TEQUE (Trippel-Ended Queue):

For å kjøre Teque-programmet, skriv følgende i terminalen…

```
$ javac *.java
$ java Teque < inputs/eksempel_input
```

…der «eksempel_input» kan byttes ut med den input-filen programmet skal testes på. 'inputs'-mappen må ligge i samme mappe som Teque.java ligger i.

a) Pseudokoder for noen Teque-operasjoner:

push_back():

```
Procedure push_back(x):
    nyNode <- new Node(x)
    size <- size + 1
    if  (empty) then:
        forste <- nyNode
        siste <- nyNode
        midten <- nyNode
    end if
    else:
        siste.neste <- nyNode
        nyNode.forrige <- siste
        siste <- nyNode
        if (size % 2 = 0) then:
            midten <- midten.neste
        end if
    end else
end
```

push_front():

```
Procedure push_front(x):
    nyNode <- new Node(x)
    size <- size + 1
    if (empty) then:
        forste <- nyNode
        siste <- nyNode
        midten <- nyNode
    end if
    else:
        nyNode.neste <- forste
        forste.forrige <- nyNode
        forste <- NyNode
        if (size % 2 = 1) then:
            midten <- midten.forrige
        end if
    end else
end
```

push_middle():

```
Procedure push_middle(x):
    nyNode <- new Node(x)
    size <- size + 1

    if (empty) then:
        forste <- nyNode
        siste <- nyNode
        midten <- nyNode
        return
    end if

    gammelMid <- midten
    midten <- nyNode

    if (size % 2 = 0) then:
        gammelNeste <- gammelMid.neste
        gammelMid.neste <- nyNode
        nyNode.forrige <- gammelMid
        nyNode.neste <- gammelNeste
        if (gammelNeste is not null) then:
            gammelNeste.forrige <- nyNode
        end if
    end if
    else
        gammelForrige <- gammelMid.forrige
        gammelMid.forrige <- nyNode
        nyNode.neste <- gammelMid
        nyNode.forrige <- gammelForrige
        if (gammelForrige is not null) then:
            gammelForrige.neste <- nyNode
        end if
    end else

    if (size = 2) then:
        siste <- nyNode
    end if
end
```

get():

```
Procedure get(index):
    if (index < 0 OR index >= size) then:
        print("Ugyldig indeks!")
        return
    end if

    counter <- 0
    peker <- forste
    while (peker is not null) do:
        if (counter = index) then:
            print(peker.data)
            return
        end if
        peker <- peker.neste
        counter <- counter + 1
    end while
end
```

b)

Se Java-program.

c)

Verste-tilfelle kjøretidsanalyse med O-notasjon (hvor N er vilkårlig stor):

push_back-operasjonen:

O(1), dvs. konstant tid. Operasjonen vil bruke like lang tid uavhengig av størrelsen på input. Årsaken er at vi har en 'siste'-peker som gir direkte aksess til det siste elementet hvor vi kan justere pekere.

push_front-operasjonen:

O(1), dvs. konstant tid. Operasjonen vil bruke like lang tid uavhengig av størrelsen på input. Årsaken er at vi har en 'første'-peker som gir direkte aksess til det første elementet hvor vi kan justere pekere.

push_middle-operasjonen:

O(1), dvs. konstant tid. Operasjonen vil bruke like lang tid uavhengig av størrelsen på input. Årsaken er at vi har en 'midten'-peker som gir direkte aksess til det midterste elementet hvor vi kan justere pekere.

