Innlevering 3 IN2010 H24

Kattungeoppgave

Kattunge-oppgaven er gjort og vedlagt som en java-fil. Den heter Kattunge.java

Sortering og Korrekthet

Sorteringsalgoritmene som er implementert er BubbleSort, HeapSort, InsertionSort og MergeSort. Disse fire algoritmene er vedlagt som Python-filer. Jeg innså at det ville bli mer kode og litt mer komplisert hvis jeg skulle skrive algoritmene i java, og i tillegg så tenker jeg at det er enklere å implementere koden i python med tanke på syntaks og forståelse av algoritmene. Algoritmene kjører og utfører operasjonene riktig og hver av algoritmene får ut riktig og samme resultat fra outputfilen. Siden det ikke er gitt hvilke og hvor mange inputfiler vi skal teste for hver av disse algoritmene, så har jeg valgt å teste den mot example.txt. Etter å ha kjørt input på programmet, der hovedprogrammet kaller på alle algoritmene og skriver ut outputfilene, med riktig endelse.

Etter å ha kjørt programmet så har alle fått riktig output, som vil si at programmet har sortert tallene i riktig rekkefølge med den minste verdien på toppen/begynnelsen. Outputfilene har fått like svar.

Input fil:

```
1  80
2  91
3  7
4  33
5  50
6  70
7  13
8  321
9  12
```

Output fil:

```
1  7
2  12
3  13
4  33
5  50
6  70
7  80
8  91
9  321
```

Programmet ble også testet for større input-filer. Denne gangen testet vi med blant annet random_100.txt. Her ble outputfilene resultatet og outputfilene helt identiske i tillegg til at de ble riktig sortert. Her var inputfilen helt random dvs. at algoritmene har litt mer arbeid å gjøre enn å velge den vanlige __100.txt filen, som ikke var helt blandet sammen. Det blir litt mange tall å legge til en screenshot av input-/output filen, så derfor har jeg lagt til alle inputfilene og outputfilene jeg har testet med i programmet inne i en egen mappe som heter input. Her vil det ligge test-filen(input) sammen med alle .out-filene fra algoritmene.

Sammenligninger, bytter og tid

Byttene som sorteringsalgoritmene ble målt med var ved hjelp av swaps fra filen «countswaps.py», mens sammenligninger «cmp» er målt i «countcompares.py» Ved hjelp av filene, blir funksjonene kalt når sorteringsalgoritmene kjøres noe som lar dem telle mens programmet kjører. Alle inputfilene som har blitt testet har lagt til en csv. -fil i input-mappen sammen med .out filene til hver sorteringsalgoritme.

Eksperimenter

Nederst ligger diagrammer fra de genererte csv-filene

1. **I hvilken grad stemmer kjøretiden overens med kjøretidsanalysene (store O) for de ulike algoritmene?**
   Ifølge grafene og store O sin representasjon, stemmer det overens med kjøretidskompleksiteten. De ulike sorteringsalgoritmene har disse kjøretidskompleksitetene (Store O):

BubbleSort = $O(n^2)$ –

Kjøretiden og antall sammenligninger øker raskt når mengden av input øker. Den sammenligner hvert element med naboen og bytter dem hvis de er i feil rekkefølge. Algoritmen er derfor ineffektiv for store datasett. Ut ifra diagrammene ser vi at BubbleSort tar mye mer tid for større n, som stemmer med $O(n^2)$

HeapSort = O (n log n) –

Dette er en algoritmen som bygger på maks-heap, som vi tidligere har sett, blitt sammenlignet med binære søketrær, men i «array-form». Her gjentar vi å bytte toppverdien med det siste elementet og omstrukturere heapen. Hjelpefunksjonen tar O (log n), og siden det utføres operasjoner på hvert element i listen på n elementer får vi totalt O (n log n). Her øker kjøretiden effektivt og balansert, uansett når inputstørrelsen øker, som betyr at den er litt bedre enn BubbleSort.

