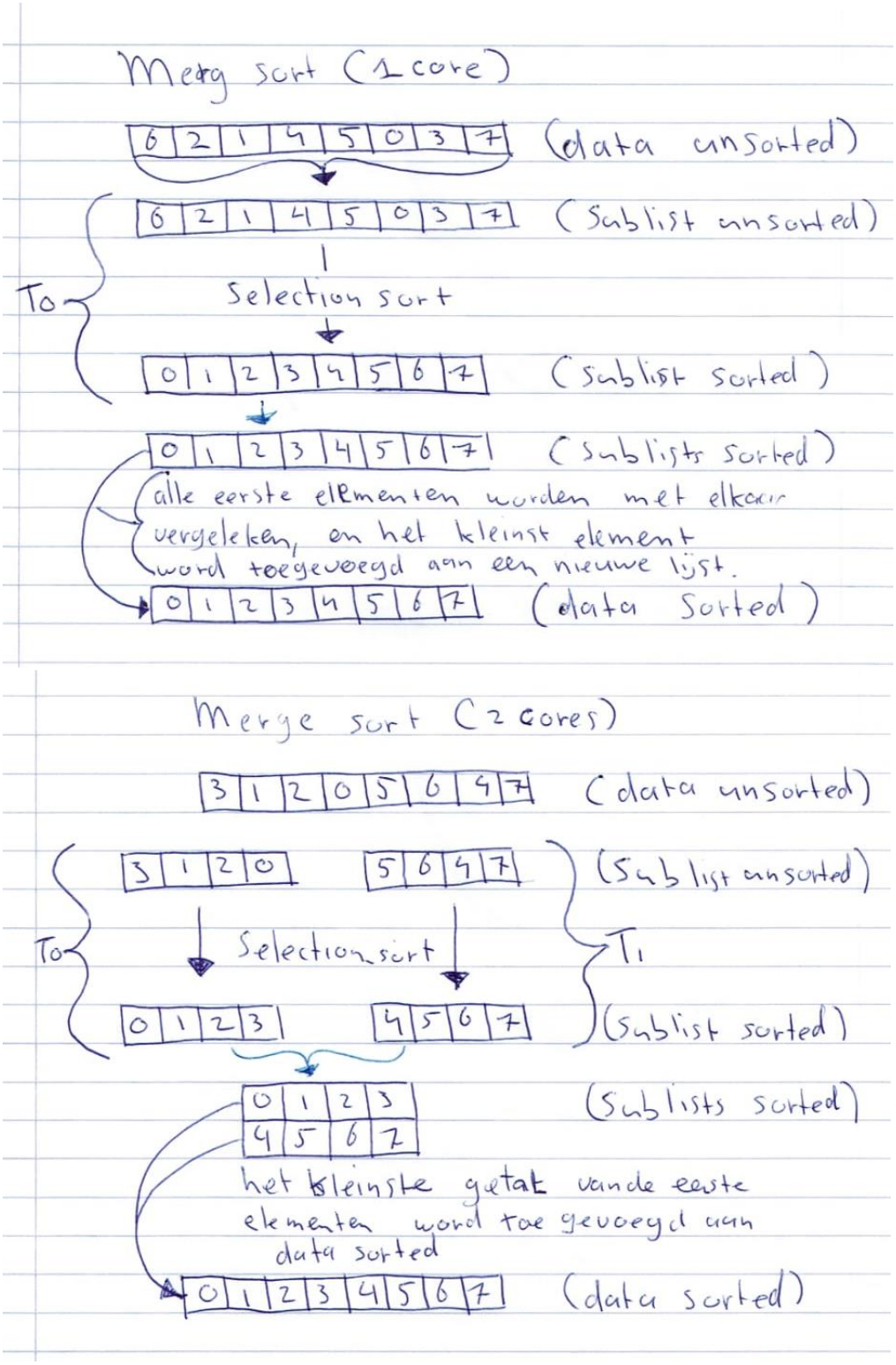
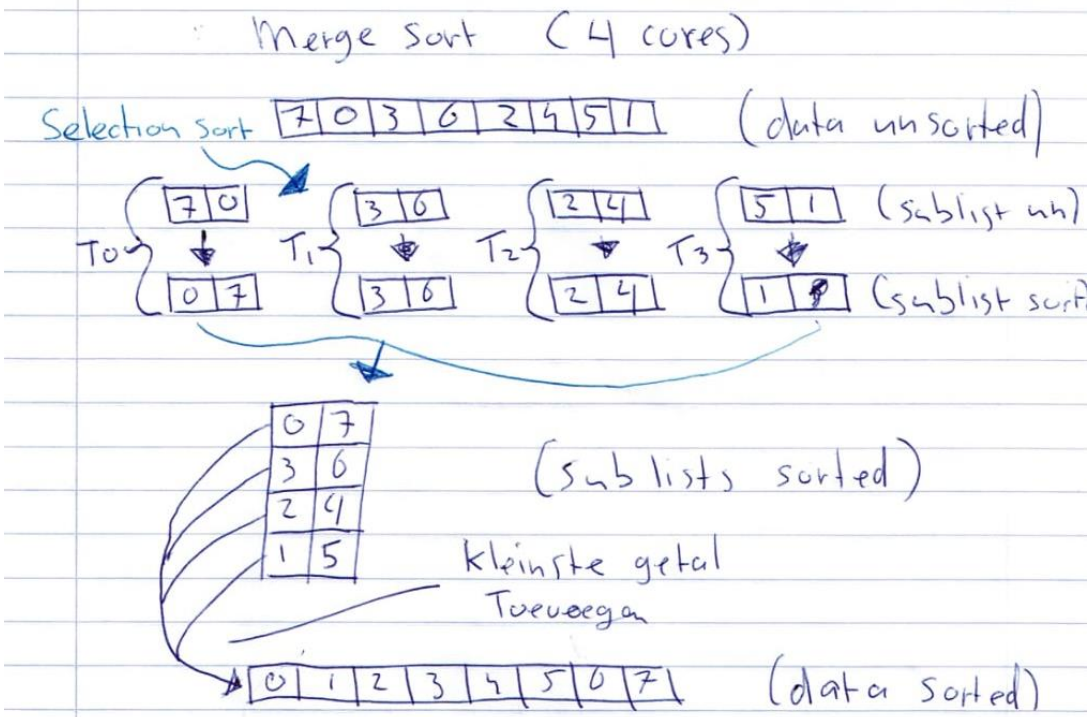
