Programmeringsprojekt: Fase 1

Hlynur Æ. Guðmundsson, Mathias B. Jensen, and Valdemar B. Reib

Fasekoordinator - Valdemar B. Reib

Institut for Matematik og Datalogi, Syddansk Universitet

November 2024

1 Problemstilling

I dette projekt er problemstillingen tredelt i faser, som giver anlæg til progressiv udvikling. I første fase har gruppen fået til opgave at udvikle de grundlæggende moduler, som resten af faserne kommer til at bygge på. Der skal til hvert modul laves et antal af funktioner, således at det færdige produkt af fase 1, kan danne et sorteringsnetværk ud fra de to modulers funktionalitet. Dette mulliggøre i sidste ende sortering af en liste heltal.

2 Introduktion

I dette projekt skal der udvikles et komparatornetværk igennem tre forskellige faser. I denne rapport dokumenterer, diskutere og forklare vi hvordan første fase er blevet gennemført. Her har udfordringen været at kreere to moduler i Python ved navn *Comparator.py* og *Network.py*, som tilsammen tillader gruppen at sortere en liste af heltal ved hjælp af komparatorer.

3 Fremgangsmåde

Da gruppen skulle udvikle to moduler, med flere forskellige funktioner har gruppen gjort brug af nogle teknikker som har gjort arbejdet lettere. Gruppen valgte at opdele versionerne i koden i forskellige revisioner, og løbende gemme dem på github[Azu24]. Da det tillod gruppen at dele kode og samle forskellige versioner. Ved at gøre brug af disse revisioner blev det også nødvendigt med fordeling af arbejdet. Gruppen har gjort brug af online møder, og fysiske møder i kombination for at sikre at hele gruppen fik en status. Dertil fordelte gruppen arbejdet ligeligt og byttede internt arbejdsopgaver hvis et individ sad fast på en enkelt funktion. Ved at møde regelmæssigt blev der planlagt arbejdsfordeling og forventning til næstkommende møde.

4 Opbygning af kode

I denne sektion vil der forklares nogle udvalgte funktioner og opbygningen af de pågældende moduler. De to moduler *Comparator.py* og *Network.py* er opbygget bag princippet omkring dataklasser og abstrakte datastrukturer. *Comparator.py* og *Network.py* har hver en dataklasse med henholdsvis to heltal i og j, samt en liste. Dette muliggjorde nem iteration af listerne. Dette tillader gruppen at udvide, rette og tilføje ting til modulerne uden videre. Dertil er projektet bygget op om ideen omkring kontraktbaseret programmering, der har gjort samarbejde omkring projektet nemt, da funktionerne skal returnere noget bestemt, kan hvert enkelt individ lave funktioner der bruger de andre funktioner uden at tænke over hvordan den pågældende funktion er kreeret.

5 Opbygning af *Comparator.py*

Komparator modulet har til formål at danne et objekt med to variabler der svarer til kanalerne de er kreeret i mellem. Modulet er bygget op rundt om flere forskellige funktioner, som tillader os at gøre brug af den effektivt i *Network.py* modulet. Komparator modulet har et par interessante funktioner, herunder *"apply()"*, *"all_comparators()"*, *"std_comparators()"*, *"to_program()"*.

5.1 *Apply()*

Apply er en funktion der varetager selve funktionaliteten af en komparatoren. Den gør brug af komparatorens variabler og en liste, til at sammenligne størrelsesorden på de to elementer i listen, komparatoren er på. Denne funktion er kreeret således at der i *Network.py* kan kaldes på den rekursivt for at løbe en hel liste igennem. Dette vil tillade gruppen at sortere en liste ved brug af flere komparatorobjekter.

5.2 *All_comparators()*

All_comparators er en funktion der tillader gruppen at hente en liste med alle de mulige komparatorer på n-kanaler. Ved at gøre brug af en *bubblesort* struktur i funktionen med brug af to *for each* lykker, for at sikre at alle kombinationer af komparatorer opnås.

5.3 *Std_comparators()*

Std_comparators er en funktion der ligeledes *all_comparators* returnere en liste med komparatorer på baggrund af n-kanaler. I denne funktion tjekkes alle de kombinationer for om de er en standardkomparator, hvilket betyder at i-variablen skal være strengt mindre end j-variablen. På den måde sikres der at listen der returneres, kun indeholder alle de mulige kombinationer af standardkomparatorer til n-kanaler.

5.4 *To_program()*

To_program er en funktion der tillader gruppen at kreere et Python program som kan køres direkte. Dette forventes brugt i de senere faser for at analysere og optimere. I komparatormodulet sættes det op således at *Network.py* kan gøre brug af det.

6 Opbygning af *Network.py*

Netværksmodulet har til formål at danne strukturen, som komparatormodulet skal agere på. Modulet er ligeledes komparatormodulet, opbygget af flere forskellige funktioner, som tillader os at bruge komparatormodulet effektivt til at simulere et sorteringsnetværk.

Netværksmodulet inkluderer funktioner, hvis funktionalitet efterligner mange af funktionerne i komparatormodulet, hvor hovedforskellen er, at de gentages for hver komparator i netværket. Udover dem er der fire essentielle funktioner i netværksmodulet, som hedder "*apply() "*, *"outputs()"*, *"all_outputs()"*, *"is_sorting()"* og *"to_program()"*.

6.1 Opbygning af *apply*()- og *outputs*()-funktionerne

Funktionerne *apply*() og *outputs*() har til formål af sortere, fra mindst til størst, en liste af lister indeholdende heltal. Funktionernes væremåde emulerer hvordan et sorteringsnetværk som helhed kan sortere talværdier.

Netværksmodulets udgave af *apply*()-funktionen efterligner i høj grad udgaven i komparatormodulet, men der er en essentiel forskel. Funktionen påkræver et ikke tomt netværk indeholdende komparatorer og en liste af heltal. Listen er den samling af tal, som der sorteres. Den fungerer ved at benytte en *for-each* lykke til at gentage *apply*()-funktionen fra komparatormodulet for hver komparator i input netværket. Dette resulterer i at alle komparator i et netværk bliver brugt en gang hver, som simulere hvordan en enkelt kanal påvirkes af flere komparatorer. Netværksmodulets udgave af *apply*()-funktionen kan dermed sortere en hel liste af heltal, i stedet for en andel, hvis et netværk har de rette komparatorer.

