Oblig3 - IN2010

Oppgave 1

Vi har valgt å implementere følgende algoritmer:

1. Insertion sort
2. Quick sort
3. Selection sort
4. Heap sort

For store input-filer vil kun heap og quicksort kjøres for begge typer inputfiler. Da Insertion er effektiv på nesten sorterte lister kunne denne også kjøres for store filer, men denne justeringen valgte vi å gjøre manuelt (for å produsere grafen i slutten av besvarelsen). Det er mulig å endre på definisjonen av “store input-filer” i oblig3.py filen. Vi valgte å sette grensen på n = 100000.

Det er opprettet en egen prosedyre, checksorting.py, som sammenligner output-filene for hver av sorteringsalgoritmene mot python sin egen sort() algoritme. Det er rimelig å anta at implementasjonen av sorteringsalgoritmene riktig utført dersom output-filene og resultatet av pythons sort() algoritme er like.

Oppgave 2

Vi brukte skallet gitt av foreleser. Her var det allerede utarbeidet metoder for registrering av antall sammenligninger, bytter og tid brukt. Formateringen av .results filen var også utarbeidet av foreleser.

I praksis betyr det at vi bruker metoden A.swap(i,j) når vi skal skal bytte elementer i sorteringsalgoritmene. For sammenligninger vil det ikke være noe endring, da CountCompares bruker @total_ordering, som betyr at vi kan bruke logiske operatorer som vanlig. Forskjellen er at CountCompares objektet registrerer hver gang en slik operator brukes.

