Øving 3, algoritmer og datastrukturer

Her er det to oppgaver å velge i, det holder å gjøre én av dem.

Uansett oppgave, må dere ha med et par tester for korrekt sortering.

Første sjekk er en sjekksum, for å unngå datatap. Beregn sum av alle tallene i tabellen, før sortering. Beregn samme sum etter at tabellen er sortert. De to summene må stemme, ellers har har koden en feil hvor tabellposisjoner overskrives og går tapt.

Andre sjekk er rekkefølgen på tallene. Sjekk at tabell[i+1] >= tabell[i] for alle i fra 0 til tabell.length-2, når sorteringen er ferdig. Dette må stemme, ellers har programmet sortert feil.

Et program som ikke har slike tester, blir ikke godkjent.

Alternativ 1, sammenligne ulike typer quicksort

Oppgaven er å sammenligne vanlig quicksort som bruker ett delingstall, med varianten som bruker to delingstall (dual pivot quicksort).

Implementer begge sorteringsalgoritmene, som metoder i samme testprogram. Dere kan f.eks. bruke min quicksort fra læreboka, og dual pivot quicksort basert på `https://www.geeksforgeeks.org/dual-pivot-quicksort/`. NB! Sorteringen fra geeksforgeeks har en svakhet. Den velger første arrayposisjon som venstre pivot (delingstall), og siste posisjon som høyre pivot. Hvis tabellen er sortert fra før, vil alle tallene havne i det midterste intervallet. Da blir det ekstrem skjevdeling, og $O\left(n^2\right)$. Dette er lett å fikse, ved å først bytte tallet i posisjon arr[low] med tallet i posisjonen arr[low+(high-low)/3], og tilsvarende bytte tallet i posisjon arr[high] med arr[high-(high-low)/3].

Bruk tidtaking, og finn ut hvilken sorteringsalgoritme som fungerer best på tabeller med 10–100 millioner tall. Det er ingen fasit på dette, det vil avhenge av pc, programmeringsspråk og implementasjonsdetaljer. De som bruker interpreterte språk som Python, kan bruke mindre datamengder som 100 000 tall.

Ta tiden på sortering av:

- Tabell med helt tilfeldige data
- Tabell med mange duplikater, f.eks. hvor annenhvert element er likt. Gode sorteringsalgoritmer er raskere på slike data, dårlige kan bli tregere.

- Tabell som er sortert fra før, da dette er et vanlig spesialtilfelle. Hvis det blir noen forskjell, bør dette gå raskere, ikke tregere.

De som vil ha ekstra utfordringer, må gjerne prøve ut ulike triks for å speede opp sorteringen enda mer. F.eks. trikset med å bruke innsettingssortering på tilstrekkelig små intervaller. (Hvor små slike intervaller bør være før man bruker innsettingssortering, er et interessant spørsmål i seg selv, det kan finnes med eksperimentering.)

Krav for godkjenning av alternativ 1

- Et program som implementerer to varianter av quicksort (single-pivot og dual-pivot), og tidsmålinger.
- Rapport med tidsmålinger som viser hvilken variant som er raskest for:
  - tabell med tilfeldige tall
  - tabell med mange duplikater
  - tabell som er sortert fra før
- I alle tilfellene må sorteringene bestå sjekksumtesten og rekkefølgetesten.

Alternativ 2, Shellsort

Som nevnt på side 55 i læreboka, fins det flere varianter av shellsort. Forskjellene går på hvordan man forandrer variabelen `s` på linje 13 i algoritmen. For enkelte varianter har *ingen* klart å beregne kompleksiteten. Men vi kan eksperimentere og måle i stedet.

1. Prøv ut ulike måter å forandre `s` på. Det går f.eks. an å dele med andre tall enn 2,2. Mange andre sekvenser er også mulige, men for å få sortert korrekt **må** `s` bli 1 til slutt. Finn ut hva som blir raskest på en tabell med mange millioner tall.

   Å dele med tall «for nære 1» vil gjøre programmet tregt, fordi det da blir veldig mange kjøringer. Å dele med for store tall, vil gjøre programmet tregt fordi det da blir for likt innsettingsortering som har kvadratisk kjøretid.

   Denne deloppgaven går altså ut på å finne et delingstall som er bra, ved at både større og mindre tall gir dårligere resultat.

2. Forsøk å måle kompleksiteten for den beste varianten av shellsort dere fant i forrige deloppgave. Gjør tidsmålinger på datasett av mange størrelser. Regn med at kompleksiteten er på formen $O\left(n^x\right)$, og finn hvilken x som ligger nærmest resultatene deres.