MergeSort = O (n log n) –

Her deler vi inputlisten rekursivt i mindre lister hvor vi sorterer dem og slår de sammen igjen. Siden listen deles i to i hvert steg har vi tiden O (n log n). På samme måte som HeapSort er denne veldig effektiv på store input og rask til å gjennomføre sorteringene. Selv om inputen øker, vil kjøretiden vokse sakte i forhold til logaritmiske økningen.

InsertionSort = $O(n^2)$ –

Denne må ta imot en liste og sortere den ved å flytte elementer til riktig posisjon for ett og ett element om gangen. Denne passer godt for små datasett og nesten sorterte liste, som vi ser fra diagrammene, i tillegg til den lineære veksten i kjøretid. Denne har også mindre sammenligninger enn HeapSort, når vi har en nesten sortert liste.

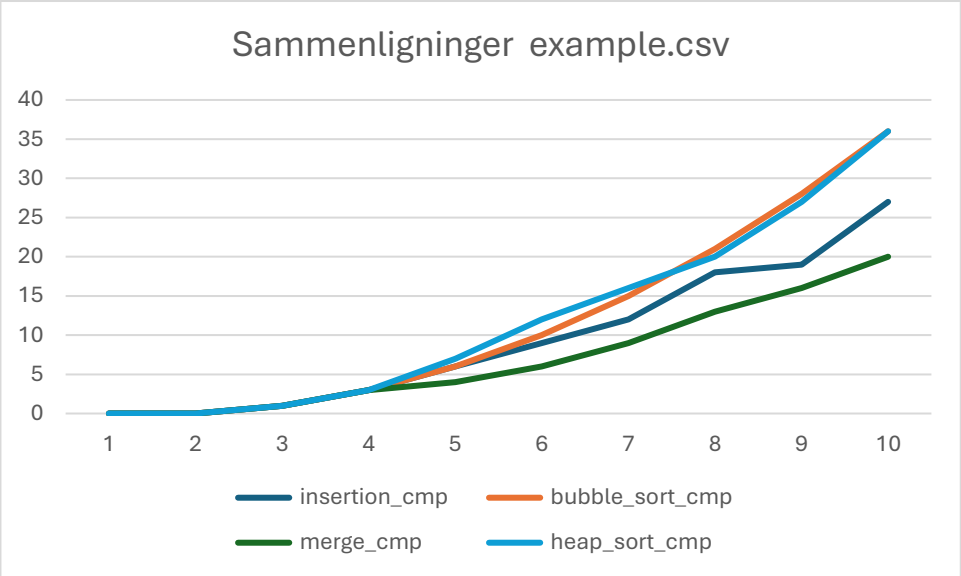
2. **Hvordan er antall sammenligninger og antall bytter korrelert med kjøretiden?**
   Algoritmene HeapSort og MergeSort har mer avanserte sammenligningsteknikker, som vil si at de bruker mindre bytter og sammenligninger som igjen gjør det til raskere kjøretid. De kan uansett få mange sammenligninger, men kjøretiden til å finne plasseringen for elementet er raskere enn f.eks. BubbleSort og insertion.
   De andre to: BubbleSort og InsertionSort har ekstremt mange sammenligninger og bytter for randomized lister, som fører til lengre kjøretid, eller stopp på algoritmen siden de har brukt mer tid enn ønsket. Vi kan se på diagrammet til random 100 at disse to algoritmene stoppet og fikk en høy «spike» når de nådde over halvparten av elementene i listen, som mest sannsynlig betyr at de fikk et element som trengte å søkes gjennom hele listen som tok ekstremt my tid.

