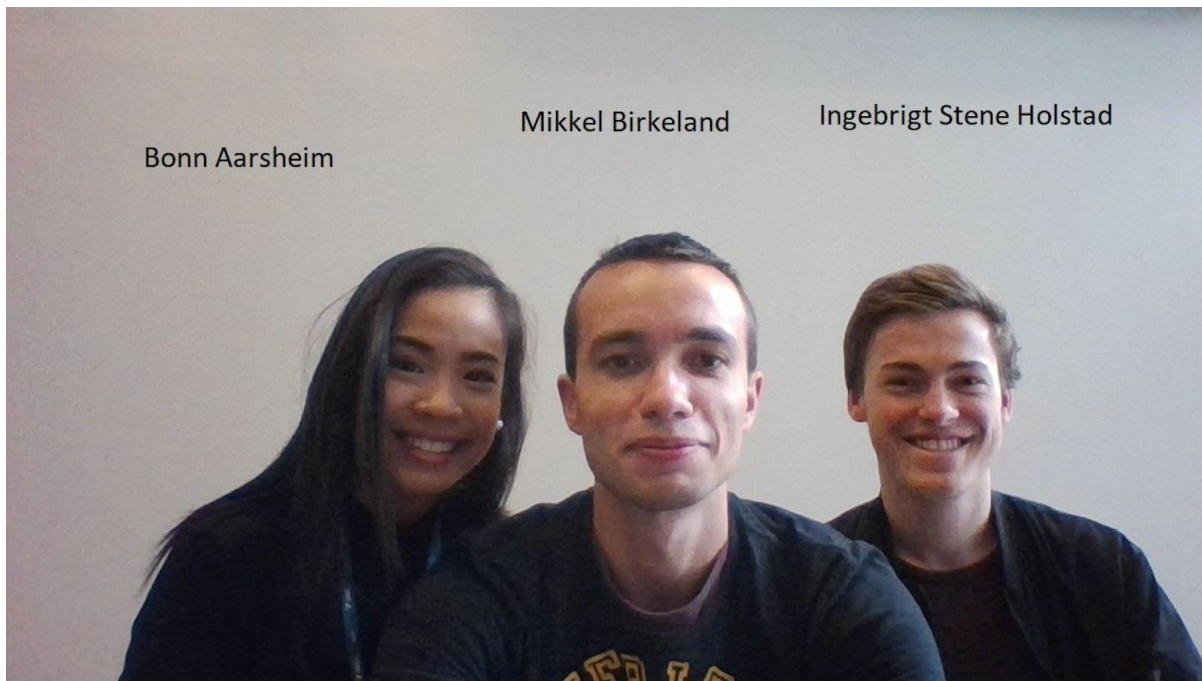


DAT102 ALGORITMER OG DATASTRUKTURER

Vår 2018 Oblig 3, øving 5



Oppgave 1)

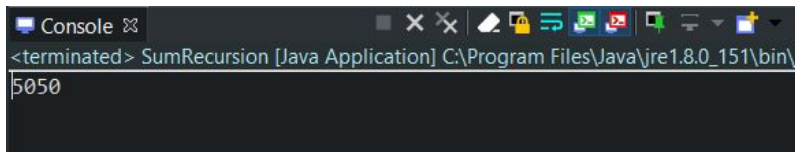
a)

```
package no.hvl.dat102;

public class SumRecursion {
    public static void main(String[] args) {
        System.out.println(sum(100));
    }

    public static int sum(int num){
        if(num == 1){
            return 1;
        }
        else {
            return sum(num - 1) + num;
        }
    }
}
```

Utskrift av kjøring:

A screenshot of a Java console window. The title bar says "Console". The text in the window is: "<terminated> SumRecursion [Java Application] C:\Program Files\Java\jre1.8.0_151\bin\j" on the first line, and "5050" on the second line.

```
<terminated> SumRecursion [Java Application] C:\Program Files\Java\jre1.8.0_151\bin\j
5050
```

b)

```
package no.hvl.dat102;

public class Oppgave1B {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            System.out.println("Ledd " + i);
            System.out.println(folge(i));
        }
    }

    public static int folge(int ledd) {
        if(ledd == 0) {
            return 2;
        }
        else if(ledd == 1) {
            return 5;
        }
        else {
            return (5*(folge(ledd-1)) - (6*(folge(ledd-2))) + 2);
        }
    }
}
```

