Rapport arbeidskrav 1 Algoritmer og Datastrukturer

Tor-Øyvind Paulsrud Bakken

Denne rapporten tar for seg oppg 1-1, 1-2 og 1-3 i boken Algoritmer og datastrukturer (Hafting og Ljosland 2003). Oppgaven handler om implementering og analysering av en algoritme for best mulig fortjeneste på en aksje. Input som blir matet inn i algoritmen er en tallrekke som angir forandring i kurs fra dag til dag.

1-1 Utsnitt av koden:

Main-metoden:

```java
public static void main(String[] args) {
    //int[] change = {-1, 0, -9, -1, 2, -1, 2, -1, 5};
    //Lager en array med tilfeldige tall mellom -10 og 10 som representerer endringen i aksjekurs fra dag til dag.
    //nSize bestemmer størrelsen på arrayen.
    //Bruker den innebygde Random-funksjonen til Java, men denne gir problemer med større nSize. Algoritmen jeg
    // bruker er likevel såpass treig at jeg ikke kommer opp i store nok n-verdier til at det gir for ulogiske svar.
    Random rand = new Random();
    int nSize = 10000;
    System.out.println("N-size: " + nSize + "\n");
    int [] change = new int[nSize];
    for (int i = 0; i < nSize; i++) {
        change[i] = (rand.nextInt( bound: 21) - 10);
    }

    Date start = new Date();
    System.out.println(stockCheck(change));
    Date slutt = new Date();
    System.out.println("Runtime in milliseconds: " + (slutt.getTime() - start.getTime()));
}
```

Lager en array med tilfeldige tall mellom -10 og 10 som representerer endringen i aksjekurs fra dag til dag.
nSize bestemmer størrelsen på arrayen.
Bruker den innebygde Random-funksjonen til Java, men denne gir problemer med større nSize. Derfor måtte jeg legge til at bound går opp til 21, fordi det gav av uviss grunn bedre normalfordeling av positive og negative verdier. Med bound 20 så fikk jeg flere negative enn positive verdier som gjorde at man mot slutten satt med en ekstremt negativ verdi på aksjen. Algoritmen jeg bruker er likevel såpass treig at jeg ikke kommer opp i store nok n-verdier til at det gir for ulogiske svar. Oppgaven er uansett ikke en analyse av Random-funksjonen og i den virkelige verden hadde man hadde bedre verdier for endring fra dag til dag.

stockCheck-metoden:

```
public static String stockCheck(int[] priceChange) {
    int bestBuyDay = 0;
    int bestSellDay = 0;
    int price = 0;
    int priceCheck = 0;
    int bestSum = 0;
    int sum;


    for (int i = 0; i < priceChange.length; i++) { //O(n)
        price += priceChange[i];
        for (int j = i; j < (priceChange.length); j++) { //O(n)
            priceCheck += priceChange[j];
            sum = priceCheck - price;
            if (sum > bestSum) {
                bestSum = sum;
                bestBuyDay = i + 1;
                bestSellDay = j + 1;
            }
        }
        priceCheck = price;
    }
    return "Best buy day: " + bestBuyDay + " \nBest sell day: " + bestSellDay +
            "\nThat gives a profit of: " + bestSum;

}
```

I denne metoden er det en rekke tilordninger, men disse er ikke like interessante for kjøretiden som for-løkkene. I den ytterste for-løkken legger jeg bare til endringen i prisen som jeg får fra tabellen som er sendt til metoden. Deretter looper jeg gjennom hele listen med priser og sjekker opp mot dagens pris for å se hvilken dag som gir best profitt. En hvis-sjekk sjekker om dette gir mer profitt enn den beste profitten som er lagret. Er så tilfellet lagres den nye dagen som den beste salgsdagen, hvis ikke fortsetter løkken gjennom alle verdiene. Når den innerste løkken har kjørt gjennom en hel gang settes sammenligningsprisen tilbake til dagens pris før den ytterste løkken tar neste runde og hele den innerste løkken kjører på nytt med neste verdi.

1-2 Analyse av algoritmen

Totalt for metoden (stockCheck) blir det O(n^2) siden det er for-løkke i en for-løkke (O(n*n)). Har 10 deklarasjoner og én sjekk O(11n), men disse har forsvinnende liten påvirkning når n går mot uendelig. Det vil si at algoritmen i teorien, i verste fall, får 100 ganger økning i tidsbruk når n ganges med 10.

1-3 Tidsmåling

```
Date start = new Date();
System.out.println(stockCheck(change));
Date slutt = new Date();
System.out.println("Runtime in milliseconds: " + (slutt.getTime() - start.getTime()));
```

Implementerer en enkel tidsmåling der jeg sjekker tiden før og etter jeg kjører metoden. Printer ut differansen mellom start og slutt.

Fra mine beregninger burde en ti-dobling av n gi ca en 100-dobling av tidsbruken. Metoden jeg bruker for å ta tiden er svært simpel, noe som fører til at java ikke greier å måle forskjeller på tidsbruk for små n(0-1000). Så den første jeg begynner på er 1000, deretter øker jeg med ti opp til 1mill. Ut fra disse resultatene så jeg at å øke det med enda en 10-gang til vil føre til at kjøretiden nærmer seg flere hundre dager, så jeg stoppet på 1mill.

Under ser man at fra 1000 til 10 000 så er det ikke helt rett ift analysen i oppg 1-2, men fra 10 000 og oppover følger nokså rett at ganger jeg n-størrelsen med 10 så får jeg en økning på tiden på 100. (nederste tallet i skjermdumpen er tidsbruk i millisekunder).

```
N-size: 1000
Best buy day: 232
Best sell day: 951
That gives a profit of: 372
Runtime in milliseconds: 11
```

```
N-size: 10000
Best buy day: 5295
Best sell day: 8119
That gives a profit of: 439
Runtime in milliseconds: 63
```

```
N-size: 100000
Best buy day: 56044
Best sell day: 68770
That gives a profit of: 1352
Runtime in milliseconds: 5721
```

```
N-size: 1000000
Best buy day: 774984
Best sell day: 935665
That gives a profit of: 6304
Runtime in milliseconds: 461038
```