Deloppgave 1: Korrekthet

Vi har implementert følgende algoritmer: insertion sort, quick sort, bubble sort og heap sort. For å teste de ulike algoritmene har vi kjørt alle inputfilene som inneholder 100 eller færre elementer, og gått over alle output-filene for å se om filene er sortert.

Deloppgave 2: Sammenligninger, bytter og tid

Vi valgte å benytte oss av prekoden som ble lagt ut i forbindelse med oppgaven.

Bytter blir målt gjennom en klasse CountSwaps, som inneholder en metode *swap(i, j)* og en variabel *swaps*. For hver gang metoden kalles på bytter den plass på *i* og *j* i arrayen, og legger til 1 i variabelen *swaps.* På denne måten holder man kontroll på antall bytter.

Når det gjelder antall sammenligninger, er dette vanskeligere å forklare. Prekoden er laget slik at man ikke trenger å endre noe i implementasjonen av algoritmene for å få måling av sammenligninger til å funke. I beskrivelsen står det følgende om måling av sammenligninger: «countcompares.py inneholder en klasse CountCompares som "wrapper" elementet som skal sorteres, og øker en teller hver gang elementet sammenlignes med et annet».

Deloppgave 3: Eksperimenter

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

Quick = n^2 i verste tilfelle, skjer svært sjeldent, n(log n) i beste tilfelle:

Når n=100 vil n(log(n)) tilsvare 200 i tid. I dette tilfellet er quick noe over dette. Vi antar at quick ligger på 400 på tid n=100. Videre ser vi at den ligger på rundt 200 i tid når listen er nesten sortert. Dette gir oss en halvering i tid.

Heap = n(log(n))

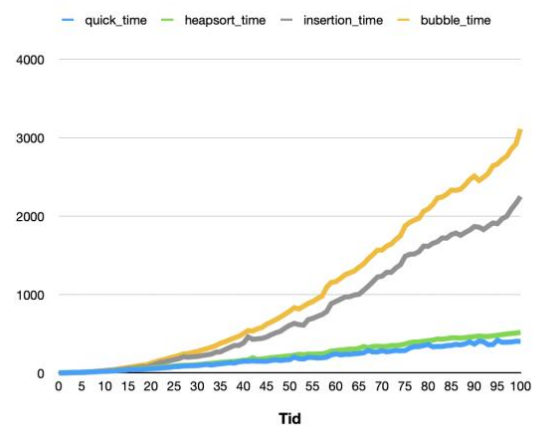
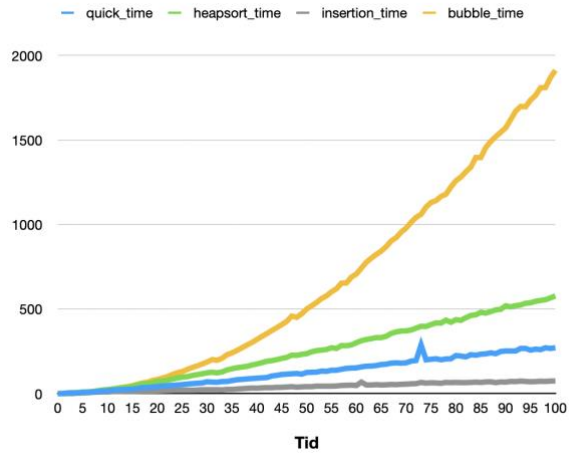
Heap sin kjøretid er lik for både random og nesten-sortert liste. Den har kjøretid som er noe over n(log(n)) når n=100. Dette kan være fordi man ser bort ifra konstantene når man ser på kjøretidskompleksitet. Hvis dette ikke er årsaken innser vi at det kan være noe feil med implementeringen.