Utskrift av kjøring:

```
<terminated>
Ledd 0
2
Ledd 1
5
Ledd 2
15
Ledd 3
47
Ledd 4
147
Ledd 5
455
Ledd 6
1395
Ledd 7
4247
Ledd 8
12867
Ledd 9
38855
```

c)

```
package no.hvl.dat102;

public class Hanoi {
    private int totalDisks;
    //Modifikasjon
    private long antall;

    public long getAntall() {
        return antall;
    }

    /**
     * Sets up the puzzle with the specified number of disks.
     *
     * @param disks
     *         the number of disks to start the towers puzzle with
     */
    public Hanoi(int disks) {
        totalDisks = disks;
    }

    /**
     * Performs the initial call to moveTower to solve the puzzle. Moves
```

```

the disks
    * from tower 1 to tower 3 using tower 2.
    */
public void solve() {
    moveTower(totalDisks, 1, 3, 2);
}

/**
 * Moves the specified number of disks from one tower to another by
moving a
 * subtower of n-1 disks out of the way, moving one disk, then moving
the
 * subtower back. Base case of 1 disk.
 *
 * @param numDisks
 *         the number of disks to move
 * @param start
 *         the starting tower
 * @param end
 *         the ending tower
 * @param temp
 *         the temporary tower
 */
private void moveTower(int numDisks, int start, int end, int temp) {
    if (numDisks == 1) {
        //moveOneDisk(start, end);
        antall++;
    }
    else {
        moveTower(numDisks - 1, start, temp, end);
        //moveOneDisk(start, end);
        antall++;
        moveTower(numDisks - 1, temp, end, start);
    }
}

/**
 * Prints instructions to move one disk from the specified start tower
to the
 * specified end tower.
 *
 * @param start

```

```

        *           the starting tower
        * @param end
        *           the ending tower
        */
    private void moveOneDisk(int start, int end) {
        System.out.print("");
        //System.out.print("Move one disk from " + start + " to " + end);
    }

    public static void main(String[] args) {
        //4 disk
        System.out.println("\nKjører med 4 disk");
        long tidStart = System.nanoTime();
        Hanoi towers = new Hanoi(4);
        towers.solve();
        long tidStop = System.nanoTime();
        System.out.println("Det tok " + (double)(tidStop-tidStart)/1000000 +
"ms");
        System.out.println("Antall flytt: " + towers.getAntall());

        //16 disk
        System.out.println("\nKjører med 16 disk");
        long tidStart2 = System.nanoTime();
        Hanoi towers2 = new Hanoi(16);
        towers2.solve();
        long tidStop2 = System.nanoTime();
        System.out.println("Det tok " + (double)(tidStop2-tidStart2)/1000000
+ "ms");
        System.out.println("Antall flytt: " + towers2.getAntall());

        //32 disk
        System.out.println("\nKjører med 32 disk");
        long tidStart3 = System.nanoTime();
        Hanoi towers3 = new Hanoi(32);
        towers3.solve();
        long tidStop3 = System.nanoTime();
        System.out.println("Det tok " + (double)(tidStop3-tidStart3)/1000000
+ "ms");
        System.out.println("Antall flytt: " + towers3.getAntall());
    }
}

```

Utskrift av kjøring:

```
Kjører med 4 disker  
Det tok 0.018773ms  
Antall flytt: 15  
  
Kjører med 16 disker  
Det tok 1.161814ms  
Antall flytt: 65535  
  
Kjører med 32 disker  
Det tok 14164.978844ms  
Antall flytt: 4294967295
```

iii)

For kjøringen med 16 disker fikk vi 1.161814ms og 65535 flytt.
Med 32 disker fikk vi 14164.97ms og 4294967295 flytt.

$$\frac{Tid_{32}}{Tid_{16}} = \frac{14164.97ms}{1.161814ms} = 12192.11 \quad \frac{2^{32}-1}{2^{16}-1} = 65537$$

Som en kan se, ble ikke de to sidene av ligningen spesielt like i vår situasjon.
Formelen for antall flytt derimot stemte perfekt.

