PG 4200: Algorithms and Data Structures

28.03.2025 - 25.04.2025

Innholdsfortegnelse

Innledning

Formålet med denne rapporten er å utforske og analysere fire ulike sorteringsalgoritmer: BubbleSort, InsertionSort, MergeSort og QuickSort, ved hjelp av et datasett bestående av byer med unike breddegrader. Vi har undersøkt hvordan disse algoritmene oppfører seg i praksis, både med og uten tilfeldig stokking av data, og vurdert deres teoretiske og observerte tidskompleksitet.

Rapporten er strukturert etter fire hovedproblemer som dekker hver av algoritmene:

1. **BubbleSort**: Vi implementerte både en optimalisert og en ikke-optimalisert versjon, og sammenlignet kjøretid og effektivitet før og etter tilfeldig stokking.
2. **InsertionSort**: Vi undersøkte hvordan algoritmen presterer med allerede sortert og tilfeldig sortert data, og vurderte hvordan dette påvirker den faktiske kjøretiden og antall operasjoner.
3. **MergeSort**: Vi implementerte en rekursiv versjon av algoritmen og målte antall merge-operasjoner som utføres under sortering, samt vurderte hvorvidt dette påvirkes av rekkefølgen på input.
4. **QuickSort**: Vi testet tre ulike pivot-strategier: første element, siste element og tilfeldig valgt, og sammenlignet antall sammenligninger for hver strategi, med mål om å finne den mest effektive for vårt datasett.

Csv filen som holder informasjon om de forskjellige byene parser vi med et program som gjør om innholdet i filen til brukbare objekter i Java. For å se tiden algoritmene bruker på å sortere har vi laget et tidtakerprogram som starter før algoritmene kjører slik at vi kan se differansen mellom start og sluttiden, for å videre kunne utregne hvilke algoritme som egner seg best til de forskjellige brukerområdene.

Gjennom oppgavene har vi fått innsikt i hvordan algoritmenes ytelse varierer i praksis, og hvilke faktorer som spiller størst rolle når man skal velge sorteringsmetode i ulike scenarier.

Problem 1. BubbleSort – Tidskompleksitet

Introduksjon

BubbleSort fungerer ved at den går gjennom listen og sjekker om forrige element er større enn nåværende element og bytter om rekkefølgen på dem dersom dette er tilfellet. Dette gjentas så lenge et bytte av elementer kan skje. Listen er sortert etter siste gjennomgang der ingen bytter skjer.

Implementering og relevans

I dette problemet har vi implementert både en optimalisert og en ikke-optimalisert versjon av BubbleSort for å sortere en liste med breddegrader tilknyttet ulike byer. Sortering av slike data er nyttig i situasjoner hvor man ønsker å visualisere byene geografisk fra sør til nord, eller analysere mønstre basert på geografisk beliggenhet. Koden under viser den optimaliserte versjonen:

```
void sort(City[] cities) {
    boolean isSorted = false;

    while (!isSorted) {
        isSorted = true;
        for (int i = 1; i < cities.length; i++) {
            if (cities[i - 1].latitude < cities[i].latitude) {
                City temp = cities[i - 1];
                cities[i - 1] = cities[i];
                cities[i] = temp;
                isSorted = false;
            }
        }
    }
}
```

Denne versjonen sorterer byene etter synkende breddegrad, altså fra nord til sør, fordi sammenligningen bruker <. Hvis variabelen isSorted forblir true gjennom en hel iterasjon,

betyr det at ingen bytter ble gjort og listen er allerede sortert. Dette gjør at algoritmen kan avslutte tidlig, noe som gir best case kjøretid på O(n).

Den optimaliserte varianten gir betydelig bedre ytelse når datasettet allerede er sortert eller nesten sortert, sammenlignet med den ikke-optimaliserte versjonen som alltid gjennomfører n^2 sammenligninger.

Selv om BubbleSort generelt er ineffektiv sammenlignet med algoritmer som QuickSort og MergeSort, har den likevel relevans i spesifikke scenarier. Algoritmen er svært enkel å implementere og forstå, noe som gjør den velegnet i undervisningssammenheng. Den kan også være tilstrekkelig for små datasett, eller i systemer hvor data kontinuerlig oppdateres og derfor er nesten sortert som f.eks. GPS-logger.