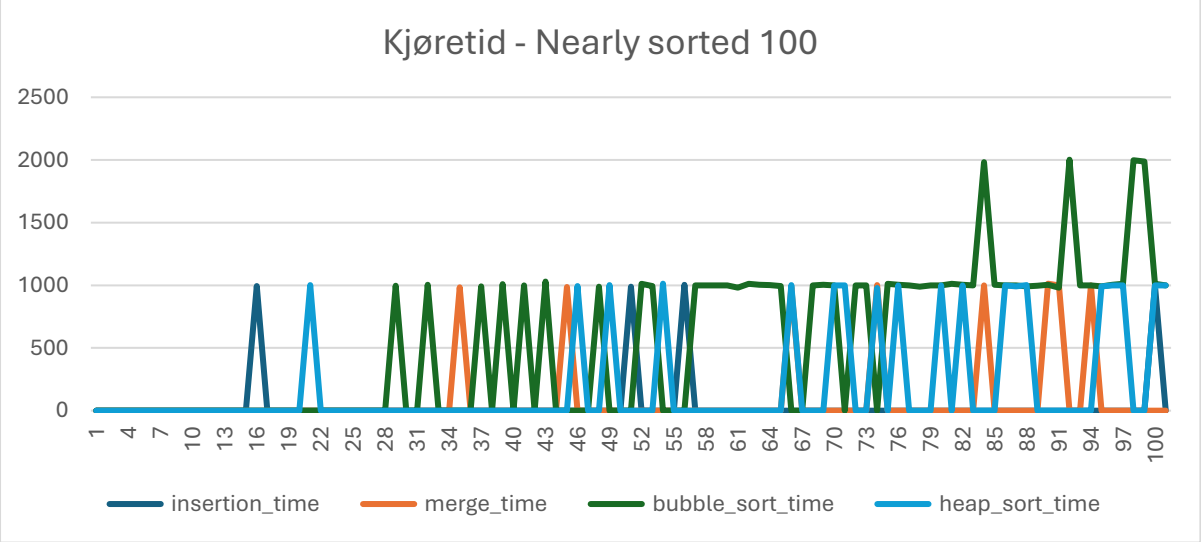
3. **Hvilke sorteringsalgoritmer utmerker seg positivt når n er veldig liten? Og når n er veldig stor?**
   Når n er liten er InsertionSort det beste valget, siden tiden er lineært og i disse tilfellene er det få elementer, som utmerker seg når elementene er nesten sortert. Den bruker også lite minne i henhold til MergeSort, og har en enkel implementasjon. Grunnen til at det ikke er HeapSort er fordi at den trenger med vedlikehold og høyrer overhead som krever at vi bygger den etter hvert element. For små elementer vil den bruke unødvendig mer tid enn den trenger. På samme måte er BubbleSort også ikke helt ideelt, siden den gjør flere sammenligninger enn insertion (se diagram under for example.txt testen).
   Når n er veldig stor passer MergeSort og HeapSort veldig god på grunn av kjøretidskompleksiteten, som gjør den effektiv mot store datasett. HeapSort bruker den vanlige heap-strukturen til å oppnå effektiv sortering, og den har en fordel om å bruke lite minne i forhold til MergeSort, som krever ekstra minne for å slå sammen listene. På samme måte er MergeSort veldig effektiv siden den deler listene i mindre deler som blir utført rekursivt, som sorterer i en systematisk måte.