Funktionen *outputs*() bygger videre på *apply*()-funktionen og udvider dets omfang således, at et netværk kan sortere flere lister af heltal, i stedet for kun en ad gangen. Udover det påkræver funktionen en liste af lister med heltal som input i stedet for en enkelt liste med heltal. Funktionen fungerer ved, at den for hver liste i input listen kalder *apply*()-funktionen med listerne af heltal fra input listen som argument. Funktionen vil trinvis gennemgå og sortere alle listerne med heltal indtil der ikke er flere tilbage. Når funktionen er færdig, så er alle tallene i alle listerne blevet flyttet rundt på af alle komparatorer, så længe der bruges et gyldigt komparatornetværk. De tre funktioner og deres ansvarsområder er illustreret på den følgende figur

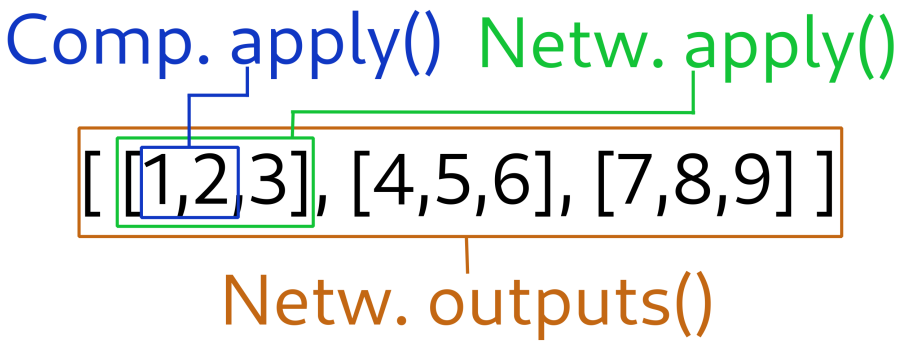

Figure 1: Diagram over omfang af funktioner. (Egen tilvirkning)

6.2 Opbygning af *permutations*() og *all_outputs*()

For at kunne teste vores sorteringsnetværk kreerede vi funktionen *all_outputs*() og en tilhørende funktion *permutations*().

Formålet med disse to funktioner er, at opstille alle permutationer af tallene 0 og 1, i form af en liste af lister med heltal, hvorefter et input netværk vil forsøge at sortere tallene i listerne. Efter netværket har sorteret tallene, så vil funktionen fjerne gentagne lister i tilfælde af, at listerne indeholder de samme tal i samme rækkefølge. Funktionerne muliggøre, at vi kan teste vores sorteringsnetværk og analysere deres væremåder, uden at skulle teste uendelige mange talværdier.

All_outputs()-funktionen gør brug af *permutations.py*-modulet sammen med *outputs*()-funktionen. Modulet *permutations.py* indeholder en enkelt funktion ved navn

permutations(), som opstiller selve permutationerne af 0 og 1 ved hjælp af *list-comprehension*, som Python understøtter. Funktionen påkræver et enkelt argument, som beskriver længden af permutationerne. For eksempel hvis længden var 3, så vil funktionen returnere:

$$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,1]]$$

Det ligger seperat i sit eget modul som et resultat af arbejdsfordelingen, men det kunne godt integreres i *network.py*-filen ud større udfordringer. Alle permutationerne af 0 og 1 bruges i funktionen $all\_outputs()$, som udover at kræve et heltal til $permutation()$-funktionen, også kræver et ikke tomt komparatornetværk. Netværket bruges til at sortere alle permutationerne ved hjælp af $output()$-funktionen. Efter at listerne bliver sorteret, så vil mange af listerne med gentagelser af hinanden, da de indeholder samme antal af 0'er og 1'er, der nu står i samme rækkefølge. Gentagelserne bliver trinvis frasorteret ved at sammensætte en simpel $for - each$ lykke med en $if\ not\ in$ commando. De vil tilsammen kun tilføje unikke indslag i en liste. Ved brug af eksemplet fra før, så vil funktionen trinvis fungerer således:

Trin 1: Opstil Permutationer
$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,1]]$

Trin 2: Sorter 0'er og 1'er
$[[0,0,0],[0,0,1],[0,0,1],[0,0,1],[0,1,1],[0,1,1],[1,1,1]]$

Trin 3: Fjern Gentagelser
$[[0,0,0],[0,0,1],[0,1,1],[1,1,1]]$

Tilsammen tillader funktionerne gruppen at sammenligne og analysere et komparatornetværk. For eksempel hvis gruppen havde to netværk, som havde de samme komparatorer men i forskellige rækkefølge, ville det være meget relevant at analysere, hvorvidt de sortere ens. Tilsvarende hvis vi havde et netværk, der manglede nogle komparatorer, så ville vi kunne opfange det ved, at analysere hvordan netværket sortere.

6.3 Opstilling af *is_sorting*()

For at kunne udnytte netværket effektivt i projektet, er det vigtigt at vide, hvilke evner og begrænsninger et individuelt netværk har. Derfor kreerede vi funktionen $is_sorting()$. Funktionen analysere om et inputnetværk kan korrekt sortere en specifikt antal kanaler, som også er et input i form af et heltal. For at kunne tjekke om et netværk kunne sortere et bestemt antal kanaler, så opstillede vi nogle minimumskrav.

- 1) Et netværk skal ikke være tomt

- 2) Skal som udgangspunkt ikke indeholde standardkomparatorer, men et gyldig netværk kan indeholde dem, så længe at fejlsorteringen bliver rettet af standardkomparatorer.

- 3) Den maksimale kanal af et netværk skal være lig med det antal af kanaler vi er interesseret i at undersøge om vores netværk kan sortere. Hvis et netværks maksimale kanal er mindre end vores antal af kanaler vi analysere, så vil et netværk ikke kunne flytte og sortere værdierne i den øverste kanal. Hvis et netværks maksimale kanal er større, så vil den heller ikke kunne sortere korrekt, da komparatorer vil forsøge at flytte værdier fra og til ikke-eksisterende kanaler.

Efter at have defineret nogle minimums krav, så skal funktionen også kunne yderligere analysere sorteringsnetværk. For selvom et netværk opfylder alle tre minimumskrav, så er det ikke garanteret, at det vil kunne sortere et bestemt antal kanaler korrekt. I planlægningsprocessen kom vi frem til to gyldige strategier, for at tjekke om et netværk kan sortere et antal kanaler eller ej.