Tidskompleksitet

BubbleSort har en best-case kjøretid på O(n) dersom listen allerede er sortert. Dette skyldes at den optimaliserte varianten avslutter tidlig etter en gjennomgang, så lenge ingen bytter forekommer. I slike tilfeller trengs kun en passering for å bekrefte at listen er i korrekt rekkefølge.

I både gjennomsnittlige og verstefallsscenarioer er tidskompleksiteten $O(n^2)$. Dette skyldes at algoritmen må gjøre gjentatte passeringer hvor flere elementer må flyttes – spesielt når listen er i omvendt rekkefølge.

Vi testet begge versjoner av algoritmen på vårt datasett med bybreddegrader:

- Den optimaliserte versjonen brukte 20 001 ms,
- Den ikke-optimaliserte versjonen brukte 49 618 ms.

Dette demonstrerer hvordan den optimaliserte varianten kan redusere kjøretiden betraktelig, men samtidig at BubbleSort fremdeles er lite effektiv på større, usorterte datasett.

Tilfeldig stokking av listen påvirker ikke den teoretiske kompleksiteten, som fortsatt er $O(n^2)$. Dette fordi sannsynligheten for at dataene er "nesten sortert" etter tilfeldig stokking er lav, og algoritmen må derfor fortsatt gjennomføre mange sammenligninger og bytter (B. Marculescu, *Runtime Analysis and Sorting*, u.å.).

Tidskompleksitetens karakteristikker

Når det gjelder tidskompleksitet, har BubbleSort følgende karakteristikker:

Best case (allerede sortert): $O(n)$. Den optimaliserte versjonen avslutter etter en passering dersom ingen bytter skjer. Dette skjer fordi algoritmen bruker en kontrollvariabel som stopper videre iterasjoner når listen er i orden.

Gjennomsnittlig tilfelle: $O(n^2)$. I gjennomsnitt må algoritmen foreta flere passeringer og mange bytter, fordi elementene i en tilfeldig sortert liste er plassert i vilkårlig rekkefølge.

Verstefall (omvendt sortert): $O(n^2)$.I dette tilfellet må hvert element flyttes hele veien gjennom listen, noe som gir maksimal arbeidsmengde for algoritmen.

Det å stokke listen tilfeldig før sortering endrer ikke den teoretiske tidskompleksiteten. Den forblir $O(n^2)$ fordi algoritmen i snitt fortsatt må gjennomføre mange bytter og sammenligninger. En tilfeldig stokket liste har ikke høyere sannsynlighet for å være "nær sortert" enn hvilken som helst annen tilfeldig rekkefølge (B. Marculescu, *Runtime Analysis and Sorting*, u.å).

Kompleksitetsklasser

BubbleSort er en algoritme som løses i polynomisk tid, og plasseres dermed i kompleksitetsklassen P. Dette betyr at både løsningen og verifikasjonen av algoritmen kan utføres i tid som er polynomisk i forhold til inputstørrelsen, altså $O(n^k)$ for en konstant k(M. A. Dahl, LO 6 (P1))

Problem 2. InsertionSort – Tidskompleksitet

Introduksjon

InsertionSort er en sorteringsalgoritme som bygger opp en sortert del av listen element for element. I hver iterasjon tar algoritmen det neste elementet fra den usorterte delen og plasserer det på riktig sted i den sorterte delen ved å sammenligne seg bakover. Denne prosessen gjentas til alle elementer er sortert (Wikipedia, *Insertion sort*, u.å).

Tidskompleksitet

InsertionSort har varierende tidskompleksitet, avhengig av rekkefølgen på input:

> *Best case (allerede sortert):* O(n).
> Ved sortert eller nesten-sortert input vil den indre løkken brytes tidlig, og hvert element plasseres direkte på riktig plass uten forskyvninger.
>
> *Gjennomsnittlig og verstefall:* $O(n^2)$.
> I slike tilfeller må mange elementer flyttes bakover i listen, og algoritmen gjennomfører et høyt antall sammenligninger og bytter (Sedgewick & Wayne, 2011, s. 250).

