# DV1625 Rapport Inlämningsuppgift A

Viktor Listi, vili22

20 Augusti 2023

## Introduktion

I dagens värld spelar olika sorteringsalgoritmer en avgörande roll på flertal aspekter av vårat samhälle även fast de oftast jobbar i det tysta. Hur effektiva dessa algoritmer är har direkt påverkan på våra livstempon. Dessa algoritmer ligger konstant i bakgrunden och ser till så att internet bland annat som är en stor del av vår dagliga värld fungerar genom att konstant hantera de otroligt stora mängder data som omsätts varje sekund. Om dessa algoritmer skulle vara tröga så skulle även vårat livstempo vara det, därför finns det en konstant strävan efter snabbare och snabbare algoritmer.

## Bakgrund

Sorterings algoritmer av olika slag är en viktig del av världen vi lever i idag och styr mycket av det vi inte ser. Om dessa bakomligande algoritmer skulle vara långsamma så skulle även våra liv vara långsammare, därför tas det konstant fram snabbare och snabbare algoritmer för att konstant höga hastigheten på det dem är tillför. I denna rapport kommer sorterings algoritmerna Insertionsort, Quicksort och Mergesort diskuteras och utredas, det kommer även utredas om dessa kan göras snabbare genom att sätta ihop dem och skapa hybridalgoritmer.

### Insertionsort

Insertionsort är en enkel sorteringsalgoritm som bygger på principen att bygga upp en sorterad sekvens stegvis genom att infoga varje element på rätt plats i den befintliga sorterade delen av listan.

Sorteringsalgoritmen har en tidskomplexitet på N^2 i medelfallet.

+ Enkel att förstå och implementera
+ Effektiv på små datamängder
- Dålig prestanda på stora listor

### Quicksort

Quicksort är en snabb och effektiv sorteringsalgoritm som använder "divide and conquer" strategin. Algoritmen väljer en pivot i listan och delar sedan upp den i två delar där den sedan sorterar listan och kombinerar dem, den appliceras oftast rekurisvt.

Sorteringsalgoritmen har en tidskomplexitet på N*log(N) i medelfallet.

+ Fungerar väl med stora datamängder
+ Snabb och effektiv algoritm med bra genomsnittlig prestanda.
- Ineffektiv om pivot elementet väljs dåligt.

### Mergesort

Mergesort är en stabil och effektiv sorteringsalgoritm som använder "divide and conquer" strategin. Algoritmen bygger på principen att dela upp listan i mindre delar och rekurisvt sortera dem och slutgiltigen slå ihop dem sorterade del mängderna.

Sorteringsalgoritmen har en tidskomplexitet på N*log(N) i medelfallet.

+ Effektiv med stora datamängder
+ Stabil algoritm som håller ordningen på liknande element
- Inte in-place, använder mycket minne

## Metod och implementationsval

Under konstruktionen av algoritmerna som ska testas har en version som är väldigt lik olika pseudo kodade versioner av algoritmerna konstruerats. Insertionsort implementationen fungerar genom att först kolla om längden av listan är större än 1, om den är det så använder den sig av en nestad loop för att sedan sortera listan. Mergesort implementationen använder sig av samma metod för att kolla längden på listan där den sedan delar upp listan och rekursivt sorterar den, och efter att den blivit sorterad så använder den sig av en loop för att sätta ihop den igen. Quicksort implementationen använder sig också av samma metod i början för att kolla längden, sedan väljs ett pivot element ut i mitten av listan och sedan rekursivt sorteras den. Alla dessa algoritmer är implementerade utan hjälp funktioner.

## Resultat och Analys

### Sorteringsalgoritmer

| Algoritm | Körtid (ms) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | Medelvärde |
| Insertionsort | 0.1276 | 0.1520 | 0.1322 | 0.1341 | 0.1417 | 0.13752 |
| Quicksort | 0.1340 | 0.1812 | 0.1415 | 0.1649 | 0.1366 | 0.15164 |
| Mergesort | 0.1353 | 0.1526 | 0.1490 | 0.1472 | 0.1725 | 0.15132 |

*Tabell 1 körtid för algoritmer där n = 10*

| Algoritm | Körtid (ms) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | Medelvärde |
| Insertionsort | 0.5410 | 0.4693 | 0.4659 | 0.3767 | 0.4903 | 0.46864 |
| Quicksort | 0.3329 | 0.2361 | 0.3536 | 0.2822 | 0.2284 | 0.28664 |
| Mergesort | 0.5285 | 0.4364 | 0.3305 | 0.3067 | 0.3178 | 0.38398 |

*Tabell 2 körtid för algoritmer där n =100*

| Algoritm | Körtid (ms) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | Medelvärde |
| Insertionsort | 31.1072 | 20.0865 | 28.1248 | 28.3679 | 35.4084 | 35.4084 |
| Quicksort | 1.9298 | 2.1701 | 1.4525 | 1.4601 | 1.4499 | 1.69248 |
| Mergesort | 2.1910 | 2.2138 | 2.1873 | 2.1673 | 2.2921 | 2.2103 |

*Tabell 3 körtid för algoritmer där n =1000*

| Algoritm | Körtid (ms) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | Medelvärde |
| Insertionsort | 2388.2962 | 2338.3079 | 2462.6624 | 2651.3088 | 2946.8612 | 2557.4873 |
| Quicksort | 25.3441 | 25.7425 | 17.0989 | 19.7859 | 24.5495 | 22.50418 |
| Mergesort | 25.9263 | 25.8382 | 26.5725 | 24.8113 | 27.7293 | 26.17552 |

