Løsningsforslag for utvalgte oppgaver fra kapittel 3

Innhold

3.3–1 Demo innsettingssortering

{8, 7, 6, 5, 4}

T[0]	T[1]	T[2]	T[3]	T[4]	j	Flytt
8	[7]	6	5	4	1	1
7	8	[6]	5	4	2	2
6	7	8	[5]	4	3	3
5	6	7	8	[4]	4	4
4	5	6	7	8	Ferdig,sum	**10**

{2, 4, 3, 6, 5}

T[0]	T[1]	T[2]	T[3]	T[4]	j	Flytt
2	[4]	3	6	5	1	0
2	4	[3]	6	5	2	1
2	3	4	[6]	5	3	0
2	3	4	6	[5]	4	1
2	3	4	5	6	Ferdig, sum	**2**

Som vi ser ble det færrest flytt i den nesten sorterte tabellen, og flest i den baklengs sorterte.

3.5–1 Demo velgesortering

{9, 8, 7, 6, 4}

T[0]	T[1]	T[2]	T[3]	T[4]	*i*
9	8	7	6	4	4
4	8	7	6	9	3
4	6	7	8	9	2
4	6	7	8	9	1
4	6	7	8	9	ferdig

{1, 3, 2, 5, 4}

T[0]	T[1]	T[2]	T[3]	T[4]	*i*
1	3	2	5	4	4
1	3	2	4	5	3
1	3	2	4	5	2
1	2	3	4	5	1
1	2	3	4	5	ferdig

3.5–2 Velgesortering

Velgesortering trenger ikke å prøve å plassere det minste tallet fordi det nødvendigvis står på rett plass når alle de andre er plassert. Når alle andre tall står på rett plass er det jo bare én ledig plass igjen, så det må være der det minste tallet står. Et tall kan simpelthen ikke stå feil plassert når alle de gale plassene er opptatt allerede.

3.8–1 Demo flettesortering

Splitting — {9, 8, 7, 6, 5, 4, 3, 2}

9 8 7 6 5 4 3 2
9 8 7 6 | 5 4 3 2
9 8 | 7 6 | 5 4 | 3 2
9 | 8 | 7 | 6 | 5 | 4 | 3 | 2

Fletting

8 9 | 6 7 | 4 5 | 2 3
6 7 8 9 | 2 3 4 5
2 3 4 5 6 7 8 9

Splitting — {55, 78, 67, 68, 67, 22, 13, 33}

55	78	67	68	67	22	13	33
55	78	67	68 |	67	22	13	33
55	78 |	67	68 |	67	22 |	13	33
55 |	78 |	67 |	68 |	67 |	22 |	13 |	33

Fletting

55	78 |	67	68 |	22	67 |	13	33
55	67	68	78 |	13	22	33	67
13	22	33	55	67	67	68	78

3.9–3 Korrekthet for median3sort

Vi kan vise at median3sort er korrekt, ved å demonstrere at algoritmen gir rett resultat på alle mulige tilfeller av input. Algoritmen bytter bare rundt på tre tall i tabellen, det er derfor ikke nødvendig å sjekke tabeller med mer enn tre elementer. Det er heller ikke nødvendig å prøve med alle mulige verdier; algoritmen sammenligner bare størrelsen på dem så det holder å sjekke alle mulige permutasjoner av tre ulike[1] verdier. Det finnes totalt 6 slike permutasjoner, som kan sjekkes ved å gå gjennom programmet linje for linje.

De seks permutasjonene er: {1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2} og {3,2,1}. I de følgende avsnittene tar jeg for meg disse permutasjonene, og viser hvordan de blir seende ut etter if-tester og *bytt*-kallene på linjene 4, 6 og 7 i programmet. Linjene 6 og 7 blir ikke nødvendigvis utført, hvis testen på linje 5 feiler. I slike fall viser jeg ikke noen informasjon for linje 6 og 7. Med en tabell som har tre elementer får vi *v=0, h=2* og *m=1*.

```
s:{1,2,3}
4:{1,2,3}
```

```
s:{2,1,3}
4:{1,2,3}
```

```
s:{3,1,2}
4:{1,3,2}
6:{1,2,3}
7:{1,2,3}
```

```
s:{1,3,2}
4:{1,3,2}
6:{1,2,3}
7:{1,2,3}
```

```
s:{2,3,1}
4:{2,3,1}
6:{2,1,3}
7:{1,2,3}
```

```
s:{3,2,1}
4:{2,3,1}
6:{2,1,3}
7:{1,2,3}
```

[1] Strengt tatt bør vi også sjekke alle permutasjoner hvor noen av tallene er like. Dette overlates til leseren.

Som vi ser ble resultater rett i alle tilfellene.

