Øving 3, algoritmer og datastrukturer

Avansert sortering, quicksort med forbedringer

Informasjon

Sorteringsoppgaven består av fire oppgaver, det holder å velge *en* av dem.

De som ønsker en utfordring, velger ikke det letteste. :-)

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.

Felles godkjenningskrav til alle deloppgavene

1. Deloppgaven er løst, med de krav som er i den.
2. Programmet sorterer korrekt, og passerer disse testene:
    a) tabell[i+1] >= tabell[i] for alle i fra 0 til tabell.length-2. 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.
3. Programmet må kunne sortere en million tall på rimelig tid. Programmet skal deretter sortere den sorterte tabellen på nytt, uten å få problemer. Altså ingen n^2-problemer med sorterte tabeller. (Om noen bruker python eller andre interpreterte språk, er det nok med 100 000 tall.)

Unngå vanlige problemer med tidtaking

- «Små» jobber går så fort at PCen ikke klarer å ta tiden nøyaktig. Hvis du er nødt til å ta tiden på en «liten» jobb, bruk trikset fra forrige gang med å kjøre mange slike småjobber og del tidsforbruket med antall jobber.
- 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 jo tabellen ferdig sortert i neste omgang, og noen sorteringsalgoritmer er mye raskere når de får en ferdig sortert tabell. Dermed tar dere tiden på feil operasjon. Det er to greie løsninger på dette problemet:
  - Ha mange små usorterte tabeller, slik at hver omgang får sin egen tabell. Problemer:
    * Kan ta for mye plass og føre til swapping.
    * Må vite på forhånd hvor mange tabeller som trengs.
  - 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. Men de som vil ha mer presis tidtaking kan kjøre en ny runde som bare kopierer tabeller, og trekke fra dette tidsforbruket.
- 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.

Tips ang. klasser i java/c++

En *klasse* er en *datatype.* Derfor gir vi klassene navn etter hva slags data de inneholder, og *ikke* etter hva slags algoritmer de inneholder. Algoritmene er *metoder,* så det er metodene som får navn etter algoritmene.

Når vi jobber med sortering, har altså *klassen* navn som «talltabell», «data» eller noe lignende. *Metodene* har navn som «`quicksort()`», «`min_perfekte_sortering()`», osv.

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 eller velgesortering. Det blir mer spennnende hvis ikke alle gruppene bruker samme hjelpealgoritme.
2. Finn ut nøyaktig 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 sannsynligvis gjort feil et sted. Ulike implementasjoner har ulik hastighet, så det kan godt hende at en gruppe får 30 mens en annen gruppe får 250 som passende størrelse for deltabeller.

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…

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 som hjelpealgoritme

Hvis dere f.eks. går inn for innsettingssortering, 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.

I den ytre løkka går altså `j` fra «`fra+1`» i stedet for «`1`», inntil «`j < til + 1`» i stedet for «`j<t.length`»

I den indre løkka må dere passe på at «`i >= fra`» i stedet for «`i >= 0`», ellers går metoden utenfor deltabellen sin. Vær dessuten oppmerksom på at boka har en trykkfeil

på linje 7 i java-eksempelet for tellesortering. Bruk linje 7 fra C-eksempelet i stedet, den er korrekt.

quicksort med shellsort som hjelpealgoritme

Variabelen `s` må initieres til «`(til-fra)/2`» i stedet for «`t.length/2`». For-løkka må gå fra «`s+fra`» i stedet for «`s`». Betingelsen blir «`i<til+1`» i stedet for «`i<t.length`».

I while-løkka må dere teste på «`j>=fra+s`» i stedet for «`j>=s`»

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.

En enkel måte å få til dette, er å lage en løkke som finner største og minste tall i deltabellen. Dermed hvilket intervall tallene ligger innenfor, og kan velge om vi vil bruke tellesortering. Løkken bør avbryte med en gang hvis den tidlig oppdager at intervallet er for stort.

For deltabeller som ikke ender på kanten av tabellen, trenger vi ikke en slik løkke. Tallet til venstre for deltabellen er et delingstall fra tidligere omganger, som er mindre (eller lik) alle tallene i deltabellen vår. Tallet til høyre er også et delingstall, som er større eller lik tallene i deltabellen. Så tallet til venstre kan brukes som minimum, og tallet til høyre som maksimum.

Denne måten har en opplagt ulempe, ved at vi går gjennom mange deltabeller en gang ekstra. 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. Finn også ut hvor mye de ekstra testene koster ved å sortere en datamengde hvor tellesortering aldri kommer til anvendelse, f.eks. 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. Det er også nødvendig å tilpasse algoritmen slik at startpunktet ikke er 0, slik at den kan sortere en del-tabell.

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 en enkel test kan vi altså droppe både kallet til «`splitt`» og de rekursive kallene til «`quicksort`» – hvis endepunktene er like. I tillegg må vi teste at deltabellen ikke er på kanten av tabellen, slik at vi ikke prøver å teste utenfor tabellen. Programkoden blir slik:

```
public static void
  quicksort(int []t, int v, int h) {
    //Ny test
    if (v > 0 && h < t.length-1 && t[v-1] == t[h+1]) return;
    //Resten av quicksort blir akkurat som før
```

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 for *alle* deltabeller som ikke ligger på kanten av tabellen. I så fall får vi bare ett rekursivt kall i stedet for to. $T(n) = 1 \cdot T\left(\frac{n}{2}\right) + n$, og formelverket for rekursive algoritmer gir oss at $T(n) \in \Omega(n)$. Her har vi ikke bare spart tid, men til og med fått lavere kompleksitet.

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 en del duplikater for å se om det blir forbedringer. Bruk evt. også en tabell uten duplikater, for å se om den ekstra testen gir målbart lenger kjøretid når den ikke er til hjelp.

Sorteringsoppgave 4, Quicksort og valg av delingstall

Quicksort i boka velger delingstall ved å bruke «midterste av 3 alternativer». Man kan få mer fart på sorteringen ved å se på mer enn 3 alternativer. Disse tallene kan f.eks. sorteres med innsettingssortering for å finne det midterste tallet.

Oppgaven er å finne ut hvor mange tall man bør se på. Jo flere tall, jo bedre vil quicsort virke. Men det tar lenger tid å finne det midterste blant mange tall. Prøv ut mange ulike størrelser, f.eks. 3, 5, 7, 9, … og opp – til det begynner å gå langsommere. Slik finner dere det beste alternativet.

NB! Når dere f.eks. tester med 7 tall, husk at quicksort etterhvert lager deltabeller med færre enn 7 tall i. Når det er så få tall igjen, kan sorteringen som finner delingstallet sortere hele deltabellen akkurat som i deloppgave 1. Dermed får dere to forbedringer!