IN2010 Oblig 1

Vetle H. Olavesen

September 28, 2023

Oppgave 1 - A

Vil legge til at jeg skriver koden i python, det betyr at jeg "pythonifiserer" koden, ved å bruke append() og insert() funksjonene. Disse har ganske god kjøretidskompleksitet. Som jeg går dypere innpå i neste deloppgave.

Pseudokode for:

Procedure *push_back*
Input: Et element 'x' som skal settes inn i enden Teque-en 'T'.

```
Procedure push_back(x):
    T.append(x)
```

Procedure *push_front*
Input: Et element 'x' som skal settes inn i fronten av Teque-en 'T'.

```
Procedure push_front(x)
    T.insert(0,x)
```

Procedure *push_middle*
Input: Et element 'x' som skal settes inn i midten av Teque-en 'T'.

```
Procedure push_middle(x)
    middle ← ⌊(n+1)/2⌋
    T.insert(middle, x)
```

Procedure *get*
Input: En int 'i' som tilsvarer indexen til elementet man ønsker ut.

```
Procedure get(i)
    return T[i]
```

Oppgave 1 - C

push_back: Siden jeg bare bruker append() her vil den ha kjøretidskompleksiteten til append(), som har O(1) ved innsetting, helt til minnet som python har allokert til listen blir full, da må den lage en kopi av listen og sette av mer plass til listen.
push_front: Denne vil ha samma kjøretidskompleksitet som python sin insert() funksjon. Som har O(n) kjøretidskompleksitet.
push_middle: Her har vi: (1+2) for å regne ut verdien av middle og sette den til verdien til variabelen middle. Så bruker vi insert() som har O(n) kjøretidkompleksitet. Push_middle har derfor en O(n+3), som når n blir veldig stor, vi kan sette til O(n) kjøretidskompleksitet.
get: Her bare henter vi ut et element på index 'i', som har O(1) kjøretidskompleksitet i python.

Oppgave 1 - D

Hvis vi har en konstant øvre grense på input størrelse, vil værste tilfelle kjøretidskompleksitet for prosedyrer som er avnhengig av størrelse på input alltid ha $O(10^6)$, eller en annen form av av det. Som betyr at alle prosedyrene teknisk sett vil ha en konstant kjøretidskompleksitet.

