

# Algoritmer Eksamen

## Introduksjon

Denne rapporten undersøker fire sentrale sorteringsalgoritmer Bubble Sort, Insertion Sort, Merge Sort og Quick Sort med fokus på både teoretisk kompleksitet og praktisk ytelse på et virkelig datasett. Datasettet består av unike breddegrader for 47 868 byer (worldcities.csv), og vi måler hver algoritmes oppførsel i tre scenarier: originalrekkefølge, allerede sortert og tilfeldig stokket.

Spesielt for Bubble Sort implementeres både en uoptimalisert og en optimalisert versjon for å demonstrere hvordan en enkel forbedring (tidlig avbrudd når ingen bytter skjer) kan endre bestetilfellet. For Insertion Sort, Merge Sort og Quick Sort gjøres en korrekt implementasjon hver, men Quick Sort tester vi tre ulike pivotstrategier (første, siste og tilfeldig element) og teller antall sammenligninger.

Alle eksperimenter kjøres i Java (IntelliJ, Java 17) på en bærbar PC med Intel Core i7-8565U og 16 GB RAM. Rapporten følger denne strukturen:

1. Oppsett og datasettbeskrivelse
2. Implementasjon og kort algoritmebeskrivelse for hver sorteringsmetode
3. Målinger av kjøretid og/eller antall operasjoner i de tre scenariene
4. Teoretisk analyse av tids- og plasskompleksitet
5. Praktiske anbefalinger for når hver algoritme er mest hensiktsmessig

## Innholdsfortegnelse

Algoritmer Eksamen .....	1
Introduksjon .....	1
Innholdsfortegnelse .....	2
Setup av Miljø. ....	4
Maskinvaregrunnlag for testene .....	4
Datasettet .....	5
City-Klassen .....	5
CsvReader-Klassen .....	7
1.a .....	9
Algorithm description .....	9
Effektiv BubbleSort .....	9
InefficientBubbleSort .....	11
Sortering av data settet ved bruk av bubblesort .....	12
1.B. ....	14
Kalkulering av tidskompleksitet .....	14
Hva skjer ved tilfeldig rekkefølge? .....	15
Ytelsessammenligning for forskjellig datasett .....	15
Måling av kjøretid for Bubble Sort .....	16
Sammenligning av kjøretid: Optimalisert vs. Uoptimalisert Bubble Sort .....	17
Space Kompleksitet: Optimalisert vs. Uoptimalisert .....	18
Når bør BubbleSort brukes? .....	19
Oppsummering .....	20
2.a .....	22
Algorithm Description .....	22
InsertionSort .....	22
Sortering ved bruk av InsertionSort .....	23
2.b .....	25
Telling av tids kompleksiteten .....	25
Bytter og sammenligninger .....	26
Måling av kjøretid .....	27

Space Kompleksitet.....	28
Hva skjer ved tilfeldig rekkefølge? .....	29
Når bør InsertionSort brukes?.....	29
Oppsummering .....	30
3.a .....	31
Algorithm Description .....	31
MergeSort .....	31
Sortering ved bruk av MergeSort.....	33
3.b.....	34
Tidskompleksitet .....	34
Måling av kjøretid.....	34
Space Kompleksitet.....	35
Antall Merges – Orginal vs Tilfeldig rekkefølge. ....	36
Når bør MergeSort brukes?.....	37
Oppsummering .....	38
4.a .....	39
Algorithm Description .....	39
QuickSort.....	39
Sortering ved bruk av QuickSortFigur 29: Utklipp av main klassen som brukes til Quicksort.....	42
4.b.....	43
Tids kompleksitet .....	43
Antall sammenligninger ved ulike pivotstrategier .....	43
Tid ved ulike pivotstrategier .....	44
Space kompleksitet .....	46
Når bør QuickSort brukes? .....	46
Oppsummering .....	47
Oppsummering/Konklusjon - Sammenligning.....	48
Kilder .....	50

## Setup av Miljø.

Først skal vi etablere et nytt utviklingsmiljø der vi organiserer prosjektet vårt på en ryddig måte. Dette innebærer at vi oppretter et prosjekt i IntelliJ, der vi strukturerer koden i flere mapper og filer for å holde alt oversiktlig. Vi vil organisere koden ved å lage en egen pakke for hver oppgave, slik at hver funksjonalitet har sitt eget område i prosjektet.

Videre vil vi opprette en klasse som har ansvar for å lese inn data fra CSV-filen. Denne CSVReader-klassen vil ta seg av all filhåndtering og parsing av dataene, slik at vi enkelt kan hente ut informasjonen vi trenger.

I tillegg vil vi opprette en egen klasse med navnet City, som fungerer som en modell for by objektene vi skal arbeide med. City-klassen vil definere hvilke egenskaper en by skal ha, som navn, breddegrad, lengdegrad og annen relevant informasjon, og gjør det enkelt å representere dataene som objekter i koden vår.

**OBS:** Variabelen filePath, som linker til .csv-filen, vil ha en annen fil sti i selve koden enn det som vises på skjermbildene/utklippene som er brukt i denne rapporten. Dette skyldes at mappenavnet ble endret i forbindelse med innlevering. Husk derfor at filstien i koden kan avvike noe fra det som vises her.

Koden som er brukt for måling av kjøretid, og minne bruk er relativt lik på alle oppgavene.

## Maskinvaregrunnlag for testene

Alle målinger av kjøretid og minnebruk i prosjektet ble gjennomført på en bærbar PC med en Intel Core i7-8565U CPU @ 1.80GHz og 16 GB RAM. Dette gir et stabilt testmiljø for sammenligning av ulike sorteringsalgoritmer. Det er viktig å merke seg at kjøretidene er relative og vil kunne variere avhengig av maskinvare, slik at resultatene først og fremst egner seg til sammenligning innenfor samme testmiljø.

## Datasettet

I vårt tilfelle består datasettet av en liste med by-objekter (`List<City>`), og vi sorterer denne listen etter breddegrad (latitude) med en enkel sammenligning av desimaltall:

```
if (cities.get(j).getLat() > cities.get(j + 1).getLat()) {
```

Figur 1: Eksempel på hvordan datasettet sorteres fra Bubble Sort algoritmen

Denne sammenligningen utføres i hver iterasjon og påvirker ikke kompleksiteten, fordi tiden det tar å sammenligne to tall er konstant.

## City-Klassen

```
public class City { 10 usages
    private String city; 2 usages
    private String cityAscii; 1 usage
    private double lat; 3 usages
    private double lng; 1 usage
    private String country; 1 usage
    private String iso2; 1 usage
    private String iso3; 1 usage
    private String adminName; 1 usage
    private String capital; 1 usage
    private String population; 1 usage
    private String id; 1 usage

    public City(String city, String cityAscii, double lat, double lng, String country, 1 usage
        String iso2, String iso3, String adminName, String capital, String population, String id) {
        this.city = city;
        this.cityAscii = cityAscii;
        this.lat = lat;
        this.lng = lng;
        this.country = country;
        this.iso2 = iso2;
        this.iso3 = iso3;
        this.adminName = adminName;
        this.capital = capital;
        this.population = population;
        this.id = id;
    }

    public double getLat() { 4 usages
        return lat;
    }

    @Override
    public String toString() {
        return city + " (" + lat + ")";
    }
}
```

Figur 2: Utklipp av city-klassen.

Klassen `City` representerer et byobjekt med en rekke attributter som beskriver ulike egenskaper ved en by, slik som navnet, en ASCII-versjon av navnet, geografiske koordinater (latitude og longitude), samt informasjon om land, ISO-koder, administrativ inndeling, hovedstadsstatus, befolkningsdata og et unikt ID-nummer. Alle disse feltene er deklartert som `private`, noe som sikrer innkapsling og begrenser direkte tilgang til dataene utenfor klassen.

Konstruktøren tar imot verdier for alle disse feltene som parametere, og initierer objektet ved å tilordne parameterverdiene til de respektive feltene. Dette garanterer at ethvert opprettet City-objekt inneholder all nødvendig informasjon, og at objektets tilstand er konsistent fra opprettelsen.

Videre inneholder klassen en metode kalt `getLat()`, som returnerer byens latitude som en `double`.

Til slutt er `toString()`-metoden overstyrt for å gi en lettfattelig tekstlig representasjon av City-objektet. Metoden returnerer byens navn etterfulgt av latitude-verdien i parentes, for eksempel "Oslo (59.9139)". Dette formatet er nyttig for å raskt kunne verifisere at sorteringen, eller annen operasjon, fungerer som forventet ved å vise de relevante dataene på en konsis måte.

## CsvReader-Klassen

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.IOException;
4  import java.util.ArrayList;
5  import java.util.List;
6  import java.util.regex.Matcher;
7  import java.util.regex.Pattern;
8
9  public class CSVReader { 1 usage
10 @ public static List<City> readCitiesFromCSV(String filePath) { 1 usage
11     List<City> cities = new ArrayList<>();
12     // Regex for å finne innholdet mellom doble anførselstegn
13     Pattern pattern = Pattern.compile(regex: "\"(.*?)\"");
14     try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
15         String line;
16         boolean isFirstLine = true;
17         while ((line = br.readLine()) != null) {
18             // Hopp over header-linjen
19             if (isFirstLine) {
20                 isFirstLine = false;
21                 continue;
22             }
23             Matcher matcher = pattern.matcher(line);
24             List<String> fields = new ArrayList<>();
25             while (matcher.find()) {
26                 fields.add(matcher.group(1));
27             }
28             // Sjekk at vi har nok felter (her forventes 11)
29             if (fields.size() < 11) {
30                 continue;
31             }
32             String city = fields.get(0);
33             String cityAscii = fields.get(1);
34             double lat = Double.parseDouble(fields.get(2).trim());
35             double lng = Double.parseDouble(fields.get(3).trim());
36             String country = fields.get(4);
37             String iso2 = fields.get(5);
38             String iso3 = fields.get(6);
39             String adminName = fields.get(7);
40             String capital = fields.get(8);
41             String population = fields.get(9);
42             String id = fields.get(10);
43
44             cities.add(new City(city, cityAscii, lat, lng, country, iso2, iso3, adminName, capital, population, id));
45         }
46     } catch (IOException e) {
47         e.printStackTrace();
48     }
49     return cities;
50 }
51 }
```

Figur 3: Utklipp fra CsvReader-klassen.

Først blir det definert en klasse kalt CSVReader med en statisk metode, readCitiesFromCSV, som leser data fra en CSV-fil og konverterer hver rad (etter header-linjen) til et City-objekt. Metoden starter med å initialisere en tom liste for å lagre byobjektene, samt å compilere et regulært uttrykk som skal matche alt som står mellom doble anførselstegn. Dette regulære uttrykket er avgjørende for å hente ut de enkelte feltene fra hver rad i CSV-filen, siden hvert felt er omsluttet av doble anførselstegn.

Når filen åpnes med en BufferedReader, leses den linje for linje. Første linje antas å være en header og blir derfor hoppet over. For hver påfølgende linje brukes det kompilerte regex-

mønsteret for å finne alle forekomster av tekst mellom doble anførselstegn, og disse blir lagret i en midlertidig liste. Det gjøres en sjekk for å forsikre seg om at linjen inneholder minst 11 felt, hvis ikke, hopper koden over linjen.

Deretter hentes verdiene fra de ulike posisjonene i listen, og de nødvendige feltene for eksempelvis latitude og longitude blir trimmet for å fjerne eventuelle ekstra mellomrom før de konverteres til datatypen double. De andre feltene, som bynavn og landinformasjon, blir lagret som strenger. Med disse verdiene opprettes et nytt City-objekt, som så legges til i listen over byer.

Til slutt håndteres eventuelle IO-unntak gjennom en try-catch-blokk, og etter at alle linjene er behandlet, returneres listen med City-objekter. På denne måten gir koden en strukturert og objektorientert tilnærming til å lese og prosessere CSV-data.



## 1.a

### Algorithm description

Bubble Sort er en sammenligningsbasert sorteringsalgoritme som organiserer en liste eller et array ved å gjentatte ganger sammenligne og bytte plass på tilstøtende elementer. Målet med algoritmen er å sortere verdiene fra den laveste til den høyeste, og navnet «Bubble» kommer av at den største verdien «bobler opp» til slutten av arrayet etter hver gjennomgang (W3Schools).

### Effektiv BubbleSort

