# Rapport til oblig 2

Denne obligatoriske innleveringen består av 2 oppgaver.

## Oppgave 1

I oppgave 1 skulle vi implementere quicksort og randomizedQuicksort. I figur 1 nedenfor kan man se en graf som viser kjøretid for ulike usorterte mengder. Jeg testet begge algoritmene med 10 elementer, 100 elementer, 1 000 elementer, 10000 elementer, 100000 elementer og 1000000 elementer. Resultatene nedenfor er målt ut ifra tiden det tok å kjøre algoritmene.

*Figur 1.1: Sammenligning mellom quicksort, randomizeQuicksort, innsettingssortering (insertion sort) og flettesortering (merge sort). Hele grafen til innsettingssorteringen ses i figur 1.2.*

*Figur 1.2: Graf som viser tidsbruken til innsettingssortering.*

Etter å ha testet de ulike usorterte elementene, genererte jeg nye filer med 10, 100, 1000, 10000, 100000, 500000 og 1000000 elementer, men denne gangen sørget jeg for at elementene var *omvendt* sortert, altså at det største elementet kom først, så det nest største, etc., hele veien ned til det siste som da var det minste elementet, for så å måle tiden det tok å kjøre quicksort og randomizeQuicksort for å sortere elementene slik at det minste elementet kommer først. Da jeg kjørte quicksort med en mengde større enn 10 000 omvendt sorterte elementer, måtte jeg legge til et ekstra argument, -XssXm under kjøring for å unngå en stack overflow error. For å kunne kjøre denne sorteringen på 1 000 000 omvendt sorterte elementer, måtte jeg kjøre *java -Xss100m quicksort reversedsorted\_1000000.txt*, der reversedsorted\_1000000.txt er filen som inneholder de 1 million omvendt sorterte tallene. Resultatene i grafene nedenfor er målt ut fra tiden det tok å kjøre algoritmene med omvendt sortert rekkefølge.

*Figur 1.3: Sammenligning mellom quicksort, randomizeQuicksort, innsettingssortering og flettesortering for omvendt sorterte elementer. Grafene til quicksort og innsettingssortering vises i figur 1.4.*

*Figur 1.4: En graf som viser sammenligningen mellom quicksort og flettesortering når elementene er omvendt sorterte*

Nedenfor viser jeg de samme resultatene i tabellform. Legg merke til at i tabellene oppgis tidsbruken i ulike enheter avhengig av hvor lang tid kjøringen av algoritmen tok.

ns = nanosekunder (10-9 s), µs = mikrosekunder (10-6 s), ms = millisekunder (10-3 s), s = sekunder

Tabell 1.1: Sammenlikning mellom quicksort, randomizeQuicksort, innsettingssortering (insertion sort) og flettesortering (merge sort) for usorterte elementer:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | quicksort | Randomize-Quicksort | Innsettings-sortering | flettesortering |
| 10 | 8 µs | 511 µs | 4,025 ns | 14 µs |
| 100 | 46 µs | 634 µs | 89 µs | 83 µs |
| 1000 | 575 µs | 2,625 ms | 6 ms | 925 µs |
| 10000 | 2,337 ms | 8,518 ms | 47 ms | 3,046 ms |
| 100000 | 14,394 ms | 23,358 ms | 1,122 s | 27,273 ms |
| 500000 | 50,227 ms | 73,503 ms | 28,541 s | 66,991 ms |
| 1000000 | 105,537 ms | 158,409 ms | 110,816 s | 135 ms |

Av tabell 1.1 ser vi at når elementene er usorterte, så bruker quicksort kortere tid enn randomizeQuicksort. Innsettingssortering er den mest effektive i starten med bare 4 nanosekunder for 10 elementer, men den er den mest ineffektive når det går over 10 000 elementer (ved innsetting av 1 000 000 usorterte elementer bruker innsettingssortering 1 minutt og 50,816 sekunder). Sammenlikner vi alle algoritmene, ser vi at når elementene er usorterte, så er quicksort gjennomsnittlig den mest effektive algoritmen.