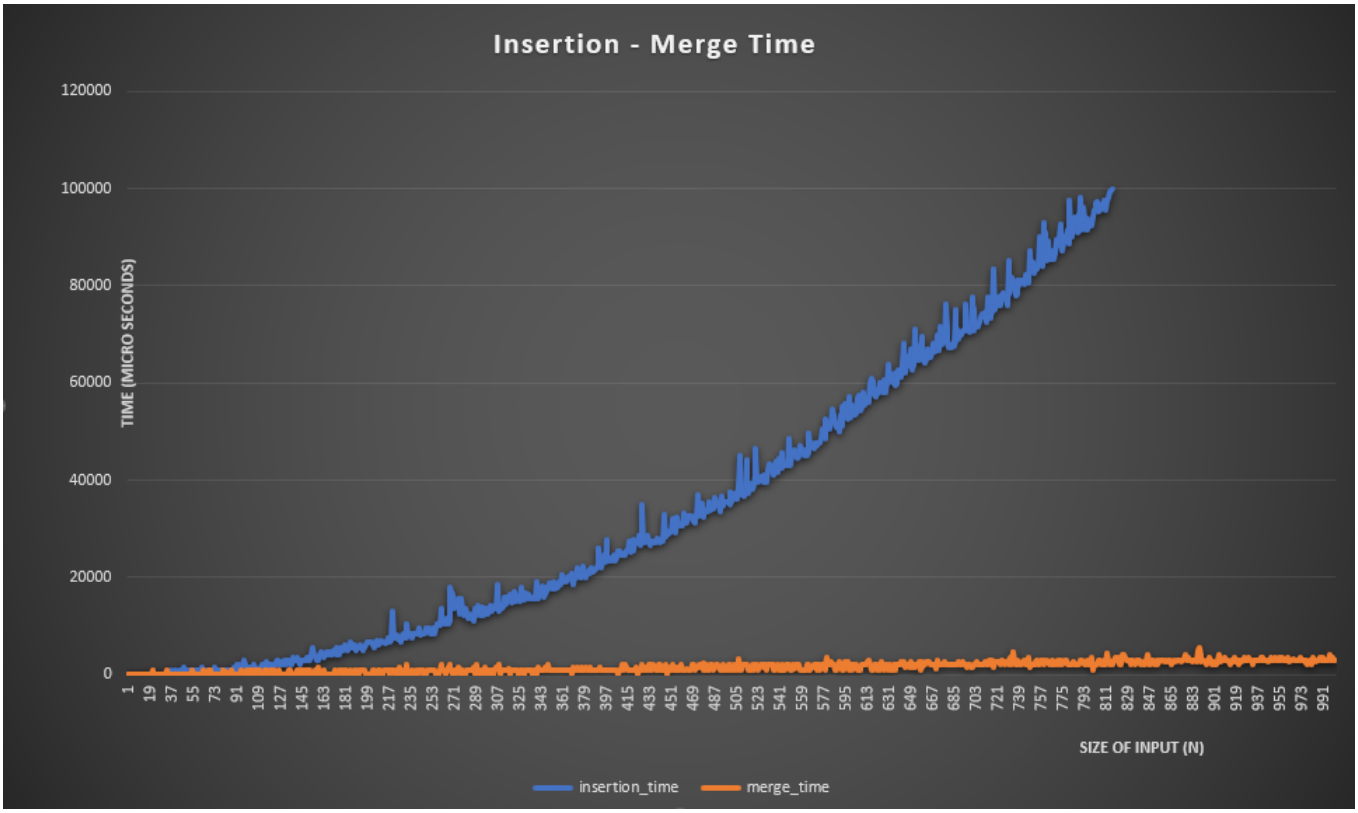
Sortering

Merk! - Jeg bruker prekodens implementasjon av telling av sammenligninger og bytter.

Sortering - Eksperimenter

Spørsmål:

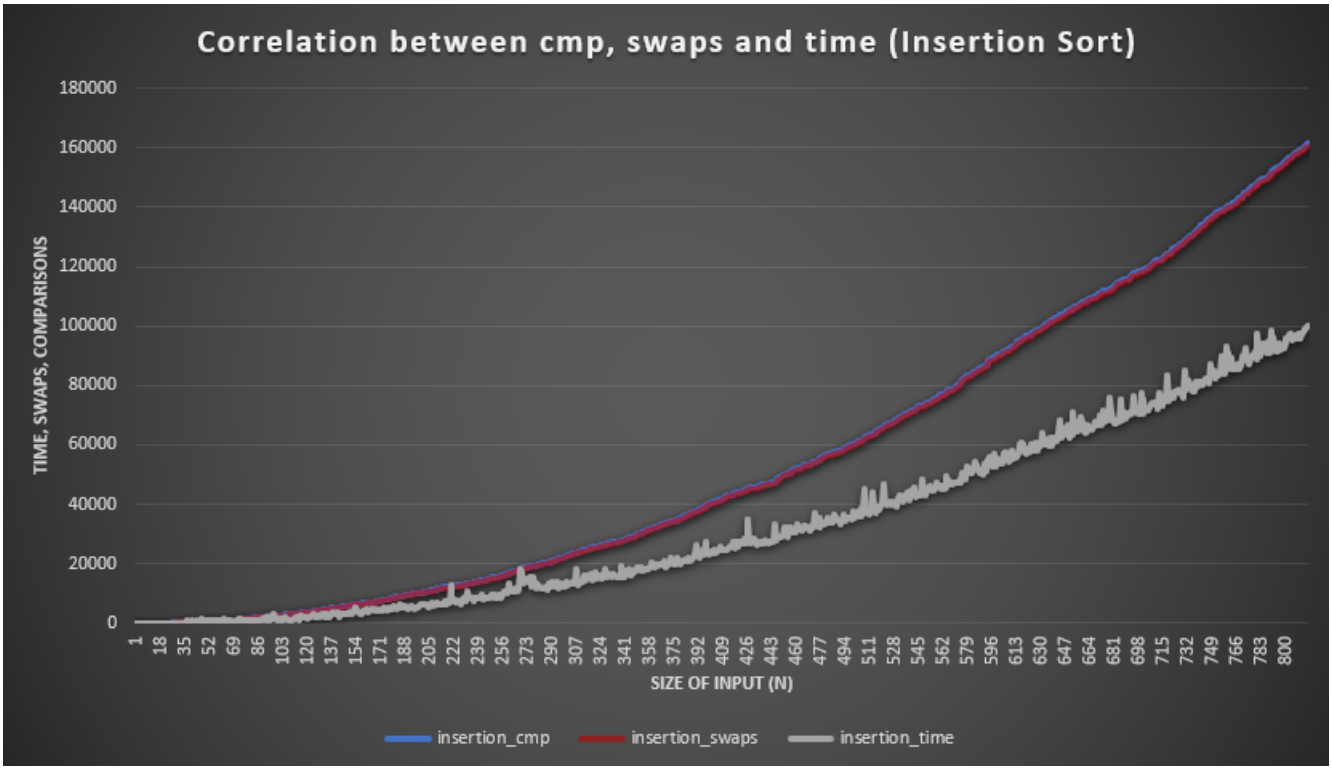
I hvilken grad stemmer kjøretiden overens med kjøretidsanalysene (store O) for de ulike algoritmene?
For dette lagde jeg en graf som viser de to algoritmene opp mot hverandre (input: 'random_1000):