- 1) Analysere om et netværk indeholder alle de nødvendige komparatorer i den rigtige rækkefølge for, at kunne sortere et antal kanaler korrekt. Matematisk tjekker vi om alle de rigtige komparatorer er en delmængde af et netværk. Teoretisk kunne det være en hurtig og effektiv løsning, men strategien har en stor ulempe. Hvis man tillader et netværk at have gentagne eller ikke standardkomparator er mængden af gyldige netværk uendelige. Hvis et netværk har de rette komparator til sidst i rækkefølgen, så vil de altid kunne rette på fejlene af de forrige, så netværket som helhed kan sortere korrekt. Det vil være intensivt at gennemgå et netværk slavisk, når der er uendelige mange gyldige muligheder. Hvis man forbyder ikke standardkomparatorer, så vil det ikke gøre den største forskel, da et netværk er tilladt at have kopier af samme komparatorer.

- 2) Lad komparatornetværk forsøge at sortere alle permutationer af 0 og 1 og derefter analysere om netværket har sorteret tallene korrekt. Strategien er forholdsvis mere simpel end den første, men den kan være intenst, når vi bearbejder mange kanaler og netværk med komparatorer. Dog undgår strategien hele problematikken med, at analysere om ikke standardkomparatorer bliver fuldstændig overskrevet af standardkomparatorer.

I sidst ende implementerede vi den anden strategi samt minimumskravende. Dette gjorde vi ved hjælp af funktionerne fra de tidligere afsnit.

6.4 *To_program*()

To_program funktionen er en funktion som ligeledes den fra komparatormodulet returnere en liste af kommandoer som kan køres direkte i Python. Den gør brug af komparatormodulets version af *to_program*, for at kunne printe en udvidelse af denne version. Dette tillader gruppen at fokusere på essensen af programmet, som vi forventer skal bruges i de senere stadier af projektet.

7 Test og evaluering

I denne sektion vil vi benævne hvordan at koden er blevet testet, og dertil også evalueret på baggrund af de forventelige resultater. Den kontraktbaseret struktur sikrer at hver funktion er klart beskrevet til at returnere specifikke ting, så som *int, str, list*[*int*]. Dette har tilladt gruppen at opskrive *DOCTEST* til hver funktion, som tydeligt angiver hvad den pågældende funktion gør. Dette er så udvidet i en separat test fil kaldet *test.py*, som har lagt an til flere test, med mere konkrete eksempler. Dette giver en samlet ide omkring hele programmets funktionalitet, ved blot at kører test filen.

7.1 Test af kode

I testen af *apply*() funktionen i *network.py* lavede vi nogle tests, for at se om funktionen virkede rigtig, derefter lavede vi mere komplekse netværk og lister. En af de tests inkluderede det samme netværk og liste fra projekbeskrivelsen (jf. 2). Dette viste gruppen at funktionen virkede korrekt, da sorteringen af den speciffike liste er betinget af det specifikke netværk af komparatorer. Testen af *to_program*() i *network.py* var mere kompliceret, da det ønskede resultat krævede at vi testede det med *exec*() funktionen, denne funktion læser Python kode fra en streng og kører det, ved brug af *exec*() skulle *aux* variablen første deklareres, fordi den kun eksisterede som et argument kaldet af *to_program*()

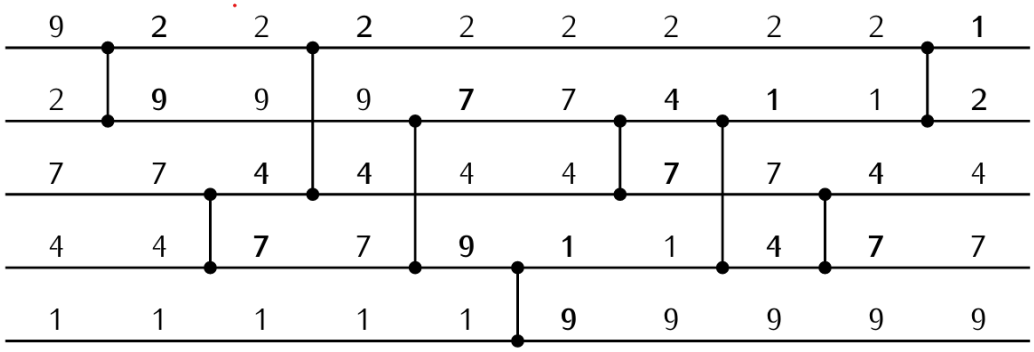

Figure 2: Komparator Netværk[Cru24]

Vi har struktureret vores test.py fil til at inkludere alle funktioner i første fase af projektet, som giver et samlet overblik. Filen er delt i to med *print*() funktioner som tydeligt angiver hvad der testes. I for eksempel test af *all_outputs*() gør vi brug af *print*() funktioner til tydeligt at vise forskellen på et sorteret og ikke sorteret output. Der gøres brug af vores hjælpefunktion permutations.py til at danne alle de permutationer, og derefter kaldes *all_outputs*() for at sortere de permutationer. Denne struktur bruges generelt i *test.py*. Et andet eksempel på dette ses i test af *is_sorting*() som også er struktureret for at tydeliggøre de forskellige scenarier hvor funktionen returnere *false*. Ved at holde den struktur og tilgang, sikre vi os at de pågældende funktioner virker som forventet.

7.2 Evaluering af kode

I løbet af udviklingen af de to moduler *Comparator.py* og *Network.py*, er der blevet taget en del valg, og ændringer. Dertil er der blevet brugt ChatGPT[Ope24] som hjælpeværktøj til sparring under udvikling af de forskellige funktioner osv. I denne sektion vil vi fokusere på at diskutere nogle af de valg.

7.3 Valg af løsningsforslag

En af de største valg der blev taget, var hvilken programmerings paradigme der ville fokuseres på. De vi i gruppen overordnet set havde mest erfaring med imperativ, er der en klar overvægt af det format. Dertil er der i vores modul permutations.py brugt funktionel programmering som løsningsforslag. Dette startede også originalt som en imperativ løsning. Dog var denne løsning delvist udformeret med ChatGPT som hjælpeværktøj, og derfor blev den senere omskrevet til en funktionel løsning.

