PG4200: Innlevering 2

av Anders Mikkelsen, Espen Rønning

Introduksjon

Oppgaven er utviklet i et miljø som benytter Java 8 JDK. JAR-filen «stdlib» ligger i default-pakken og for at man skal kunne kjøre programmet fra terminal nyttes kommandoene:

- `javac –cp .;stdlib.jar ProgramNavn.java`
- `java –cp .;stdlib.jar ProgramNavn`

Oppgave 1)

I denne oppgaven har vi valgt å sortere navn alfabetisk stigende, sist modifisert og filstørrelse synkende og kjørbarhet synkende (altså kjørbare først). Vi antar at dette blir riktig rekkefølge på kjørbarhet da dokumentasjonen sier at metoden returner -1 hvis ikke kjørbar og 1 hvis kjørbar.

Oppgave 2)

I denne oppgaven har vi hardkodet enkodingen av klassekampen.no som er «Windows-1252». Dette var nødvendig for å implementere søkene våre på navn som inneholdt æøå. Vi har også hardkodet lokasjonen for stortinget2014.txt til storting/stortinget2014.txt.

I metoden getURL har vi valgt å kun søke etter stortingsrepresentanter i urlene som har path http://www.klassekampen.no/article/xxx. Dette fordi vi tror at disse sidene vil gi flest resultater, redusere trafikken og gi oss den informasjonen som faktisk er interessant.

Vi presiserer at oppgaven kun responderer på navn når de er skrevet korrekt. Altså med store bokstaver på riktige plasser. Dette fordi det er sånn man skriver navn.

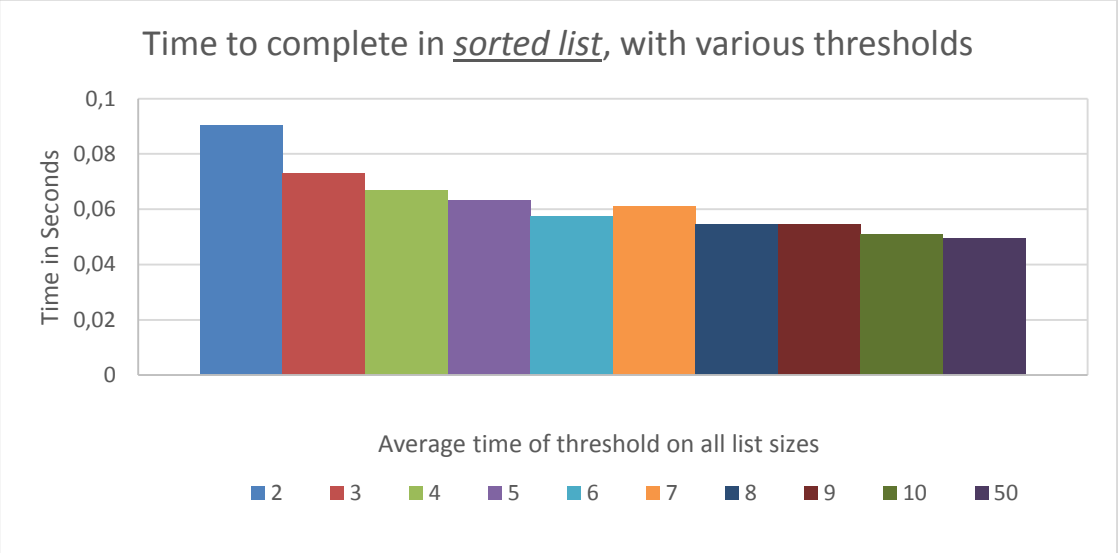
Oppgave 3b i)

I denne oppgaven valgte vi først å implementere en klasse kalt MergeSortStatistics samt gjøre noen endringer i MergeSort klassen i den hensikt å kjapt generere statistikk på forskjellige thresholds opp mot forskjellige størrelser av lister. Vi går her fra threshold 2 til threshold 50. Listene går fra størrelse 0 til 10 000 000, i intervaller på 100 000.