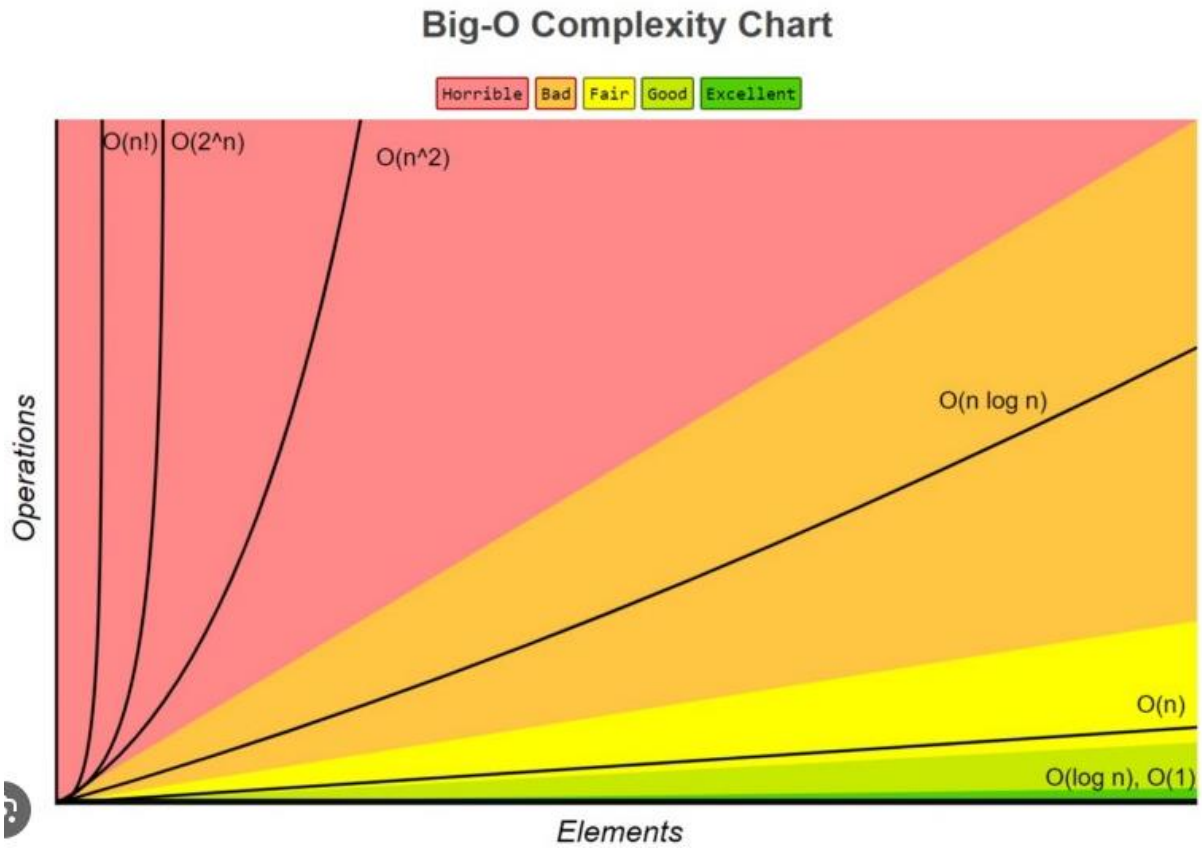
get-operasjonen:

O(N), dvs. lineær tid. Operasjonen vil i verste tilfelle måtte iterere over hele lenkelisten for å få tak i elementet av interesse. Tendensen i kjøretid er derfor lineær med størrelsen på input.

d)

10^6 er en konstant, og i O-notasjon fjerner vi konstanter og lavere ordens termer. Ved å gjøre dette fokuserer vi på den dominerende faktoren som påvirker kjøretiden når inndata vokser. Grunnen er at vi ønsker å få en mer generell forståelse for hvordan algoritmene presterer på store inndata. Hvis N er begrenset, blir det vanskelig å skille mellom algoritmer med ulik kjøretid fordi forskjellene i kjøretid på små inndata kan være ubetydelige. Dessuten vil det være vanskelig å si noe pålitelig om

atferden til algoritmene på store datasett når N er begrenset. Derfor er det viktig å fjerne begrensningen på N når vi analyserer algoritmers kompleksitet, slik at vi kan sammenligne deres ytelse og forstå hvordan de skalerer med store datasett.



På Store O-diagrammet ser vi at kurvene nærmest overlapper hverandre når N (Elements) er veldig liten.

SORTERING:

For å kjøre Sorterings-programmet; kompiler først, deretter kan du eventuelt skrive følgende i terminalen…

```
$ java Main inputs/random_100
```

… der «random_100» kan byttes ut med den input-filen programmet skal testes på. Bruk formen…

```
$ java Main ../inputs/nearly_sorted_10
```

…dersom inputs-mappen ligger over arbeidsmappen i mappehierarkiet. Merk at OUT-filene og CSV-filen også havner inne i inputs-mappen.

Implementerte sorteringsalgoritmer:

- Insertion Sort
- Merge Sort
- Bubble Sort
- Selection Sort

Jeg har valgt å ta i bruk pre-koden som ble gitt på GitHub, slik at hver sorteringsalgoritme implementeres som en klasse i hver sin fil og er subklasser av 'Sorter'-superklassen. 'Sorter' har en funksjon for bytting, samt flere for ulike sammenligninger. Hver gang disse blir påkalt så blir teller-variabler inkrementert. **Hvordan sammenligninger og bytter skjer generelt sett**:

- En sammenligning skjer i form av en if-test der et element på en indeks blir sammenlignet med et annet element på en annen indeks (betingelsessjekkene i løkkene telles altså ikke med).
- Et bytte ('swap') skjer dersom if-testen passerer og to elementer bytter plass.

