Algoritmer og datastruktur - Arbeidskrav 01 - Analyse

Gruppe: John Ivar Eriksen, Emil Slettbakk

Mulige array-typer

int-array (int[]) vs. ArrayList: I denne løsningen falt valget på et enkelt `int[] array`, i stedet for de mer teknisk avanserte alternativene, som f.eks ArrayList.

Et `int[] array` er bedre egnet til denne type oppgave, siden vi har et array med en gitt lengde, og ønsker å ha en prosess som går så rasks som mulig. Et `int[] array` raskere enn `ArrayList`, blant annet fordi `ArrayList` har dynamisk størrelse hvilket krever ytterligere minne for å holde på strukturen. Som følge av at det er dynamisk, lagres elementene forskjellige steder i minnet, siden den ikke kan forhåndsreservere plasser til elementer som ennå ikke er opprettet. Arrayet må dermed utvides, og minnelokasjoner reserveres, etterhvert som den får nye instrukser Et enkelt `int[] array` er "fixed size" og har alle plassene/elementene på rekke og rad i minnet.

Analyse av valgt løsning

Den første/øverste løkken er lineær, altså $O(n)$, fordi den har en tidskompleksitet som er direkte proporsjonal med mengden elementer. Operasjonen er den samme, bare at den må utføres på færre eller flere elementer.

```
    int[] runningSumArray = new
int[stockTickerArray.length];
    int runningSum = 0;

    // Running sum array for stock value
    for (int s = 0; s < stockTickerArray.length; s++) {
      runningSum = runningSum + stockTickerArray[s];
      runningSumArray[s] = runningSum;
    }
```

Hvis vi ser på den neste løkken av koden, som utgjør selve algoritmen for å beregne beste kjøp og salg, er det to `for`-løkker, i en såkalt "nested loop".

```
// Compare values and find best profit
  for (int i = 0; i < runningSumArray.length - 1; i++)
{
    for (int j = i + 1; j < runningSumArray.length;
j++) {
      potentialProfit = runningSumArray[j] -
runningSumArray[i];
      if (potentialProfit > maxProfit) {
        maxProfit = potentialProfit;
        buyDay = i + 1; // +1 to convert index to day.
        sellDay = j + 1; // +1 to convert index to day.
      }
    }
  }
```

- Den ytterste løkken må kjøre n ganger, der $n = runningSumArray.length - 1$.
- Den indre løkken må kjøre $n - i$ ganger, hvor i er den gjeldende iterasjonen av den ytre løkken.

Vi får altså at den indre løkkens iterasjoner er avhengig av den inneværende iterasjonen i den ytre løkken. Når i er liten, blir $n - i$, altså den indre løkken er stor. Og etterhvert som `i` øker, blir iterasjonene til den indre løkken færre.

For å finne O-notasjonen for kompleksitet, tar vi utgangspunkt i "worst case scenario", dvs at algoritmen må kjøre gjennom "alle" iterasjonene for å finne de "riktige" tallene. Da må den innerste løkka, `j` kjøre gjennom alle `n` elementer i arrayet for hver inkrementering av i. Med n elementer i arrayet, vil algoritmen måtte kjøre $n * n$ ganger. Størrelsen/lengden av den inneste løkken avhengig av hvor i arrayet den ytterste løkken er. Vi får følgende tilnærming av iterasjonsmengden:

$$n + (n-1) + (n-2) + \ldots + 2 + 1 \approx \frac{n^2}{2}$$

Siden vi ser bort fra lavere ordens faktorer i O-notasjon, står vi igjen med $O(n^2)$.

Vi kan kjøre en timing test på algoritmen ved å lage et nytt array som inneholder tilfeldige tall, og kjører dette i en løkke for å gjøre flere timinger på rad. Tidsforbruket summeres og deles til slutt på antall runder kjørt.

Resultatet er, i mitt tilfelle, en tilnærmet kvadrering av tidsforbruk. Ved å kjøre opprettelse av et nytt array i en løkke som kjører 100 ganger, med 4000 og så doblet til 8000 elementer, observeres en økning fra 3-5 millisekunder, til omkring 13-15 ms.

Elements in array	**Number of runs**	**Avg. time, millisec.**	**Avg. time, nanosec.**	**Avg. of avg.**
4 000	100	3,9892	3 989 158	
4 000	100	4,3110	4 311 004	
4 000	100	3,5917	3 591 671	
4 000	100	4,7011	4 701 066	**4.15**

Elements in array	Number of runs	Avg. time, millisec.	Avg. time, nanosec.	Avg. of avg.
8 000	100	13,5244	13 524 391	
8 000	100	14,3092	14 309 175	
8 000	100	14,1843	14 184 253	
8 000	100	13,5860	13 586 029	**13.90**