Øving 3

Jeg valgte deloppgave 3, som handler om å forbedre quicksort-algoritmen ved å hoppe over deltabeller der de avgrensende pivot-elementene er like.

I programmet har jeg programmert to algoritmer. Den ene algoritmen er en standard quicksort-algoritme. Den andre er en kopi av den første, men med den forbedringen som er beskrevet i oppgaven.

**Merk!** *Resultatene i denne rapporten har blitt produsert av å kompilere programmet på Windows med VS2022-kompilatoren, med C-standard 14.*

# Hva er det programmet printer ut?

Programmet printer først ut en rekke tidsmålinger for begge algoritmene. Dette er testene i riktig rekkefølge:

* Sortering av tilfeldige verdier
* Sortering av tilfeldige verdier med mange duplikater
* Sortering av ferdigsortert data
* Sortering av ferdigsortert data med mange duplikater
* Kompleksitets-måling av den gitte algoritmen. Altså at algoritmen påkalles flere ganger med en n-verdi som øker med en faktor på 10.

Deretter printer programmet ut en lineær målserie mellom 3,000,000 og 10,000,000 for hver av algoritmene. Dette brukes til å estimere en graf for hver algoritmes kompleksitet. Input-dataene som sorteres er tabeller med tilfeldige verdier.

# Output for godkjenningskravene til deloppgave 3

Dette er et resultat av målingene fra når jeg har kjørt programmet selv.

1. 0.542717s – vanlig quicksort på mange duplikater.
2. 0.354419s – forbedret quicksort på mange duplikater.
3. 0.624043s – forbedret quicksort på tilfeldige tall.
   1. 0.728514s – vanlig quicksort på tilfeldige tall.

Oppgaveteksten hinter til at forandringen vil gjøre algoritmen tregere for data som kjører Som vi kan se, lider ikke den forbedrede algoritmen av høyere kjøretid på vanlige sorteringer av tilfeldige tall. Det er fordi jeg fant ut at det er mer effektivt å videreføre tabell-kantene til hver rekursjon, enn å lete etter min/max-verdier for hver eneste deltabell. Den negative siden ved dette er at forbedringen aldri vil kjøre for deltabeller som ligger på kantene til den originale tabellen. Men utfra måleresultatene, ser vi at dette ikke er et stort problem.

# Hvordan testes algoritmene?

Algoritmene testes hver eneste gang de blir påkalt, uansett situasjon. Dette gjøres ved at programmet alltid påkaller en template funksjon ‘test\_sort\_function’. Den tar en sorteringsalgoritme som input, gjennomfører checksum-validering (med std::accumulate før og etter sortering), og sjekker at tabellen faktisk er sortert (via std::is\_sorted etter sortering).

Bruken av STL-funksjoner er for å unngå feil i valideringsmetode. Det fjerner en kritisk fallgruve der det er feil i valideringsmetodene jeg lager selv.

# Algoritmene

Dette er den vanlige quicksort algoritmen jeg har laget:

A screen shot of a computer code

Description automatically generated

Dette er den forbedrede versjonen av algoritmen. Jeg har markert de viktigste forskjellene.

Måten jeg har implementert forbedringen, er ved å lage en inner-funksjon som den offentlige funksjonen påkaller. Den offentlige funksjonen fyller inn ‘bound\_begin’ og ‘bound\_end’, og innerfunksjonen viderefører disse verdiene. På denne måten har alltid innerfunksjonen direkte tilgang på original-tabellens grenser, og kan sjekke om den nåværende del-tabellen grenser

A screenshot of a computer program

Description automatically generated

# Resultat

Her er en linær måleserie for begge algoritmene. Begge algoritmene har sortert

|  |  |  |
| --- | --- | --- |
| N-verdi | Quicksort (s) | Quicksort med forbedring (s) |
| 3000000 | 0.210071 | 0.189333 |
| 3250000 | 0.226494 | 0.203236 |
| 3500000 | 0.245356 | 0.220587 |
| 3750000 | 0.260281 | 0.232495 |
| 4000000 | 0.281686 | 0.245342 |
| 4250000 | 0.295004 | 0.260181 |
| 4500000 | 0.316466 | 0.280357 |
| 4750000 | 0.33577 | 0.298179 |
| 5000000 | 0.349851 | 0.310329 |
| 5250000 | 0.366912 | 0.328523 |
| 5500000 | 0.381151 | 0.333745 |
| 5750000 | 0.408172 | 0.357405 |
| 6000000 | 0.422121 | 0.368845 |
| 6250000 | 0.435679 | 0.379885 |
| 6500000 | 0.462744 | 0.402744 |
| 6750000 | 0.480546000000001 | 0.420034 |
| 7000000 | 0.499253000000001 | 0.433949 |
| 7250000 | 0.521472000000001 | 0.45196 |
| 7500000 | 0.530699000000001 | 0.456754 |
| 7750000 | 0.550377000000001 | 0.475657000000001 |
| 8000000 | 0.571091000000001 | 0.493472000000001 |
| 8250000 | 0.585284000000001 | 0.505308000000001 |
| 8500000 | 0.602384000000001 | 0.520224000000001 |
| 8750000 | 0.627852000000001 | 0.540457000000001 |
| 9000000 | 0.641316000000001 | 0.558360000000001 |
| 9250000 | 0.667941000000001 | 0.582934000000001 |
| 9500000 | 0.678741000000001 | 0.585340000000001 |
| 9750000 | 0.694589000000001 | 0.594657000000001 |
| 10000000 | 0.710804000000001 | 0.609456000000001 |

Når vi plotter disse verdiene i Geogebra, og bruker funksjonen «fit» med som grunn-funksjon, får vi disse to grafene. Grafen med lavest stigning tilhører den forbedrede quicksort-algoritmen. Her kan vi se tydeligere at den forbedrede algoritmen har en bedre utvikling enn den vanlige quicksort-algoritmen.

A graph with black dots and blue lines

Description automatically generated