Oppgave 2

a) Og b)

```
@Override  
public void leggTil(T el) {  
  
    // Setter inn ordnet før den noden p peker på  
    DobbeltNode<T> p;  
  
    if ((el.compareTo(foerste.getElement()) <= 0) ||  
        (el.compareTo(siste.getElement()) >= 0)) {  
        // Ugyldig. Alternativt kan vi ha det som et forkrav!  
        System.out.println("Ugyldig verdi. verdi > " +  
            foerste.getElement() + "verdi < " + siste.getElement());  
    } else { // Kun lovlige verdier
```

```

        antall++;

        if (el.compareTo(midten.getElement()) >= 0) { // Finn plass i
siste

            // halvdel
            p = midten.getNeste();
        } else { // Finn plass i første halvdel
            p = foerste.getNeste();
        }

        while (el.compareTo(p.getElement()) >= 0) {
            p = p.getNeste();
        } // while

        // Setter inn:
        // Innsett foran noden som p peker på

        DobbelNode<T> nyNode = new DobbelNode<T>(el);
        nyNode.setNeste(p);
        nyNode.setForrige(p.getForrige());
        p.getForrige().setNeste(nyNode);
        p.setForrige(nyNode);

        // Fyll ut med noen få setninger

        // Oppdaterer ny midten
        nyMidten();

    } // else lovlige

} //

```

```

@Override
public T fjern(T el) {
    T resultat = null;
    DobbelNode<T> p = null;

    if ((el.compareTo(foerste.getElement()) <= 0) ||
        (el.compareTo(siste.getElement()) >= 0)) {

```

```

        // Ugyldig. Alternativt kan vi ha det som et forkrav!
        System.out.println("Ugyldig verdi. verdi > " +
foerste.getElement() + "verdi < " + siste.getElement());

    } else { // Kun lovlige verdier

        p = finn(el);

        if (p != null) {
            // Tar ut
            antall = antall - 1;
            // Fyll ut med noen få setninger.

            p.getForrige().setNeste(p.getNeste());
            p.getNeste().setForrige(p.getForrige());

            // Oppadtere midten
            nyMidten();

            resultat = p.getElement();

        } // funnet
        else {
            System.out.println("Finner ikke verdien du vil fjerne");
        }

    } // lovlige
    return resultat;
} //

```


Utskrift av kjøring:

```
Opprinnelig liste
AAA a c e k m o s zzz Foerste:AAA Midten:k Siste:zzz

Elementet Kalle fins ikke

Fjerner a
AAA c e k m o s zzz Foerste:AAA Midten:m Siste:zzz

Fjerner m
AAA c e k o s zzz Foerste:AAA Midten:k Siste:zzz

Fjerner e
AAA c k o s zzz Foerste:AAA Midten:o Siste:zzz

Fjerner o
AAA c k s zzz Foerste:AAA Midten:k Siste:zzz
```

```
Fjerner k
AAA c s zzz Foerste:AAA Midten:s Siste:zzz

Fjerner c
AAA s zzz Foerste:AAA Midten:s Siste:zzz

Fjerner s
AAA zzz Foerste:AAA Midten:zzz Siste:zzz
```

c)

NyMidt() er av $O(n)$ siden den bruker av for-løkke som går fra første ledd i kjeden og til midten. I verste fall er midten av kjeden det siste ledd, og med n antall ledd, blir O -notasjonen $O(n)$.

For å gjøre metoden om til $O(1)$ må vi ta imot informasjon om det ble utført “fjern” eller “leggTil”. Om det nye antallet er partall gjør vi ingenting, midt kan bli stående. Ellers så sjekker vi om elementet i noden som ble lagt til eller ble fjernet av kom før eller etter midt-noden. Om det ble utført en fjerning, skal midt flyttes en node til venstre om noden som ble fjernet var på høyre side, og omvendt om noden var på venstre. Samme regel gjelder for “leggTil”, men alt blir reversert i forhold til når noder ble fjernet.

d)

- i) Uten midtpeker må vi gjennom alle noder, så n operasjoner.
- ii) Med midtpeker og søking som en veg vil vi sjekke $n/2$ noder. Om det bare er 2 noder vil vi sjekke n noder. Så dette blir mer effektivt når det er mange noder.
- iii) Når vi skal søke fra midten og mot start kan vi gå både fra starten og mot midten, samtidig som vi går fra midten mot starten. Da vil vi i snitt finne elementet vårt på $n/8$ trekk. Denne metoden fungerer for alle situasjonene i oppgaven.

Oppgave 3

a)

Binærsøking fungerer på enhver SORTERT tabell/liste.

Den fungerer ved at vi holder styr på 3 verdier: bunn, topp og midt.

Bunn er den første posisjonen i den "aktive" listen, og topp er siste posisjon i den "aktive" listen. Midt finner vi ved å ta $(\text{bunn} + \text{topp}) / 2$.