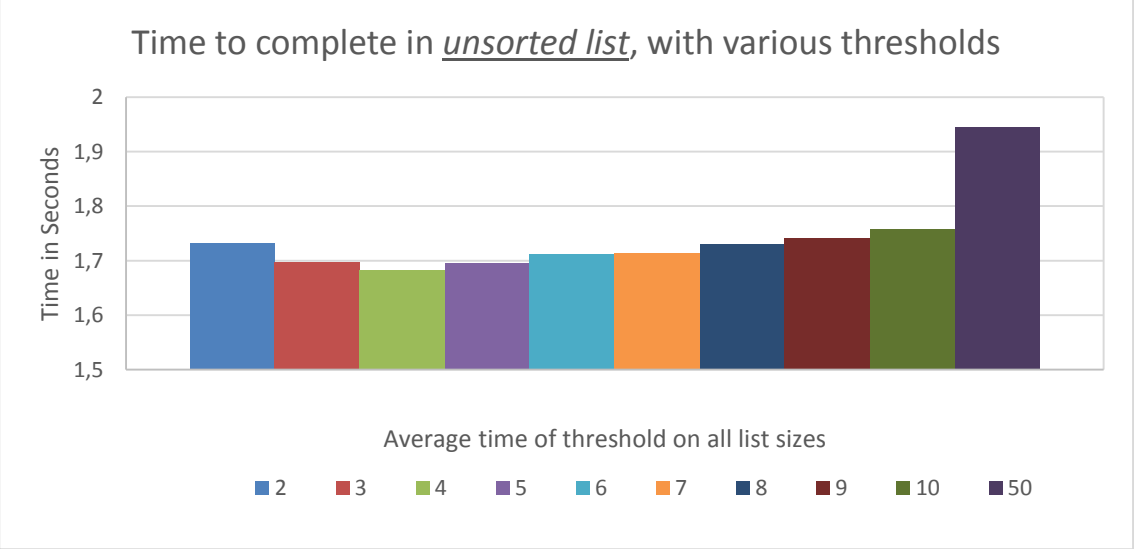
Vi har så satt opp to blokkskjema som viser gjennomsnittstiden over alle listerstørrelser for sorterte og usorterte lister. De usorterte listene er nygenerert for hver eneste listestørrelse og hver eneste threshold. Dette gir ett mer realistisk bilde på hva den faktiske gjennomsnittstiden blir når algoritmen er i praktisk bruk og ikke testes i ett lab.

Grunnen til at vi har valgt å benytte oss av gjennomsnittstiden denne gangen i stedet for å sette opp en graf på alle de forskjellige tidene er fordi det med så mange forskjellige listestørrelser mer skaper en uryddig graf enn å gi så mange svar. Vi mener at dette oppsettet gir oss ett riktig svar men på en mye mer ryddig og stabil måte, uten tydelige tegn til HW overhead i grafen. Ref. Innlevering 1.

Sortert Liste:



Usortert Liste:



Grafene er hentet ut av vedlagt exceldokument.

På sorterte lister ser vi at eksekveringstiden synker jo høyere threshold vi har. Dette er logisk fordi insertionSort har en veldig høy ytelse på nesten sorterte lister (ekstremt bra på ferdigsorterte). Grunnen til dette er at insertionSort er en adaptiv metode som er nærmere O(n) enn $O(n^2)$ for nesten sorterte lister og selve algoritmen har veldig lite HW overhead. Den er adaptiv i den forstand at den benytter seg av graden av uorden når den sorterer. Hvis den kun må iterere noen få plasser tilbake for å plassere en verdi vil den bli ferdig fort. Hvis listen er i «maks uorden» som ironisk nok vil si at den er sortert descending i stedet for ascending vil insertionSort ha $O(n^2)$.

Det vil si at for en sortert liste blir det slik at desto nærmere thresholden er størrelsen på listen, jo fortere vil det gå fordi man vil trenge færre og færre rekursive kall til mergeSort og insertionSort løper med O(n) som er en raskere grad enn O(n (log n)) til MergeSort. Grunnen til det er at når listen allerede er sortert er splittingen mergeSort gjør helt unødvendig. Heldigvis vil mergeSort som er implementert i Arrays (som MergeSort.java er basert på) sjekke om listen er sortert og deretter hoppe over merging om det er tilfelle. Hadde det ikke vært for denne modifiseringen hadde det vært enda mer overhead.

På usorterte lister ser vi en annerledes oppførsel. Som vi vil se synker eksekveringstiden gradvis mot threshold 4 før den øker igjen mot 7. Fra 7 og oppover ser vi en markant økning i eksekveringstid. Dette er fordi insertionSort i teorien er treigere $O(n^2)$ men i praksis er raskere på svært små lister fordi den har svært liten overhead fordi den gjør in-place endringer og gjør færre compares enn mergeSort.