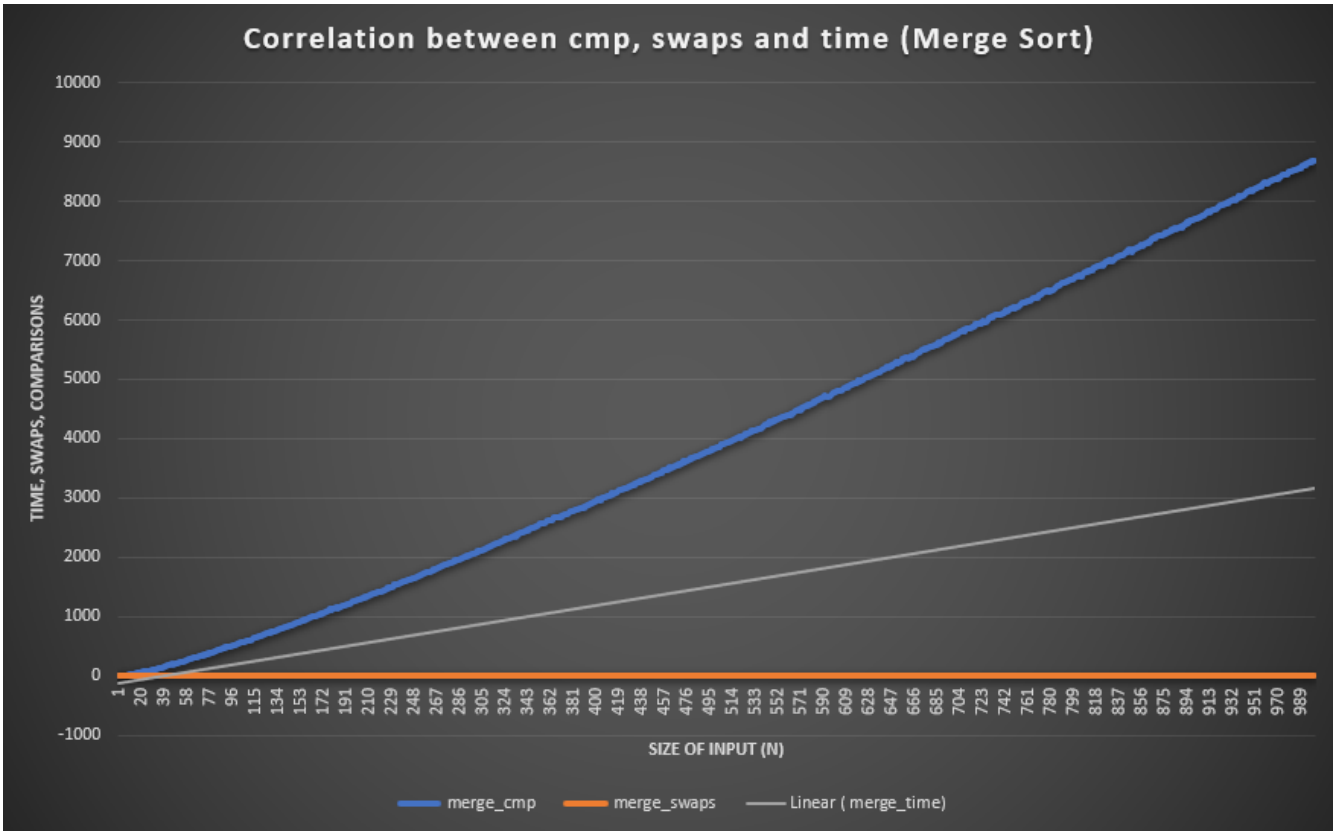
Her kan man tydelig se at merge sort er mye raskere enn insertion sort, hvertfall ved større inputs. Det ser også ut til at de følger O(n) og O(nlog(n)). Her brukte jeg 'random_1000' input filen, grunnen til dette er at det måler bedre 'worst case' kjøretiden til de to algoritmene.