I vårt tilfelle brukte algoritmen 3 670 ms på en tilfeldig stokket liste med by-breddegrader. Dette reflekterer at algoritmen i praksis ikke drar nytte av noen forhåndssortering, og opererer nær sin teoretiske gjennomsnittlige ytelse.

Implementering

Under vises vår implementasjon av InsertionSort, brukt for å sortere en liste med City-objekter basert på breddegrad:

```
void sort(City[] cities) {
   int pivot = 1;
   for (int i = pivot; i < cities.length; i++) {
```

```
        for (int x = pivot; x > 0; x--) {
            if (cities[x].latitude < cities[x - 1].latitude) {
                City temp = cities[x];
                cities[x] = cities[x - 1];
                cities[x - 1] = temp;
            } else {
                break;
            }
        }
        pivot++;
    }
}
```

Denne implementasjonen sorterer fra nord til sør (synkende breddegrad), siden den bruker < i sammenligningen. Den indre løkken sammenligner hvert element med det forrige og flytter det bakover til det finner sin riktige posisjon. Dersom et element er større enn det foregående, avsluttes løkken med break, noe som gir algoritmen dens effektive ytelse i best case.

Analyse

Tidskompleksiteten forblir $O(n^2)$ i både gjennomsnitt og verstefall fordi hvert nytt element potensielt må sammenlignes med alle tidligere elementer og flyttes gjennom hele listen. Likevel er InsertionSort svært effektiv ved allerede sorterte eller nesten-sorterte datasett, hvor den kun trenger én sammenligning per element.

Effekt av tilfeldig stokking

Dersom listen stokkes tilfeldig før sortering, reduseres sannsynligheten for at dataene er nær sortert. Dermed må algoritmen i større grad bruke den indre løkken fullt ut, og kjøretiden nærmer seg verstefall $O(n^2)$. Selv om stokking ikke endrer algoritmens teoretiske kompleksitet, medfører det at algoritmen i praksis oppnår svakere ytelse særlig på større datasett.

Problem 3. MergeSort – Antall merges

Introduksjon

MergeSort er en klassisk divide-and-conquer algoritme som sorterer en liste ved å dele den rekursivt i mindre deler, for deretter å slå sammen delene i sortert rekkefølge. Hver deling og sammenslåing utgjør en "merge"-operasjon.

Et viktig kjennetegn ved MergeSort er at antall merges kun avhenger av datasettets størrelse, ikke av rekkefølgen på elementene. Dette gjør algoritmen forutsigbar og stabil, uavhengig av hvordan dataene er sortert på forhånd (W3Schools, u.å.).

Implementering

Under er en forenklet implementasjon brukt i vår løsning. Den sorterer byer etter breddegrad:

```
void mergeSort {
   int merges = 0;

   void sort(City[] cities) {

       if (cities.length > 1) {

           int mid = cities.length / 2;

           City[] left = Arrays.copyOfRange(cities, 0, mid);
           City[] right = Arrays.copyOfRange(cities, mid, cities.length);

           sort(left);
           sort(right);

           merge(cities, left, right);
       }
    }
}
```

Denne implementasjonen teller antall merge-operasjoner, altså hver sammenligning mellom to lister.

Analyse

Ved kjøring på datasettet vårt (med unike breddegrader for byer), brukte algoritmen kun 19 ms og gjennomførte 47 867 merges. Dette samsvarer godt med den teoretiske tidskompleksiteten på O(n log n), hvor n er antall elementer og log n representerer dybden i den rekursive delingsstrukturen.

Uansett rekkefølge på dataene vil MergeSort alltid utføre omtrent samme antall merges, fordi:

- Algoritmen deler listen i to like store halvdeler hver gang.
- Antall nivåer i delingsprosessen er $\log_2(n)$.
- På hvert nivå gjøres en full gjennomgang (O(n)) for å slå sammen delene.

Dette gjør at MergeSort har svært forutsigbar ytelse, i motsetning til enklere algoritmer som BubbleSort og InsertionSort, hvor rekkefølgen på input kan gi store variasjoner i kjøretid.

Effekt av tilfeldig stokking

Å stokke listen tilfeldig før sortering påvirker ikke ytelsen til MergeSort nevneverdig. Antall merge-operasjoner og tidsbruk forblir tilnærmet likt, fordi algoritmens struktur er uavhengig av dataenes opprinnelige rekkefølge. Dette er en viktig styrke ved algoritmen, og en av hovedgrunnene til at den benyttes i praktiske sorteringssystemer hvor konsekvent ytelse er ønsket.