I løbet af udviklingen er der også sket fejl, da programmet er komplekst, kan der ske uventet ting under test og udvikling. En af de største fejl var den første implementering af *Comparator.py* modulets funktion *apply*() da den gjorde brug af to for-each lykker som essentielt set er en *bubblesort* algoritme. Det resulterede i at output fra funktionen, var en allerede sorteret liste. Det blev opdaget senere hen i testforløbet, da valget af komparator var intet sigende. Det er et godt eksempel på hvorfor vi har valgt at fokusere så meget på testning af koden undervejs. Da vi ellers kunne stå i en situation hvor fremtidige funktioner fejler uventet, og i værste tilfælde de næste to faser. Et andet godt eksempel omhandlede funktionen *outputs*(), som bruges flere steder i *Network.py* da den er fundamentet i for eksempel *all_outputs*(). Da vi valgte at skrive et adskilt modul til at danne permutationerne af 0 og 1, baseret på n, oplevede vi et problem i testning af samtlige funktioner, da vi konsekvent fik en *Out of Range* fejl. Efter lang tid opdagede vi at problemet lå i implementeringen af de funktioner. Modulet *permutations.py* returnerede en liste med lister, og i implementeringen af *outputs*() blev den allerede samlet i en liste. Det gjorde at vores komparatorer ikke kunne få fat i de elementer der skulle sorteres, da det var en tredimensionel liste og ikke todimensionel som kontrakten beskrev. Den type fejl er ekstremt svær at finde da det bygger på flere elementer der fejler. Da hver funktion bruger den foregående. Dette er endnu et eksempel på hvor testning undervejs af koden forhindrede fremtidige problemer.

8 Konklusion

I denne rapport har vi gennemgået den første fase i det tredelte projekt. Der er kreeret to moduler i Python *Comparator.py* og *Network.py* som udgør første fase. Funktionaliteten af de to moduler er blevet tydeliggjort igennem eksempler på test og opbygning. Dertil er gruppens fremgangsmåde dokumenteret og diskuteret løbende. Det ses igennem udførlige tests og dokumentation at det færdige produkt af Fase 1 opfylder de krav der er stillet i problemstillingen, og ligger dertil grund til de næstkommende faser.

9 Litteraturliste

References

[Azu24] AzureX. *SDU-Programmeringsprojek*. Accessed: 08-11-2024. 2024. URL: https://github.com/xXmemestarXx/SDU-Programmeringsprojekt/tree/AzureX/.

[Cru24] Luis Cruz-Filipe. *Gruppeprojekt — Efterår 2024 — Introduktion*. PDF document. 2024.

[Ope24] OpenAI. *ChatGPT*. Accessed: 08-11-2024. 2024. URL: https://openai.com/chatgpt.

10 Bilag

10.1 comparator.py

```python
"""
comparator.py
Made by H8G03: Valdemar, Mathias og Hlynur
"""

from dataclasses import dataclass

"""
Implements a custom dataclass that uses two variables i and j as integers.
This enables creating a instance of a comparator that symbolizes'
the gates of a comparator network.
"""
@dataclass
class Comparator:
    i: int
    j: int

def make_comparator(i: int, j: int) -> Comparator:
    """
    Takes the arguments i and j and returns a Comparator instance with the arguments.

    DOCTEST
    i = 0
    j = 2
    >>> make_comparator(i, j)
    Comparator(0, 2)
    """
    return Comparator(i, j)

def get_i(c: Comparator):
    """
    Auxillary function to get the i-value of a Comparator
    """
    return(int(c.i))

def get_j(c: Comparator):
    """
    Auxillary function to get the j-value of a Comparator
    """
    return int(c.j)

def get_i_and_j(c: Comparator):
    """
    Auxillary function to get the contents of a Comparator
    as a tuple
    """
    return (int(c.i), int(c.j))

def min_channel(c: Comparator) -> int:
    """
    Compares the value of i and j in c and returns the lower value
```

```
52
53      DOCTEST
54      i = 0
55      j = 2
56      >>> min_channel(Comparator(i, j))
57      0
58      """
59      if(c.i > c.j):
60          return c.j
61      else:
62          return c.i
63
64  def max_channel(c: Comparator) -> int:
65      """
66      Compares the value of i and j in c and returns the higher value
67
68      DOCTEST
69      i = 0
70      j = 2
71      >>> max_channel(Comparator(i, j))
72      2
73      """
74      if(c.i < c.j):
75          return c.j
76      else:
77          return c.i
78
79  def is_standard(c: Comparator) -> bool:
80      """
81      Checks if c is a standard comparator (it sets the lowest value on the lowest channel)
82
83      DOCTEST
84      i = 0
85      j = 2
86      c = make_comparator(i, j)
87      >>> is_standard(c)
88      True
89      """
90      return (c.i < c.j) and (c.i != c.j)
91
92  def apply(c: Comparator, w: list[int]) -> list[int]:
93      """
94      Uses a comparator to compare 2 elements in a list of integers and
95      swaps them if needed.
96
97      DOCTEST
98      c = make_comparator(i, j)
99      w = [3,4,2,5]
100     >>> apply(c, w)
101     [2,3,4,5]
102     """
103
104     if(is_standard(c) and w[c.i] > w[c.j]):
105         w[c.i], w[c.j] = w[c.j], w[c.i]
106     return w
107
```

```
108 def all_comparators(n: int) -> list[Comparator]:
109     """
110     Returns a list of all possible comparators on n-channels.
111 
112     DOCTEST
113     n = 3
114     >>> all_comparators(n)
115     [Comparator(i=0, j=1), Comparator(i=0, j=2), Comparator(i=1, j=0),
116      Comparator(i=1, j=2), Comparator(i=2, j=0), Comparator(i=2, j=1)]
117     """
118     comparators=[]
119     # This iterates the nested part of the for loop,
120     # and ensures we print all combinations of i and j.
121     for i in range(n):
122         for j in range(n):
123             # Ensures that i and j are not equal, to comply with the defintion of a comparator
124             if i != j:
125                 comparators.append(Comparator(i,j))
126     return comparators
127 
128 def std_comparators(n: int) -> list[Comparator]:
129     """
130     Returns a list of all standard comparators on n-channels.
131     Standard mean that the j-value is always larger than the
132     i-value.
133 
134     DOCTEST
135     n = 3
136     >>> std_comparators(n)
137     [Comparator(i=0, j=1), Comparator(i=0, j=2), Comparator(i=1, j=2)]
138     """
139     comparators=[]
140     # This iterates the nested part of the for loop,
141     # and ensures we print all combinations of i and j.
142     for i in range(n):
143         for j in range(n):
144             # Checks if i and j are not equal and is a standard comparator
145             if(i != j and is_standard(Comparator(i,j))):
146                 comparators.append(Comparator(i,j))
147     return comparators
148 
149 def to_program(c: Comparator, var: str, aux: str) -> list[str]:
150     """
151     Returns a list of instructions that simulates the Comparator.
152 
153     DOCTEST
154     i = "0"
155     j = "1"
156     var = "new_list"
157     aux = "temp"
158     c = make_comparator(i, j)
159     >>> to_program(c, var, aux)
160     ['if new_list[0] > new_list[1]:', 'temp = new_list[0]',
161     'new_list[0] = new_list[1]', 'new_list[1] = temp']
162     """
163     return [
```

```
    f"if {var}[{c.i}] > {var}[{c.j}]:",
    f"    {aux} = {var}[{c.i}]",
    f"    {var}[{c.i}] = {var}[{c.j}]",
    f"    {var}[{c.j}] = {aux}"]
```