Hvordan er antall sammenligninger og antall bytter korrelert med kjøre- tiden?

Har valgt å lage to grafer, en for hver av algoritmene, starter med insertion, hvor jeg satt opp sammenligninger, bytter og tid sammen i en graf (input: 'random_1000'):



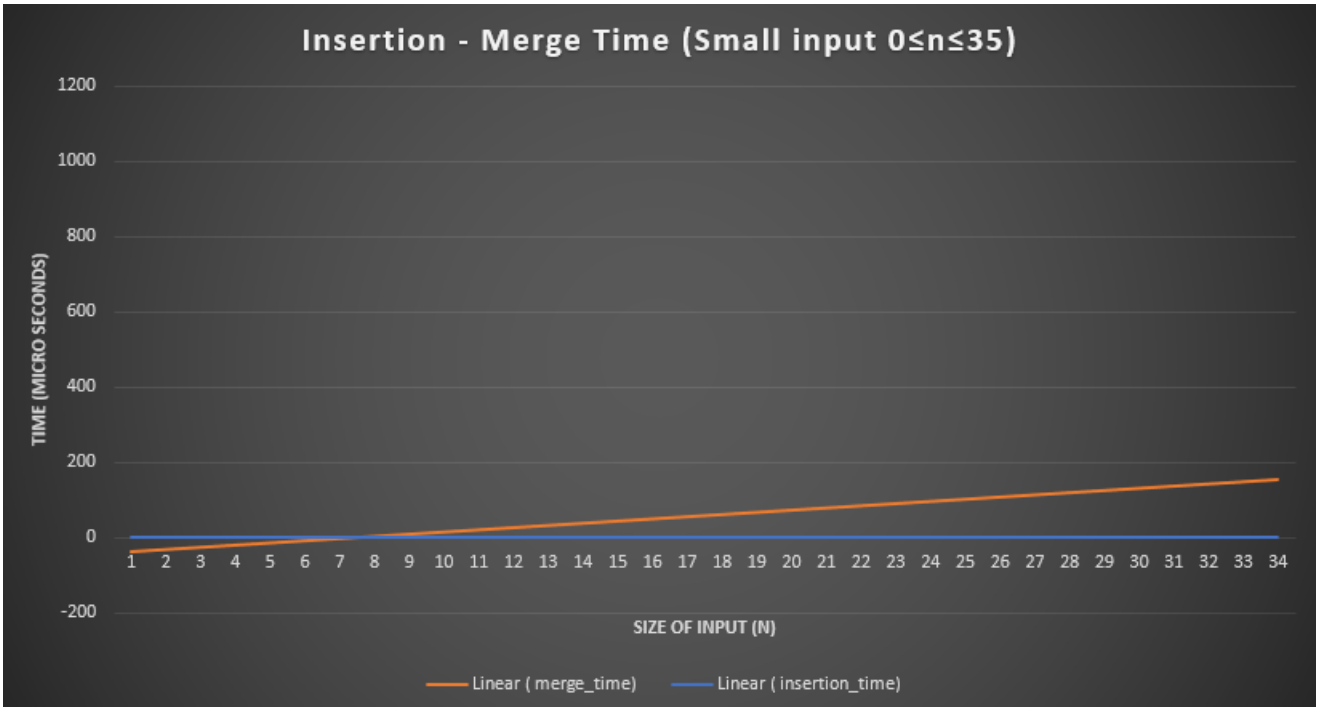
For insertion sort ser man at antall bytter og antall sammenligninger ligger oppå hverandre. Avviket mellom sammenligninger/bytter og tiden blir større og større jo større inputet er.
Under kommer samme type graf for merge algoritmen:



Fra dataen ser jeg at merge sort ikke gjør noen bytter, hvertfall ved prekodens definisjon av bytter. Det den egentlig gjør er å sammenligne to elementer også putte de sortert tilbake i den opprinnelige listen, uten å faktisk bytte på to elementer. Derfor er bytter for merge alltid 0. Ellers ser man at avviket mellom tid og antall sammenligninger vokser når størrelsen på input vokser.