Problem 4. QuickSort – Antall sammenligninger

Introduksjon

QuickSort er en effektiv sammenligningsbasert sorteringsalgoritme som følger en divide-and-conquer-strategi. Algoritmen velger et pivot-element og omorganiserer (partisjonerer) resten av elementene slik at alle verdier mindre enn pivot havner på venstre side, og alle større på høyre.

Vi implementerte og analyserte tre ulike pivot-strategier:

1. Første element som pivot
2. Siste element som pivot
3. Tilfeldig valgt pivot

Antall sammenligninger som kreves for å sortere listen varierer betydelig med pivot-valget. Følgende resultater ble registrert på vårt datasett:

- Første element som pivot: 1 128 176 sammenligninger, 16 ms
- Siste element som pivot: 1 177 589 sammenligninger, 15 ms
- Tilfeldig valgt pivot: 1 080 614 sammenligninger, 48 ms

Implementasjon

Under ser du kjernen i vår QuickSort-implementasjon. Selve pivot-strategien bestemmes av en ekstern funksjon (utelatt her), men partisjoneringen og rekursjonen er identisk for alle strategiene:

```
void sort(City[] cities, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(cities, low, high);
        sort(cities, low, pivotIndex);
        sort(cities, pivotIndex + 1, high);
    }
}
```

partition() returnerer indeksen til pivot-elementet etter at listen er delt. Antall sammenligninger under sorteringen avhenger direkte av hvor balansert denne partisjoneringen er, noe som igjen påvirkes av hvordan pivot velges.

Analyse

Analysen viser tydelig at pivot-strategien har stor innvirkning på ytelsen:

- Når første eller siste element velges som pivot, kan partisjonene bli svært ubalanserte, særlig hvis dataene allerede er sortert eller omvendt sortert. Dette fører til worst-case kompleksitet $O(n^2)$.
- Den tilfeldig valgte pivoten gir oftere balanserte delinger, og nærmer seg den forventede tidskompleksiteten O(n log n).

Selv om den tilfeldige pivoten brukte litt mer kjøretid i millisekunder, reduserte den det totale antallet sammenligninger betydelig, noe som viser bedre skalering på store datasett.

Dette illustrerer et viktig poeng i algoritmeanalyse: antall operasjoner er ofte viktigere enn rå millisekund-ytelse, spesielt når algoritmen skal brukes på større eller mer varierende datasett.

Tidskompleksitet

QuickSorts teoretiske tidskompleksitet varierer med pivot-valget og dataenes struktur:

Best case: O(n log n) – oppstår når pivot gir to like store delmengder.

Gjennomsnittlig: O(n log n) – forutsatt god pivot-strategi (typisk med tilfeldig pivot).

Verstefall: $O(n^2)$ – hvis pivot alltid er det minste eller største elementet.

I praksis er QuickSort svært rask, og med en tilfeldig pivot-strategi unngår man i stor grad worst-casescenarier.

Kompleksitetsklasser

QuickSort er en algoritme som løses i polynomisk tid, og tilhører derfor kompleksitetsklassen P. Dette betyr at både løsning og verifikasjon skjer i tid som vokser polynomisk med størrelsen på input. Sortering regnes ikke som et NP-problem, ettersom det ikke involverer noen form for ikke-determinisme.

