Øving 3, algoritmer og datastrukturer

Avansert sortering, forske på quicksort med forbedringer

29. august 2025

Innhold

Informasjon, velg 1 av 3 alternativer

Sorteringsoppgaven inneholder tre oppgaver, velg *en* av dem.

Les oppgaven *nøye,* før du gjør den. Så slipper du underkjenning for noe du har oversett. Det er også en del tid å spare på å unngå «vanlige feil».

De som ønsker en utfordring, velger ikke den letteste, men den mest interessante deloppgaven. :-)

Dere velger selv om dere bruker vanlig quicksort som utgangspunkt, eller dual-pivot. Uansett, vær oppmerksom på at en del eksempler på nett har dårlig valg av delingstall/pivot. Et slikt program vil ikke klare å sortere en million tall, hvis de er sortert rett fra før. En enkel implementasjon av vanlig quicksort, kan bruke midten av intervallet som delingstall. En dual-pivot quicksort kan bruke tallene på $1/3$ og $2/3$ av intervallet, for å unngå problemer.

Om valg av pivot

Quicksort i læreboka *fungerer.* Andre varianter *kan* ha problemer med pivot. Mange eksempler bruker første (eller siste) tall i deltabellen som pivot. De får problemer. Hvis du tilpasser en slik algoritme, husk at resten av algoritmen forutsetter at pivot ligger i starten (eller slutten). Hvis du bare velger en annen pivot, gjør programmet feil når det forsøker å sette pivot på rett plass. Dette løses ved å bytte om tall, slik at din ideelle pivot legges på starten (eller slutten) der algoritmen forventer at den ligger. For single-pivot quicksort, er det midterste tallet i deltabellen en god pivot.

Om dual-pivot

De fleste eksempler på dual-pivot velger første og siste tall som pivots. Det går dårlig med en sortert tabell, fordi alle tall havner i det midterste intervallet. Løsningen er å bytte første tall i deltabellen med tallet på $1/3$-plassen, og siste tall med tallet på $2/3$-plassen. Dermed får vi perfekt oppdeling av sorterte tabeller, og dual-pivot går mye bedre. Slik bytting må gjøres i den rekursive delen.

Også dual-pivot kan få problemer med mange duplikater. Men det har en enkel løsning. Etter at tallene er fordelt på tre intervaller, sjekker man om første og andre pivot er like. I så fall trenger en ikke sortere det midterste intervallet videre, fordi alle tallene der er like.

Felles godkjenningskrav til alle deloppgavene

1. Deloppgaven er løst, med de krav som er i den. Så sjekk oppgavens egne krav.
2. Kildekode og måleresultater er med.
3. Programmet sorterer korrekt, og passerer disse testene:
   a) tabell[i] >= tabell[i-1] for alle i fra 1 til tabell.length-1. En slik test avslører feil sortering.
   b) Sjekksum på tabellen er den samme før og etter sortering. Sjekksummen er summen av alle tallene i tabellen. Denne testen oppdager feil hvor tall blir overskrevet.

   Hvis ikke disse testene gjøres på hver eneste sortering, underkjennes oppgaven.
4. Programmet må kunne sortere en million tall på rimelig tid. (Om noen bruker python eller andre interpreterte språk, er det nok med 100 000 tall.)
5. Programmet må kunne sortere en million tall med mange duplikater, på rimelig tid. En tabell med duplikater lages slik at annenhvert tall er tilfeldig, og annenhvert tall er tallet 42. (For python holder det med 100 000 tall.) På nett og i noen andre lærebøker fins dessverre eksempler som ikke klarer dette.
6. Etter en sortering, skal programmet deretter sortere den sorterte tabellen på nytt, uten å få problemer. Altså ingen n^2-problemer med sorterte tabeller. Hvis det tar over dobbel tid å sortere en allerede sortert tabell, er oppgaven underkjent! (Varianter med dårlig valg av pivot, får dette problemet.)

Unngå vanlige problemer med tidtaking

- Når dere kjører samme jobb mange ganger, pass på å sortere et nytt datasett hver gang! Hvis dere sorterer samme tabell to eller flere ganger, er tabellen ferdig sortert i neste omgang, og gode sorteringsalgoritmer er mye raskere når de får en ferdig sortert tabell. Dermed tar dere tiden på feil operasjon. Et par løsninger på dette problemet:
  - Sorter en tabell som er så stor, at tidtaking ikke er noe problem. (Jobben tar minst et tiendels sekund)
  - Ta en ny kopi av en usortert tabell for hver omgang, og sorter kopien. Tidtakingen lider noe under dette, da man nå tar tiden på kopieringen i tillegg. Det

er ikke så farlig i dette tilfellet, da kopiering tar lineær tid, mens sortering tar mer enn lineær tid.

