Deloppgave 1:

Har implementert algoritmene fra prekoden og koblet disse på prekoden. Har også lagd en liten modifikasjon på prekoden som lager en fil, _is_sorted.out. Som sjekker om svarene produsert av algoritmene faktisk er 100% sortert.

Insertion Sort	**Quick sort**	**Merge sort**	**Selection sort**	**Bubble sort**
$O(n^2)$	O(n(log n))	O(n(log n))	$O(n^2)$	$O(n^2)$
insertion.py	quick.py	merge.py	selection.py	bubble.py

Deloppgave 2:

Hvordan swaps og comparisons måles

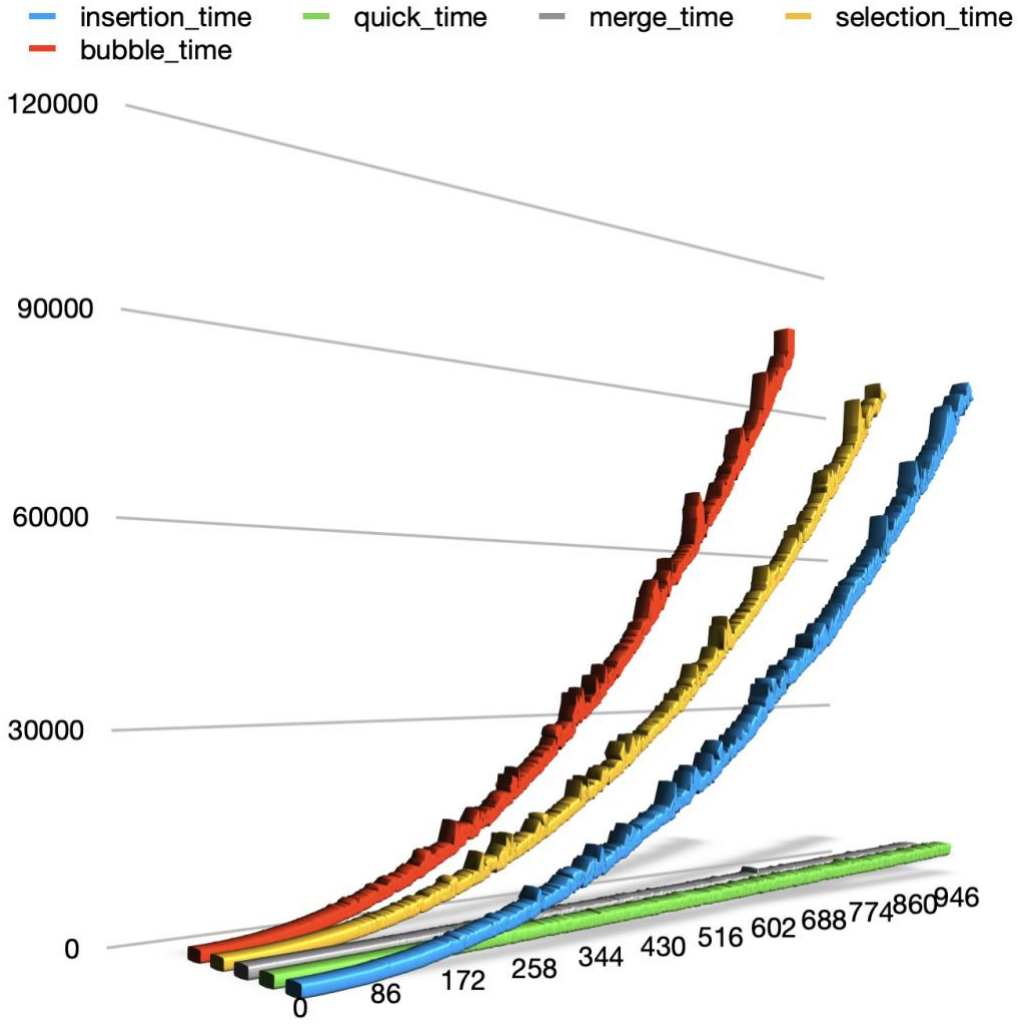
I denne obligen måles antall bytter og sammenligninger ved hjelp av to klasser, CountCompares (CC) og CountSwaps (CS). CS holder en liste med CC objekter. Listen som sendes inn til sortering i de forskjellige algoritmene er et objekt av klassen CS. For å omrokere elementer bruker vi CS sin metode, swap(i,j). Denne bytter plass på to interne CC objekter og øker swaptelleren med en. Comparisons telles vet at klassen CC har implementert magiske metoder som __lt__ og __eq__, som gjør at man kan sammenligne objekter i CS listen med hverandre som om de var tall, samtidig som vi automatisk får telt antall sammenligninger. Denne dataen brukes så i innlevering2runner.py for å printe ut summen av swaps og comparisons.