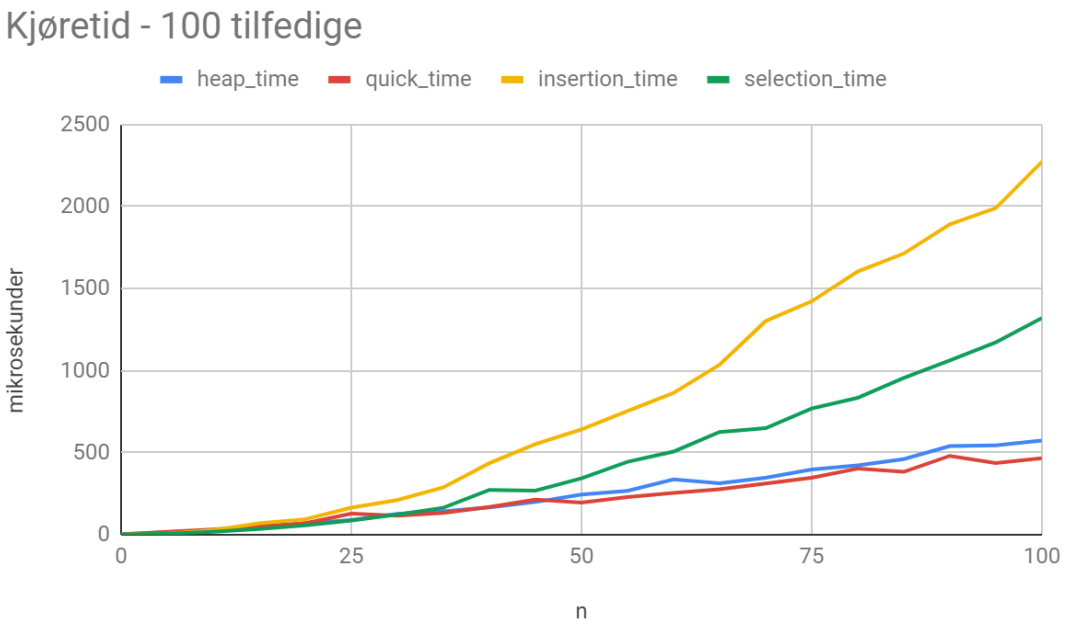
Oppgave 3

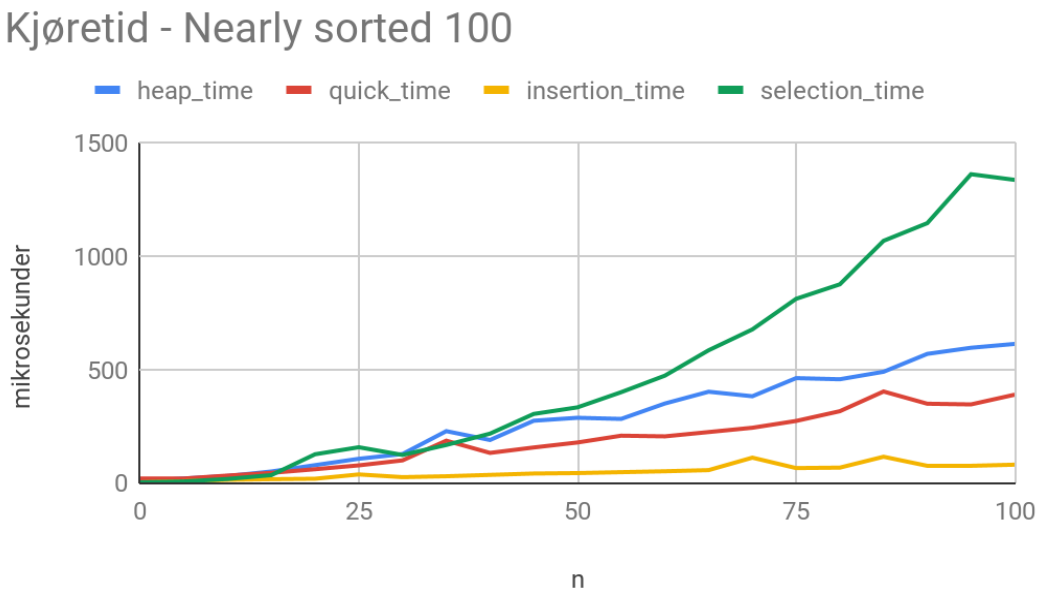
I hvilken grad stemmer kjøretiden overens med kjøretidsanalysene (store O) for de ulike algoritmene?

Algoritmene skal ha følgende kjøretidskompleksitet:

1. Insertion - $O(n^2)$
2. Selection - $O(n^2)$
3. Heap - $O(n \log(n))$
4. Quick - Skiller seg ut da den har stor forskjell på normal kjøretid og “worst case” der de er henholdsvis $O(n \log(n))$ og $O(n^2)$.