IKKE skru tidtaking av og på mellom kopiering og sortering! Da mangedobler dere problemet med unøyaktig maskinklokke…

- Regnereglene for asymptotisk notasjon gjelder for store n, ikke for små n. Så ikke bruk for små datasett i oppgave 1, selv om dere klarer å ta tiden presist.

Sorteringsoppgave 1, quicksort med hjelpealgoritme

Kapittel 3.9.1 i læreboka forteller at selv om quicksort er rask på store datamengder, så er den ikke raskest på små. Ettersom quicksort kaller seg selv rekursivt, vil en stor sortering nødvendigvis føre til mange ineffektive små sorteringer også.

1. Lag en variant av quicksort hvor rekursjonen brytes når deltabellen kommer under en «passende» størrelse. Når dette skjer skal quicksort-metoden benytte en annen sortering som hjelpealgoritme i stedet. Hjelpealgoritmen kan være en enklere sortering, som innsettingssortering, boblesortering, velgesortering eller shellsort. Det blir mer spennnende hvis ikke alle gruppene bruker samme hjelpealgoritme.
2. Finn ut hvor stor den «passende» størrelsen er ved å kjøre tester med tidtaking på store datamengder. Finn altså hvilken størrelse som gir raskest sortering for en gitt stor datamengde. Hvis svaret på dette blir 1 eller ∞ har dere gjort feil et sted. Ulike implementasjoner har ulik hastighet, så det kan godt hende at en gruppe får 35 mens en annen gruppe får 2 300 som passende størrelse for å bryte rekursjonen.

Egne godkjenningskrav for deloppgave 1

1. Måleserie (gjerne med kurve) som viser hvordan dere fant passende størrelse for å bytte til hjelpealgoritmen
2. Tidtaking både for quicksort med hjelpealgoritme, og quicsort uten. Noe bedre tid *med* hjelpealgoritmen. (En god implementasjon vil alltid hjelpe – det interessante her er *hvor mye*.)
3. Felleskravene er også oppfylt.

Tips deloppgave 1

Vanlig feil

Vanligste feil her, er å lage en test slik at programmet enten sorterer *hele* tabellen med quicksort, eller med innsetting/boblesort. Men det er ikke det oppgaven spør etter. Les i så fall oppgaven på nytt.

Nest vanligste feil er vel å ikke tilpasse hjelpealgoritmen, så den ender opp med å gjøre for mye jobb. Den jobber seg hele veien ned til 0, når den bare skal ned til starten på deltabellen. Sorteringen blir forsåvidt korrekt, men tidsforbruket altfor høyt. Det er meningen å gjøre det *bedre* enn standard quicksort her.

quicksort med hjelpealgoritme

Slik quicsort er beskrevet i boka, begynner det med en test på om `h-v>2`. Her tester dere i stedet om `h-v` er større enn den passende delingsverdien. I så fall brukes `splitt` og rekursive kall til `quicksort` som vanlig. Hvis ikke, brukes hjelpealgoritmen i stedet for `median3sort`.

quicksort med innsettingssort/velgesort/boblesort som hjelpealgoritme

Hvis dere går inn for en av disse hjelpealgoritmene, må dere tilpasse algoritmen slik at den kan sortere en deltabell i stedet for en hel tabell. Normalt opererer innsettingssortering fra `0` til `t.length-1`, nå skal den i stedet kunne sortere f.eks. fra posisjon 45 til posisjon 67. «`fra`» og «`til`» må overføres som parametre til metoden, som må ta hensyn til de nye endepunktene.

Både indre og ytre løkke må tilpasses de nye endepunktene. Dere er programmerere, og bør få til såpass. En enkel test på at du er på rett vei: hvis «fra» er 0, og «til» er `t.length-1`, skal den nye koden gjøre akkurat det samme som den gamle.

Vær dessuten oppmerksom på at boka har en trykkfeil på linje 7 i java-eksempelet for innsettingssortering. Bruk linje 7 fra C-eksempelet i stedet, eller fra forelesningen.

quicksort med shellsort som hjelpealgoritme

Variabelen `s` må initieres til «`(til-fra)/2`» i stedet for «`t.length/2`». Endepunktene for indre og ytre løkker må også tilpasses.

Sorteringsoppgave 2, quicksort med tellesortering

Når quicksort lager del-tabeller, vil vi noen ganger ende opp med en deltabell hvor alle verdiene ligger innenfor et smalt område. Hvis dette området er smalere enn størrelsen på deltabellen, kan den sorteres kjapt med tellesortering – uansett hvor stor den er. Implementer dette.

Men hvordan vet vi hvilket intervall tallene ligger i? Vi kan se på tallet før og tallet etter del-tabellen vi sorterer. Dette er pivots fra tidligere sorteringsomganger. Alt som ligger ovenfor en pivot, er større eller lik. Alt som ligger nedenfor en pivot, er mindre eller lik. Dermed kan vi bruke slike pivots som max og min, og slipper dermed å gå gjennom deltabellen en ekstra gang!