```
1 import java.util.List;
2
3 public class Bubblesort { 3 usages
4     @ public static int bubbleSortByLatitude(List<City> cities) { 3 usages
5         int n = cities.size();
6         boolean swapped;
7         int swapCount = 0;
8
9         for (int i = 0; i < n - 1; i++) {
10             swapped = false;
11             for (int j = 0; j < n - i - 1; j++) {
12                 if (cities.get(j).getLat() > cities.get(j + 1).getLat()) {
13                     City temp = cities.get(j);
14                     cities.set(j, cities.get(j + 1));
15                     cities.set(j + 1, temp);
16                     swapped = true;
17                     swapCount++;
18                 }
19             }
20             if (!swapped) break;
21         }
22
23         return swapCount;
24     }
25 }
```

Figur 4: Utklipp av Bubblesort klassen

Koden implementerer en optimalisert bubble sort-algoritme for å sortere en liste av City-objekter basert på deres latitude-verdi. Metoden tar inn en liste og returnerer antall bytter som ble gjort under sorteringen.

I starten beregnes antallet elementer i listen med `int n = cities.size();`. To variabler deklarerer: `swapped` (en boolean som indikerer om det ble utført noen bytter i løpet av en iterasjon) og `swapCount` (en teller som holder oversikt over det totale antallet bytter).

Deretter starter den ytre løkken, som går fra  $i = 0$  til  $i < n - 1$ . For hver iterasjon settes `swapped` til false, og en indre løkke starter fra  $j = 0$  til  $j < n - i - 1$ . Denne indre løkken sammenligner naboelementene ved å sjekke om latitude-verdien til elementet på indeks  $j$  er større enn den på indeks  $j + 1$ . Hvis dette er tilfellet, bytter koden plass på disse to elementene ved hjelp av en midlertidig variabel *temp*, og både `swapped` og `swapCount` økes.

Etter at den indre løkken er ferdig, sjekker algoritmen om ingen bytter ble gjort i løpet av den passeringen (dvs. `swapped` forblir false). Hvis dette er tilfelle, betyr det at listen allerede er sortert, og løkken brytes ut tidlig med `break`.

Til slutt returnerer metoden `swapCount`, som indikerer det totale antallet bytter som ble gjort under sorteringen. Dette gir en målestokk på hvor mye listen måtte "omrøkkers" for å bli sortert, og demonstrerer dermed effektiviteten til sorteringsprosessen.

## InefficientBubbleSort

```
public class InefficientBubbleSort { no usages
    public static void sortByLatitude(List<City> cities) { no usages
        int n = cities.size();
        // Denne ytre løkken kjører n ganger, selv om listen kan bli sortert før vi ha
        for (int i = 0; i < n; i++) {
            // Den indre løkken sammenligner og bytter naboelementer for hver passering
            for (int j = 0; j < n - 1; j++) {
                if (cities.get(j).getLat() > cities.get(j + 1).getLat()) {
                    // Bytt plass på elementene
                    City temp = cities.get(j);
                    cities.set(j, cities.get(j + 1));
                    cities.set(j + 1, temp);
                }
            }
        }
    }
}
```

Figur 5: Utklipp av InefficientBubbleSort klassen

Denne koden definerer en klasse kalt InefficientBubbleSort som inneholder en statisk metode, sortByLatitude, for å sortere en liste med City-objekter basert på byenes latitude. Metoden tar inn en liste som parameter og bestemmer antallet elementer med cities.size(). Deretter benytter den to nestede løkker for å gjennomføre sorteringen.

I den ytre løkken itererer koden fra 0 til  $n$  (antall elementer) og gjennomfører en full passering av listen for hver iterasjon. Den indre løkken sammenligner tilstøtende elementer fra indeks 0 til  $n - 2$ , og for hver sammenligning sjekker den om latitude-verdien til elementet på indeks  $j$  er større enn den til elementet på indeks  $j + 1$ . Dersom dette er tilfelle, byttes de to elementene ved hjelp av en midlertidig variabel temp.

Denne implementasjonen er klassifisert som ineffektiv fordi den ytre løkken alltid kjører  $n$  ganger, uavhengig av om listen allerede er sortert før alle iterasjoner er fullført. Det innebærer at selv om listen kan være sortert tidlig, vil algoritmen fortsette å gjennomføre unødvendige sammenligninger.

## Sortering av data settet ved bruk av bubblesort

```
1 package Task1;
2
3 import java.util.List;
4 import shared.CSVReader;
5 import shared.City;
6
7 public class SortMain {
8     public static void main(String[] args) {
9         String filePath = "Eksamen Algoritmer - Kopi/Databaser/worldcities.csv";
10        List<City> cities = CSVReader.readCitiesFromCSV(filePath);
11        System.out.println("Før sortering: ");
12
13
14        for(City city : cities) {
15            System.out.println(city);
16        }
17
18        Bubblesort.bubbleSortByLatitude(cities);
19        System.out.println("\nEtter sortering (basert på latitude): ");
20
21        for(City city : cities) {
22            System.out.println(city);
23        }
24    }
25
26 }
```

Figur 6: Utklipp av main ved kjøring

Denne koden leser inn en fil, og lagrer dem som en liste av City-objekter, og skriver deretter ut listen før og etter at den har blitt sortert etter latitude ved hjelp av bubble sort. Først spesifiseres filstien til filen ("databaser/worldcities.csv"), og metoden readCitiesFromCSV henter ut dataene. Deretter skrives alle byene ut i sin opprinnelige rekkefølge. Etterpå kalles metoden bubbleSortByLatitude for å sortere byene basert på deres latitude, og til slutt skrives den sorterte listen ut slik at man kan se effekten av sorteringen.

Det vil da "printes" ut en liste som den man ser i figur 7.

```
Etter sortering (basert på latitude):  
Puerto Williams (-54.9333)  
Ushuaia (-54.8019)  
Kaiken (-54.5062)  
King Edward Point (-54.2833)  
Grytviken (-54.2806)  
Río Grande (-53.7833)  
Punta Arenas (-53.1667)  
Puerto Natales (-51.7333)  
Stanley (-51.7)  
Veintiocho de Noviembre (-51.65)  
Río Gallegos (-51.6233)
```

*Figur 7: Utklipp av output etter kjøring*

## 1.B.

### Kalkulering av tidskompleksitet

I både gjennomsnittstilfellet (for eksempel en tilfeldig ordnet liste) og verstefallstilfellet (der listen er sortert i motsatt rekkefølge), må algoritmen i `bubbleSortByLatitude` sammenligne og potensielt bytte nesten hvert eneste element med sine naboer gjennom flere passeringer.

Koden består av to nestede løkker – en ytre løkke (for (int i = 0; i < n - 1; i++)) som kjører opptil n ganger, og en indre løkke (for (int j = 0; j < n - i - 1; j++)) som i verste fall også kjører opptil n ganger per passering.

Totalt gir dette et antall operasjoner på omtrent  $\frac{n(n-1)}{2}$ , noe som tilsvarer en kompleksitet på  $O(n^2)$  i både gjennomsnittstilfellet og verstefallstilfellet. Siden hver sammenligning og bytte utføres på konstant tid, bekrefter dette at implementasjonen av algoritmen `bubbleSortByLatitude` har en total tidskompleksitet på  $O(n^2)$ .

I gjennomsnittstilfellet, når elementene er tilfeldig ordnet, må algoritmen fremdeles gjennomføre et betydelig antall sammenligninger og bytter før listen blir sortert. I verstefallet, når elementene ligger i motsatt rekkefølge, må hvert element flyttes hele veien gjennom listen, og ingen av iterasjonene kan avsluttes tidlig. Hele den kvadratiske strukturen i de to nestede løkkene må derfor gjennomføres, noe som resulterer i dårlig ytelse for store datasett.

### Bestefall for algoritmen

Når listen allerede er sortert, vil implementasjonen av `bubbleSortByLatitude` oppdage dette allerede i første iterasjon fordi variabelen `swapped` aldri blir satt til true. Dermed vil algoritmen umiddelbart avslutte etter en full gjennomgang av listen med kun  $n - 1$  sammenligninger og ingen bytter. Dette gir algoritmen en total tidskompleksitet på  $O(n)$  i beste tilfelle. Denne optimaliseringen gir altså en betydelig ytelsesforbedring sammenlignet med uoptimaliserte versjoner.

## Hva skjer ved tilfeldig rekkefølge?

Dersom du stokker om listen tilfeldig før sortering, vil listen i de aller fleste tilfeller være langt unna å være sortert. Algoritmen vil derfor ikke kunne avslutte tidlig, og man må gjøre nesten like mange sammenligninger som i verstefall. Tidskompleksiteten forblir  $O(n^2)$ . En tilfeldig rekkefølge gir ingen garanti for færre operasjoner.

## Ytelsessammenligning for forskjellig datasett

I praksis demonstrerte vi hvordan Bubble Sort oppfører seg under tre ulike tilstander av datasettet ved å måle antall bytter som utføres under sorteringen. Først brukte vi datasettet slik det leses inn fra CSV-filen, og målte antall bytter i den opprinnelige rekkefølgen. Dette ga oss et grunnlag for å forstå hvordan algoritmen håndterer data uten forbehandling.

Deretter sorterte vi datasettet fullstendig etter breddegrad for å simulere bestefall. Ved å kjøre sorteringsalgoritmen på en allerede sortert liste, oppdaget vi at den optimaliserte implementasjonen raskt konkluderte med at ingen bytter var nødvendige, resultatet ble 0 bytter, noe som demonstrerer en lineær ytelse  $O(n)$  i dette scenariet.

Til slutt stokket vi datasettet tilfeldig med *Collections.shuffle()* og utførte sorteringen flere ganger for å sikre pålitelighet i målingene. Antallet bytter ble beregnet som gjennomsnittet av fem separate kjøring, noe som ga et gjennomsnitt på 573 451 918 bytter. Dette bekrefter tydelig at algoritmen opererer med en kvadratisk tidskompleksitet  $O(n^2)$  i gjennomsnittlige eller verstefallstilfeller.

Samlet viser denne praktiske demonstrasjonen at mens en allerede sortert liste gir en vesentlig raskere ytelse, reduserer ikke en tilfeldig rekkefølge den asymptotiske kompleksiteten til Bubble Sort. Datasettet i sin opprinnelige rekkefølge ligger et sted mellom disse ytterpunktene, men det er først i bestefall-situasjonen at den optimale ytelsen oppnås.