.

Deloppgave 3:

Hvordan stemmer kjøretid overens med store O

Kjøretidsanalysen ser visuelt ut til å stemme godt for alle algoritmene ved random_1000. Som jeg vil gå inn på i senere deloppgaver skiller insertion sort seg spesielt ut ved nesten sorterte datasett. Vi kan lese fra dataen i random_1000 at vi fra n=500 til n=1000 får en ca. firedobling i kjøretid. I det samme spannet får vi litt over en dobling i kjøretid på n log(n) algoritmene. Dette og tyder på kjøretid lik O-notasjonen.

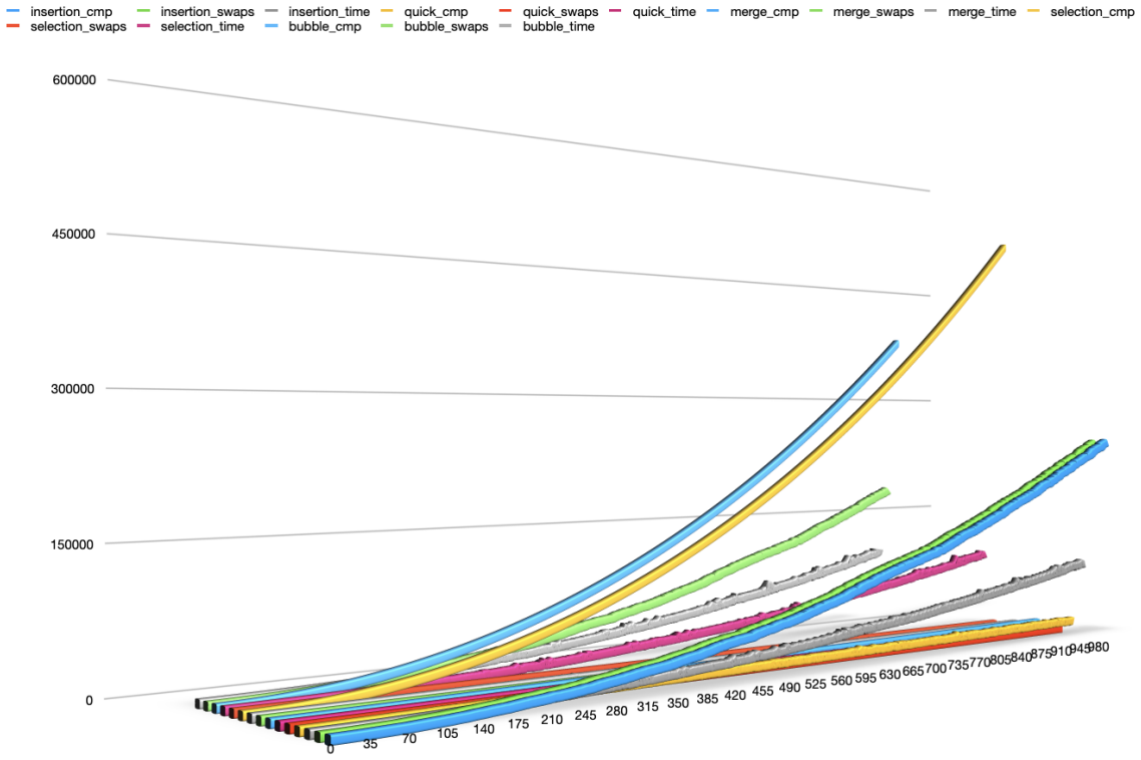


Figur 2: Kjøretider, random_1000.

Forholdet mellom bytter, sammenligninger og kjøretid

Vedlagt ser vi grafene som viser sammenhengene mellom compare, swap og tid på forskjellige sorteringsalgoritmer.