10.2 permutations.py

```
import functools as func

def permutations(n) -> list[list[int]]:
    """
    Strategy:
    Start with all zeros and then edit
    the lists one-by-one
    """

    """Start by making an empty list"""
    all_permu = []

    """Then make a list that contain n amount of 0's """
    all_zero = ["0"] * n

    """Calculate hvor many permutations of n there is"""
    amount_permu = 2**n

    """
    Start by going through the lists of 0's one-by-one
    until reaching the number of permutations
    """

    i = 0
    while i < amount_permu:

        v = format(i,"b").zfill(n)

        all_permu.append(list(map(lambda x,y : int(x) | int(y),all_zero,v,)))
        i = i + 1

    return all_permu
```

10.3 network.py

```
"""
network.py
Made by H8G03: Valdemar, Mathias og Hlynur
"""

from dataclasses import dataclass
import comparator as Comp
from permutations import * #Note: We made this module ourselves

"""
```

```
11 We start by importing the dataclass functionality,
12 the comparator module and our own permutations module.
13 """
14
15 @dataclass
16 class Network():
17     """
18     The network is implemented as a dataclass,
19     whose only property is to create a list of comparator objects.
20     """
21     network: list
22
23 def to_string(net: Network) -> str:
24     """
25     Writes the contents of a Network object to a String
26     Note: Useful for testing
27
28     DOCTEST
29     >>> net.network = [obj_1, obj_2]
30     '[obj_1, obj_2]'
31     """
32     return (f"{net.network}")
33
34 def empty_network() -> dataclass:
35     """
36     Creates an empty Network object that only contains an empty list
37     """
38     return Network([])
39
40 def append(c: Comp, net: Network) -> None:
41     """
42     Appends a comparator c to the Network, net.
43
44     Iterates on net and does not make a copy of it in memory.
45
46     DOCTEST
47     >>> append(Comparator_1,Network)
48     >>> append(Comparator_2,Network)
49     Network = [Comparator_1, Comparator_2]
50     """
51     net.network.append(c)
52
53 def size(net: Network) -> int:
54     """
55     Returns the amount of Comparators in a Network
56     in the form of an integer
57
58     DOCTEST
59     Network.network = [Comparator_1,Comparator_2]
60     >>> size(Network)
61     2
62     """
63     return len(net.network)
64
65 def max_channel(net: Network) -> int:
66     """
```

```
    Returns the largest j-value in the network. The main difference in not using
    max.channel() from comparator.py is that we avoid
    non-standard Comparators.

    Instead it uses get_j() from comparator as an auxiliary function
    Therefore Comparator.py needs to be imported.

    Requirement:
    size(net) > 0

    DOCTEST
    Comparator_1.i = 2
    Comparator_1.j = 4

    Comparator_2.i = 1
    Comparator_2.j = 5

    Comparator_3.i = 2
    Comparator_3.j = -1

    append(Comparator_1,Network)
    append(Comparator_2,Network)
    append(Comparator_3,Network)

    >>> max_channel(Network)
    5
    """

    max = Comp.get_j(net.network[0])

    """
    We start by equating the maximum channel of the network to
    the maximum channel of the first comparator of the network.

    This also ensures that if there is only one comparator in the
    network, the following for-each loop is skipped.
    """
    for i in range(1,size(net)):
        """
        Thereafter we check the maximum channel of each Comparator one-by-one
        except the first one, since we already checked it.
        """

        if Comp.get_j(net.network[i]) > max:
            """
            If the function finds a bigger number via the for-loop
            and if-statement, it overwrites the previous maximum channel
            and continues to check the other Comparators
            """
            max = Comp.get_j(net.network[i])
    return max

def is_standard(net: Network) -> bool:
    """
    Checks whether the input Network only
    contains standard comparators or not
```

```
123
124     Note:
125     A standard comparator is a comparator where
126     its j-value is larger than its i-value
127
128     Uses is.standard() from comparator module as an
129     auxiliary function. Therefore Comparator.py
130     needs to be imported.
131
132     Requirement:
133     size(net) > 0
134
135     DOCTEST
136     Net_1 = empty_network()
137     Com_1 = comparator_1.make_comparator(7,1)
138     Com_2 = comparator_1.make_comparator(3,5)
139     append(Com_1,Net_1)
140     append(Com_2,Net_1)
141
142     >>> is_standard(Net_1)
143     False
144     """
145
146     for i in range(0,size(net)):
147         """
148         We go through the list of Comparators and check one-by-one if they
149         are standard or not. If at least one of them is non-standard then
150         the whole function returns false.
151         """
152
153         if Comp.is_standard(net.network[i]) is False:
154             return False
155
156     """
157     If the for-each loop reached the end of the list without ever finding at
158     least one non-standard Comparator then the function returns True.
159
160     None are False => All are True
161     """
162     return True
163
164 def apply(net: Network, w: list[int]) -> list[int]:
165     """
166     Sorts a single list of integers using comparators
167     in a network.
168
169     Requirement:
170     size(net) > 0
171     len(w) > 0
172
173     DOCTEST
174     net = empty_network()
175     Com_1 = Comp.make_comparator(1,3)
176     Com_2 = Comp.make_comparator(2,3)
177