```
// 1. ORIGINAL REKKEFØLGE (slik dataene kommer fra CSV-filen)
List<City> originalList = new ArrayList<>(cities);
int swapsOriginal = Bubblesort.bubbleSortByLatitude(originalList);
System.out.println("Antall bytter (original rekkefølge): " + swapsOriginal);

// 2. ALLEREDER SORTERT LISTE (bestefall)
List<City> sortedList = new ArrayList<>(cities);
Bubblesort.bubbleSortByLatitude(sortedList); // Første sortering for å gjøre listen helt sortert
int swapsSorted = Bubblesort.bubbleSortByLatitude(sortedList); // Nå skal den oppdage at listen er sortert
System.out.println("Antall bytter (allerede sortert liste): " + swapsSorted);

// 3. TILFELDIG STOKKET LISTE (gjennomsnitt/verste fall)
List<City> shuffledList = new ArrayList<>(cities);
Collections.shuffle(shuffledList); // Shuffler kortene til en tilfeldig rekkefølge
int swapsShuffled = Bubblesort.bubbleSortByLatitude(shuffledList);
System.out.println("Antall bytter (etter tilfeldig stokking): " + swapsShuffled);
```

Figur 8: Utklipp av kode som teller antall bytter i forskjellig datasett

```
Antall bytter (original rekkefølge): 540321435
Antall bytter (allerede sortert liste): 0
```

Figur 9: Output av antall bytter gjort

## Måling av kjøretid for Bubble Sort

For å støtte analysen av tidskompleksitet ble kjøretid målt for tre ulike tilstander av datasettet: original rekkefølge, allerede sortert og tilfeldig stokket. I koden brukte vi `System.currentTimeMillis()` før og etter sorteringskallet for å måle hvor lang tid algoritmen brukte på hvert scenario. For scenarioet med tilfeldig rekkefølge ble sorteringen kjørt flere ganger, og gjennomsnittlig kjøretid ble brukt for å gi et bedre sammenligningsgrunnlag. Tiden for tilfeldig rekkefølge ble 19 803 milisekund i gjennomsnitt.

```
List<City> originalList = new ArrayList<>(cities);
long startOriginal = System.currentTimeMillis();
Bubblesort.bubbleSortByLatitude(originalList);
long endOriginal = System.currentTimeMillis();
System.out.println("Tid (original rekkefølge): " + (endOriginal - startOriginal) + " ms");
```

Figur 10: Utklipp av kode som måler tid i ms

Resultatene bekreftet observasjonene fra byttemålingene, at Bubble Sort har lineær ytelse i bestefall (allerede sortert), og kvadratisk ytelse i både tilfeldig og original rekkefølge. Den



sorterte listen ble behandlet på kun 1 ms, mens de to andre krevde betydelig mer tid. Dette illustrerer hvordan algoritmens kjøretid i praksis følger det teoretiske ytelsesmønsteret.

```
Antall byer: 47868  
Tid (original rekkefølge): 19182 ms  
Tid (allerede sortert): 1 ms
```

*Figur 11: Output av millisekund brukt*

## Sammenligning av kjøretid: Optimalisert vs. Uoptimalisert Bubble Sort

Vi valgte å sammenligne den optimaliserte og den uoptimaliserte versjonen av Bubble Sort for å tydeliggjøre effekten av selv en enkel optimalisering på algoritmens praktiske ytelse. Ved å inkludere begge versjonene i testen ønsket vi å vise hvordan dette slår ut i praksis, særlig når datasettet allerede er sortert eller delvis sortert.

Resultatene viser tydelig hvorfor dette er en verdifull sammenligning, mens den optimaliserte algoritmen brukte kun 1 millisekund på å sortere en allerede sortert liste, brukte den uoptimaliserte versjonen 37 216 millisekunder på nøyaktig samme datasett. Det skjer fordi den uoptimaliserte versjonen ikke har noen mekanisme for å oppdage at sorteringen er ferdig, og derfor fortsetter å gjøre unødvendige passeringer og sammenligninger helt til slutten, selv om listen allerede er sortert.

Ved å også inkludere original rekkefølge og tilfeldig stokket rekkefølge i begge versjoner, fikk vi et helhetlig bilde av ytelsen. Her så vi at selv i ikke-bestefall, reduserte den optimaliserte algoritmen kjøretiden med flere sekunder sammenlignet med den uoptimaliserte. Sammenligningen viser dermed at valg av implementasjonsstrategi har stor betydning, og at forståelse for når og hvordan man kan forbedre en algoritme er essensielt, særlig når man arbeider med større datasett.

```

Tid (original rekkefølge): 19342 ms
Tid (allerede sortert): 1 ms
Tid (stokket): 19611 ms
-----
Tid (original rekkefølge uoptimalisert): 41334 ms
Tid (allerede sortert uoptimalisert): 37216 ms
Tid (stokket uoptimalisert): 30932 ms

```

Figur 12: Forskjellen på optimalisert og uoptimalisert

## Space Kompleksitet: Optimalisert vs. Uoptimalisert

Bubble Sort er en in-place algoritme med en teoretisk plasskompleksitet på  $O(1)$ , ettersom den ikke bruker ekstra datastrukturer uavhengig av størrelsen på datasettet. Algoritmen sorterer direkte i den eksisterende listen og benytter kun noen få hjelpevariabler for sammenligning og bytte, noe som gjør den svært plass-effektiv.

For å bekrefte dette i praksis, målte vi minneforbruket før og etter sortering i tre ulike scenarioer: original rekkefølge, allerede sortert og tilfeldig stokket. Testen ble gjennomført både for den optimaliserte og uoptimaliserte versjonen av algoritmen. Minneverdiene ble innhentet ved hjelp av `Runtime.getRuntime()`, og garbage collection ble utløst før måling for å forbedre nøyaktigheten.

Resultatene vises i tabellen under:

Versjon	Scenario	Før sortering (bytes)	Etter sortering (bytes)	Endring (bytes)
<b>Optimalisert</b>	Original rekkefølge	26 835 048	26 849 640	+14 592
<b>Optimalisert</b>	Allerede sortert (bestefall)	27 031 928	27 031 688	-240
<b>Optimalisert</b>	Tilfeldig stokket	27 225 520	27 228 560	+3 040
<b>Uoptimalisert</b>	Original rekkefølge	27 421 520	27 426 816	+5 296
<b>Uoptimalisert</b>	Allerede sortert	27 620 952	27 620 848	-104
<b>Uoptimalisert</b>	Tilfeldig stokket	27 813 528	27 814 776	+1 248

Tabell 1: Oversikt over minnebruk fra BubbleSort

Alle målingene viser svært små endringer i brukt minne, typisk i området fra noen hundre til få tusen bytes. Disse variasjonene er så små at de mest sannsynlig skyldes Java Virtual Machines interne håndtering av minne og garbage collection, og ikke selve sorteringsalgoritmen. Både den optimaliserte og uoptimaliserte versjonen bekrefter derfor den teoretiske plasskompleksiteten på  $O(1)$  og viser at Bubble Sort har lav og stabil minnebruk i praksis, uavhengig av datasettets rekkefølge

### Når bør BubbleSort brukes?

Bubble Sort er enkel å forstå og implementere, men lider av en kvadratisk tidskompleksitet  $O(n^2)$  både i gjennomsnitts- og verstefall, noe som gjør den svært treg for store datamengder. På grunn av dette har Bubble Sort i praksis nesten ingen reelle anvendelser utenom som et pedagogisk verktøy i akademiske sammenhenger for å illustrere grunnleggende sorteringsprinsipper. Algoritmen brukes sjelden i produksjonssystemer, der man heller foretrekker mer effektive metoder som QuickSort eller Merge Sort (GeeksForGeeks, 2025).

## Oppsummering

- Verste fall (**Big-O**):  $O(n^2)$
- Gjennomsnitt (**Theta**):  $O(n^2)$
- Beste fall (**Omega, med optimalisering**):  $O(n)$

Bubble Sort har en teoretisk tidskompleksitet på  $O(n^2)$  i både gjennomsnitt og verste fall, fordi den består av to nestede løkker som sammenligner og eventuelt bytter naboelementer. Siden vårt datasett inneholder 47 868 byer, vil algoritmen i verste fall utføre omtrent  $n \frac{(n-1)}{2} \approx 1,14$  milliarder operasjoner. I beste fall, når listen allerede er sortert og vi bruker en optimalisert versjon av algoritmen, er kompleksiteten redusert til  $O(n)$ , fordi algoritmen avslutter etter en gjennomgang.

Testscenario	Versjon	Bytter	Kjøretid	Minnebruk
Original rekkefølge	Optimalisert	540 321 435	19 342 ms	320 bytes
Allerede sortert	Optimalisert	0	1 ms	-48 bytes
Tilfeldig stokket	Optimalisert	573 451 918	19 803 ms	-48 bytes
Original rekkefølge	Uoptimalisert	540 321 435	41 334 ms	384 bytes
Allerede sortert	Uoptimalisert	0	37 216 ms	128 bytes
Tilfeldig stokket	Uoptimalisert	570 883 740	30 932 ms	-80 bytes

Tabell 2: Oversikt over målinger fra BubbleSort

Tabellen oppsummerer forskjellene i antall bytter, kjøretid og minnebruk mellom optimalisert og uoptimalisert Bubble Sort for tre ulike datasett-tilstander. Resultatene viser at begge versjoner gjør omtrent like mange bytter når det trengs, men at den uoptimaliserte versjonen bruker betydelig mer tid, særlig i allerede sorterte tilfeller hvor den ikke avslutter tidlig. Dette bekrefter at optimalisering har stor praktisk betydning for ytelse, selv om den ikke påvirker algoritmens asymptotiske kompleksitet.

## 2.a

### Algorithm Description

Insertion Sort fungerer ved at man bygger opp en sortert del av lista en og en. Først antas det første elementet som sortert. For hvert nytt element («key») sammenlignes det med elementene i den sorterte delen fra høyre mot venstre; alle elementer som er større enn «key» flyttes en plass mot høyre for å gjøre plass. Når man treffer et element som er mindre enn eller lik «key», settes «key» inn i den ledige plassen, og algoritmen går videre til neste usorterte element (W3Schools, u.å.). Denne prosessen gjentas til alle elementene er inkorporert i den sorterte delen.

### InsertionSort

