IN2010 høsten 2019

Obligatorisk oppgave 3: Sortering

Innleveringsfrist fredag 1. november kl. 23.59.

1. Innledning

Denne oppgaven er laget for at du skal få praktisk erfaring med en del av sorteringsalgoritmene fra pensum. Du skal implementere algoritmene i Java, teste dem på ulike typer input og levere en rapport (i pdf-format) som beskriver arbeidet du har gjort. Som dokumentasjon skal du også levere en javafil med all koden din samlet, men det er pdf-rapporten som vil være utgangspunktet for vurdering og tilbakemelding fra retterne.

Det finnes selvfølgelig mange implementasjoner av dette på nettet og ellers. I denne oppgaven forventes du å skrive dine egne versjoner. Vi tror du vil lære mest av denne oppgaven ved å gjøre den alene, men diskuter gjerne med andre underveis. Pass på at det du leverer inn er et resultat av ditt eget arbeid og i tråd med instituttets generelle krav til innleveringer.

2. Algoritmene

Algoritmene du kan jobbe med i denne oppgaven er:

- *Utplukkssortering (selection-sort; algoritme 5.3 s. 160)*
- *Innstikksortering (insertion-sort; algoritme 5.5 s. 162)*
- **Heapsortering (heap-sort; algoritme 5.16 s. 176)**
- **Kvikksortering (quick-sort; algoritme 8.9/8.11 s.255/6)**
- **Flettesortering (merge-sort; algoritme 8.3 s. 245)**
- Bøttesortering (bucket-sort; algoritme 9.2 s. 267)
- Radixsortering (radix-sort; algoritmen beskrevet s. 268-9, jf. oppg. R-9.2)

Du skal velge minst tre algoritmer, én av *de to første* og to av **de tre i midten**. Vær oppmerksom på at alle algoritmene er pensum, så det kan være klokt å velge mer enn tre, og å velge dem du synes virker vanskeligst.

3. Implementasjon

Implementer minst tre av algoritmene nevnt over i Java. Implementasjonen din skal følge algoritmebeskrivelsen så tett som mulig, men slik at det som skal sorteres er arrayer med heltall (int). De fire første algoritmene skal ikke ha ekstra datastruktur som overgår O(1). De skal operere direkte på arrayen de får som parameter (in-place).

I pdf-rapporten skal du for hver algoritme:

- Inkludere Java-koden (kun implementasjonen av selve algoritmen).
- Beskrive eventuelle valg du har tatt og utfordringer du har møtt i arbeidet med å konvertere fra algoritme til Java-implementasjon.

4. Test av korrekthet og mønster

Lag et lite program som tester hver av implementasjonene over på sortering av ti tall (int). For hver algoritme skal du teste at den fungerer på input som er

- tilfeldig
- sortert etter stigende verdier
- sortert etter fallende verdier (dvs reversert)

I denne deloppgaven må du gjerne begrense verdiområdet til å for eksempel sortere tall fra 0 til 9. For å generere tilfeldig input kan du bruke java.util.Random og metodene nextInt() og nextInt(int bound). Vi anbefaler at du lager en god struktur på testprogrammet ditt, men det er ikke noe krav.

Skriv ut verdiene i arrayen for hver iterasjon i algoritmen(e). I pdf-rapporten skal du for hver implementasjon:

- Beskrive for hver type input om det fremkommer noe spesielt mønster, og eventuelt hvilket.
- Gi en forklaring på disse mønstrene ut fra hvordan algoritmen fungerer.

5. Test av hastighet

Utvid programmet ditt til å teste sortering av vilkårlig mange elementer. Du skal nå teste hvor lang tid de ulike implementasjonene bruker for tilfeldig, sortert og reversert input. Test for antall elementer lik 1 000, 5 000, 10 000, 50 000, 100 000 osv så langt oppover som det er praktisk mulig (avbryt kjøringene for eksempel etter 5-10 minutter). Test for arrayer med unike verdier og arrayer med flere elementer som har lik verdi. Test også både med og uten ulike begrensninger på verdiområdet for elementene. I pdf-rapporten skal du lage en tabell som for hver implementasjon viser kjøretiden for antall elementer som angitt over. Test også kjøretiden ved å bruke java.util.Arrays.sort(…) med samme antall elementer og legg dette til i tabellen.

Uthev i tabellen hvilken implementasjon som var raskest og tregest for hver kombinasjon av type input og antall elementer. Faktisk kjøretid i Java kan du finne ved å pakke koden din inn i:

```
long t = System.nanoTime(); //nanosekunder

//  din kode her

double tid = ( System.nanoTime() - t ) / 1000000.0;    //millisekunder
```

I pdf-rapporten skal du også svare på:

- Er det noen av resultatene i tabellen som overrasker deg?
- Sammenlign faktisk kjøretid med forventet kjøretid basert på O-notasjon for hver av algoritmene. Stemmer teorien med praksis?

6. Ekstraoppgaver (frivillige)

Ofte anbefales det å bruke quicksort ved store datamengder, men skifte til innstikksortering mot slutten i stedet for å gjennomføre rekursjonen helt til bunns. Implementer denne kombinasjonen og test programmet for å gi et anslag på for hvor mange elementer det eventuelt kan lønne seg å gå fra quicksort til innstikksortering.

Du kan også implementere, teste og ta med i rapporten flere av sorteringsalgoritmene enn de tre obligatoriske.

7. Krav til innleveringen

Vi forventer at du leverer en godt strukturert og ryddig pdf-rapport. Denne må inkludere alle punktene over, det vil si:

- For hver av algoritmene i seksjon 2:
    - Inkludere javakoden (kun implementasjonen av selve algoritmen).
    - Beskrive eventuelle valg du har tatt og utfordringer du har møtt i arbeidet med å konvertere fra algoritme til Java-implementasjon.
    - Beskrive for hver type input (tilfeldig/sortert/reversert) om det fremkommer noe spesielt mønster, og eventuelt hvilket.
    - Gi en forklaring på disse mønstrene ut fra hvordan algoritmen fungerer.
- En tabell med test-resultater som beskrevet i seksjon 5 (husk å teste også med Arrays.sort), og med svar på de to spørsmålene:
    - Er det noen av resultatene i tabellen som overrasker deg?
    - Sammenlign faktisk kjøretid med forventet kjøretid basert på O-notasjon for hver av algoritmene. Stemmer teorien med praksis?

I tillegg skal selve javafilene leveres for dokumentasjon og etterprøvbarhet.