Insertion = n^2

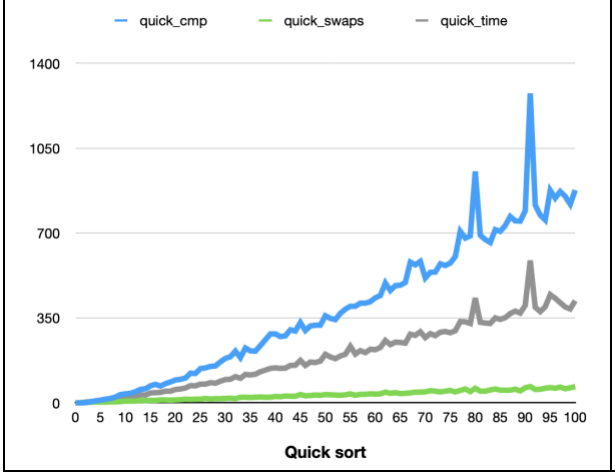
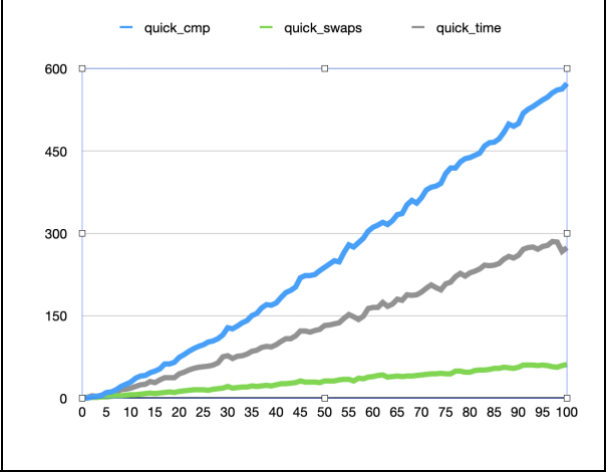
Insert sin kjøretid er veldig rask for nesten-sortert liste. Den tilsvarer kjøretiden på n^2.