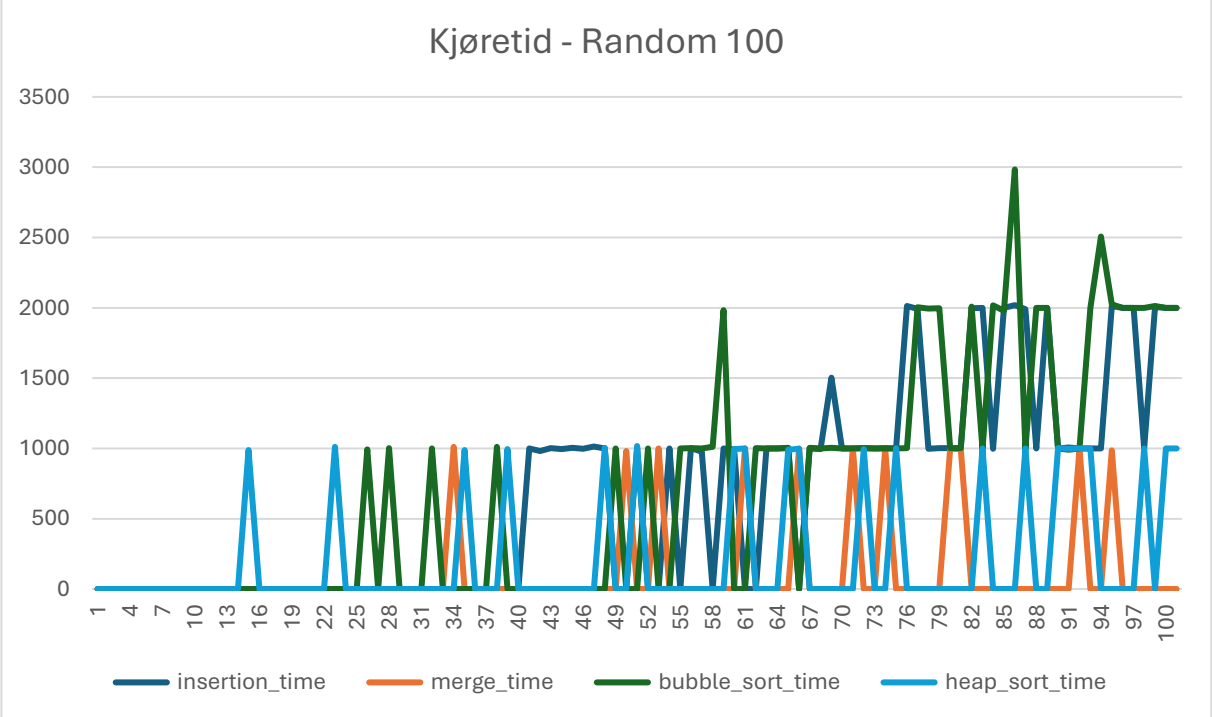
4. **Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfilene?**
   Når vi testet nearly sorted lister: av alle sorteringsalgoritmer var det InsertionSort som fungerte veldig godt blant disse. Den var raskest og sorterte listene med kortest tid. Algoritmen trenger bare å gjøre noen få bytter, og når elementene er nesten sortert, blir antallet cmp og bytter mye mindre.

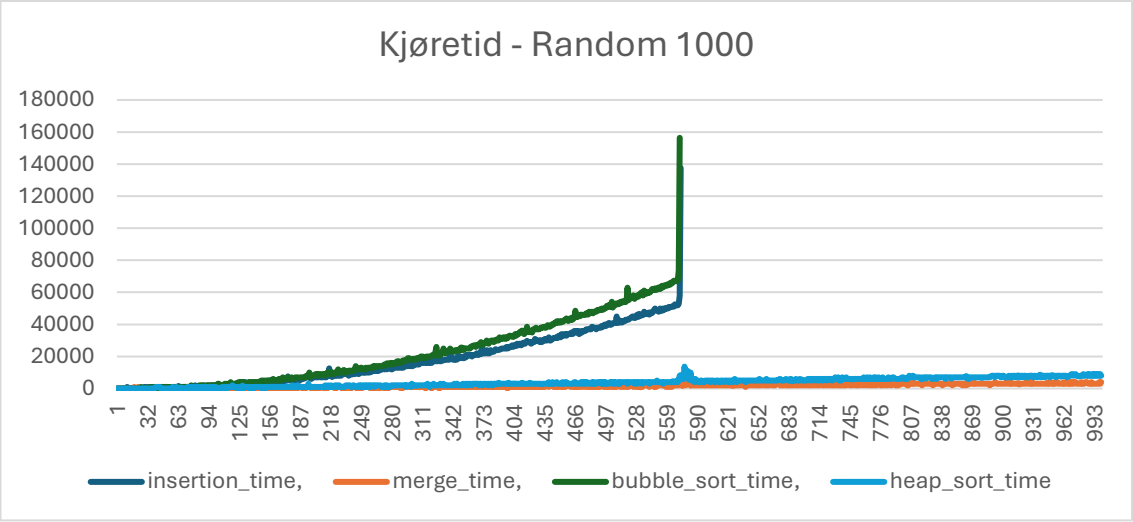
   På random data, det vil si elementer som er helt blandet i listen, så var det MergeSort og HeapSort som dominerte hele kjøretiden. De to andre algoritmene ble stoppet siden de brukte ekstra mye tid, og var mye mer tidskrevende i sorteringsfasen. Siden MergeSort alltid deler listene i mindre lister så blir den ikke påvirket av graden av sortering eller usortering i listene. Vi kan også nesten si det samme for HeapSort. Her er den også veldig rask men ikke like rask som MergeSort i praksis. Hvis vi ser på random 10000-diagrammet ser vi at HeapSort stopper opp. Dette var veldig overraskende, siden jeg trodde den også ville sortere ferdig hele listen. En av grunnene kan være at algoritmen har nådd den maksimale minnet som programmet kan bruke eller at den har brukt mer tid enn det som maksimalt er avsatt.