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

Fra grafen ved spørsmål 1, kan man enkelt se hvilken av de to algoritmene som fungerer best ved store inputs, altså merge sort.
For mindre input har jeg lagd en ny graf som viser hvem som fungerer best. Da har jeg valgt $0 \geq n \leq 35$:

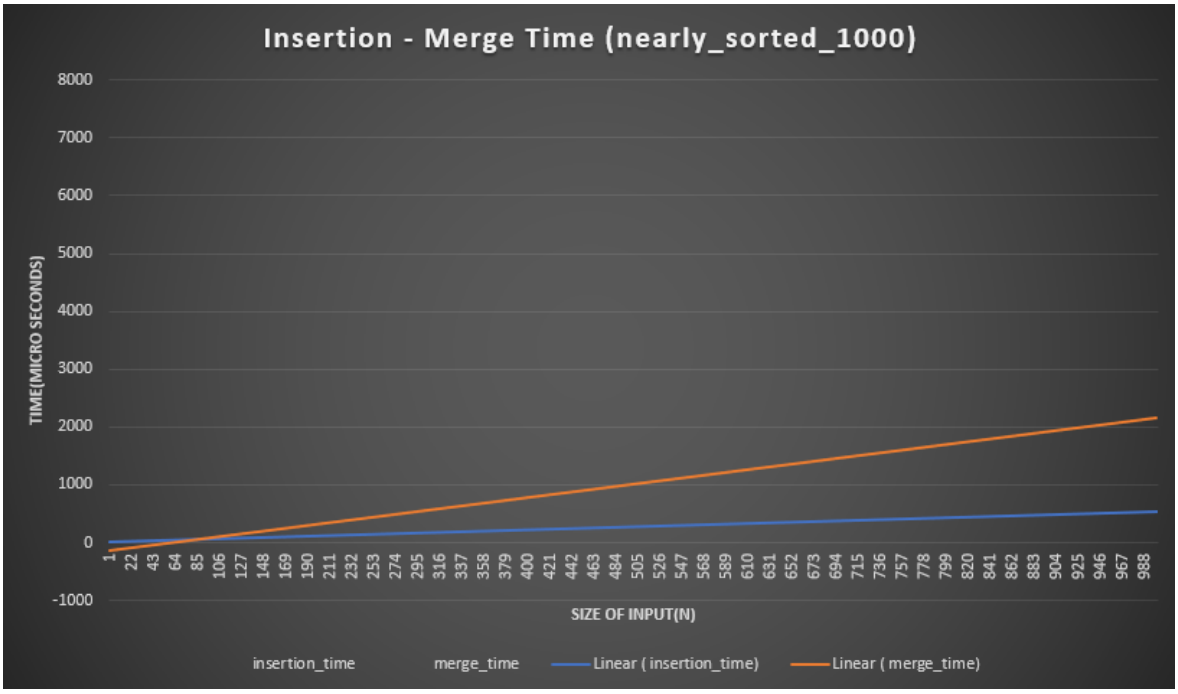


Time dataen for små input var litt vriene, dette er fordi tiden ikke ble registrert med mindre den var over en viss tid. Derfor fikk jeg 'spikes' i dataen, altså at tiden hoppet opp og ned fra 0 til 1000 mikrosekunder. Valgte derfor å lage grafen bare på 'trend lines' i excel, og fikk grafen over, viser godt trenden ved små input. Siden insertion ikke hadde noen spikes velger jeg å si at insertion sort egner seg best for dataset på $0 \geq n \leq 35$. Ved input større enn n = 35 viser dataen at merge sort er raskere.

Dataen til grafen over kommer fra den første biten av 'random_1000' input filen.

Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfile- ne?

Grafen 'Insertion - Merge Time' ble produsert ved bruk av 'random_1000' input filen. Som klart viser at merge sort egner seg best til helt tilfeldige inputs.
Lager også en graf på input settet 'nearly_sorted_1000' og får dette som tids-graf:



Her fikk jeg samme problem som jeg gjorde ved de små inputsa, altså at algoritmene var begge så raske at det var vanskelig å få bra data på tiden. Valgte derfor å bruke 'trend lines' igjen. Som viser ganske tydelig at insertion algoritmen fungerer en god del bedre når inputet allerede nesten er sortert.

Har du noen overraskende funn å rapportere?

Jeg ble overraska over hvor mye bedre insertion sort var på inputs som nesten var sortert. Før jeg startet trodde jeg at merge sort algortimen skulle slå insertion sort på tid i alle tilfeller, men at insertion kom til å være nærme på 'nearly_sorted' input filene, forventet ikke at insertion var bedre der.