Tips: Slik beregner vi kompleksitet

Eksempel på hvordan man gjør beregninger som i deloppgave 2:

$n_1 = 500\,000$ elementer: $t_1 = 12\,\mathrm{s}$. $n_2 = 2\,500\,000$ elementer: $t_2 = 1500\,\mathrm{s}$. Her ser vi at datamengden ble femdoblet, fordi $\frac{n_2}{n_1} = \frac{2\,500\,000}{500\,000} = 5$. Når vi antar en kompleksitet på formen $O\left(n^x\right)$, betyr det at tidsforbruket øker med 5^x når datamengden femdobles. Vi får altså ligningen $5^x = \frac{1500}{12}$, altså $5^x = 125$. I dette tilfellet er det ikke vanskelig å se hva x blir, men så lett er det som regel ikke. Vi kan fortsette med å ta logaritmen på begge sider, og får da $\log 5^x = \log 125 \Longrightarrow x \log 5 = \log 125 \Longrightarrow x = \frac{\log 125}{\log 5} = 3$.

Generell formel, som forhåpentligvis er grei å forstå etter dette eksempelet: $x = \frac{\log \frac{t_2}{t_1}}{\log \frac{n_2}{n_1}}$.

Med *fler enn to* målinger finner en flere slike *x*-verdier, som forhåpentligvis er temmelig like når n_1 og n_2 blir store nok. Det kan være *store* avvik for små n-verdier. Derfor, bruk store nok datasett til at x varierer lite. I følge teorien kan ikke x bli eksakt lik 1, så beregn x med et par desimaler.

For å få god tidtaking på en algoritme som nesten er lineær, får dere sannsynligvis bruk for tabeller med titusener av elementer eller mer. Gode implementasjoner på raske maskiner kan trenge tabeller med flere millioner tall.

Tips: forstå siste linja i shellsort, og unngå trøbbel

En del har ikke sett programkonstruksjonen `?:` før. Siste linje i shellsort ser slik ut:

```
s = (s == 2) ? 1 : (int)(s / 2.2);
```

For de som ikke har sett dette før, så er det bare en kortere måte å gjøre dette:

```
if (s == 2) s = 1; else s = (int)(s / 2.2);
```

Når dere gjør om på dette, så husk at denne linja har hele tre viktige formål:

1. Variabelen `s` skal gå nedover, hver gang.
2. `s` må ende opp på 1 etterhvert, ellers blir sorteringen feil. Hvis dere deler med større tall enn 3, kan `s` gå rett fra 3 til 0. Unngå det! (Det unngås ved å tilpasse testen, *ikke* ved å holde *s* under 3 for enhver pris.) Det er også derfor dere må teste resultatet av sorteringen; med en feil her kan en shellsort som hopper over det siste steget bli veldig kjapp — men med feil resultat.
3. Etter at `s` har vært 1, må `s` videre ned til 0. Ellers blir ikke shellsort ferdig.

Hvis tallet vi deler på ligger i en variabel kalt *deletall,* kan vi bruke en linje som dette for å sikre at det går bra også for deletall større enn 3:

```
if (s > 1 && s < deletall) s = 1; else s = (int)(s / deletall);
```

Pass på å prøve noen deletall som ikke er for «runde». Tidligere forsøk antyder at en graf for kjøretid som funksjon av deletall, er en bue med et bunnpunkt. Men *hele* tall ser ut til å fungere spesielt dårlig, og blir små topper i denne grafen. Vi vet ikke hvorfor, men det er mulig at å dele med heltall som 3.0 eller 4.0 fører til at det stort sett er de samme elementene som sammenlignes, og dermed blir andre tall stående innbyrdes feil ganske lenge. Kanhende det lønner seg med noen veldig u-runde tall, som e, π, $\sqrt{5}$ og lignende?

Krav for godkjenning av alternativ 2

- Et program som prøver ut shellsort, på tabeller med tilfeldige tall.
- Programmet må bestå sjekksumtesten og rekkefølgetesten for alle sorteringer dere måler på. (Men disse testene skal selvsagt ikke være med i tidsmålingen.)
- Rapporten må vise måleresultatene som gjorde det mulig å finne et beste delingstall. Med mange nok målinger vil dere finne noen bra tall, med dårligere tall både over og under.
- For den beste shellsorteringen dere fant, må dere også ha målinger og utregninger som gir kjøretid på formen $O\left(n^x\right)$. x forventes være mellom 1 og 2.