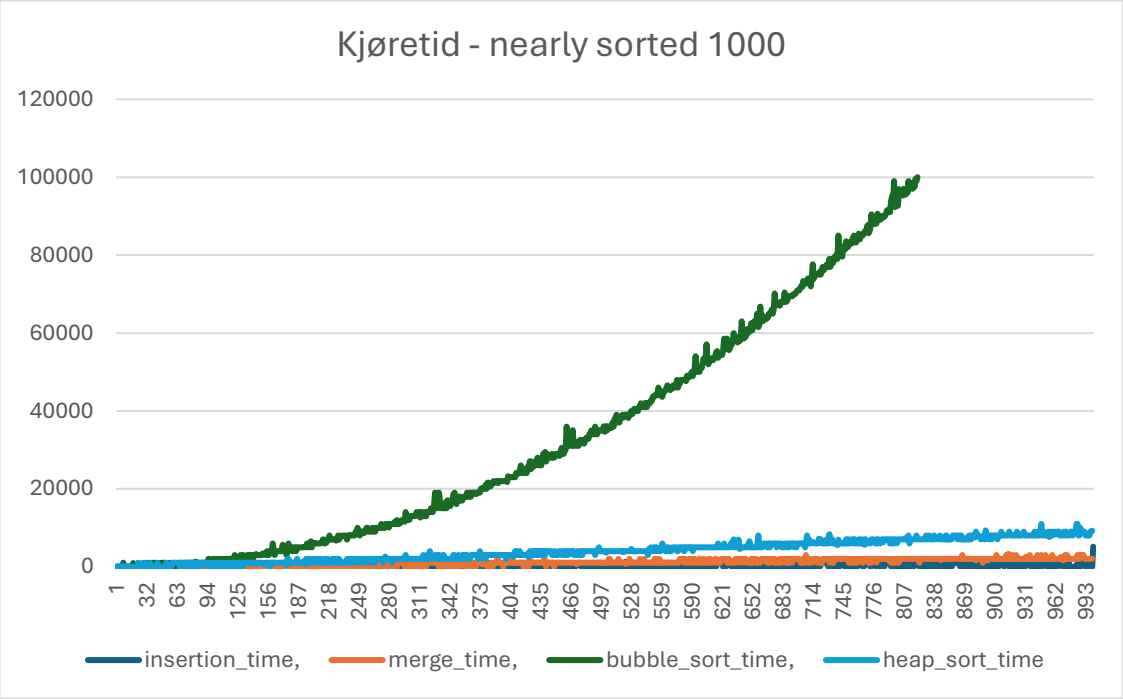
5. **Har du noen overraskende funn å rapportere?**
   Jeg ble litt overasket over hvor godt BubbleSort fungerte. Jeg trodde den ville stoppe helt og ikke kjøre når jeg testet den med nearly_sorted 1000, men den kom et stykke fram. Grunnen til at jeg tenkte det slik var at kjøretiden allerede var høy når den ble testet mot nearly og random 100, og antall sammenligninger var også veldig høy og den hadde høyest kjøretid i y-aksen enn de andre.