3.11–2 Kompleksitet for algoritme 3.11 på side 70

Her skal vi finne kompleksiteten for algoritmen på side 70 i læreboka, «Tellesortering av heltall». Linjene frem til og med linje 4 har kompleksiteten $\Theta(1)$. Linje 5 har kompleksiteten $\Theta(k)$, og linje 6 har kompleksiteten $\Theta(n)$.

Linje 7 har en løkke med kompleksitet $\Theta(k)$. Inne i den, på linje 8, har vi imidlertid en while-løkke som avhenger av inndata. Nå vet vi imidlertid at tabellen *ht* inneholder antall tall av hver størrelse. ($ht[0]$ inneholder antall nuller i datasettet vårt, $ht[1]$ inneholder antall enere i datasettet vårt, osv. Summen av alle tallene i *ht* blir altså antall tall totalt i datasettet, altså n.) Løkken på linje 7 går gjennom alle elementene i *ht*, og løkken på linje 8 teller hvert element ned til 0. Dermed ser vi at den totale arbeidsmengden er summen av alle tallene i *ht*, som er n, antall tall i datasettet. Kombinasjonen av linje 7 og linje 8 er altså $\Theta(n)$ hvis $n > k$, eller $\Theta(k)$ ellers. Dette kan også skrives som $\Theta(n+k)$.

Legger vi sammen kompleksiteten for de forskjellige linjene, får vi at denne algoritmen har kompleksiteten $\Theta(n+k)$. Normalt brukes ikke tellesortering hvis $k > n$, og med denne forutsetningen kan en si at algoritmen har kompleksiteten $\Theta(n)$.

3–2 Modifisert quicksort

Først den modifiserte algoritmen. Sortering gjøres ved å kalle opp *quickstart*, som beregner maksimal dybde og deretter kaller den modifiserte rutinen *quicksort2*.

```
void
  quicksort2(int *t, int v, int h, int dybde) {
    if (dybde > 0) {
    if (h - v > 2) {
      int delepos = splitt(t, v, h);
      quicksort2(t, v, delepos - 1, dybde - 1);
      quicksort2(t, delepos + 1, h, dybde - 1);
    } else median3sort(t, v, h);
  } else {
    mergesort(t, v, h);
  }
}

quickstart(int *t, int v, int h) {
```

```
    int dybde = 1 + 2*log(h-v) / log(2);
    quicksort2(t, v, h, dybde);
}
```

Kompleksiteten blir $\Theta(n \log n)$ fordi:

Quicksort kjører på $\Theta(n \log n)$ i gjennomsnittelige tilfeller. Hvis vi får rimelig heldig oppdeling av datamengden vil rekursjonsdybden være $\log n$. Da vil dybde-testen alltid slå til, og vi får en normal kjøring av quicksort.

Hvis quicksort derimot kommer ut for et uheldig tilfelle vil rekusjonen bli dyp. Men ikke dypere enn $\lceil 2\log_2 n \rceil$, for da feiler den nye testen og vi får brukt mergesort på de «vanskelige» delene av datasettet i stedet. Når dette skjer har vi allerede brukt $2n \log n$ tid på quicsort, og bruker så i tillegg $n \log n$ tid på mergesort. Med regnereglene for asymptotisk notasjon kan vi se bort fra konstante faktorer, og ser at vi fortsatt klarer oss med en kompleksitet på $\Theta(n \log n)$.

3–5 Kvadratsortering

Et kvadrat med areal n, har sidelengder $\sqrt{n}$. Tidsforbruket for å finne det største tallet på en linje er proporsjonalt med hvor mange tall det er på linja, altså $\Theta(\sqrt{n})$. For å finne det største tallet på hver linje multipliserer vi med antall linjer, som jo også er $\sqrt{n}$. Tidsforbruket for initialiseringen, å finne det største tallet på hver linje, blir dermed $\Theta(\sqrt{n} \cdot \sqrt{n}) = \Theta(n)$.

Deretter kommer selve sorteringen, som er en løkke som plukker ut det største tallet i første kolonne og gjemmer det unna, i tillegg til å finne det neste største tallet i den aktuelle raden.

Å finne det største tallet i kolonnen har, som vi har sett, kompleksiteten $O(\sqrt{n})$. (Bruker ikke Θ, da «kvadratet» blir mindre med tiden.) Å finne det neste største tallet i den aktuelle raden har også kompleksiteten $O(\sqrt{n})$. Kombinerer vi disse to får vi kompleksiteten $O(\sqrt{n})$ for en slik operasjon.

Det blir n slike operasjoner, fordi vi sorterer helt til det ikke er noen tall igjen. Dermed får kvadratsortering et tidsforbruk på $O(n\sqrt{n})$.