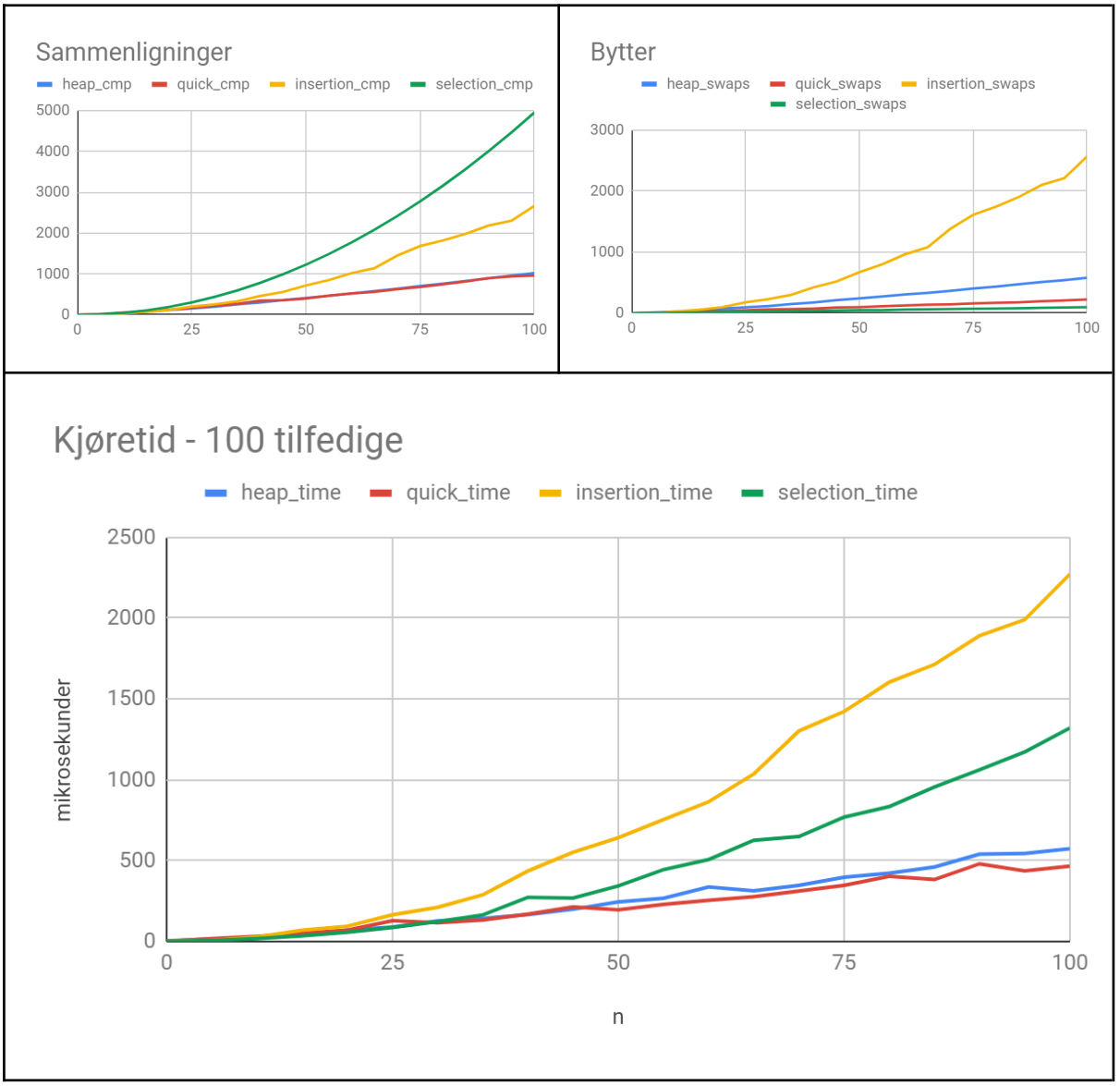
- Quick og Heap har omtrent lik kjøretid uavhengig av input og ser ut til å være nært n log (n)
- Selection er også uavhengig av hvor sortert input er og kjøretid er eksponentiell som O(n^2)
- Insertion er avhengig av input og for tilfeldige tall ser kjøretiden eksponentiell ut O(n^2). (se overraskende funn for mer)