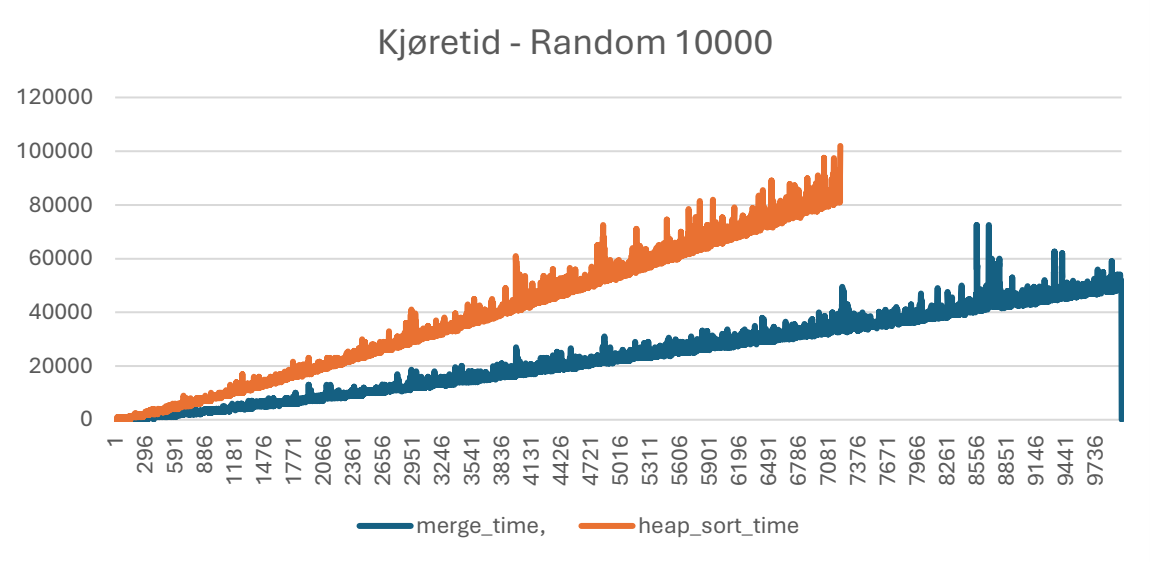

Sammenligninger example.csv
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
9
10
insertion_cmp
bubble_sort_cmp
merge_cmp
heap_sort_cmp