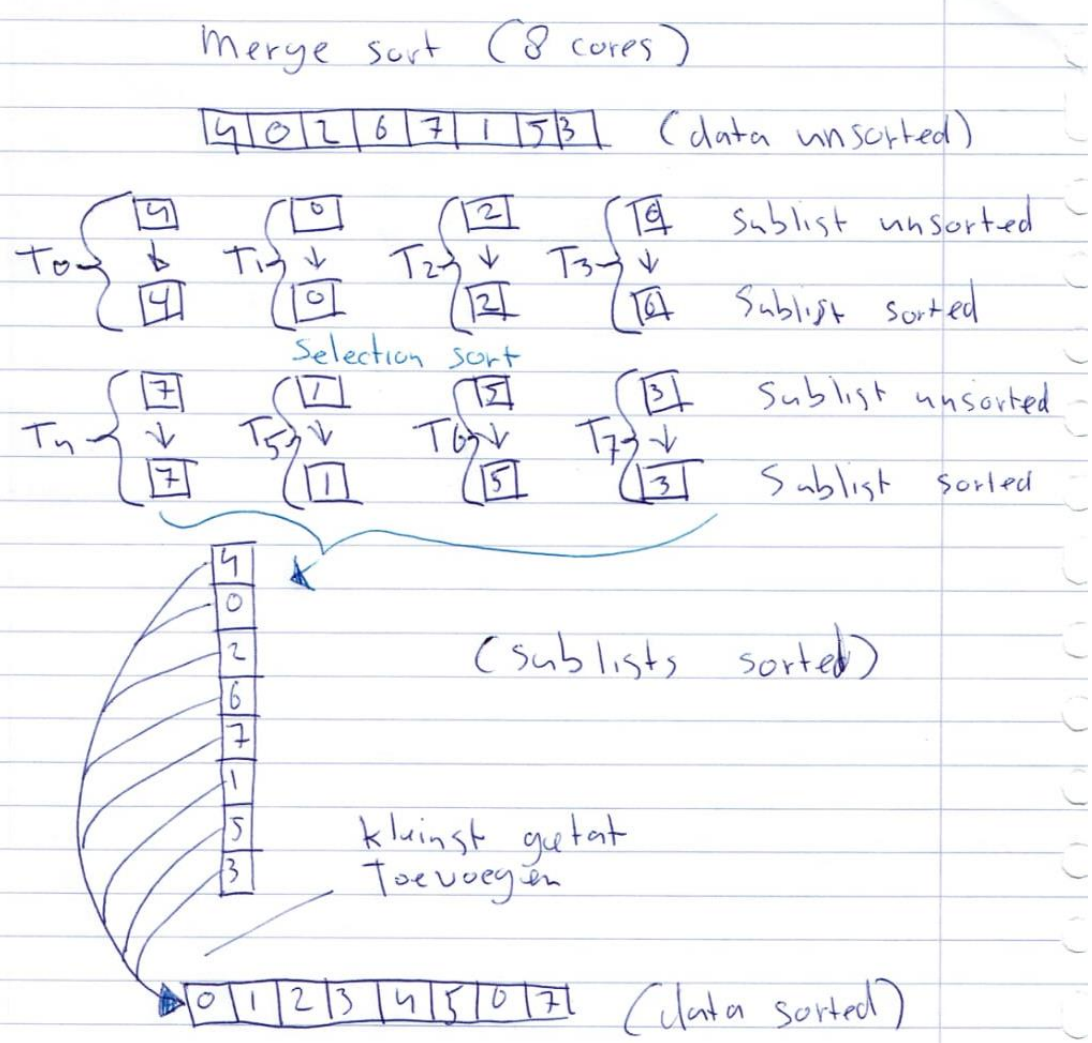
**1. Ontwerp**