```
1 package Task2;
2
3 import java.util.List;
4 import shared.City;
5
6 public class InsertionSort { 13 usages
7
8
9 @ public static void insertionSortByLatitude(List<City> cities) { 13 usages
10     // Starter fra indeks 1 siden det første elementet antas å være "sortert"
11     for (int i = 1; i < cities.size(); i++) {
12         City key = cities.get(i);
13         int j = i - 1;
14
15         // Flytt alle elementer med høyere latitude til en posisjon høyre
16         while (j >= 0 && cities.get(j).getLat() > key.getLat()) {
17             cities.set(j + 1, cities.get(j));
18             j--;
19         }
20
21         // Sett inn key på riktig posisjon
22         cities.set(j + 1, key);
23     }
24 }
25 }
```

Figur 13: InsertionSort Klassen.

For hvert element som skal sorteres, hentes objektet fra den gjeldende posisjonen og lagres midlertidig i en variabel kalt "key". Deretter settes en tellevariabel til posisjonen rett før elementet som skal settes inn. Koden går så inn i en while-løkke som går baklengs gjennom

den allerede sorterte delen av listen. I denne løkken sammenlignes breddgraden til hvert element med breddgraden til "key". Så lenge et element i den sorterte delen har en høyere latitude enn "key", flyttes dette elementet en posisjon til høyre. På den måten blir det opprettet ledig plass for "key" til å settes inn på riktig plass.

Når den riktige posisjonen er funnet det vil si når enten starten av listen er nådd, eller når et element med lavere (eller lik) breddegrad er funnet settes "key" inn på plassen som er opprettet. Denne prosessen gjentas for hvert element i listen, slik at den sorterte delen vokser med ett element for hver iterasjon, helt til hele listen er sortert.

## Sortering ved bruk av InsertionSort

```
1 package task2;
2
3 import java.util.List;
4 import shared.CSVReader;
5 import shared.City;
6
7 public class SortMain {
8     public static void main(String[] args) {
9         String filePath = "Eksamen Algoritmer - Kopi/Databaser/worldcities.csv";
10        List<City> cities = CSVReader.readCitiesFromCSV(filePath);
11        System.out.println("Før sortering: ");
12
13
14        for(City city : cities) {
15            System.out.println(city);
16        }
17
18        InsertionSort.insertionSortByLatitude(cities);
19        System.out.println("\nEtter sortering (basert på latitude): ");
20
21        for(City city : cities) {
22            System.out.println(city);
23        }
24    }
25 }
```

Figur 14: Bruk av insertionSort til å sortere listen.

Koden er lik som i Oppgave 1, men i stedet for å bruke bubble sort for å sortere byene etter latitude, benyttes nå insertion sort.

```
Etter sortering (basert på latitude):  
Puerto Williams (-54.9333)  
Ushuaia (-54.8019)  
Kaiken (-54.5062)  
King Edward Point (-54.2833)  
Grytviken (-54.2806)  
Río Grande (-53.7833)  
Punta Arenas (-53.1667)  
Puerto Natales (-51.7333)  
Stanley (-51.7)
```

*Figur 15: Sorterte listen.*



## 2.b

### Telling av tids kompleksiteten

For å analysere hvor effektiv Insertion Sort er, teller vi hvor mange operasjoner algoritmen utfører i forhold til datasettets størrelse. Insertion Sort (se klassen 'InsertionSort') består av en ytre løkke (linje 19) som itererer gjennom elementene i listen fra indeks 1 og utover.

For hvert element («key») sammenligner den bakover mot tidligere sorterte elementer i listen (linje 24–33). Dersom tidligere elementer har høyere verdi enn «key», flyttes de ett hakk mot høyre, og dette telles som en forskyvning (linje 28).

I verstefall, når listen er sortert i motsatt rekkefølge, vil hvert element måtte flyttes helt tilbake til starten av listen, noe som gir summen:

$$1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2}$$

av sammenligninger og flyttinger (ref. 'comparisonCount' og 'swapCount'). Selv om det i tillegg utføres noen små operasjoner som indeksjusteringer, endrer dette ikke kompleksitetsklassen, som fortsatt blir  $O(n^2)$ .

I gjennomsnitt, når listen er tilfeldig stokket, vil hvert element flyttes omtrent halvparten så langt, altså rundt:

$$\frac{\left\lfloor \frac{n(n - 1)}{2} \right\rfloor}{2} = \frac{n(n - 1)}{4}$$

Også her blir veksten kvadratisk, og tidskompleksiteten forblir  $O(n^2)$ .

### Bestefall for algoritmen

Bestefallet for Insertion Sort oppstår når listen allerede er sortert stigende etter latitude. Da utfører algoritmen nøyaktig en sammenligning per element (linje 25-26), men ingen elementer flyttes. Totalt blir dette  $(n - 1)$  sammenligninger, som gir lineær tidskompleksitet  $O(n)$ . Derfor er Insertion Sort spesielt effektivt for små eller nesten-sorterte datasett, i kontrast til Bubble Sort, som alltid gjennomfører fullstendige gjennomganger av listen uavhengig av hvor sortert dataene er.

## Bytter og sammenligninger

For å oppnå en rettferdig sammenligning mellom Insertion Sort og Bubble Sort, benytter vi nøyaktig samme testmiljø og datasett som i Bubble Sort-eksperimentene. Dette innebærer de samme tre scenarioene: original rekkefølge, allerede sortert og tilfeldig stokket.

Vi måler både antall bytter (flyttinger av elementer) og sammenligninger (hver gang algoritmen sjekker om et element må flyttes). Som vist i tabellen, ligner verdiene for Insertion Sort mye på Bubble Sort, da begge algoritmene har en kvadratisk kompleksitet  $O(n^2)$  i gjennomsnittlig og verstefall.

- **Original rekkefølge**

Antall bytter er fremdeles i nærheten av kvadratisk, siden dataene ikke er spesielt “snille” for Insertion Sort. Algoritmen må gjøre mange sammenligninger og forskyvninger før listen er sortert. I eksempelet ser vi også at antall sammenligninger ligger svært tett opp mot antall bytter, noe som bekrefter den kvadratiske veksten.

```
Sortert (original rekkefølge):  
Bytter: 540321435, Sammenligninger: 540369293
```

Figur 16: Viser antall bytter og sammenligninger til original rekkefølge

- **Allerede sortert**

Antall bytter er 0 (bestefall) fordi Insertion Sort raskt oppdager at hvert element allerede er i riktig rekkefølge og ikke trenger å flyttes. Likevel er antall sammenligninger ikke nødvendigvis 0; algoritmen må fortsatt utføre en rask sjekk for hvert nytt element. Dette speiler også optimalisert Bubble Sorts bestefall, der algoritmen kan avslutte tidlig uten unødvendige ombyttinger.

```
Sortert (allerede sortert):  
Bytter: 0, Sammenligninger: 47867
```

Figur 17: Viser antall bytter og sammenligninger til allerede sortert liste

- **Tilfeldig stokket**

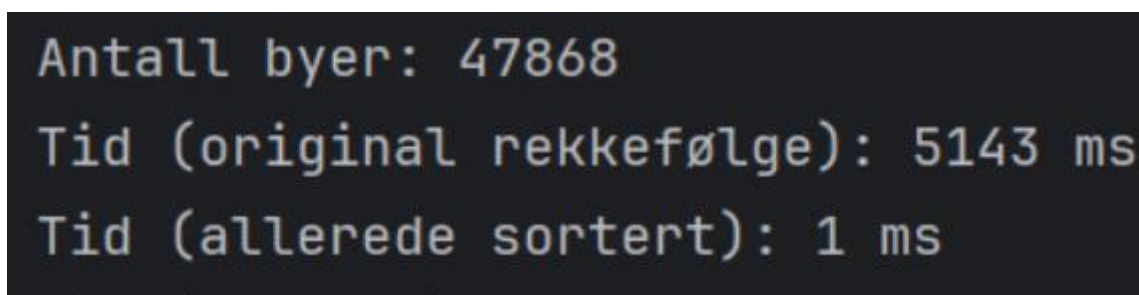
For å få et mer pålitelig bilde av ytelsen i gjennomsnittstilfellet, ble datasettet stokket tilfeldig og kjørt fem ganger. Vi tok deretter gjennomsnittet av resultatene. Antall bytter endte på 572 548 774 og antall sammenligninger lå på 572 596 631. Dette bekrefter den kvadratiske oppførselen i praksis, siden både bytter og sammenligninger øker raskt med datasettets størrelse. Resultatene ligger på samme nivå som for Bubble Sort, noe som viser at Insertion Sort også har kvadratisk tidskompleksitet i gjennomsnitt når listen er stokket.

## Måling av kjøretid

For å telle tidskompleksiteten i praksis, målte vi hvor lang tid Insertion Sort brukte på å sortere datasettet under de tre forskjellige rekkefølgene. Målingene ble registrert med *System.currentTimeMillis()*.

Resultatene viste at bestefall kun tok rundt 1 ms, noe som bekrefter en omtrentlig  $O(n)$  oppførsel når datasettet allerede er sortert. Ved original rekkefølge og tilfeldig stokket datasett økte derimot kjøretiden, noe som illustrerer algoritmens  $O(n^2)$  tidskompleksitet i både gjennomsnitt og verstefall. For å gi et mer pålitelig resultat i scenarioet med tilfeldig stokket datasett, gjennomførte vi fem kjøring og brukte gjennomsnittet, som endte på 5539 milisekund. Dette støtter den teoretiske analysen om at Insertion Sort har lineær ytelse i bestefall og kvadratisk ytelse ellers.

En kort sammenligning med Bubble Sort viser at begge algoritmene har liknende tidskompleksitet. Likevel kan Insertion Sort ofte utføre færre unødvendige operasjoner på nesten sorterte data, slik at det i praksis kan oppleves noe raskere enn Bubble Sort.