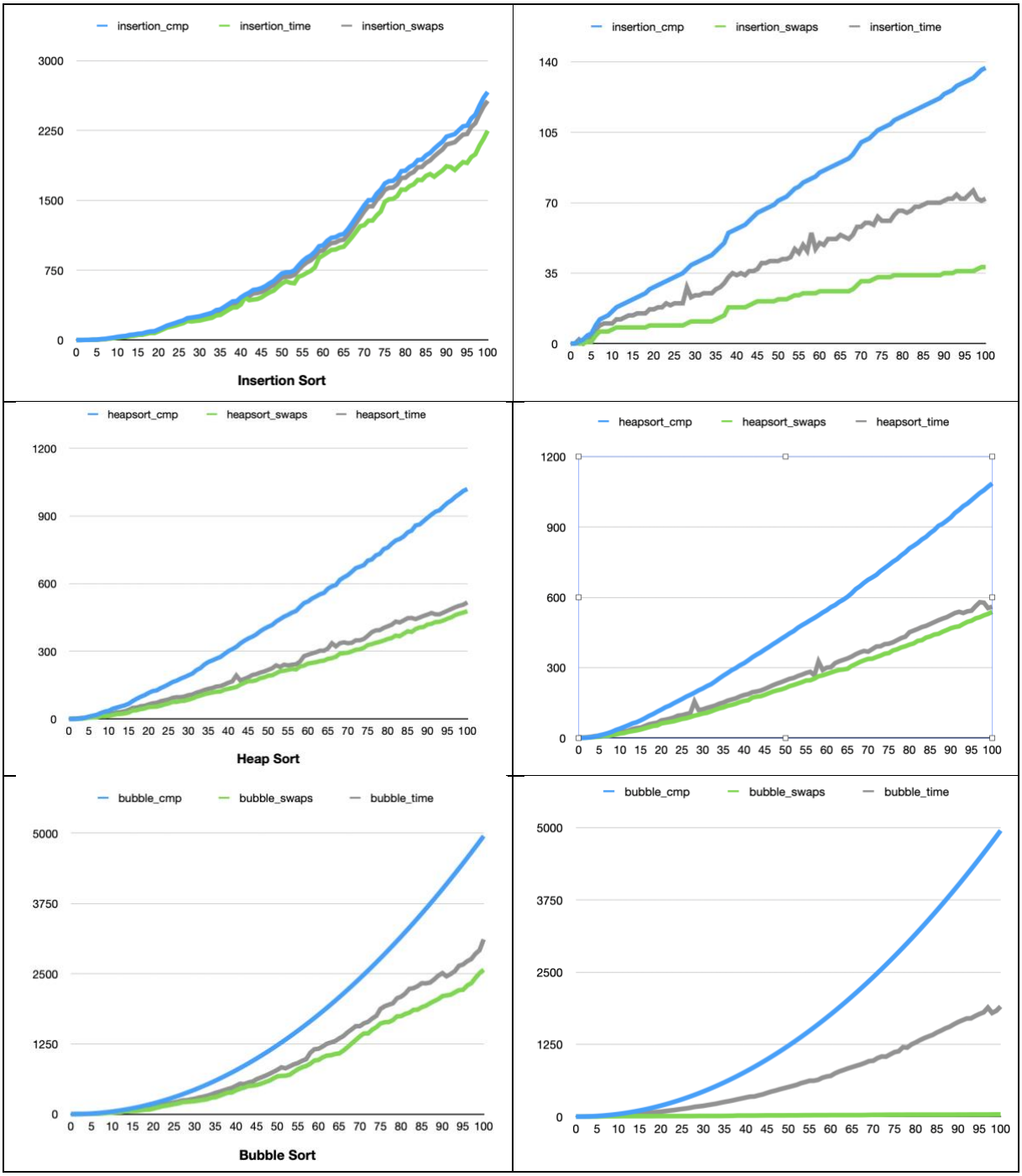
Bubble = n^2

Bubble er den tregeste sorteringsmetoden. Den følger også kjøretidskompleksiteten sin.

Random 100:	Nearly sorted 100:
	

Hvordan er antall sammenligninger og antall bytter korrelert med kjøretiden?

Random	Nearly sorted
	



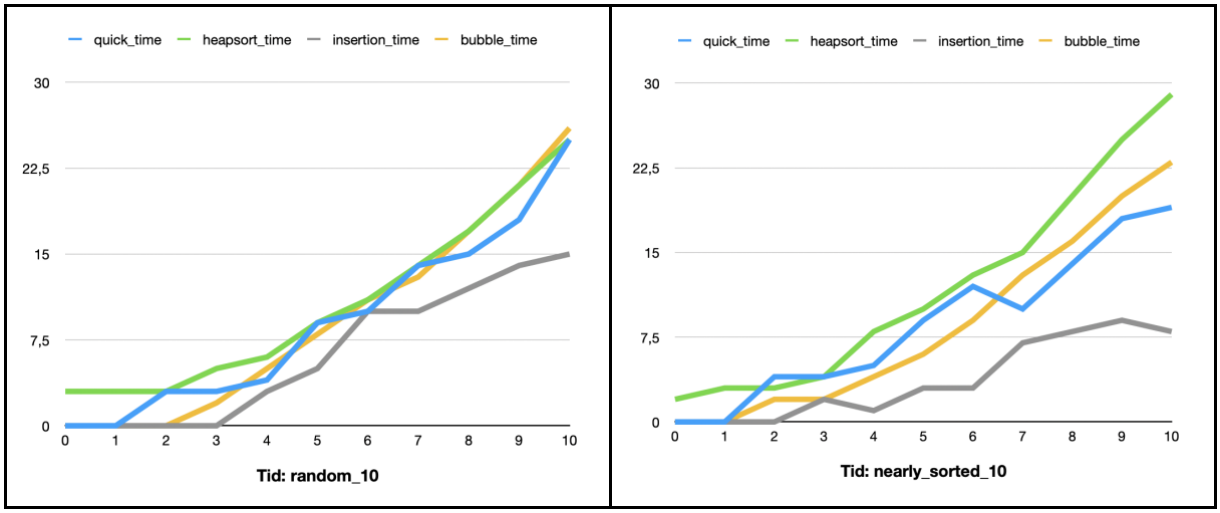
Quick sort: kjøretiden er omtrent det firedobbelte av antall bytter, og halvparten av antall sammenligninger.

