Øving 1 Algoritmer og datastrukturer Jacob Lein

```java
public static class SearchAlg {

    // Returns array where first element is the best profit, second element is the
    // buy-date and third element is the sell-date
    public static int[] findBestProfit(int[] pricePerDayList) {
        int startDate = 0;
        int endDate = 0;
        int maxProfit = Integer.MIN_VALUE;

        for (int i = 0; i < pricePerDayList.length - 1; i++) {
            for (int j = i + 1; j < pricePerDayList.length; j++) {
                int profit = sum(pricePerDayList, i, j);

                if (profit > maxProfit) {
                    maxProfit = profit;
                    startDate = i + 1;
                    endDate = j + 1;
                }
            }
        }
        int[] returnArray = { maxProfit, startDate, endDate };
        return returnArray;
    }
```

```java
// Returns the sum of the elements in the array from start to end
private static int sum(int[] arr, int start, int end) {
    int sum = 0;
    for (int i = start + 1; i <= end; i++) {
        sum += arr[i];
    }
    return sum;
}
```

1.1 Algoritmen søker etter beste mulige profitt for hver kjøpsdag, der hvis profitten er høyere enn tidligere profitter (maxProfit) så lagres datoene for dette kjøpet. Koden regner med at kjøp og salg blir gjort på slutten av dager. Altså at hvis beste kjøps- og salgdato er fra dag 5 til 7 så tilsier det at man kjøpte etter kursendring på kjøpsdato og solgte etter kursendring på salgsdato.

1.2 Tidskompleksiteten til algoritmen vil være O(n^3) siden findBestProfit() metoden inneholder to for-løkker med n-antall iterasjoner, der innerste løkke kaller på metoden sum() som igjen inneholder en løkke med n-antall iterasjoner. De røde pilene i bildet

peker til løkkene, mens metode-kallet er understreket. Det finnes nok algoritmer med bedre effektivitet.

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Table table = new Table(new int[2]);

        int[] testColumnAmount = { 1000, 1500, 2000, 3000, 4000, 6000, 8000, 10000 };
        for (int i = 0; i < testColumnAmount.length; i++) {
            Table newTable = table.generateTable(testColumnAmount[i]);

            long startTime = System.currentTimeMillis();
            int[] result = SearchAlg.findBestProfit(newTable.table);
            long elapsedTime = System.currentTimeMillis() - startTime;

            System.out.println(x:"\n");
            System.out.println("Columns: " + testColumnAmount[i] + " Time: " + elapsedTime + "ms ");
            System.out.println("Best profit: " + result[0] + " from day: " + result[1] + " to day: " + result[2]);
        }
        int[] testArray = { -1, 3, -9, 2, 2, -1, 2, -1, -5 };
        int[] exampleResult = SearchAlg.findBestProfit(testArray);
        System.out.println(x:"\n");
        System.out.println(x:"Example array: ");
        System.out.println(
            "Best profit: " + exampleResult[0] + " from day: " + exampleResult[1] + " to day: " + exampleResult[2]);

    }
```

1.3 Her vises det som kjøres i rekkefølge. Tiden startes rett før algoritmen finner beste kjøp og salg og alt printes etter det igjen. Til høyre er utskriften ved en kjøring. Siden tidskompleksiteten kan antas å kunne beskrives ved O(n^3) så kan man dobbeltsjekke dette med utskriften. 4000 dager tar 2980ms mens dobbelt mengde, 8000 dager, tar 24300ms.

```
Columns: 1000 Time: 95ms
Best profit: 325 from day: 123 to day: 911


Columns: 1500 Time: 337ms
Best profit: 586 from day: 272 to day: 1478


Columns: 2000 Time: 357ms
Best profit: 498 from day: 1184 to day: 1992


Columns: 3000 Time: 1281ms
Best profit: 299 from day: 2274 to day: 2774


Columns: 4000 Time: 2980ms
Best profit: 470 from day: 3062 to day: 3574


Columns: 6000 Time: 10194ms
Best profit: 196 from day: 2518 to day: 2824


Columns: 8000 Time: 24300ms
Best profit: 277 from day: 143 to day: 585


Columns: 10000 Time: 48028ms
Best profit: 941 from day: 3644 to day: 8645


Example array:
Best profit: 5 from day: 3 to day: 7
jacoblein@dhcp-10-22-107-22 Øvinger %
```

$$\frac{24300\ ms}{2980\ ms} = 8.15$$

Dette forholdet blir omtrent det samme som 8, som er det forventes ved dobling av data til en algoritme med kubisk kompleksitet. Hvis man øker datamengden med 2.5 altså til 10 000 dager, får man en responstid på 48028ms.

$$\frac{48028\ ms}{2980\ ms} = 16.1 \qquad 2.5^3 = 15.6$$

Selv om forholdene vist over har en differanse på 0.5 så stemmer det omtrentlig som forventet i forhold til O(n^3). Vi kan til slutt også se på doblingen av data mellom 3000 og 6000 dager. ($\frac{10194\ ms}{1281\ ms} = 7.95$) Tidsmålingene ser ut til å stemme i forhold til analysen.