Merg sort (1 core)
6 2 1 4 5 0 3 7
(data unsorted)
6 2 1 4 5 0 3 7
(Sublist unsorted)
To
Selection sort
0 1 2 3 4 5 6 7
(Sublist sorted)
0 1 2 3 4 5 6 7
(Sublists sorted)
alle eerste elementen worden met elkaar vergeleken, en het kleinst element word toegevoegd aan een nieuwe lijst.
0 1 2 3 4 5 6 7
(data Sorted)
Merge sort (2 cores)
3 1 2 0 5 6 4 7
(data unsorted)
3 1 2 0
5 6 4 7
(Sublist unsorted)
To
Selection sort
T1
0 1 2 3
4 5 6 7
(Sublist sorted)
0 1 2 3
4 5 6 7
(Sublists sorted)
het kleinste getal vande eerste elementen word toe gevoegd aan data sorted
0 1 2 3 4 5 6 7
(data sorted)


Merge Sort (4 cores)
Selection sort
7 0 3 6 2 4 5 1
(data unsorted)
T0
T1
T2
T3
7 0
3 6
2 4
5 1
(sublist un)
0 7
3 6
2 4
1 5
(sublist sort
(sublists sorted)
kleinste getal
Toevoegen
0 1 2 3 4 5 6 7
(data sorted)


Merge sort (8 cores)
4 0 2 6 7 1 5 3
(data unsorted)
T0
T1
T2
T3
Sublist unsorted
Sublist sorted
Selection sort
T4
T5
T6
T7
Sublist unsorted
Sublist sorted
(sublists sorted)
kleinst getal
Toevoegen
0 1 2 3 4 5 6 7
(data sorted)

2.

2.

2.

**2. Complexiteit**

Ik er op uit gekomen dat de volgende notatie de complexiteit aangeeft van het algoritme. Wat hierbij niet vergelijkbaar is, is dat je hier rekening moet houden met het aantal threads wat je gebruikt. Je moet dus bij het proces waarbij je multi threading gebruikt nog delen door het aantal threads dat je gebruikt.