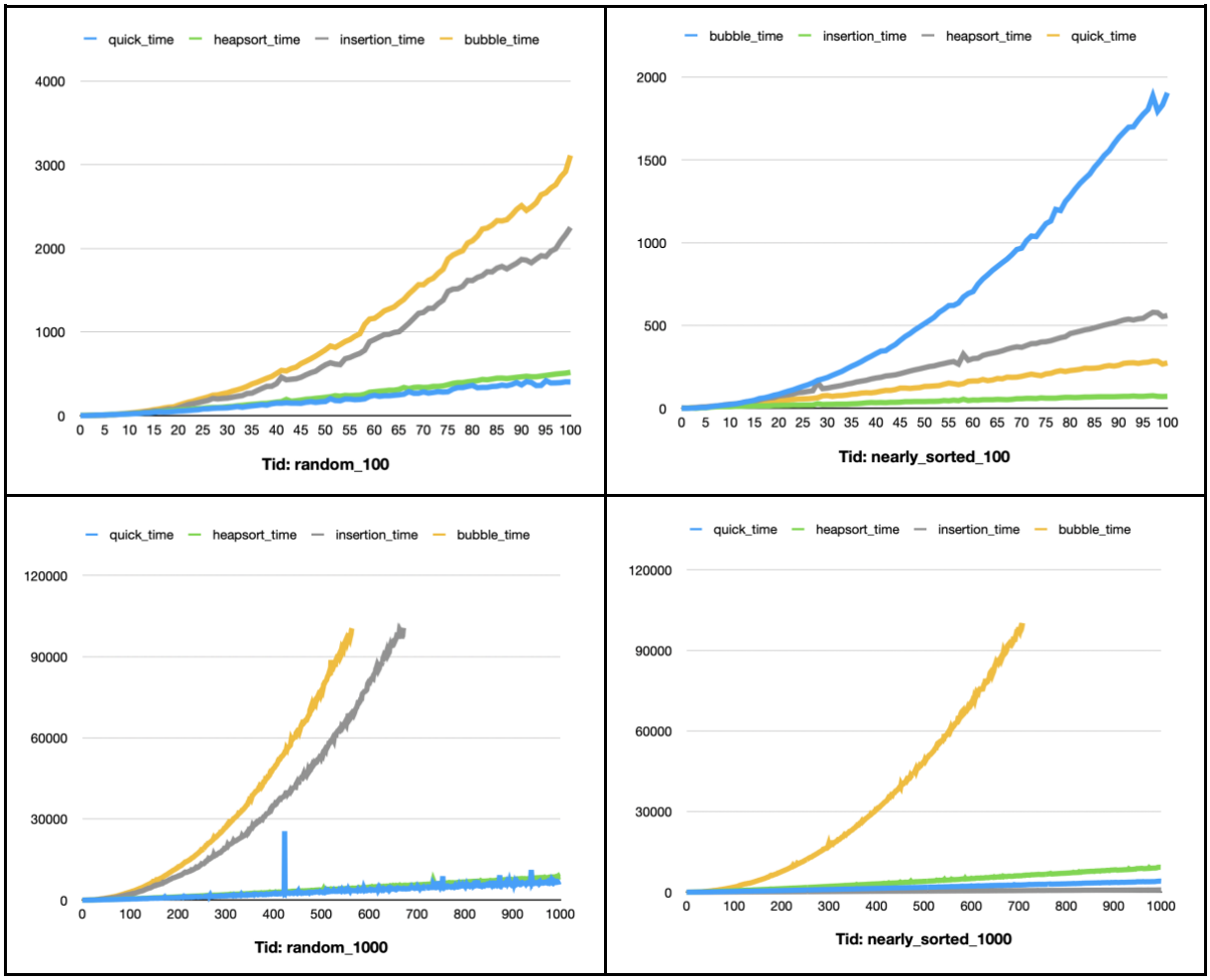
Insertion sort: i nearly sorted-filen bruker insertion sort veldig mye kortere tid, fordi antall swaps og sammenligninger reduseres kraftig. Med andre ord henger de tett sammen i denne algoritmen.

Heap sort: ingen endring mellom random og nearly sorted-filen, så det er vanskelig å si hvordan kjøretiden avhenger av antall bytter og sammenligninger. Når det er sagt, kan man se at kjøretid og antall bytter følger hverandre veldig tett.

Bubble sort: for denne algoritmen endres ikke antall sammenligninger mellom random og nearly sorted-filen, mens antall bytter blir tilnærmet lik null. Kjøretiden reduseres med nærmere 50%, som forteller oss at kjøretiden påvirkes av både antall sammenligninger og bytter.

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





Her har vi valgt å kjøre alle inputfiler som har 1000 eller færre elementer.