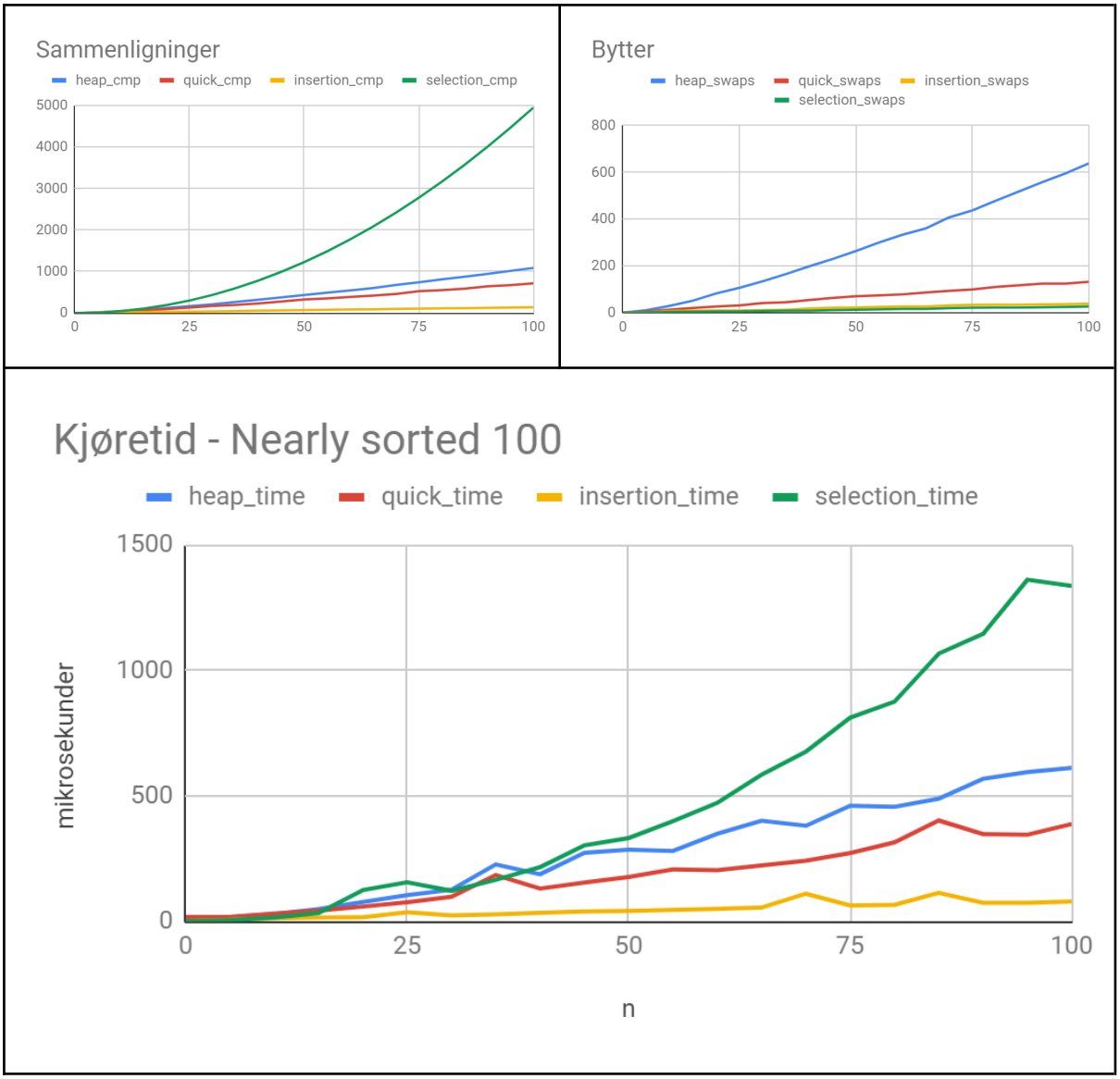




Hvordan er antall sammenligninger og antall bytter korrelert med kjøretiden?

Når man ser på sammenligner og bytter er de begge sterkt korrelert med kjøretid. Merk spesielt hvordan kjøretiden til Insertion (gul) er stor for tilfeldige tall og liten for nesten sorterte tall. Kjøretiden minker betraktelig i tilfellet Insertion sort har lite sammenligninger og bytter.





Hvilke sorteringsalgoritmer utmerker seg positivt når n er veldig liten? Og når n er veldig stor?

Generelt
Kjøretiden for de forskjellige algoritmene er:
Insertion(nesten sortert) < Quick < Heap < Selection < Insertion(tilfeldig)

n er veldig liten
n^2-leddet har ikke vokst så mye enda så Selection og Insertion(tilfeldig) er ikke så mye dårligere enn Quick og Heap.

n er veldig stor
Da slår n^2 mer ut på kjøretiden slik at Selection og Insertion(tilfeldig) blir flere ganger tregere enn Quick og Heap.

Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfilene?

Tallene er tilfeldige
Tilfeldige tall gir forventet kjøretid for algoritmene. Her skiller Quick og Heap seg ut med Quick hakket bedre. Det er tydelig at Quick ikke kjører “worst-case” tilfellet som er O(n^2).

Tallene er nesten sortert
Dette er tilnærmet “best-case” for Insertion, som påvirker dens kjøretid drastisk og gjør den til den beste. For de resterende oppfører de seg omtrent likt.

Har du noen overraskende funn å rapportere?

At insertion sort er så raskt på nesten sorterte tall. Til og med raskere enn Quick og Heap. Men når man tenker over hvordan algoritmen fungerer gir det likevel mening, fordi hvis listen er helt sortert vil “while-løkken” aldri “slå inn”. Altså går man bare gjennom alle elementer 1 gang som gir lineærtid O(n). For denne nesten sorterte listen er vi derfor svært nær lineær kjøretid.