Noen detaljer:

I Bubble Sort skjer det en sammenligning mellom parvise elementer. Dersom et større element står foran, bytter det plass med elementet bak. Slik vil de største elementene gradvis flyttes bakerst til en sortert partisjon (de «bobbler» opp til slutten av listen).

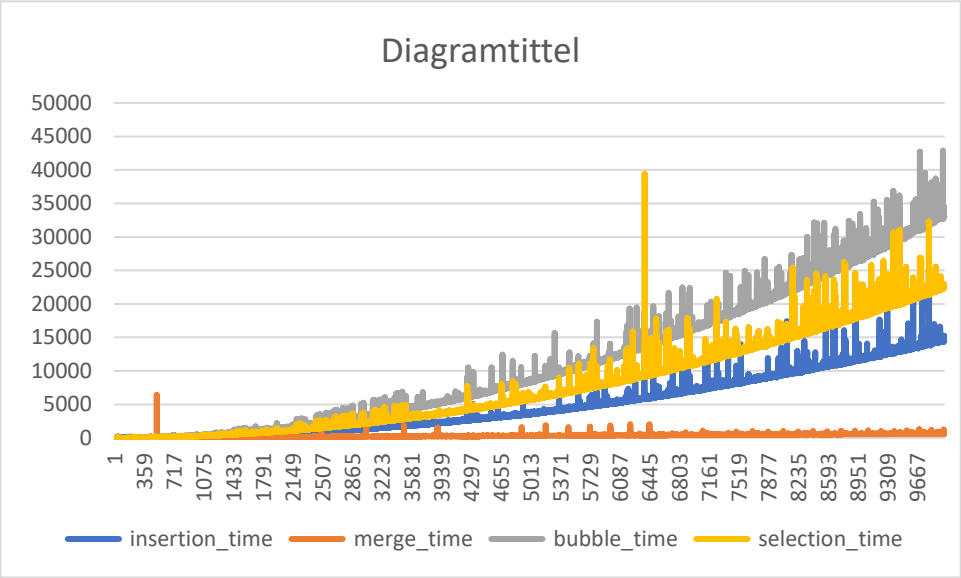
I Selection Sort skjer det en sammenligning både for å sjekke om vi må oppdatere minste-indeksen og for å sjekke om nåværende indeks i ytre loop har blitt forskjellig fra minste-indeksen. Et bytte skjer bare dersom nåværende indeks i ytre loop har blitt forskjellig fra minste-indeksen; i så fall skal elementene på de indeksene bytte plass. Slik vil de minste elementene gradvis flyttes fremst til en sortert partisjon.

I Insertion Sort skjer det en sammenligning mellom parvise elementer, men som en implisitt del av en while-loop som jobber «baklengs». Så lenge while-loopen går, blir det utført sammenligninger. Dersom elementet foran er større enn elementet bak, bytter de plass. Fordi while-telleren først settes til å være det samme som ytre loop-teller, så vil vi etter hvert jobbe oss bakenifra og fremover for å «rydde opp i» elementer som står i feil rekkefølge (litt som å sortere kort på hånda).

I Merge Sort skjer det en sammenligning bare ett sted, og det er under flettingen av del-arrayene vi ender opp med. Dersom elementet på gitt indeks i del-array 1 er mindre enn eller lik elementet i del-array 2, så setter vi vedkommende element inn i det flettede arrayet som vil bli sortert. Altså tolker jeg det slik at det ikke gjøres noen bytter fordi vi gjør fletting i stedet.

Eksperimenter

Det ble ikke tid til å gjøre noe særlig med eksperimenter, men med det lille jeg gjorde så resultatene ut til å avsløre at Bubble Sort generelt er en ineffektiv algoritme, og at de andre algoritmene kan prestere bedre enn Merge Sort selv på store datasett hvis de er nesten-sorterte. Merge Sort så ut til å være «vinneren» på store datasett med vilkårlig ordning av tallene. Her er et forsøk å lage et diagram i Excel basert på tid for 'random_10000':



Siden det er 10 000 (ganske stort datasett) elementer, og de ikke er nesten-sorterte, ser Merge Sort ut til å gjøre det veldig bra, mens de andre får en mer og mer lineær tendens i tidsbruk med økt datamengde.