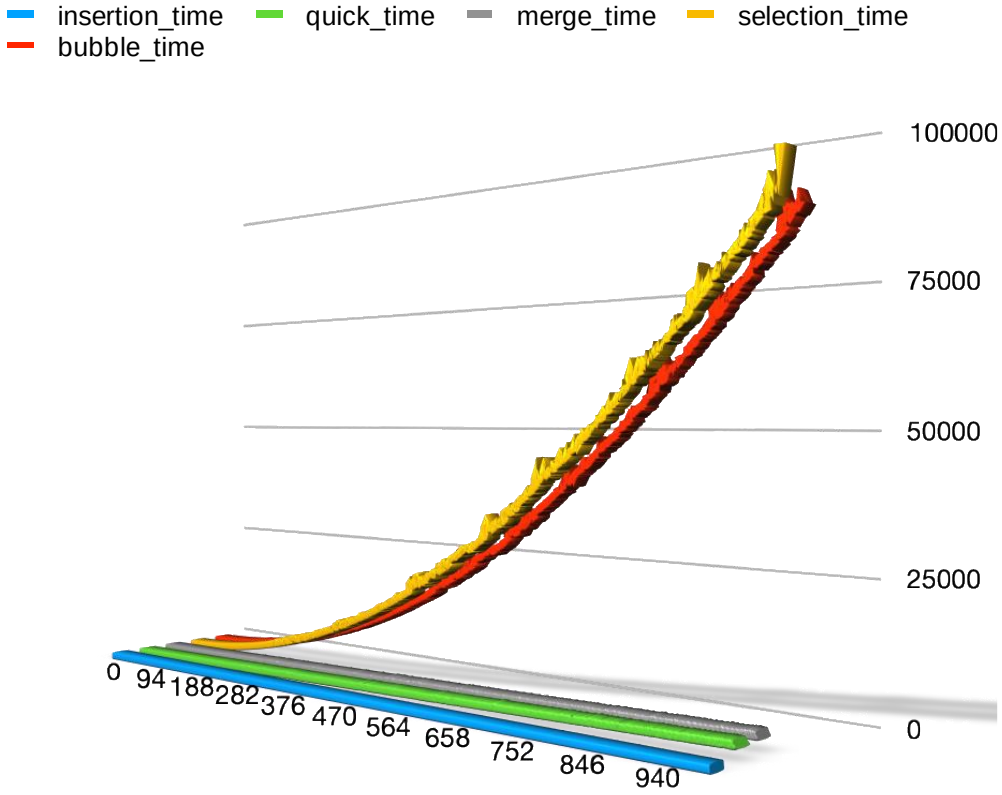
Den generelle informasjonen jeg får fra disse grafene er at vi alltid vil ha flere comparisons enn swaps. Det er også tydelig at tiden er et resultat av antall swaps og comparisons. Dette kan vi blant annet se på merge sort og selection sort. Selection sort har svært mange comparisons, men veldig få swaps i forhold. Tiden ender på ca. ¼ av høyden til compare, og swap sine høyde i grafoversikten er tilnærmet null. Merge har tilnærmet like grafer for swaps og comparison. Tidsgrafen ligger på ca. halve høyden av disse to. Fra disse to observasjonene kan man trekke konklusjonen at min implementasjon av disse algoritmene, har relativ lik kjøretid på en swap og på en comparison. (eller?) Se figur to.



Figur 2*: Random_1000_out, all data*

Algoritmer som utmerker seg

Fra figur tre kan vi tolke alle de forskjellige algoritmene som utmerker seg. Ved særlig stor input vil merge og quicksort utmerke seg, både på nær sortert, og tilfeldig input, sett opp mot bubble- og selection sort. Insertion sort utmerker seg derimot ytterligere på nær-sorterte datamengder, og er her den raskeste ved n = 1000. Dette er fordi når insertion sort først begynner å flytte på et element flytter den det helt til det møter ett som er mindre, og i en nesten sortert mengde vil antall plasser insertion flytter i gjennomsnitt per gang være betydelig høyere enn i et tilfeldig datasett, og vi sparer dermed tid på sammenligninger. Fra dataen jeg har analysert har jeg funnet at ingen av algoritmene utmerker seg noe spesielt på veldig små datasett, her får vi relativt like kjøretider. Dette kan vi jo også på starten av figur tre, for eksempel frem til n=100. Her er det ikke veldig store forskjeller.



Figur 3*: nearly_sorted_1000 kjøretider.*

Interessant funn

I implementeringen av mergesort gjorde jeg en feil med bruk av en O(n) metode i den indre loopen i sorteringsalgoritmen. Dette førte til at merge sort ble $O(n^2 \log(n))$, og følgelig ble den tregeste algoritmen. Beskrives nærmere i linje 22 i merge.py.