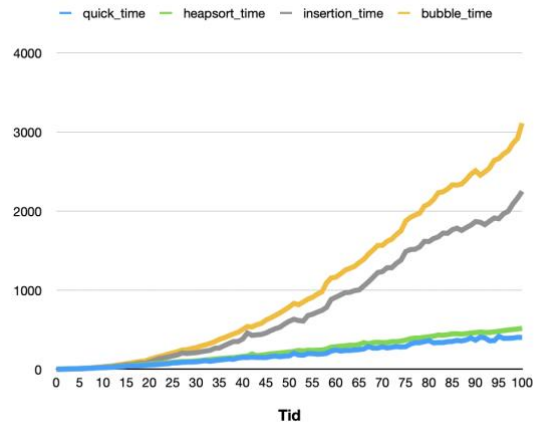
På de minste inputfilene kan man se at insertion sort skiller seg ut, og er betydelig raskere enn de andre. Heap sort, som i utgangspunktet skal være en av de raskeste, er tregest på både random og nearly sorted-filen.

Når vi ser på filene med 100 elementer, blir forskjellene større. I random-filen ser vi at de to antatt raskeste algoritmene skiller seg ut positivt, og er ganske like. Bubble sort er tregest, men også insertion er treg. På nearly sorted-filen derimot, ser vi at insertion sort er klart raskest. Også bubble sort og quick sort forbedrer seg litt, mens heap sort bruker omtrent samme tid som for random-filen.

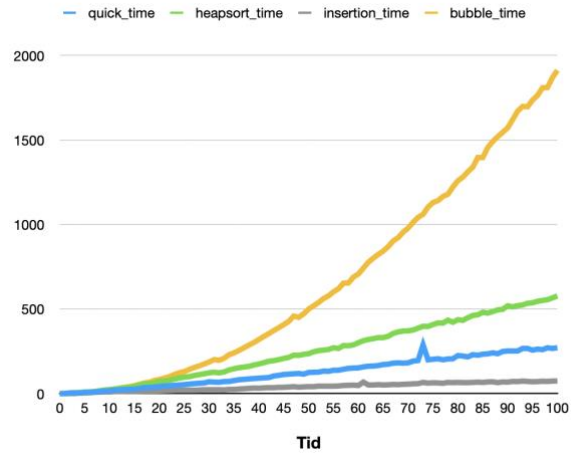
Filene med 1000 elementer kan man se at ligner mye på 100-filene, men forskjellene blir enda større. I random-filen gir programmet opp både bubble og insertion sort når kjøretiden blir for høy, mens heap sort og quick sort fullfører. De er ganske like, men quick sort er hakket raskere. For nearly sorted-filen ser vi at bubble er den klart tregeste. Det blir også vanskelig å se forskjellene på de andre algoritmene, fordi alle ser veldig raske ut sammenlignet med bubble sort. Man ser likevel at insertion er raskest foran quick sort, mens heap sort bruker litt lengre tid. Dette stemmer også bra med det vi så i 100-filene.

Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfilene?

Random 100:



Nearly sorted 100:



Alle algoritmene er raskere om inputfila er nearly sorted, utenom heap sort. Både bubble sort og quick sort forbedrer seg en god del, men insertion sort skiller seg ut. Den er nesten like treg som bubble sort på random-filen, men er raskest på nearly sorted-filen. Dette er fordi den ikke trenger å gjøre like mange sammenligninger når tallene allerede kommer nesten sortert. Heap sort forbedrer seg ikke fordi den må gjøre omtrent samme antall sammenligninger uansett.

Har du noen overraskende funn å rapportere?

Vi ble overrasket over at algoritmer med lik kjøretidskompleksitet, presterte såpass ulikt. For eksempel ser vi at quick, bubble og insertion sort, ikke bruker like lang tid på å kjøre, selv om de alle har kjøretidskompleksitet $O = n^2$. Vi ble også overrasket over at grafene stemmer dårlig med hva vi får når vi setter inn ulike tall for n i uttrykket for O. I omtrent alle tilfeller blir dette tallet lavere enn tiden algoritmene brukte i våre tester.

En annen ting vi ble overrasket over, var hvor stor forskjell det var på insertion sort når vi kjørte random-filen, og nearly sorted-filen. Eksempelvis gikk den fra å ikke fullføre random_1000-filen, til å være klart raskest i nearly_sorted_1000. Videre ble vi også overrasket over at heap sort ikke ble noe raskere selv om elementene var nesten sortert.

En siste refleksjon er at vi synes det virker som at vår heap sort er for treg, uten at vi skjønner hvorfor. Det kan ha noe med implementeringen å gjøre.