Tabell 1.2: Sammenlikning mellom quicksort, randomizeQuicksort, innsettingssortering (insertion sort) og flettesortering (merge sort) for omvendt sorterte elementer:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | Quicksort | Randomize-Quicksort | Innsettings-sortering | Flettesortering |
| 10 | 9,583 µs | 514,268 µs | 4,525 µs | 14,573 µs |
| 100 | 342,509 µs | 1,199 ms | 126,258 µs | 70,955 µs |
| 1 000 | 10,415 ms | 2,671 ms | 6,740 ms | 798,142 µs |
| 10 000 | 72,672 ms | 8,757 ms | 52,724 ms | 2,454 ms |
| 100 000 | 5,101 s | 19,849 ms | 2,133 s | 12,039 ms |
| 500 000 | 1 min 45,166 s | 57,042 ms | 53,456 s | 30,348 ms |
| 1 000 000 | 8 min 3,787 s | 119,444 ms | 3 min 36,276 s | 52,789 ms |

Av tabellene ser vi at når elementene er motsatt sortert, så er quicksort uhyre ineffektivt ved innsetting av et stort antall elementer. RandomizedQuicksort bruker lang tid når det er få elementer, men er svært effektivt når det er mange elementer som settes inn. Flettesortering er treg når det settes inn få elementer, men er den aller raskeste av de 4 sammenlignede algoritmene når det settes inn mange omvendt sorterte elementer.

## Oppgave 2

I oppgave 2 skulle vi ta for oss radixSort. Implementeringen skulle gjøres slik at r kunne ha ulike verdier og plotte resultatene av disse inn i en tabell og deretter en graf. Inputstørrelsene jeg brukte var de samme som i oppgave 1: 10,100, 1 000, 10 000, 100 000, 500 000 og 1 000 000.

Tabell 2.1: Tiden radixSort brukte på å sortere ulike inputstørrelser med usorterte heltall:

|  |  |  |
| --- | --- | --- |
| Inputstørrelse | R | Tid |
| 10 | 8 | 41,681 µs |
| 10 | 16 | 2,566 ms |
| 10 | 32 | 14,274 µs |
| 100 | 8 | 98, 918 µs |
| 100 | 16 | 2,593 ms |
| 100 | 32 | 13,327 µs |
| 1 000 | 8 | 354,367 µs |
| 1 000 | 16 | 4,214 ms |
| 1 000 | 32 | 86,704 µs |
| 10 000 | 8 | 3,153 ms |
| 10 000 | 16 | 3,383 ms |
| 10 000 | 32 | 822,257 µs |
| 100 000 | 8 | 7,697 ms |
| 100 000 | 16 | 6,437 ms |
| 100 000 | 32 | 3,770 ms |
| 500 000 | 8 | 26,658 ms |
| 500 000 | 16 | 20,152 ms |
| 500 000 | 32 | 8,692 ms |
| 1 000 000 | 8 | 47,469 ms |
| 1 000 000 | 16 | 34,812 ms |
| 1 000 000 | 32 | 16,566 ms |

Når R er 32, ser vi at raidxSort bruker kortere tid enn for de andre verdien av R det er sammenliknet med i tabellen 2.1 for én inputstørrelse. Vi ser for eksempel at når det er 10 000 usorterte elementer, bruker radixsort 3,1 millisekunder når R er 8. Når R er 16 bruker radixSort 3,3 millisekunder på å sortere 10 000 elementer, mens radixSort kun bruker 822 mikrosekunder (eller 0,8 millisekunder) når R er 32 for 10 000 usorterte elementer.

Når inputstørrelsen er liten, bruker radixSort lengst tid når R er 16, men fra 100 000 elementer og oppover, bruker radixSort lengst tiden når R er 8.

*Figur 2.1: En graf som viser kjøretiden for radixSort med ulik inputstørrelse og ulike verdier til R.*