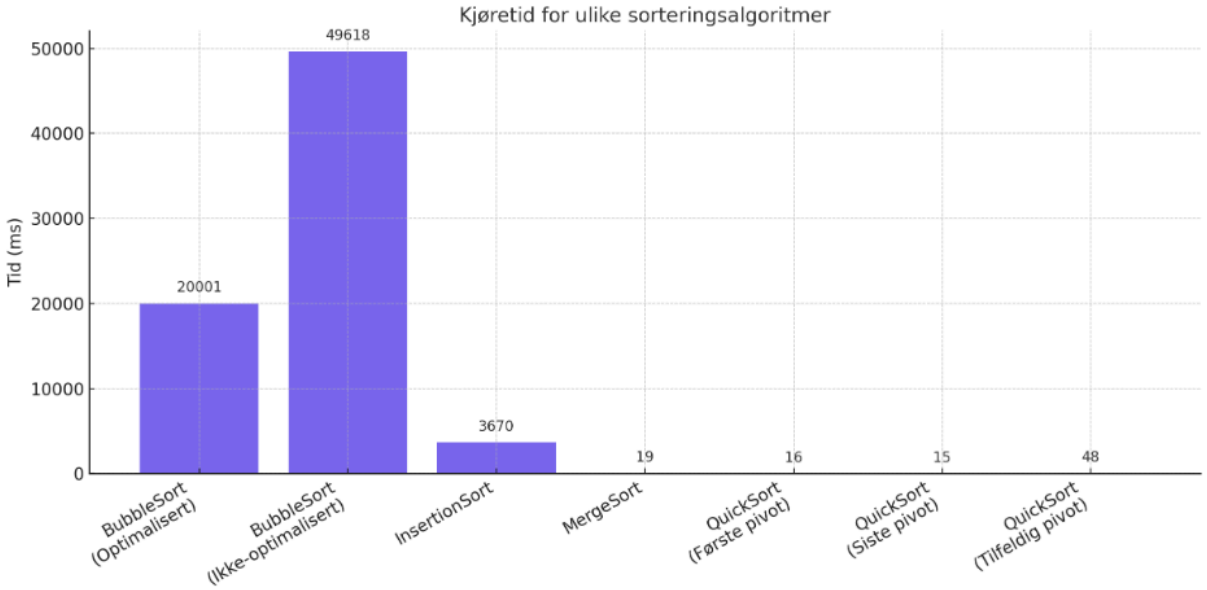
Selv om begrepet "NP" ofte brukes i diskusjoner om komplekse problemer, er det ikke relevant for deterministiske algoritmer som QuickSort, som har effektiv implementering og verifisering i praksis.

Resultater

Selv om tilfeldig pivot tok litt lengre tid i kjøretid (48 ms), ga den færrest sammenligninger og mer balansert partisjonering. Dette understøtter at tilfeldig valgt pivot gir mer robust ytelse, med kompleksitet nær $O(n \log n)$, i motsetning til de andre strategiene som lett havner i worst-case $O(n^2)$ ved ugunstig sorterte datasett (Sedgewick & Wayne, 2011, s. 288).

Visualisering av resultater

For å gi et tydelig bilde av hvor effektive de ulike algoritmene er, har vi laget et stolpediagram som viser kjøretiden i millisekunder for hver algoritme:



Konklusjon

I denne rapporten har vi undersøkt og analysert fire klassiske sorteringsalgoritmer: BubbleSort, InsertionSort, MergeSort og QuickSort basert på et datasett bestående av byer med unike breddegrader.

Resultatene viser tydelig at både teoretisk og praktisk ytelse varierer sterkt mellom algoritmene. BubbleSort og InsertionSort er enkle å implementere, men skalerer dårlig på store, tilfeldig sorterte datasett, med en tydelig $O(n^2)$ kjøretid. Samtidig viste vi at de kan være effektive ved nesten sortert data.

MergeSort leverte svært stabil ytelse uansett datasettets rekkefølge, med en kjøretid som samsvarer med den teoretiske kompleksiteten $O(n \log n)$. QuickSort presterte best når vi benyttet en tilfeldig pivot, og demonstrerte at valg av strategi har stor påvirkning på antall sammenligninger og dermed ytelse.

Gjennom arbeidet har vi fått bedre forståelse for hvordan algoritmer fungerer i praksis, og hvor viktig det er å vurdere både inputstruktur og ressursbruk når man velger sorteringsmetode.

Referanseliste

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Upper Saddle River, N.J.: Addison-Wesley.

W3Schools. (u.å.). *Merge Sort Algorithm*. Hentet 19. april 2025, fra https://www.w3schools.com/dsa/dsa_algo_mergesort.php

W3Schools. (u.å.) *Quick Sort.* Hentet 24. april 2025, fra https://www.w3schools.com/dsa/dsa_algo_quicksort.php

Wikipedia. (u.å.). Insertion sort. Hentet 24. april 2025, fra https://en.wikipedia.org/wiki/Insertion_sort