Vi ser av boka side 275 at best case sort med top-down mergesort (som er typen implementert i Arrays og MergeSort.java) er NlgN og på side 250 ser vi at insertionSort har en best case på $O(N^2 / 2)$. Vi vet også at insertionSort er en adaptiv metode som vil si at kjøretid for en delvis sortert liste vil være $O(n^2)$ ~ O(nk) hvor k er maksimalt antall plasser hvert enkelt element er unna sin sorterte posisjon. Vi kan derfor sette opp ett best case scenario for insertion- og mergeSort og på den måten regne oss frem til hvorfor insertionSort trumfer top-down mergeSort på alt under størrelse 7 og over størrelse 3 i best case. Dette antar vi er grunnen til at INSERTIONSORT_THRESHOLD i Arrays er satt til 7, for å kompensere for alle best case. Grunnen til at threshold 2 og 3 vises som tregere er at på disse størrelsene fører kallet på insertionSort stort sett til mer overhead for ingen verdi, fordi det kun ville vært ett rekursivt kall til mergeSort på threshold 3 og ingen på threshold 2. Dette unngår man ved å sette grensen til 7.

Eksempel :

Size: 4	Size: 5	Size: 6	Size: 7
(4 * 3) / 2 = 6	(5 * 4) / 2 = 10	(6 * 5) / 2 = 15	(7 * 6) / 2 = 21
4(lg 4) = 8	5(lg 5) ~ 11,6	6(lg 6) ~ 15,5	7(lg 7) ~ 19,65

Oppgave 3b ii)

Hvis vi ser på dette utdraget av rådataen vår for threshold 7 vil vi se at antall kall til compareTO er en valid måte metode for å måle kjøretid da den følger vekstfaktoren til mergeSort.

7	100000	0,001	99999	0,017	1572324
7	200000	0,002	199999	0,037	3345381
7	300000	0,002	299999	0,059	5176924
7	400000	0,003	399999	0,084	7089769
7	500000	0,006	499999	0,106	8989888

Vi ser at for sorterte lister (som er best case) så følger compares en modell av O(n). Altså når mengden dobles så dobles også antall av compareTo. Vi ser også at for en usortert liste følger den modellen for N(log N), slik som tiden gjør. Når antallet dobles, dobles også antallet kall på compareTo, pluss en konstant av verdi log N. I dette tilfelle er det ca. 300000. Det er selvfølgelig avhengig av hvor usortert lista er for hver iterasjon.

Oppsummert ser vi at antall kall på compareTo følger modellen for best og worst case og er derfor en god indikasjon på kjøretid.

Hvis du ønsker å produsere egen statistikk for MergeSort kan du kompilere og kjøre Driver.java. (Avhengig av stdlib.jar)

Oppgave 4)

I denne oppgaven valgte vi å lage en klassisk Top-down mergeSort basert på den i Arrays klassen. Vår implementasjon av bottom-up sort endte med å bli en slags halveisløsning. Vi bruker en queue for å simulere en bottom-up. Hvordan dette fungerer mener vi er tilstrekkelig beskrevet i koden og relativt greit å sette seg inn i.

Vi gjorde endel tester med checkforSorted men kom frem til at man uansett må iterere gjennom lista for å finne midten og at når man må det kan man like så godt bare merge med en gang. Hvis LinkedListen vi hadde å gjøre med var Double-Linked og hadde kontroll på sin egen tail hadde det vært en annen sak. Vi prøvde å implementere en checkSorted som forbedret ytelsen for Top Down da den uansett splitter lista på midten men lyktes med ikke dette, fordi nodene blir sortert og da er ikke midten det vi tror den er lenger. Etterhvert forstod vi at dette ville være uhensiktsmessig også i Top Down så lenge listen er Single-Linked. Dette fordi vi aldri vil kunne sjekke enden på den venstre lista vår uten å iterere gjennom hele. I en Double-Linked kunne vi bare kalt tail. Når vi uansett må iterere gjennom lista for å kunne sammenligne enden blir det samme problemet som i mergeSortBottomUp.

Vi hadde orginalt med en merging som baserte seg på rekursiv merging men dette viste seg å skape stack overflow errors rimelig fort. Endret derfor til en iterativ versjon.