```
Antall byer: 47868
Tid (original rekkefølge): 5143 ms
Tid (allerede sortert): 1 ms
```

Figur 18: Viser målt tid i ms med insertion sort

## Space Kompleksitet

Insertion Sort er en *in-place* algoritme, noe som betyr at den sorterer dataene direkte i den eksisterende listen uten å bruke ekstra datastrukturer. Den benytter kun et fåtall hjelpevariabler som *key*, *i* og *j*, uavhengig av størrelsen på datasettet. Dette gir algoritmen en teoretisk plasskompleksitet på  $O(1)$ , altså konstant ekstra minneforbruk.

Vi testet dette i praksis ved å måle minnebruk før og etter sortering i tre scenarioer: original rekkefølge, allerede sortert og tilfeldig stokket. Målingen ble gjort med `Runtime.getRuntime()` og viser kun små variasjoner, for eksempel økte minnebruken med 688 bytes i original rekkefølge, mens den sank med 208 bytes i de to andre tilfellene. Små variasjoner i målt minnebruk reflekterer sannsynligvis hvordan Java rydder opp i hukommelsen, ikke ulikt det som ble observert med Bubble Sort.

```
Måling av minnebruk (Insertion Sort):
Test 1: Original rekkefølge:
  Før sortering: 24651552 bytes
  Etter sortering: 24652240 bytes
  Endring: 688 bytes

Test 2: Allerede sortert liste (bestefall):
  Før sortering: 24886248 bytes
  Etter sortering: 24886040 bytes
  Endring: -208 bytes

Test 3: Tilfeldig stokket liste:
  Før sortering: 25081960 bytes
  Etter sortering: 25081752 bytes
  Endring: -208 bytes
-----
```

Figur 19: Viser endring i minne bruk ved de forskjellige datasettene

Sammenlignet med Bubble Sort viser Insertion Sort samme minneprofil, begge sorterer direkte i datastrukturen uten å bruke ekstra lagring. Målingene våre bekrefter derfor at Insertion Sort i praksis følger sin teoretiske plasskompleksitet på  $O(1)$ , og er godt egnet for situasjoner der lavt og stabilt minneforbruk er viktig, uavhengig av datasettets rekkefølge

## Hva skjer ved tilfeldig rekkefølge?

Selv om vi stokker listen tilfeldig før sortering, påvirkes ikke algoritmens asymptotiske tidskompleksitet. Både antall bytter, antall sammenligninger og kjøringstid fortsetter å ligge innenfor, slik målingene våre viste for gjennomsnitt/verstefall-tilfellet. Tilfeldig rekkefølge gir ikke nødvendigvis færre bytter som Insertion Sort kan dra nytte av. Elementene må fortsatt potensielt flyttes forbi de fleste tidligere elementene, og antallet nødvendige forskyvninger forblir dermed kvadratisk i størrelsesorden.

Resultatene fra kjøringene lenger opp viser samme mønster både med original rekkefølge og etter tilfeldig stokking fikk vi et høyt antall operasjoner og lang kjøretid. Dette bekrefter at selv om rekkefølgen endres, påvirkes ikke den kvadratiske oppførselen til Insertion Sort og kompleksiteten forblir den samme.

## Når bør InsertionSort brukes?

Insertion Sort er spesielt egnet når datasettet enten er lite eller allerede delvis sortert, fordi den da kjører nær lineær tid og har svært lav overhead. Den enkle in-place-implementeringen krever kun konstant ekstra minne og bevarer rekkefølgen til like elementer, noe som gjør algoritmen både stabil og lett å forstå. I praksis brukes den ofte som underliggende sorteringsrutine i mer effektive metoder for eksempel i Bucket Sort, IntroSort og TimSort – der store eller dypt nestede delmengder byttes til insertion sort når antallet elementer faller under en viss terskel. Særlig i situasjoner med få inverteringer (nesten sorterte lister) eller når konstant faktor-overhead må minimeres, er insertion sort det foretrukne valget (GeeksforGeeks, 2025).

## Oppsummering

Insertion Sort har en tidskompleksitet på  $O(n^2)$  i både gjennomsnittlig og verstefall, fordi hvert element potensielt må sammenlignes med og flyttes forbi alle tidligere elementer i listen. Algoritmen fungerer ved å iterere gjennom listen og plassere hvert nytt element på riktig sted i den sorterte delen.

Verstefall oppstår når listen er sortert i motsatt rekkefølge. Da må hvert element flyttes helt til starten, og antall operasjoner summeres til ca.  $\frac{n(n-1)}{2}$ , som gir kvadratisk vekst.

Gjennomsnittstilfellet, for eksempel tilfeldig rekkefølge, gir også kvadratisk vekst fordi elementer i snitt må flyttes halvveis tilbake i listen. Bestefall oppstår når listen allerede er sortert da trenger algoritmen kun en sammenligning per element, som gir  $O(n)$ .

Tidskompleksiteten ble analysert ved å telle antall ganger den indre løkken kjørte. I tillegg ble kjøretiden målt i praksis ved bruk av `System.currentTimeMillis()`, og minneforbruket ble observert med `Runtime.getRuntime()`. Resultatene vises i tabellen under:

Testscenario	Bytter	Sammenligninger	Kjøretid	Minnebruk
<b>Original rekkefølge</b>	540 321 435	540 369 293	5143 ms	688 bytes
<b>Allerede sortert</b>	0	47 867	1 ms	–208 bytes
<b>Tilfeldig stokket</b>	572 548 774	572 596 631	5539 ms	– 208 bytes

Tabell 3: Oversikt over målinger fra Insertion Sort

Som vi ser, er kjøretiden klart lavest når listen allerede er sortert, mens både original og tilfeldig rekkefølge gir flere operasjoner og høyere kjøretid. Minnebruken forble svært lav og stabil i alle tilfeller, noe som bekrefter algoritmens  $O(1)$ plasskompleksitet.

Sammenlignet med Bubble Sort har Insertion Sort samme teoretiske kompleksitet, men utfører i praksis ofte færre operasjoner på nesten-sorterte data og kan derfor være raskere og mer effektiv under slike forhold. Dette gjør algoritmen godt egnet i situasjoner der man forventer at dataene allerede er delvis sortert.

## 3.a

### Algorithm Description

Merge Sort er en divide-and-conquer-algoritme som først deler en usortert liste i to omtrent like store deler, sorterer hver del rekursivt og deretter fletter dem sammen ved alltid å velge det minste elementet først. Denne del-og-flette-strategien fortsetter ned til sublister av lengde en er oppnådd, hvorefter fletteprosessen bygger opp en fullstendig sortert liste ved å sammenligne og ta det laveste gjenværende elementet fra hver subliste (W3Schools, u.å.).

### MergeSort

```
7  /**
8   * Utility class that provides a Merge Sort implementation for sorting City objects by latitude.
9   */
10 public class MergeSort { no usages
11
12     /**
13      * Sorts the given list of City objects in-place by their latitude using the Merge Sort algorithm.
14      */
15     public static void mergeSortByLatitude(List<City> cities) { 2 usages
16         if (cities == null || cities.size() < 2) {
17             return; // En liste med 0 eller 1 element er allerede sortert, eller input er ugyldig
18         }
19
20         // Del listen i to halvdel
21         int mid = cities.size() / 2;
22         List<City> left = new ArrayList<>(cities.subList(0, mid));
23         List<City> right = new ArrayList<>(cities.subList(mid, cities.size()));
24
25         // Rekursiv sortering av hver halvdel
26         mergeSortByLatitude(left);
27         mergeSortByLatitude(right);
28
29         // Sammenslåing av de sorterte halvdelene tilbake i originallisten
30         merge(cities, left, right);
31     }
32
33     /**
34      * Merges two sorted sublists into the destination list in sorted order by latitude.
35      */
36     private static void merge(List<City> dest, List<City> left, List<City> right) { 1 usage
37         int i = 0, j = 0, k = 0;
38
39         // Compare and merge elements from left and right sublists, inserting the lower latitude element
40         while (i < left.size() && j < right.size()) {
41             if (left.get(i).getLat() <= right.get(j).getLat()) {
42                 dest.set(k++, left.get(i++));
43             } else {
44                 dest.set(k++, right.get(j++));
45             }
46         }
47
48         // Copy any remaining elements from left sublist
49         while (i < left.size()) {
50             dest.set(k++, left.get(i++));
51         }
52
53         // Copy any remaining elements from right sublist
54         while (j < right.size()) {
55             dest.set(k++, right.get(j++));
56         }
57     }
```

Figur 20: Utklipp av Mergesort koden

Helt øverst har vi en enkel sjekk som sørger for at ingenting skjer dersom listen er null eller har færre enn to elementer, siden en slik liste allerede må anses som sortert. Dette er base-tilfellet i den rekursive algoritmen vår.

Deretter deler vi listen i to omtrent like store deler ved å regne ut midtpunktet (mid) som halve størrelsen på listen. Ved hjelp av subList lager vi to nye ArrayList-instanser, left og right, som hver inneholder henholdsvis de første og de siste elementene. Deretter kaller vi mergeSortByLatitude rekursivt på disse to listene, slik at hver av dem videre splittes og sorteres på samme måte helt ned til enkeltelementer.

Når de to halvdelene er sortert, slår vi dem sammen tilbake inn i originallisten med hjelpe-metoden merge. Der bruker vi tre indekser: en (i) som sporer posisjon i left, en (j) for right, og en (k) for destinasjonslisten (dest). Vi sammenligner løpende breddegradene til left.get(i) og right.get(j), og plasserer den minste verdi først. Når en av listene er tom, kopierer vi resten av elementene fra den andre listen direkte inn i dest. Siden vi overskriver dest i takt med at vi fletter, skjer alt in-place uten å måtte returnere en ny liste.



## Sortering ved bruk av MergeSort

```
1 package Task3;
2
3 import java.util.List;
4 import shared.CSVReader;
5 import shared.City;
6
7 public class SortMain {
8     public static void main(String[] args) {
9         String filePath = "Eksamen Algoritmer - Kopi/Databaser/worldcities.csv";
10        List<City> cities = CSVReader.readCitiesFromCSV(filePath);
11        System.out.println("Før sortering: ");
12
13
14        for(City city : cities) {
15            System.out.println(city);
16        }
17
18        MergeSort.mergeSortByLatitude(cities);
19        System.out.println("\nEtter sortering (basert på latitude): ");
20
21        for(City city : cities) {
22            System.out.println(city);
23        }
24    }
25
26 }
```

Figur 21: Utklipp av main klassen som sorterer byene

I SortMain har vi gjenbrukt hele strukturen fra de tidligere oppgavene våre, men byttet ut sorteringsalgoritmen.