178     w = [1,2,4,3]
```

```
179
180     append(Com_1, net)
181     append(Com_2, net)
182
183     >>> apply(net, w)
184     [1, 2, 3, 4]
185     """
186
187     for i in range(0, size(net)):
188         """
189         For every Comparator in the network, apply the
190         Comparators on the list, so every Comparator is
191         used at least once
192         """
193         Comp.apply(net.network[i],w)
194
195     return w
196
197 def outputs(net: Network, w: list[list[int]]) -> list[list[int]]:
198     """
199     Returns a sorted list of lists containing no duplicates
200     The list themselves are not sorted
201
202     Requirement:
203     size(net) > 0
204     len(net) > 0
205
206     DOCTEST
207     net = empty_network()
208
209     Com_1 = Comp.make_comparator(0,1)
210     Com_2 = Comp.make_comparator(1,2)
211
212     append(Com_1,net)
213     append(Com_2,net)
214
215     v = [[36,25,25],[36563236,63425,4433660]]
216
217     >>> outputs(net, v)
218     [[25,25,36],[63425,4433660,36563236]]
219     """
220
221     for i in range(0,len(w)):
222         """
223         Sorts the individual lists inside the w
224         by using apply() as an auxiliary function
225         """
226         apply(net,(w[i]))
227
228         """
229         Depends on if we want to also remove the duplicate
230         numbers in the list
231         """
232
233         """
234         Python supports the datatype sets which are unorderd
```

```
        collections of elements with no duplicates.
        Python can convert a list to a set using the set()
        function which will also remove any duplicate elements.

        But by converting the set back to a list using the
        list() function, we essentially get the original list
        without duplicates, since we didn't change the order
        of the set or modified it in any other way.

        w[i] = set(w[i])
        w[i] = list(w[i])
        """
    return w


def all_outputs(net: Network, n: int) -> list[list[int]]:
    """
    Returns all permutations of 0 and 1 of n length,
    essentially the same as counting from 0 to n in binary,
    and then sortes them and removes repeats.

    Requirement:
    n > 0

    DOCTEST
    >>> all_outputs(1)
    [[0],[1]]

    >>> all_outputs(3)
    [[000],[001],[010],[011],[100],[101],[111]]
    """

    permu = permutations(n)

    permu = outputs(net,permu)

    permu_no_dupe = []

    for i in range(0,len(permu)):
        if permu[i] not in permu_no_dupe:
           permu_no_dupe.append(permu[i])

    return permu_no_dupe


def is_sorting(net: Network, size: int) -> bool:
    """
    We will refer the variable size to letter n

    Checks wheter a sorting network is able to correctly sort
    a network with n amount of channels with the comparators
    it has.

    DOCTEST
    Net_1 = empty_network()
    Com_1 = Comp.make_comparator(1,3)
```

```
    Com_2 = Comp.make_comparator(2,4)
    append(Com_1,Net_1)
    append(Com_2,Net_1)

    Net_2 = empty_network()
    Com_3 = Comp.make_comparator(0,1)
    com_4 = Comp.make_comparator(0,2)
    Com_5 = Comp.make_comparator(1,2)
    append(Com_3,Net_2)
    append(Com_4,Net_2)
    append(Com_5,Net_2

    >>> is_sorting(Net_1, 3)
    False

    >>> is_sorting(Net_2, 3)
    True
    """
    """
    Checking the minimum criteria for the network.
    """
    if len(net.network) == 0:
        return False

    if max_channel(net) != (size-1) : #Code using 0-indexing
        return False

    """
    First we create all permutations of 0 and 1
    of n length and place it in a list
    """
    permu = all_outputs(net,size)

    """
    We check if the list of lists of integers are
    sorted correctly be checking every integer
    one-by-one. If a previous value is larger
    than the following value then the list
    is sorted incorrectly. If all previous values
    are less than the following values then the list
    is sorted correctly.
    """

    for i in range(0,len(permu)-1):
        for j in range(0,len(permu[i])-1):
            if permu[i][j] > permu[i][j+1]:
                return False
    return True

def to_program(net: Network, var: str, aux: str)-> list[str]:
    """
    Returns a list of instructions that simulates the Comparator network.

    DOCTEST
    net = empty_network()
    Com_1 = Comp.make_comparator(0,1)
```

```
    Com_2 = Comp.make_comparator(2,3)
    Com_3 = Comp.make_comparator(0,2)
    append(Com_1, net)
    append(Com_2, net)
    append(Com_3, net)
    >>> to_program(Net1, var, aux)
    [['if var[0] > var[1]:', 'aux = var[0]', 'var[0] = var[1]', 'var[1] = aux'],
    ['if var[2] > var[3]:', 'aux = var[2]', 'var[2] = var[3]', 'var[3] = aux'],
    ['if var[0] > var[2]:', 'aux = var[0]', 'var[0] = var[2]', 'var[2] = aux']]
    """
    # Note: The output from this program can be run by the command exec()
    # and should result in the same execution as apply()
    returned_list = []

    # Iterates through the entire network
    for i in range(size(net)):
         # Iterates through the list from to_program() in comparator.py
        for j in range(len(Comp.to_program(net.network[i], var, aux))):
            # Appends each line as a new element in returned_list
            returned_list.append(Comp.to_program(net.network[i], var, aux)[j])
            # Next line is only needed if you need to be able to run the code
            # Creates a new line to seperate each output from to_program() in comparator.
            returned_list.append("\n")
    return returned_list
```