*Tabell 4 körtid för algoritmer där n =10000*

| Algoritm | Körtid (ms) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | Medelvärde |
| Insertionsort | 1079534.8324 | 782764.7321 | 971917.3388 | 917897.5711 | 826179.4124 | 915658.7774 |
| Quicksort | 334.7611 | 240.2468 | 296.0796 | 228.0164 | 266.0541 | 273.0316 |
| Mergesort | 381.3817 | 374.0679 | 314.0504 | 472.9154 | 350.738 | 378.63068 |

*Tabell 5 körtid för algoritmer där n =100000*

| Algoritm | *c*-konstanter | | | | | |
|---|---|---|---|---|---|---|
| | $c_{10}$ | $c_{100}$ | $c_{1000}$ | $c_{10000}$ | $c_{100000}$ | $c_{medel}$ |
| Insertionsort | 0.0013752 | 0.000046864 | 0.0000354084 | 0.000025574873 | 0.000915658774 | 0.0003149 |
| Quicksort | 0.015164 | 0.0014332 | 0.00056416 | 0.0005626045 | 0.0005460632 | 0.003654 |
| Mergesort | 0.015134 | 0.0019199 | 0.00073677 | 0.000654338 | 0.00075726 | 0.0038406 |

*Tabell 6 Tabell för C konstanter av algorithmerna vid olika mängder tal*

## Identifiering av brytpunkter

Genom att analysera den informationen vi nu har om de olika algoritmerna kan vi ställa upp ekvationer för att konstatera brytningspunkten där quicksort liksom mergesort blir mer effektiva även insertionsort. Vi använder oss av ekvationen *C*TimeComplexity = Time.* Med hjälp av denna kan vi sedan ställa upp ekvationer för att beräkna brytningspunkterna. Vi började analysen straxt under N = 100 då det var där vi under våra tidigare tester kunde se att dem passerade varandra i körtiden. Vi såg dock snabbt att även på N = 80 var skillnaden för stor så vi började jämförelsen från mindre mängd av tal och jobbade oss uppåt tills att vi hittat brytpunkterna.

## Mergesort:

N = 80 => 0.0038406 * 80 * log(80) = 0.58472005923

N = 20 => 0.0038406 * 20 * log(20) = 0.099934716

N = 15 => 0.0038406 * 15 * log(15) = 0.06775344134

N = 14 => 0.0038406 * 14 * log(14) = 0.06162548067

N = 13 => 0.0038406 * 13* log(13) = 0.05561674091

### Insertionsort:

N = 80 => 0.0003149 * 80^2 = 2.01536

N = 20 => 0.0003149 * 20^2 = 0.12596

N = 15 => 0.0003149 * 15^2 = 0.0708525

N = 14 => 0.0003149 * 15^2 = 0.0617204

N = 13 => 0.0003149 * 15^2 = 0.0532181

N = 12 => 0.0003149 * 15^2 = 0.0453456

### Quicksort:

N = 80 => 0.003654* 80 * log(80) = 0.556311265

N = 20 => 0.003654* 20 * log(20) = 0.09507927208

N = 15 => 0.003654* 15 * log(15) = 0.06446156191

N = 14 => 0.003654* 14 * log(14) = 0.05863132579

N = 13 => 0.003654* 13* log(13) = 0.05291453712

N = 12 => 0.003654* 12* log(12) = 0.04731993928

Genom att analysera detta så kan vi se att mergesort är snabbare är insertionsort när N>13 och att quicksort är snabbare än insertionsort när N>12.

### Hybridsorteringsalgoritmer

| Filstorlek | Algoritm | Körtid (ms) | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | Medelvärde |
| 10 | $Hybrid_Q$ | 0.1401 | 0.1274 | 0.1308 | 0.1320 | 0.1383 | 0.13372 |
| | $Hybrid_M$ | 0.1267 | 0.1273 | 0.1359 | 0.1289 | 0.1285 | 0.12946 |
| 100 | $Hybrid_Q$ | 0.2743 | 0.2155 | 0.2052 | 0.2043 | 0.2281 | 0.22548 |
| | $Hybrid_M$ | 0.2328 | 0.2332 | 0.2315 | 0.2351 | 0.2301 | 0.23254 |
| 1000 | $Hybrid_Q$ | 1.1740 | 1.2337 | 1.1849 | 1.1981 | 1.1826 | 1.19466 |
| | $Hybrid_M$ | 1.5265 | 1.4698 | 1.5559 | 1.5368 | 1.4766 | 1.51312 |
| 10000 | $Hybrid_Q$ | 15.0268 | 15.8093 | 16.2510 | 23.0187 | 15.3574 | 17.09264 |
| | $Hybrid_M$ | 19.0863 | 19.4287 | 19.31380 | 19.6474 | 19.4680 | 19.38884 |
| 100000 | $Hybrid_Q$ | 209.5491 | 214.4832 | 222.5871 | 223.5125 | 230.4958 | 220.12554 |
| | $Hybrid_M$ | 256.9866 | 271.5171 | 264.8028 | 274.6864 | 274.1359 | 268.42576 |

*Tabell 7 Hybridsorteringsalogirtmer och körtider (ms)*

# Slutsatser

Genom att analysera körtiderna från våra originella algoritmer mot de hybridalgoritmerna som vi skapat kan vi se att dessa hybridalgoritmer är märkvärdigt snabbare än de original algoritmerna detta bekräftar att kombinera olika tillvägagångssätt kan leda till betydande förbättringar av prestandan. Genom detta har vi bevisat att man kan sammanföra olika metoder för att uppnå ett bättre resultat.