Hvis intervallet som sorteres ligger på kanten av tabellen, kan vi ikke sjekke utenfor. Men dette har en løsning: før sorteringen finner dere max og min i hele tabellen. Bytt min med tallet på første plass i tabellen, og bytt max med tallet på siste plass. Dermed er max og min plassert allerede. Kjør deretter sortering fra index 1 til tab.length-2, i stedet for 0 til tab.length-1.

Dette har sin pris. Vi får noen ekstra sammenligninger, og et søk etter max og min i starten. Det interessante er å finne ut hvorvidt denne ekstra innsatsen hjelper oss, i og med at tellesortering er enda raskere enn quicksort. Sjekk om dere fikk quicksort til å bli enda raskere. Det vil opplagt avhenge av hva slags intervall det er på tallene dere sorterer. En tabell med en del duplikater, kan f.eks. genereres ved å ha mindre spredning enn størrelsen på tabellen. Som en million tall hvor alle ligger mellom null og femti tusen. Finn også ut hvor mye de ekstra testene koster ved å sortere en datamengde hvor tellesortering aldri kommer til anvendelse, f.eks. en stokking av rekka 1, 3, 6, 9, 12, 15, … og sammenlign med quicksort uten tellesortering.

Standard tellesortering sorterer tall i intervallet $0..k$, men her må intervallet forskyves så det sorteres i intervallet *min…max.* Det er også nødvendig å tilpasse algoritmen slik at endepunktene ikke er 0 og t.length-1, slik at den kan sortere en del-tabell. En del tilpasning av tellesorteringen, altså.

Egne godkjenningskrav for deloppgave 2

1. Tidtaking både for quicksort med tellesortering, og tilsvarende quicsort uten. Målinger både på tabeller med en del duplikater, og tabeller med god spredning. (Så vi både ser hvor mye tellesorteringen hjelper, og hva den koster når den ikke kan hjelpe.)
2. Felleskravene er også oppfylt.

Sorteringsoppgave 3, quicksort med min forbedring

Innledning

Kjøretiden for quicsort er i beste fall $\Omega(n \log n)$. Dette kan forbedres til $\Omega(n)$, ved å innføre noen ekstra tester.

Den varianten av quicksort vi har sett på i boka, er slik at delingstallet havner *mellom* de to deltabellene som skal sorteres rekursivt. Det betyr også at når quicksort sorterer en deltabell som ikke ligger på kanten av tabellen, så ligger det et slikt delingstall på hver side av deltabellen.

Det interessante er konklusjonen vi kan trekke, hvis delingstallene på begge sidene av en deltabell viser seg å være like. (Forutsetter at tabellen vår har en del duplikater.) Vi vet jo:

- Alle tall på lavere indekser «til venstre» er mindre enn eller lik delingstallet.
- Alle tall på høyere indekser «til høyre» er større enn eller lik delingstallet.

Hvis samme delingstall fins på begge sider av en deltabell, må altså alle tallene i denne deltabellen være både «mindre eller lik» og samtidig «større eller lik» dette delingstallet. Den eneste måten det kan skje, er ved at alle tallene i deltabellen er helt like!

Når en deltabell består av bare like tall, er det opplagt ikke nødvendig å sortere den videre. Med noen enkle test kan vi altså returnere fra quicksort() uten å jobbe, hvis endepunktene er like.

Hvis deltabellen er på kanten av tabellen, kan vi ikke teste utenfor tabellen. Dette løses ved å finne min og max før sorteringen starter. min byttes med første tall i tabellen, og max byttes med siste tall i tabellen. Deretter kjøres sortering fra index 1 til tab.length-2, i stedet for den vanlige sorteringen fra 0 til tab.length-1.

Test altså:

> Hvis elementet foran det første er lik elementet etter det siste, kan quicksort() returnere uten å kjøre splitt og rekursive kall.

Hvis tabellen vi sorterer inneholder en del duplikater, vil testen slå til nå og da, og spare en del arbeid. I «beste fall» har tabellen bare like tall, og testen slår til på første kall. Sorteringen droppes, algoritmen har bare gjort arbeidet med å finne min og max – og det er lineært!

Programmering

Lag to varianter av quicksort — en med, og en uten denne forbedringen. Sammenlign kjøretidene for tabeller med ulik størrelse og innhold. Bruk en tabell med mange duplikater, for å se om det blir forbedringer. Mange duplikater kan f.eks. oppnås ved å la annenhvert tall være 42. Bruk også en tabell uten duplikater, for å se om den ekstra testen gir målbart lenger kjøretid når den ikke kan hjelpe.

Egne godkjenningskrav for deloppgave 3

1. Tidtaking for:
    a) quicksort uten forbedring, på tabell med mange duplikater (basis)
    b) quicksort med forbedring, på tabell med mange duplikater (hjelper dette?)
    c) quicksort med forbedring, på tabell uten duplikater (for å se hva dette koster)
2. Felleskravene er også oppfylt.