10.4 test.py

```
import network as Netw
import comparator as Comp

print(f"")
print(f"------- test comparator.py -------")
print(f"")
print(f"------- test 1 begin -------")
print(f"")
print(f"Testing make_comparator()")
print(f"")
i = 0
j = 2
c = Comp.make_comparator(i, j)
print(f"i = 0, j = 2 => {c}")

print(f"")
print(f"------- test 1 end -------")
print(f"")

print(f"------- test 2 begin -------")
print(f"")
print(f"Testing min_channel() & max_channel()")
print(f"")
i = 0
j = 2
c = Comp.make_comparator(i, j)
print(f"{c} => min = {Comp.min_channel(c)}, max = {Comp.max_channel(c)}")
```

```
28
29  print(f"")
30  print(f"------- test 2 end -------")
31  print(f"")
32
33  print(f"------- test 3 begin -------")
34  print(f"")
35  print(f"Testing is_standard()")
36  print(f"")
37  i = 0
38  j = 2
39  c = Comp.make_comparator(i, j)
40  print(f"i = {i} and j = {j} => {Comp.is_standard(c)}")
41  i = 2
42  j = 0
43  c = Comp.make_comparator(i, j)
44  print(f"i = {i} and j = {j} => {Comp.is_standard(c)}")
45  i = 2
46  j = 2
47  c = Comp.make_comparator(i, j)
48  print(f"i = {i} and j = {j} => {Comp.is_standard(c)}")
49
50  print(f"")
51  print(f"------- test 3 end -------")
52  print(f"")
53
54  print(f"------- test 4 begin -------")
55  print(f"")
56  print(f"Testing apply()")
57  print(f"")
58  comp = Comp.make_comparator(i = 0, j = 0)
59  w = [3,4,2,5]
60  print(f"Normal: [3,4,2,5] => {Comp.apply(comp, w)}")
61  v = [1,2,4,5]
62  print(f"Sorted: [1,2,4,5] => {Comp.apply(comp, v)}")
63  z = [1,3,4,4]
64  print(f"Duplicated Sorted: [1,3,4,4] => {Comp.apply(comp, z)}")
65  k = [2,1,1,3]
66  print(f"Duplicated: [2,1,1,3] => {Comp.apply(comp, k)}")
67  print(f"")
68  print(f"------- test 4 end -------")
69  print(f"")
70
71  print(f"------- test 5 begin -------")
72  print(f"")
73  print(f"Testing all_comparators()")
74  print(f"")
75  n = 3
76  print(f"n = 3 => {Comp.all_comparators(n)}")
77  n = 0
78  print(f"n = 0 => {Comp.all_comparators(n)}")
79  n = -3
80  print(f"n = -3 => {Comp.all_comparators(n)}")
81  print(f"")
82  print(f"------- test 5 end -------")
83  print(f"")
```

```
84
85  print(f"------- test 6 begin -------")
86  print(f"")
87  print(f"Testing std_comparators()")
88  print(f"")
89  n = 3
90  print(f"n = 3 => {Comp.std_comparators(n)}")
91  n = 0
92  print(f"n = 0 => {Comp.std_comparators(n)}")
93  n = -3
94  print(f"n = -3 => {Comp.std_comparators(n)}")
95  print(f"")
96  print(f"------- test 6 end -------")
97  print(f"")
98
99  print(f"------- test 7 begin -------")
100 print(f"")
101 print(f"Testing to_program()")
102 print(f"")
103 i = "i"
104 j = "j"
105 to_program_c = Comp.make_comparator(i = str(i), j = str(j))
106 print(Comp.to_program(Comp.Comparator(i, j), var = "network", aux = "temp"))
107 print(f"")
108 print(f"------- test 7 end -------")
109
110 print(f"")
111 print(f"------- test network.py -------")
112 print(f"")
113 print(f"------- test 1 start -------")
114 print(f"")
115 print(f"Testing to_string()")
116
117 Net = Netw.empty_network()
118 Com_1 = Comp.make_comparator(0,1)
119 Com_2 = Comp.make_comparator(1,2)
120 Netw.append(Com_1,Net)
121 Netw.append(Com_2,Net)
122
123 print(f"The network object to convert: {Net}")
124 print(f"Converted to string: {Netw.to_string(Net)}")
125
126 print(f"")
127 print(f"------- test 1 end -------")
128 print(f"")
129
130 print(f"")
131 print(f"------- test 2 start -------")
132 print(f"")
133 print(f"Testing empty_network()")
134 print(f"Test a empty network {Netw.empty_network()}")
135
136
137 print(f"")
138 print(f"------- test 2 end -------")
139 print(f"")
```

```
140
141 print(f"")
142 print(f"------- test 3 start -------")
143 print(f"")
144
145 Net = Netw.empty_network()
146 Com_1 = Comp.make_comparator(0,1)
147 Com_2 = Comp.make_comparator(1,2)
148
149 print(f"Testing append()")
150 print(f"Before append(): {Net}")
151 Netw.append(Com_1,Net)
152 Netw.append(Com_2,Net)
153 print(f"After append(): {Net}")
154
155 print(f"")
156 print(f"------- test 3 end -------")
157 print(f"")
158
159 print(f"")
160 print(f"------- test 4 start -------")
161 print(f"")
162
163 print(f"Testing size()")
164 Net = Netw.empty_network()
165 print(f"Before append, {Net}")
166 print(f"Before append, size: {Netw.size(Net)}")
167
168 Com_1 = Comp.make_comparator(0,1)
169 Com_2 = Comp.make_comparator(1,2)
170 Netw.append(Com_1,Net)
171 Netw.append(Com_2,Net)
172
173 print(f"After append, {Net}")
174 print(f"After append, size: {Netw.size(Net)}")
175
176 print(f"")
177 print(f"------- test 4 end -------")
178 print(f"")
179
180 print(f"")
181 print(f"------- test 5 start -------")
182 print(f"")
183
184 print(f"Testing max_channel()")
185
186 Net = Netw.empty_network()
187 Com_1 = Comp.make_comparator(0,1)
188 Netw.append(Com_1,Net)
189 print(f"Network to test: {Net}")
190 print(f"Max channel: {Netw.max_channel(Net)}")
191
192 Com_2 = Comp.make_comparator(1,2)
193 Com_3 = Comp.make_comparator(3,4)
194
195 Netw.append(Com_2,Net)
```

```
196 Netw.append(Com_3,Net)
197
198 print(f"Network to test: {Net}")
199 print(f"Max channel: {Netw.max_channel(Net)}")
200
201 print(f"")
202 print(f"------- test 5 end -------")
203 print(f"")
204
205 print(f"")
206 print(f"------- test 6 start -------")
207 print(f"")
208
209 print(f"Testing is_standard()")
210
211 Net = Netw.empty_network()
212 Com_1 = Comp.make_comparator(0,1)
213 Com_2 = Comp.make_comparator(1,2)
214 Com_3 = Comp.make_comparator(4,3)
215 Netw.append(Com_1,Net)
216 Netw.append(Com_2,Net)
217 Netw.append(Com_3,Net)
218
219 print(f"")
220 print(f"Network to test: {Net}")
221 print(f"Does the network only contain std comparators: {Netw.is_standard(Net)}")
222 print(f"")
223 Net_1 = Netw.empty_network()
224 Com_1 = Comp.make_comparator(0,1)
225 Com_2 = Comp.make_comparator(4,5)
226 Com_3 = Comp.make_comparator(2,5)
227 Netw.append(Com_1,Net_1)
228 Netw.append(Com_2,Net_1)