Algoritmen går ut på at vi går inn i midten av listen og sjekker om verdien som ligger på den posisjonen er større, mindre eller lik den verdien vi skal finne.

Er verdien vi finner i posisjonen midt STØRRE enn den verdien vi skal finne, da vet vi at vår verdi IKKE ligger på en senere posisjon i listen. Så vi oppdaterer topp posisjonen til å være den nåværende midt-posisjon.

Etter dette på det regnes ut en ny midt-posisjon ved hjelp av den nye toppverdien, og den gamle bunnverdien.

Er verdien vi finner i midt MINDRE enn den verdien vi søker, da oppdaterer vi bunn-posisjonen. Dette er på grunn av at vi vet verdien vår garantert IKKE ligger tidligere i listen.

Vi utfører denne algoritmen helt til vi enten finner verdien vår, eller $\text{topp} - \text{bunn} == 1$.

b)

Dette er vår tabell: 2 4 5 7 8 10 12 15 18 21 23 27 29 30 31

Bunn = 0, topp = 14 og midt = 7

Vi går inn i posisjon 7, her finner vi verdien 15. Denne er større enn 8, som vi søker. Dermed setter vi topp = 7 og regner ut ny midt = 4.

Deltabell: 2 4 5 7 8 10 12 15

Så går vi inn i posisjon 4, finner 8, som er lik vår verdi og søket avsluttes.

c)

Dette er vår tabell: 2 4 5 7 8 10 12 15 18 21 23 27 29 30 31

Bunn = 0, topp = 14 og midt = 7

Vi skal finne tallet 16.

Vi går inn i posisjon 7, finner verdien 15. 15 er mindre enn 16, så vi setter en ny bunn = 7 og ny midt = 11

Deltabell: 15 18 21 23 27 29 30 31

Vi går inn i posisjon 11, finner verdien 27. 27 er større enn 16, så vi setter ny topp = 11 og ny midt = 9

Deltabell: 15 18 21 23 27

Vi går inn i posisjon 9, finner verdien 21. 21 er større enn 16, så vi setter ny topp = 9 og ny midt = 8

Deltabell: 15 18 21

Vi går inn i posisjon 8, finner verdien 18. 18 er større enn 16, så vi setter ny topp = 8. Nå er bunn = 7 og topp = 8. Da har vi topp-bunn == 1. Vi kan da konkludere med at verdien 16 ikke finnes i listen.

Deltabell: 15 18

I dette tilfellet fikk vi 4 kall, eller vi brukte compareTo fire ganger. $\log_2 15 = 3.9$, så dette stemmer meget godt overens med at O-notasjonen til binær søk er $\log_2 n$