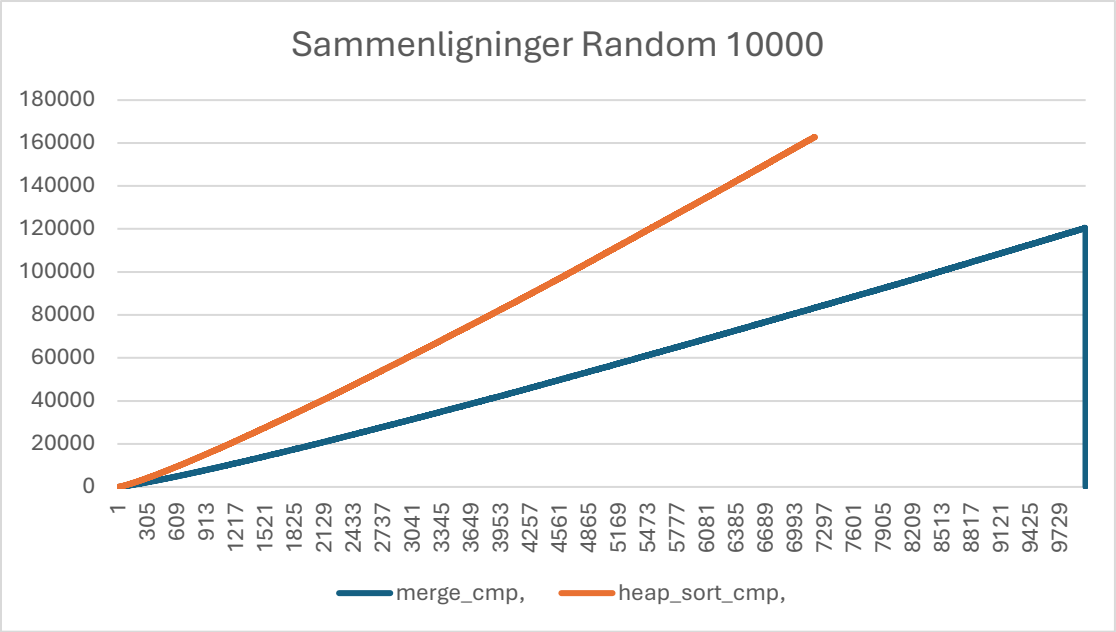

Kjøretid - Nearly sorted 100
2500
2000
1500
1000
500
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
insertion_time
merge_time
bubble_sort_time
heap_sort_time

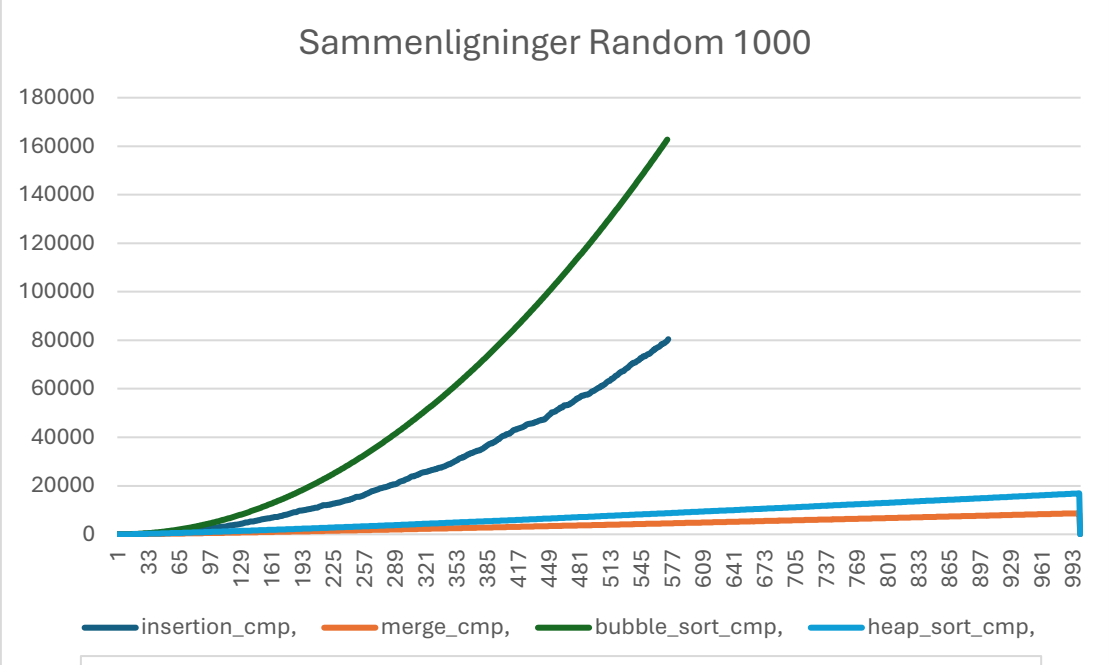

Kjøretid - Random 100
3500
3000
2500
2000
1500
1000
500
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
insertion_time
merge_time
bubble_sort_time
heap_sort_time