229 Netw.append(Com_3,Net_1)
230
231 print(f"Network to test: {Net_1}")
232 print(f"Does the network only contain std comparators: {Netw.is_standard(Net_1)}")
233
234 print(f"")
235 print(f"------- test 6 end -------")
236 print(f"")
237
238 print(f"")
239 print(f"------- test 7 start -------")
240 print(f"")
241
242 print(f"Testing apply()")
243
244 Net = Netw.empty_network()
245 Com_1 = Comp.make_comparator(1,2)
246 Com_2 = Comp.make_comparator(3,4)
247 Com_3 = Comp.make_comparator(0,1)
248
249
250 Netw.append(Com_1,Net)
251 Netw.append(Com_2,Net)
```

```
252 Netw.append(Com_3,Net)
253
254 v = [1,2,0,4,3]
255
256 print(f"Testing sorting the network with correct comparators")
257 print(f"")
258 print(f"{Net}")
259 print(f"")
260 print(f"List before: {v}")
261 print(f"")
262 print(f"Each step taken for sorting the list")
263 print(f"")
264 print(f"List after: {Netw.apply(Net,v)}")
265 print(f"")
266
267 Net_1 = Netw.empty_network()
268 Com_1 = Comp.make_comparator(1,3)
269 Com_2 = Comp.make_comparator(1,2)
270 Com_3 = Comp.make_comparator(3,4)
271 Com_4 = Comp.make_comparator(2,3)
272
273 Netw.append(Com_1,Net_1)
274 Netw.append(Com_2,Net_1)
275 Netw.append(Com_3,Net_1)
276 Netw.append(Com_4,Net_1)
277
278 w = [1,2,0,4,3]
279
280 print(f"Testing sorting the network with wrong comparators")
281 print(f"")
282 print(f"{Net_1}")
283 print(f"")
284 print(f"List before: {w}")
285 print(f"")
286 print(f"List after: {Netw.apply(Net_1,w)}")
287
288 print(f"")
289 print(f"------- test 7 end -------")
290 print(f"")
291
292 print(f"")
293 print(f"------- test 8 start -------")
294 print(f"")
295
296 print(f"Testing outputs()")
297
298 Net = Netw.empty_network()
299 Com_1 = Comp.make_comparator(0,1)
300 Com_2 = Comp.make_comparator(0,2)
301 Com_3 = Comp.make_comparator(1,2)
302
303 Netw.append(Com_1,Net)
304 Netw.append(Com_2,Net)
305 Netw.append(Com_3,Net)
306
307 w = [[36,25,25],[36563236,63425,4433660]]
```

```
308 print(f"Testing sorting the network with correct comparators")
309 print(f"")
310 print(f"{Net}")
311 print(f"List before: {w}")
312 print(f"")
313 print(f"List after: {Netw.outputs(Net,w)}")
314 print(f"")
315
316 print(f"")
317 print(f"------- test 8 end -------")
318 print(f"")
319
320 print(f"")
321 print(f"------- test 9 start -------")
322 print(f"")
323
324 print(f"Testing all_outputs()")
325
326 Net = Netw.empty_network()
327 Com_1 = Comp.make_comparator(0,1)
328 Com_2 = Comp.make_comparator(0,2)
329 Com_3 = Comp.make_comparator(1,2)
330
331 Netw.append(Com_1,Net)
332 Netw.append(Com_2,Net)
333 Netw.append(Com_3,Net)
334
335 print(f"Network to test with: {Net}")
336 print(f"All permutations unsorted: {Netw.permutations(3)}")
337 print(f"All permutations sorted: {Netw.all_outputs(Net, 3)}")
338
339 print(f"")
340 print(f"------- test 9 end -------")
341 print(f"")
342
343 print(f"")
344 print(f"------- test 10 start -------")
345 print(f"")
346
347 print(f"Testing is_sorting()")
348
349 print(f"")
350 print(f"For us to check is_sorting(), we need to check the cases where,")
351 print(f"Evertyhing is OK, and then all the false cases possible, being")
352 print(f"len == 0, max_channel != size-1, all if statements is true,")
353 print(f"but one comparator is missing")
354 print(f"")
355
356 Net = Netw.empty_network()
357
358 com1 = Comp.make_comparator(0,1)
359 com2 = Comp.make_comparator(0,2)
360 com3 = Comp.make_comparator(1,2)
361
362 Netw.append(com1,Net)
363 Netw.append(com2,Net)
```

```
364 Netw.append(com3,Net)
365
366 print(f"First case where the three criteria are OK")
367 print(f"Max: {Netw.max_channel(Net)}")
368
369 print(f"The network contains: {Netw.to_string(Net)}")
370 print(f"")
371
372 print(f"Is the network able to sort a network of size {3}: {Netw.is_sorting(Net,3)}")
373
374 Net = Netw.empty_network()
375
376 print(f"")
377 print(f"Second case where the network is empty")
378 print(f"Max: 0")
379
380 print(f"The network contains: {Netw.to_string(Net)}")
381 print(f"")
382
383 print(f"Is the network able to sort a network of size {0}: {Netw.is_sorting(Net,0)}")
384
385 print(f"")
386 print(f"Third case where the network max_channel is not equal to size-1")
387
388 Net = Netw.empty_network()
389
390 Com_1 = Comp.make_comparator(0,1)
391 Com_2 = Comp.make_comparator(0,2)
392 Com_3 = Comp.make_comparator(1,3)
393
394 Netw.append(Com_1,Net)
395 Netw.append(Com_2,Net)
396 Netw.append(Com_3,Net)
397
398
399 print(f"Max: {Netw.max_channel(Net)}")
400
401 print(f"The network contains: {Netw.to_string(Net)}")
402 print(f"")
403
404 print(f"Is the network able to sort a network of size {4}: {Netw.is_sorting(Net,5)}")
405
406 print(f"")
407 print(f"Fourth case where the network is missing one singular comparator")
408
409 Net = Netw.empty_network()
410
411 Com_1 = Comp.make_comparator(0,1)
412 Com_2 = Comp.make_comparator(1,2)
413
414 Netw.append(Com_1,Net)
415 Netw.append(Com_2,Net)
416
417 print(f"Max: {Netw.max_channel(Net)}")
418
419 print(f"The network contains: {Netw.to_string(Net)}")
```

```
420 print(f"")
421
422 print(f"Is the network able to sort a network of size {3}: {Netw.is_sorting(Net,3)}")
423
424 print(f"")
425 print(f"------- test 10 end -------")
426 print(f"")
427
428 print(f"")
429 print(f"------- test 11 start -------")
430 print(f"")
431
432 print(f"Testing to_program()")
433 print(f"")
434
435 Net = Netw.empty_network()
436 v = [9,5,8,23]
437 len_v = len(v)
438 for i in range(len_v - 1):
439     for j in range(len_v - 1):
440         Com = Comp.make_comparator(j, j+1)
441         Netw.append(Com, Net)
442 outputs = Netw.to_program(Net, "v", "aux")
443 print(outputs)
444
445 empty_string = ""
446
447 for x in range(len(outputs)):
448     empty_string += outputs[x]
449
450 print(f"list before: {v}")
451 aux:int
452 exec(empty_string)
453 print(f"list after: {v}")
454
455 print(f"")
456 print(f"------- test 11 end -------")
```