Oppgave 4

a)

```
Innsettingsmetoden med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 668.4ms for 32000 verdier  
Sorteringsmetode: Innsettingsmetoden  
  
Innsettingsmetoden med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 2495.4ms for 64000 verdier  
Sorteringsmetode: Innsettingsmetoden  
  
Innsettingsmetoden med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 8771.4ms for 128000 verdier  
Sorteringsmetode: Innsettingsmetoden  
  
Utvalgsmetoden med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 631.8ms for 32000 verdier  
Sorteringsmetode: Utvalgsmetoden  
  
Utvalgsmetoden med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 2668.8ms for 64000 verdier  
Sorteringsmetode: Utvalgsmetoden  
  
Utvalgsmetoden med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 11415.8ms for 128000 verdier  
Sorteringsmetode: Utvalgsmetoden  
  
Boblesortering med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 3411.4ms for 32000 verdier  
Sorteringsmetode: Boblesortering  
  
Boblesortering med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 15053.4ms for 64000 verdier  
Sorteringsmetode: Boblesortering  
  
Boblesortering med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 61776.8ms for 128000 verdier  
Sorteringsmetode: Boblesortering  
  
Quicksortering med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 3.8ms for 32000 verdier  
Sorteringsmetode: Quicksortering
```

```
Quicksortering med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 3.2ms for 64000 verdier  
Sorteringsmetode: Quicksortering  
  
Quicksortering med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 8.0ms for 128000 verdier  
Sorteringsmetode: Quicksortering  
  
Flettesortering med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 9.4ms for 32000 verdier  
Sorteringsmetode: Flettesortering  
  
Flettesortering med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 12.8ms for 64000 verdier  
Sorteringsmetode: Flettesortering  
  
Flettesortering med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 23.6ms for 128000 verdier  
Sorteringsmetode: Flettesortering  
  
Radixsortering med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 22.6ms for 32000 verdier  
Sorteringsmetode: Radixsortering  
  
Radixsortering med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 14.0ms for 64000 verdier  
Sorteringsmetode: Radixsortering  
  
Radixsortering med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 28.0ms for 128000 verdier  
Sorteringsmetode: Radixsortering
```

```

Hybrid Quicksort med 32000 elementer i listen.
Snitttid over 5 iterasjoner 6.8ms for 32000 verdier
Sorteringsmetode: Hybrid Quicksort

Hybrid Quicksort med 64000 elementer i listen.
Snitttid over 5 iterasjoner 4.4ms for 64000 verdier
Sorteringsmetode: Hybrid Quicksort

Hybrid Quicksort med 128000 elementer i listen.
Snitttid over 5 iterasjoner 8.2ms for 128000 verdier
Sorteringsmetode: Hybrid Quicksort

Quicksort som i forelesning: med 32000 elementer i listen.
Snitttid over 5 iterasjoner 35.8ms for 32000 verdier
Sorteringsmetode: Quicksort som i forelesning:

Quicksort som i forelesning: med 64000 elementer i listen.
Snitttid over 5 iterasjoner 142.2ms for 64000 verdier
Sorteringsmetode: Quicksort som i forelesning:

Quicksort som i forelesning: med 128000 elementer i listen.
Snitttid over 5 iterasjoner 742.0ms for 128000 verdier
Sorteringsmetode: Quicksort som i forelesning:

Hybrid Quicksort som i forelesning: med 32000 elementer i listen.
Snitttid over 5 iterasjoner 33.2ms for 32000 verdier
Sorteringsmetode: Hybrid Quicksort som i forelesning:

Hybrid Quicksort som i forelesning: med 64000 elementer i listen.
Snitttid over 5 iterasjoner 139.8ms for 64000 verdier
Sorteringsmetode: Hybrid Quicksort som i forelesning:

Hybrid Quicksort som i forelesning: med 128000 elementer i listen.
Snitttid over 5 iterasjoner 740.4ms for 128000 verdier
Sorteringsmetode: Hybrid Quicksort som i forelesning:

```

Vi bruker $n = 32\,000$ for å regne ut konstanten c for hver metode.

Innsettingsmetoden

n	Antall målinger	Målt til (gjennomsnitt)	Teoretisk tid $c \cdot f(n)$
32000	5	667ms	667ms
64000	5	2495ms	2671ms
128000	5	8771ms	10 787ms

Uvalgsmetoden

n	Antall målinger	Målt til (gjennomsnitt)	Teoretisk tid $c \cdot f(n)$
32000	5	631ms	631ms
64000	5	2668ms	2523ms
128000	5	11415ms	10092ms

Boblesortering

n	Antall målinger	Målt til (gjennomsnitt)	Teoretisk tid $c \cdot f(n)$
32000	5	3411ms	3411ms
64000	5	15 053ms	13 639ms
128000	5	61 776ms	54 558ms

Quicksort

n	Antall målinger	Målt til (gjennomsnitt)	Teoretisk tid $c \cdot f(n)$
32000	5	3.8ms	3.8ms
64000	5	3.2ms	8.1ms
128000	5	8.0ms	17.2ms

Flettesortering

n	Antall målinger	Målt til (gjennomsnitt)	Teoretisk tid $c \cdot f(n)$
32000	5	9.4ms	9.4ms
64000	5	12.8ms	20ms
128000	5	23.6ms	42.6ms

Radix

n	Antall målinger	Målt til (gjennomsnitt)	Teoretisk tid $c \cdot f(n)$
32000	5	22.6ms	22.6ms
64000	5	14ms	45ms
128000	5	28ms	90.4ms

b)

I våre metoder blir quicksort en del raskere. Vi har implementert 2 forskjellige variasjoner av quicksort. Den ene er mye raskere enn den andre i tilfellene vi har testet den på. Fra skjermbildene kan vi se at metoden “quicksort fra forelesning” er mye tregere enn den andre. Den fra forelesningsnotatene velger pivot på en annen måte enn den fra nettet, og ender opp med å bli mye tregere i våre tester. Quicksorten fra forelesningsnotatene ender opp med å være en del tregere enn flettesortering.

c)

Radix-sortering er en type sortering som baserer seg på strukturen til nøklene. Dersom vi skulle sortere tall fra 0-9, så hadde radixen vært 10, og vi vil få 10 forskjellige køer, en for hver mulighet. I radix-sortering vil hvert tall bli undersøkt (fra høyre til venstre) og bli lagt inn i en kø etter verdien til det tallet vi ser på. Når alle verdiene har blitt lagt inn i riktig kø, så tømmes køen en etter en og det samme blir gjentatt, bare at vi nå ser på neste verdi i tallet(fremdeles fra høyre til venstre). Til slutt vil vi få ut en liste med alle tallene sortert.