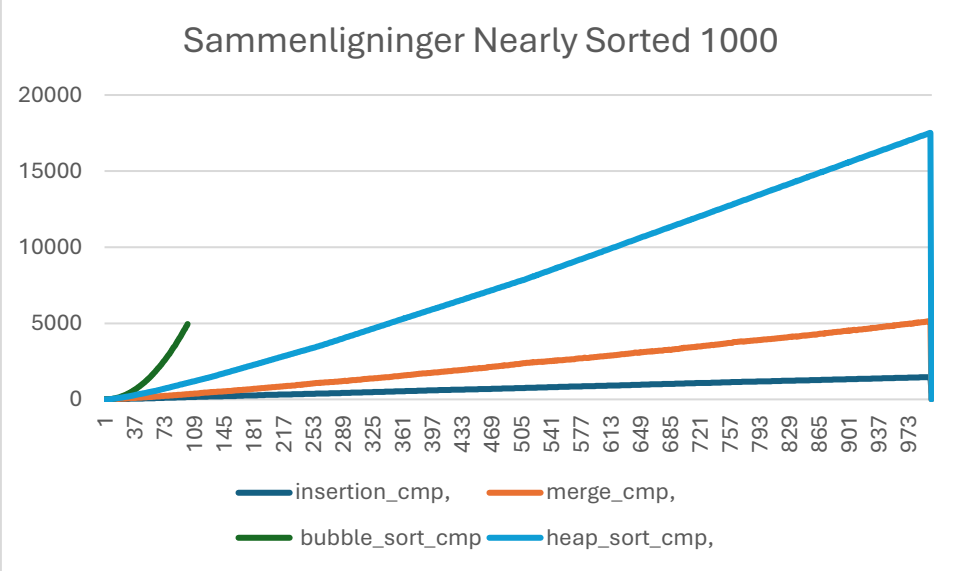

Kjøretid - Random 1000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
1 32 63 94 125 156 187 218 249 280 311 342 373 404 435 466 497 528 559 590 621 652 683 714 745 776 807 838 869 900 931 962 993
insertion_time,
merge_time,
bubble_sort_time,
heap_sort_time


Kjøretid - nearly sorted 1000
120000
100000
80000
60000
40000
20000
0
1 32 63 94 125 156 187 218 249 280 311 342 373 404 435 466 497 528 559 590 621 652 683 714 745 776 807 838 869 900 931 962 993
insertion_time,
merge_time,
bubble_sort_time,
heap_sort_time


Kjøretid - Random 10000
120000
100000
80000
60000
40000
20000
0
1 296 591 886 1181 1476 1771 2066 2361 2656 2951 3246 3541 3836 4131 4426 4721 5016 5311 5606 5901 6196 6491 6786 7081 7376 7671 7966 8261 8556 8851 9146 9441 9736
merge_time,
heap_sort_time


Sammenligninger Random 10000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
1
305
609
913
1217
1521
1825
2129
2433
2737
3041
3345
3649
3953
4257
4561
4865
5169
5473
5777
6081
6385
6689
6993
7297
7601
7905
8209
8513
8817
9121
9425
9729
merge_cmp,
heap_sort_cmp,


Sammenligninger Random 1000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
1
33
65
97
129
161
193
225
257
289
321
353
385
417
449
481
513
545
577
609
641
673
705
737
769
801
833
865
897
929
961
993
insertion_cmp,
merge_cmp,
bubble_sort_cmp,
heap_sort_cmp,


Sammenligninger Nearly Sorted 1000
20000
15000
10000
5000
0
1
37
73
109
145
181
217
253
289
325
361
397
433
469
505
541
577
613
649
685
721
757
793
829
865
901
937
973
insertion_cmp,
merge_cmp,
bubble_sort_cmp
heap_sort_cmp,