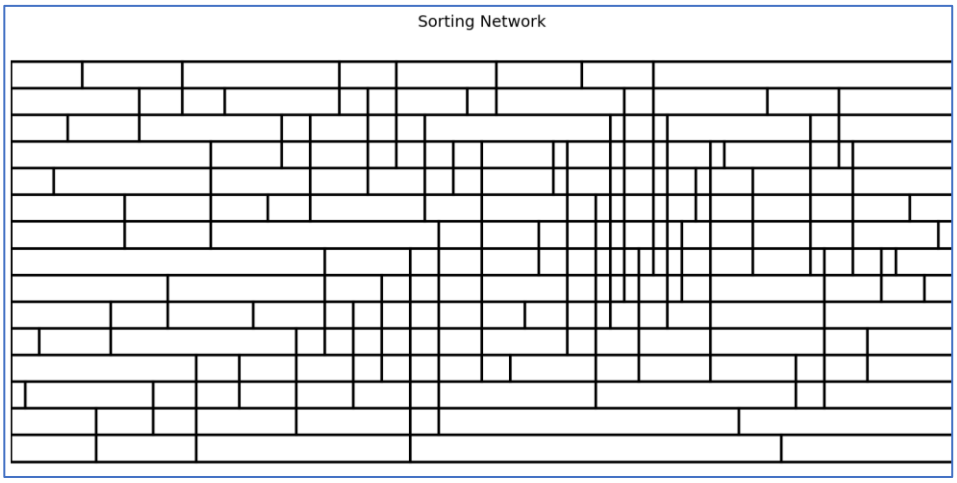
$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$

- For the 6:
  - If the vector was already sorted we would only need 5 comparators to ensure that.
  - Maximum number is 15 according to the formula.
  - The most complex vector we were able to find is (84, 86, 64, 52, 93, 87).
  - The most efficient network we were able to find contains 8 comparators.



- For the 16:
  - If the vector was already sorted we would only need 15 comparators to ensure that.
  - Maximum number is 120 according to the formula.
  - We aimed, of course, towards 60 comparators as was discussed in the class. Our first attempt was pretty horrible with 120 comparators but over time we managed to reduce it to 65.
  - The most complex vector we were able to find is

    (65, 9, 84, 75, 31, 17, 79, 76, 93, 10, 46, 55, 20, 69, 5, 70)
  - The most efficient network we were able to find contains 65 comparators


Sorting Network