Figur som viser hvordan nøklene blir fordelt i køer for de tre fasene. Starter med å sjekke det bakerste tallet og legger inn i riktig kø, det samme med det midterste tallet, så det første tallet. I radix-sortering sjekker vi alltid fra høyre til venstre.

d)

```
Quicksort som i forelesning: med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 43.2ms for 32000 verdier  
Sorteringsmetode: Quicksort som i forelesning:  
  
Quicksort som i forelesning: med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 142.6ms for 64000 verdier  
Sorteringsmetode: Quicksort som i forelesning:  
  
Quicksort som i forelesning: med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 778.0ms for 128000 verdier  
Sorteringsmetode: Quicksort som i forelesning:  
  
Hybrid Quicksort som i forelesning: med 32000 elementer i listen.  
Snitttid over 5 iterasjoner 66.2ms for 32000 verdier  
Sorteringsmetode: Hybrid Quicksort som i forelesning:  
  
Hybrid Quicksort som i forelesning: med 64000 elementer i listen.  
Snitttid over 5 iterasjoner 59.8ms for 64000 verdier  
Sorteringsmetode: Hybrid Quicksort som i forelesning:  
  
Hybrid Quicksort som i forelesning: med 128000 elementer i listen.  
Snitttid over 5 iterasjoner 6.6ms for 128000 verdier  
Sorteringsmetode: Hybrid Quicksort som i forelesning:
```

Her ser vi resultatet fra kjøringen når MIN = 8000. Da blir en ganske liten del kjørt som quicksort før den går over på innsetting. Dette er noe av den største forbedringen vi har sett. Med MIN verdi på rundt 10, ser vi ingen forskjell i tiden mellom vanlig quicksort, og hybrid quicksort.

Her er koden for Hybrid-metoden:

```
public class QuickSortNyDAT102 implements Sortering {  
    private static final int MIN = 8000;  
    private Integer[] sortert;  
  
    public void sorter(Integer[] liste) {  
        sortert = liste;  
        sort();  
    }  
  
    public void sort() {  
        kvikkSort(0, sortert.length - 1);  
        sorteringVedInnsetting2(0, sortert.length-1);  
    }  
  
    public void sorteringVedInnsetting2(int forste, int siste) {  
        for (int indeks = forste + 1; indeks <= siste; indeks++) {  
  
            Integer nokkel = sortert[indeks];  
            int p = indeks;  
            // Forskyv større verdier mot høyre  
            while (p > 0 && sortert[p - 1].compareTo(nokkel) > 0) {
```

```

        sortert[p] = sortert[p - 1];
        p--;
    }
    sortert[p] = nokkel;
} // ytre
}

public void kvikkSort(int min, int maks) {
    int posPartisjon;
    if (maks - min + 1 > MIN) { // minst to element
        /** Lager partisjon */
        posPartisjon = finnPartisjon(min, maks);
        /** Sorterer venstre side */
        kvikkSort(min, posPartisjon-1);
        /** Sorterer høyre side */
        kvikkSort(posPartisjon + 1, maks);
    }
}

public int finnPartisjon(int min, int maks) {
    int venstre, hoyre;
    Integer temp, pivot;
    pivot = sortert[min];
    venstre = min;
    hoyre = maks;
    while(venstre < hoyre) {
        while(venstre < hoyre && sortert[venstre].compareTo(pivot)<=0)
        {
            venstre++;
        }
        while(sortert[hoyre].compareTo(pivot)>0) {
            hoyre--;
        }
        if(venstre < hoyre) {
            temp = sortert[venstre];
            sortert[venstre] = sortert[hoyre];
            sortert[hoyre] = temp;
        }
    }
    temp = sortert[min];
    sortert[min] = sortert[hoyre];
    sortert[hoyre] = temp;
}

```



```
    return hoyre;  
}  
  
}
```