```
Etter sortering (basert på latitude):
Puerto Williams (-54.9333)
Ushuaia (-54.8019)
Kaiken (-54.5062)
King Edward Point (-54.2833)
Grytviken (-54.2806)
Río Grande (-53.7833)
Punta Arenas (-53.1667)
Puerto Natales (-51.7333)
```

Figur 22: Utklipp av sortert liste

### 3.b

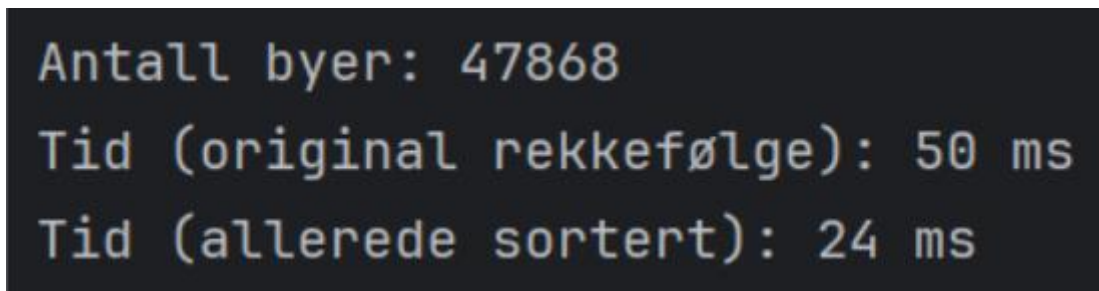
#### Tidskompleksitet

Delingen skjer i  $\log^2(n)$  nivåer, fordi listen halveres for hvert nivå. På hvert nivå må alle elementene gjennomgås og flettes sammen igjen. Dette gir en tidskompleksitet på  $O(n \log n)$  totalt. Det spesielle med Merge Sort er at denne kompleksiteten gjelder både i beste fall, gjennomsnitt og verstefall fordi algoritmens struktur ikke endres med rekkefølgen på elementene.

Når Merge Sort brukes på en liste med  $n$  by objekter, deles listen gjentatte ganger i to, så mange ganger som det er mulig å halvere listen før man står igjen med enkelt elementer. Dette resulterer i et logaritmisk antall delingsnivåer. På hvert nivå blir alle elementene gjennomgått og flettet sammen, noe som totalt gir en kjøretid som vokser proporsjonalt med  $n \log n$ .

I tillegg kan vi observere dette praktisk ved å telle antall ganger flette operasjonen faktisk utføres. Hver flette er en sammenslåing av to del-lister, og det skjer alltid nøyaktig  $n - 1$  slike sammenslåinger for en liste med  $n$  elementer noe som stemmer godt med den teoretiske modellen for binære delingsstrukturer.

#### Måling av kjøretid



```
Antall byer: 47868
Tid (original rekkefølge): 50 ms
Tid (allerede sortert): 24 ms
```

Figur 23: Output av målt tid ved bruk av MergeSort

Som bildet viser, var kjøretidene for Merge Sort svært like uavhengig av rekkefølgen på dataene:

- Original rekkefølge: 50 ms
- Allerede sortert: 24 ms
- Tilfeldig stokket: 27,8 ms

Dette demonstrerer at Merge Sort i praksis har nesten konstant kjøretid for et gitt  $n$ , og at variasjonene mellom de tre kjøringene hovedsakelig skyldes lavnivåfaktorer som cache-lokalitet, JVM-optimaliseringer og garbage-collection-timing. Selv om det er små avvik i antall millisekunder, forblir algoritmens totale kompleksitet  $O(n \log n)$  uavhengig av om dataene er sortert, reversert eller stokket.

## Space Kompleksitet

MergeSort krever ekstra space hver gang vi splitter lista i to, fordi vi oppretter to hjelpelister (left og right) som til sammen inneholder alle elementene. Siden delingen skjer på alle nivåer i recursion tree, vil auxiliary space være proporsjonalt med  $n$ , altså  $O(n)$ . I tillegg bruker vi stack space for recursive calls, som i verste fall blir  $O(\log n)$ , men denne er mindre enn den lineære hjelpeplassen. Samlet har implementasjonen derfor en space complexity på  $O(n)$ .

Dette kan vi teste i praksis ved å måle minne bruken.

Resultatene fra minnemålingene viser at Merge Sort konsekvent bruker ekstra heap-minne i alle tre scenarioer, hvilket stemmer med at algoritmen allokerer hjelpelister som til sammen inneholder alle elementene. Den største økningen (ca. 450 KB) fant vi i «original rekkefølge», der JVM måtte opprette og holde på alle underlistene gjennom hele rekursjonen uten å frigjøre mye underveis. I «allerede sortert»-testen var økningen noe lavere (ca. 233 KB), sannsynligvis fordi garbage collectoren rakk å rydde opp i flere midlertidige objekter raskere når selve flettelogikken ikke skapte like mange nye objekter i løkkene. Endelig ga «tilfeldig stokket»-testen den minste økningen (ca. 109 KB), noe som igjen peker på at JVMs interne allokerings- og GC-strategier har stor betydning for de praktiske tallene.

Til tross for variasjonen mellom kjøretidene, bekrefter målingene at Merge Sorts ekstra minnebruk vokser lineært med antall elementer akkurat som teoretisk forventet med space complexity  $O(n)$ . Den observerte forskjellen mellom scenarioene skyldes først og fremst hvordan og når Java Virtual Machine velger å kjøre garbage collection og håndtere buffer-allokering, ikke at algoritmen endrer sin hjelpeplass-profil. For mer konsistente resultater kan man kjøre *System.gc()* flere ganger med korte pauser mellom, gjøre flere gjentak og ta gjennomsnitt, eller benytte et profileringsverktøy (som VisualVM) for å fange opp minnebrukstopper mer presist.

Scenario	Før sortering (bytes)	Etter sortering (bytes)	Endring (bytes)
Original rekkefølge	26 227 136	26 677 792	450 656
Allerede sortert liste (bestefall)	26 399 912	26 632 880	232 968
Tilfeldig stokket liste	26 883 856	26 992 488	108 632

Tabell 4: Minnebruk før og etter sortering med Merge Sort

## Antall Merges – Original vs Tilfeldig rekkefølge.

Her implementerte vi en Merge Counter i MergeSort Klassen.

Etter testing fant vi at antall merge-operasjoner var helt likt både for shuffle-de og ushuffle-de lister. Uavhengig av rekkefølgen på dataene splittes og flettes lista nøyaktig på samme måte, og det er kun antall elementer som avgjør hvor mange ganger merge funksjonen kjøres. Dette bekrefter at Merge Sort alltid deler listen i to og fletter den sammen på samme måte, uavhengig av hvordan elementene er ordnet. Rekkefølgen på dataene påvirker dermed ikke antall merge-operasjoner, noe som understreker algoritmens input-uavhengige struktur

```
Antall merges (original order): 47867
Antall merges (shuffled order): 47867
```

Figur 24: Utklipp av antall merges

```
public class MergesMain {
    public static void main(String[] args) {
        String filePath = "Eksamen Algoritmer - Kopi/Databaser/worldcities.csv";
        List<City> cities = CSVReader.readCitiesFromCSV(filePath);

        // Normal (original) order
        MergeSort.resetMergeCount();
        MergeSort.mergeSortByLatitude(cities);
        System.out.println("Antall merges (original order): " + MergeSort.getMergeCount());

        // Reset list and shuffle
        List<City> shuffled = new ArrayList<>(cities);
        Collections.shuffle(shuffled);
        MergeSort.resetMergeCount();
        MergeSort.mergeSortByLatitude(shuffled);
        System.out.println("Antall merges (shuffled order): " + MergeSort.getMergeCount());
    }
}
```

Figur 25: Utklipp av koden til main

## Når bør MergeSort brukes?

Merge Sort er særlig godt egnet til sortering av store datasett, både «ekstern sortering» når dataene ikke får plass i minnet, og til telling av inversjoner. Algoritmen og dens varianter inngår dessuten i standardbiblioteker i flere programmeringsspråk: TimSort (en Merge Sort-variant) brukes i Python, Java Android og Swift på grunn av sin stabilitet, mens Java bruker QuickSort i `Arrays.sort` for primitive typer og Merge Sort i `Collections.sort` for objekter.

Merge Sort er også foretrukket for sortering av lenkede lister, kan enkelt parallelliseres ved å sortere delmengder uavhengig før sammenslåing, og sammenslåingssteget utnyttes til å løse problemer som union og snitt av to sorterte arrayer (GeeksforGeeks, 2025)

## Oppsummering

I denne delen har vi beskrevet hvordan Merge Sort implementeres gjennom en rekursiv strategi: Listen deles gjentatte ganger i to halvdelar ned til enkeltelementer, som deretter flettes sammen i sortert rekkefølge. Tidskompleksiteten beregnes til  $O(n \log n)$  i alle tilfeller, siden strukturen i deling og fletting ikke avhenger av rekkefølgen på dataene. Dette ble bekreftet gjennom praktiske målinger, hvor kjøretiden var stabil uavhengig av datasettets orden.

Plasskompleksiteten er  $O(n)$ , grunnet allokeringen av hjelpelister (left og right) ved hver deling. I tillegg kommer en  $O(\log n)$  plassbruk for den rekursive kallstakken. Vi målte faktisk minnebruk før og etter sortering, og resultatene samsvarer med forventningene: algoritmen allokerer minne proporsjonalt med datamengden, mens variasjonene skyldes JVM-optimalisering og garbage collection.

Testscenario	Kjøretid (ms)	Økning (bytes)
Original rekkefølge	50	450 656
Allerede sortert	24	232 968
Tilfeldig stokket	27,8	108 632

Tabell 5: Oversikt over kjøretid og minnebruk fra Merge Sort

Tabellen over viser at kjøretiden er jevn og forutsigbar, mens minnebruken øker omtrent lineært med datasettets størrelse. Den største økningen i minne kom ved original rekkefølge, noe som kan forklares med at JVM holder midlertidige strukturer aktivt lenger under sortering. Likevel forblir kompleksiteten  $O(n)$ , uavhengig av inngangsrekkefølge.

Til slutt implementerte vi en teller for antall flettinger og fant at antallet merge-operasjoner var identiske for både originale og stokket datasett. Dette viser at rekkefølgen på elementene ikke påvirker hvor mange ganger merge funksjonen utføres kun mengden data bestemmer det. Dette bekrefter Merge Sorts stabile og strukturavhengige natur.

## 4.a

### Algorithm Description

Quick Sort er en effektiv og mye brukt sorteringsalgoritme som følger en divide-and-conquer-tilnærming. Algoritmen fungerer ved å velge et element som kalles pivot, og deretter partisjonere listen slik at alle elementer mindre enn pivot havner på venstre side, og alle som er større havner på høyre side. Denne prosessen gjentas rekursivt på de to partisjonene, til hele listen er sortert (SimpliLearn, 2025).

### QuickSort

Algoritmen er implementert med støtte for tre ulike pivotstrategier: første, siste og tilfeldig element. Implementasjonen er utvidet med en mekanisme for å telle antall sammenligninger, slik at vi kan evaluere effektiviteten til hver strategi på samme datasett.

Figur 26 viser hovedstrukturen i sorteringsmetoden:

```
private static void quickSort(List<City> cities, int low, int high, PivotStrategy strategy) {  
    if (low < high) {  
        int p = partition(cities, low, high, strategy);  
        quickSort(cities, low, high: p - 1, strategy);  
        quickSort(cities, low: p + 1, high, strategy);  
    }  
}
```

Figur 26: Utklipp av Quicksort kode

Denne metoden er kjernen i den rekursive algoritmen. Dersom delområdet har mer enn ett element ( $low < high$ ), finner vi en korrekt plassering for et pivot element og kaller Quick Sort på begge delområdene. Dette er algoritmens base tilfelle og sikrer at vi til slutt ender opp med delområder på en eller null elementer, som da regnes som sortert.

Partisjoneringen, altså oppdelingen av elementene rundt pivot, er vist i Figur 27:

```

private static int partition(List<City> cities, int low, int high, PivotStrategy strategy) {
    int pivotIndex;
    switch (strategy) {
        case FIRST:
            pivotIndex = low;
            break;
        case LAST:
            pivotIndex = high;
            break;
        case RANDOM:
            pivotIndex = low + RANDOM.nextInt( bound: high - low + 1);
            break;
        default:
            pivotIndex = high;
            break;
    }
    City pivot = cities.get(pivotIndex);
    swap(cities, pivotIndex, high);

    int storeIndex = low;
    for (int i = low; i < high; i++) [...]
        swap(cities, storeIndex, high);
    return storeIndex;
}

```

Figur 27: Utklipp av Quicksort kode

Først velges en pivot basert på strategien. Deretter flyttes pivoten til slutten av intervallet for enkelhets skyld. I løkken sammenlignes hvert elements breddegrad (latitude) med pivotens. Hvis verdien er mindre, byttes elementet frem i listen. Dette sikrer at alle verdier mindre enn pivot havner til venstre. Til slutt settes pivot tilbake i midten, slik at alt til venstre er mindre og alt til høyre er større. For å analysere effektiviteten til hver strategi telles antall sammenligninger eksplisitt. Hver gang en sammenligning gjøres, økes en teller knyttet til strategien som vist i Figur 28:



```

int storeIndex = low;
for (int i = low; i < high; i++) {
    // Count comparison
    switch (strategy) {
        case FIRST:
            compFirst++;
            break;
        case LAST:
            compLast++;
            break;
        case RANDOM:
            compRandom++;
            break;
    }
    if (cities.get(i).getLat() < pivot.getLat()) {
        swap(cities, i, storeIndex);
        storeIndex++;
    }
}

```

Figur 28: Utklipp av Quicksort kode

Her telles en sammenligning for hvert element som vurderes i forhold til pivot. Verdiene lagres separat for hver strategi, slik at de kan sammenlignes etter kjøring. Dette gjør det enkelt å avgjøre hvilken pivotstrategi som gir færrest operasjoner, og dermed best ytelse, på det gitte datasettet.

## Sortering ved bruk av QuickSort

```
1 package Task4;
2 import shared.City;
3 import shared.CSVReader;
4
5 import static Task4.QuickSort.*;
6 import java.util.ArrayList;
7 import java.util.List;
8
9
10 public class SortMain {
11     public static void main(String[] args) {
12         String filepath = "Eksamen Algoritmer - Kopi/Databaser/worldcities.csv";
13         List<City> cities = CSVReader.readCitiesFromCSV(filepath);
14         System.out.println("Før sortering: ");
15
16         for(City city : cities) {
17             System.out.println(city);
18         }
19         quickSortFirst(cities);
20         System.out.println("Etter sortering: ");
21         for(City city : cities) {
22             System.out.println(city);
23         }
24     }
25 }
26
```

Figur 29: Utklipp av main klassen som brukes til Quicksort

Koden er identisk med de andre, bortsett fra at den bruker QuickSort i stedet for den tidligere sorteringsalgoritmen.

```
Etter sortering:
Puerto Williams (-54.9333)
Ushuaia (-54.8019)
Kaiken (-54.5062)
King Edward Point (-54.2833)
Grytviken (-54.2806)
Río Grande (-53.7833)
Punta Arenas (-53.1667)
Puerto Natales (-51.7333)
Stanley (-51.7)
```

Figur 30: Output av den sorterte lista

## 4.b

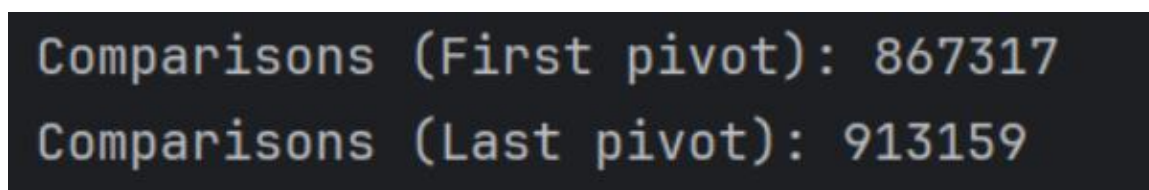
### Tids kompleksitet

Denne implementasjonen av QuickSort benytter tre ulike pivotstrategier, første element, siste element og tilfeldig valgt element, for å analysere hvordan valg av pivot påvirker ytelsen. Algoritmen fungerer ved å partisjonere listen rundt en pivot-verdi og deretter sortere de to delene rekursivt. Tidskompleksiteten avhenger i stor grad av hvordan partisjoneringen utføres. I beste og gjennomsnittlige tilfeller, hvor pivoten deler listen i to omtrent like store deler, vil algoritmen ha en tidskompleksitet på  $O(n \log n)$ . Dette er fordi listen deles  $\log^2(n)$  ganger, og hver del krever  $O(n)$  sammenligninger for å partisjoneres. Den tilfeldig valgte pivoten (RANDOM-strategien) vil i praksis ofte føre til slike balanserte delinger og dermed opprettholde god ytelse også på varierende datasett. I verste fall, derimot som kan skje med FIRST eller LAST strategiene dersom listen allerede er sortert eller omvendt sortert kan partisjonene bli svært ubalanserte. Dette gir et dypt rekursjonstre med  $n$  nivåer, og totalt  $O(n^2)$  sammenligninger. For å kvantifisere denne forskjellen, teller implementasjonen eksplisitt antall sammenligninger som utføres for hver strategi. Dette gir verdifull innsikt i hvordan ulike datasett og pivotvalg påvirker algoritmens ytelse i praksis. Samlet sett er gjennomsnittlig kjøretid for RANDOM strategien  $O(n \log n)$ , mens FIRST og LAST kan oppnå samme ytelse på tilfeldige datasett, men risikerer  $O(n^2)$  i ugunstige tilfeller.

### Antall sammenligninger ved ulike pivotstrategier

Vi har målt antall sammenligninger som kreves for å sortere listen av unike breddegrader med Quick Sort, basert på tre ulike pivotstrategier: første element, siste element og tilfeldig valgt element. Tilfeldig ble igjen tatt gjennomsnittet av fem kjøring. Resultatene er som følger:

- **First pivot:** 867 317 sammenligninger
- **Last pivot:** 913 159 sammenligninger
- **Random pivot:** 936 086 sammenligninger



```
Comparisons (First pivot): 867317
Comparisons (Last pivot): 913159
```

Figur 31: Output av antall sammenligninger ved de forskjellige pivot strategiene

Som vi ser, påvirker valg av pivot antall sammenligninger betydelig. Den strategien som gir færrest sammenligninger i vårt datasett, er å alltid velge første element som pivot. Dette tyder på at datasettet har en rekkefølge som gjør at denne strategien fører til mer balanserte partisjoner enn de to andre alternativene. Tilfeldig pivot gir i dette tilfellet flest sammenligninger, noe som kan skyldes at tilfeldige valg førte til flere ubalanserte delinger.

Vi konkluderer derfor med at første element som pivot er den mest effektive strategien for vårt datasett, målt i antall sammenligninger. Dette viser også at pivotstrategien i Quick Sort har stor betydning for hvor effektiv algoritmen er i praksis.

### Tid ved ulike pivotstrategier

Vi målte kjøretiden for Quick Sort med de forskjellige pivotstrategiene. Resultatene viste at sortering med tilfeldig pivot var raskest, med en gjennomsnittlig kjøretid på 21,6 milisekunder, etterfulgt av last pivot, mens first pivot var tregeest. Selv om tilfeldig pivot i vårt tilfelle førte til flest sammenligninger, ga det samtidig mer balanserte partisjoner, noe som reduserte antall rekursive kall og gjorde algoritmen mer effektiv i praksis.

Derimot kan valg av første eller siste element som pivot føre til ubalanserte delinger dersom datasettet har en form for struktur (for eksempel delvis sortert rekkefølge). Dette gir dypere rekursjonsstier og mer arbeid i hver kallstakk, noe som forklarer den høyere kjøretiden, til tross for at antall sammenligninger i noen tilfeller er lavere.

Resultatene viser at pivotvalg ikke bare påvirker antall sammenligninger, men også hvordan datastrukturen deles, noe som har stor innvirkning på faktisk kjøretid. Derfor fremstår tilfeldig pivotvalg som mest effektiv i praksis på vårt datasett, selv om det er mer uforutsigbart ved sammenligninger