O((n^2)/num_threads+(n^2))

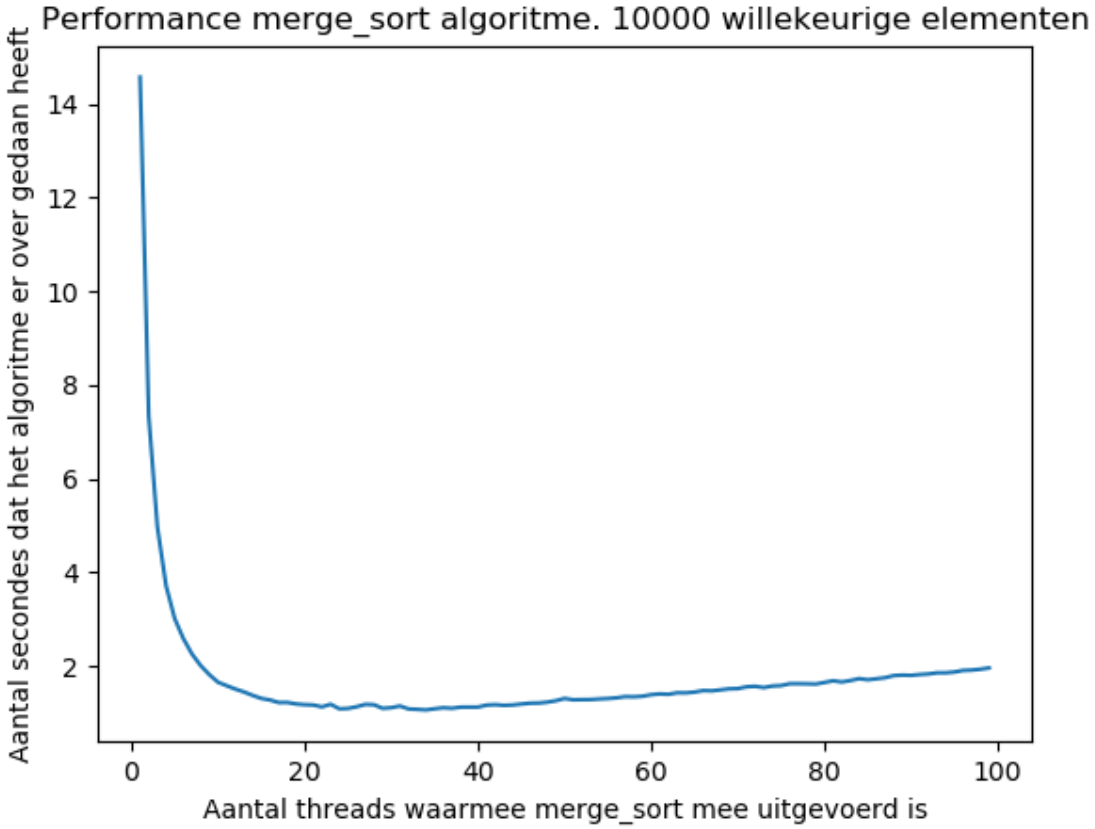
**3. communicatie overhead**

De hoeveelheid communicatie die er plaats vindt tijdens het uitvoeren van het algoritme hangt af van hoeveel threads je gebruikt voor het algoritme. Elke thread geef ik een sublijst, die sublijst wordt vervolgens gesorteerd en tot slot word die sublijst weer uit de thread gehaald. Dit betekent dus dat de communicatie overhead voor mijn algoritme *aantal_threads * 2* is.

**4. Resultaten**

Ik heb hier de resultaten van de performance van mijn sorteer algoritme, mijn algoritme wordt er wel degelijk sneller van als ik hem meer threads geef, waarbij het algoritme ongeveer op de 30 threads het snelste is en daarne weer langzaam langzamer wordt, al zou dit niet moeten kunnen in python.

merge_sort started with 1 thread(s).
merge_sort ended with a total runtime of 14.570921 seconds.
---
merge_sort started with 2 thread(s).
merge_sort ended with a total runtime of 7.331074 seconds.
---
merge_sort started with 3 thread(s).
merge_sort ended with a total runtime of 4.969496 seconds.
---
merge_sort started with 4 thread(s).
merge_sort ended with a total runtime of 3.711963 seconds.
---
merge_sort started with 5 thread(s).
merge_sort ended with a total runtime of 3.009457 seconds.
---
merge_sort started with 6 thread(s).
merge_sort ended with a total runtime of 2.582066 seconds.
---
merge_sort started with 7 thread(s).
merge_sort ended with a total runtime of 2.252439 seconds.
---
merge_sort started with 8 thread(s).
merge_sort ended with a total runtime of 2.005697 seconds.
---


Performance merge_sort algoritme. 10000 willekeurige elementen
Aantal secondes dat het algoritme er over gedaan heeft
14
12
10
8
6
4
2
0
20
40
60
80
100
Aantal threads waarmee merge_sort mee uitgevoerd is