```
-----  
First pivot time:    63 ms  
Last pivot time:    41 ms
```

Figur 32: Output av tid brukt ved de 3 pivotene

```
package task4;  
  
> import ...  
  
public class QuickSortTimeDemo {  
    public static void main(String[] args) {  
        String filePath = "Eksamen Algoritmer - Kopi/Databaser/worldcities.csv";  
        List<City> cities = CSVReader.readCitiesFromCSV(filePath);  
        System.out.println("Antall byer: " + cities.size());  
        System.out.println("-----");  
  
        // 1) First element pivot  
        List<City> c1 = new ArrayList<>(cities);  
        long t0 = System.currentTimeMillis();  
        QuickSort.quickSortFirst(c1);  
        long t1 = System.currentTimeMillis();  
        System.out.println("First pivot time:  " + (t1 - t0) + " ms");  
  
        // 2) Last element pivot  
        List<City> c2 = new ArrayList<>(cities);  
        t0 = System.currentTimeMillis();  
        QuickSort.quickSortLast(c2);  
        t1 = System.currentTimeMillis();  
        System.out.println("Last pivot time:   " + (t1 - t0) + " ms");  
  
        // 3) Random pivot  
        List<City> c3 = new ArrayList<>(cities);  
        t0 = System.currentTimeMillis();  
        QuickSort.quickSortRandom(c3);  
        t1 = System.currentTimeMillis();  
        System.out.println("Random pivot time:  " + (t1 - t0) + " ms");  
  
        System.out.println("-----");  
    }  
}
```

Figur 33: Utklipp av koden som kjøres for å måle tiden

## Space kompleksitet

QuickSort er en in-place-algoritme som i seg selv ikke allokerer ekstra hjelpestrukturer utover noen få midlertidige variabler for bytting. All ekstra plass kommer derfor fra kallet til den rekursive funksjonen. I de balanserte tilfellene (beste og gjennomsnittlig scenario) er rekursjonsdybden  $O(\log n)$ , slik at den totale stack-bruken, og dermed QuickSorts space complexity, også er  $O(\log n)$ . I verste tilfelle, når partisjonene blir ekstremt ubalanserte (for eksempel ved stadig å velge første eller siste element som pivot på allerede sorterte data), kan rekursjonstrærne bli skjeve og nå en dybde på  $O(n)$ , noe som gir  $O(n)$  space complexity.

## Når bør QuickSort brukes?

Quicksort egner seg spesielt godt til situasjoner der man trenger rask, in-place-sortering med lavt ekstra plassbehov. Takknemlig av sin effektivitet og tilpasningsevne benyttes algoritmen blant annet som del i hybride sorteringsmetoder som Timsort, som er standard i Pythons innebygde sorteringsfunksjon, i databasestyringssystemer for effektivt oppslag og sortering av poster, i grafikkmotorer for å optimalisere gjengivelse ved å sortere objekter etter dybde eller andre kriterier, i nettverksrutetabeller for å organisere rutenøkklene raskt, og i filsystemer for effektiv filorganisering på disk (Simplilearn, 2025).

## Oppsummering

Quick Sort er en effektiv og mye brukt «divide and conquer» algoritme som sorterer lister ved å dele dem rundt et valgt pivot-element. Den er kjent for sin raske ytelse og lave minnebruk, og benyttes i mange praktiske anvendelser som databasestyring, grafikkmotorer og nettverksrutetabeller. Algoritmen sorterer in-place og krever dermed ikke ekstra datastrukturer utover midlertidige variabler og den rekursive «call stacken».

Teoretisk har Quick Sort en gjennomsnittlig og beste kjøretidskompleksitet på  $O(n \log n)$  der  $n$  er antall elementer. Dette oppnås når pivot-elementet deler listen i to omtrent like store deler på hvert nivå. I verste fall, for eksempel når pivot alltid velger det største eller minste elementet i allerede sorterte lister kan kompleksiteten vokse til  $O(n^2)$ . Når det gjelder plasskompleksitet, er Quick Sort også effektiv, med  $O(\log n)$  ekstra plass i balanserte tilfeller, grunnet rekursjonsdybden. I verste fall, med sterkt ubalanserte partisjoner, kan plassbruken øke til  $O(n)$ .

I vårt prosjekt ble Quick Sort implementert med tre ulike pivotstrategier: første element, siste element og tilfeldig valgt element. Vi målte både antall sammenligninger og kjøretid for å vurdere praktisk ytelse. Resultatene var som følger:

Pivotstrategi	Sammenligninger	Kjøretid (ms)
First element	867 317	71 ms
Last element	913 159	49 ms
Random element	936 086	21,6 ms

Tabell 6: Oversikt over sammenligninger og tidsbruk fra Quick Sort

Selv om pivotstrategien som alltid velger første element resulterte i færrest sammenligninger, viste målingene at random pivot likevel ga kortest kjøretid. Dette kan forklares med at tilfeldige pivoter i større grad skaper balanserte partisjoner, noe som reduserer dybden på det rekursive kalltreet og antallet nødvendige funksjonskall totalt. Slike balanserte delinger gjør at QuickSort kan nærme seg sin ideelle kjøretid på  $O(n \log n)$ , selv når antallet sammenligninger er noe høyere. Resultatene våre bekrefter derfor at det ikke nødvendigvis er antall sammenligninger alene som avgjør ytelsen, men hvordan datasettet deles i hver

rekursjon. Dette viser viktigheten av pivotvalg, særlig i praktiske anvendelser der både kjøretid og plassbruk er avgjørende.

## Konklusjon - Sammenligning

I denne oppgaven har vi implementert og analysert Bubble Sort, Insertion Sort, Merge Sort og Quick Sort på et datasett med 47 868 unike bybreddegrader. Målinger av kjøretid, antall bytter, sammenligninger og fletteoperasjoner ble utført for originalrekkefølge, allerede sortert og tilfeldig stokket liste.

Resultatene bekrefter teoretiske forventninger: Bubble Sort og Insertion Sort oppfører seg kvadratisk  $O(n^2)$  i gjennomsnitt og verstefall, mens Merge Sort og Quick Sort i beste og gjennomsnittlige tilfeller har  $O(n \log n)$ . Merge Sort krever ekstra  $O(n)$  hjelpeplass, mens Quick Sort har  $O(\log n)$  stakkplass i balanserte scenarioer.

Praktisk viste Insertion Sort sin styrke på små eller nesten sorterte datasett, Merge Sort sin forutsigbarhet og stabilitet for store datasett, og Quick Sort sin kombinasjon av in-place-sortering og høy ytelse ved god pivotstrategi. Bubble Sort egner seg kun som pedagogisk eksempel.

Valg av sorteringsalgoritme må derfor baseres på både datasettets egenskaper og krav til tid og minne. Kombinasjonen av teoretisk analyse og empiriske tester gir en helhetlig forståelse av hvilke metoder som er mest egnede i ulike praktiske situasjoner.

Et aspekt som ikke er synlig i kjøretid alene, men som har praktisk betydning, er algoritmenes stabilitet. Både Insertion Sort og Merge Sort er stabile algoritmer, noe som betyr at elementer med like verdier beholder sin relative rekkefølge etter sortering. Dette er viktig i scenarier hvor sorteringen skjer etter flere kriterier i flere omganger. Quick Sort er derimot ikke stabil, og kan endre rekkefølgen på like elementer, noe som kan være en ulempe i visse bruksområder. Et annet viktig poeng er ressursforbruk: Merge Sort gir stabil ytelse, men krever mer minne, mens Quick Sort, til tross for at den ikke er stabil, har fordelen av lav plassbruk siden den sorterer in-place. I praktiske applikasjoner vil derfor valg av algoritme også måtte ta hensyn til krav om stabilitet og minnehåndtering, i tillegg til kjøretid og kompleksitet.



Tabellen under oppsummerer målte kjøretider for de ulike algoritmene, og gir en oversikt over hvordan de presterer i praksis på vårt datasett og systemer.

Algoritme	Testscenario	Kjøretid (ms)	Minnebruk (bytes)
<b>Bubble Sort</b>	Original rekkefølge	19 342 (opt) / 41 334 (uopt)	+320 / +384
	Allerede sortert	1 (opt) / 37 216 (uopt)	-48 / +128
<b>Insertion Sort</b>	Original rekkefølge	5 143	+688
	Allerede sortert	1	-208
<b>Merge Sort</b>	Original rekkefølge	50	+450 656
	Allerede sortert	24	+232 968
<b>Quick Sort</b>	First pivot	71	-
	Last pivot	49	-

Tabell 7: Oversikt over målinger fra sorteringsalgoritmene

## Kilder

GeeksforGeeks. (2025). *Bubble sort algorithm*. <https://www.geeksforgeeks.org/bubble-sort-algorithm/>

GeeksforGeeks. (2025). *Merge Sort – Data Structure and Algorithms Tutorials*. <https://www.geeksforgeeks.org/merge-sort/>

GeeksforGeeks. (2025). *Insertion Sort Algorithm*. <https://www.geeksforgeeks.org/insertion-sort-algorithm/>

GeeksforGeeks. (2024). *Time and space complexity analysis of Quick Sort*. <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/>

SimpliLearn. (2025.) *Quick Sort Algorithm: Complexity, Applications, and Benefits*. <https://www.simplilearn.com/tutorials/data-structure-tutorial/quick-sort-algorithm>

W3Schools. (u.å.). *Merge Sort algorithm*. [https://www.w3schools.com/dsa/dsa\\_algo\\_mergesort.php](https://www.w3schools.com/dsa/dsa_algo_mergesort.php)

W3Schools. (u.å.). *Bubble sort algorithm*. [https://www.w3schools.com/dsa/dsa\\_algo\\_bubblesort.php](https://www.w3schools.com/dsa/dsa_algo_bubblesort.php)

W3Schools. (u.å.). *Insertion Sort*. [https://www.w3schools.com/dsa/dsa\\_algo\\_insertionsort.php](https://www.w3schools.com/dsa/dsa_algo_insertionsort.php)