Institut for Matematik og Datalogi
Syddansk Universitet, Odense

16. februar 2023
RF

Opgaver Uge 8

DM507/DM578/DS814/SE4-DMAD

Opsummering fra forelæsninger: der er mindst tre naturlige niveauer at beskrive en algoritme på:

1. Som idé beskrivet i ord og tegninger.
2. Som pseudo-kode.
3. Som færdig kode (i f.eks. Java eller Python).

Dette er også de naturlige stadier at passere, når man skal udvikle algoritmer til et program i den virkelig verden. Hovedidéen i en algoritme er allerede på plads på niveau 1, og korrekthedsanalyse og asymptotisk analyse med grovsortering af algoritmer efter voksehastighed kan normalt finde sted der. Niveau 2 tilføjer flere detaljer, som skal være på plads for at sikre korrektheden af koden i sidste ende. Det kan ofte tage overraskende lang tid. Det kan betale sig at være omhyggelig med at lave korrekthedsanalyse af detaljerne (initialisering og opdatering af variable, f.eks.) i ens forslag til pseudo-kode, før man går videre til næste niveau. Hvis niveau 2 er lavet grundigt, bliver niveau 3 ofte relativt nemt.

Til forelæsningerne udtrykker vi os primært på niveau 1 og til dels på niveau 2. Lærebogen udtrykker sig også på niveau 1 og niveau 2, med de fleste algoritmer angivet meget detaljeret på niveau 2. I programmeringsopgaverne til øvelsestimerne og (for DM507) i projektet arbejder vi med niveau 3 – ofte baseret på niveau 2 fra bogen, andre gange baseret på niveau 1 og niveau 2, som man selv skal lave først.

Til øvelsestimerne er det i opgaver, hvor man skal udvikle, beskrive og/eller analysere algoritmer, *nok at gøre dette på niveau 1*, medmindre andet er angivet (dvs. medmindre man direkte bliver bedt om pseudo-kode eller Java/Python-implementation).

A: Løses i løbet af øvelsestimerne i uge 8

1. Eksamen juni 2008, opgave 2 (tidligere eksamensopgaver kan findes øverst på kursets hjemmeside).

2. Eksamen juni 2011, opgave 2. [Hint: For et sandt svar, argumenter ud fra definitionerne på O og Ω. For et falsk svar, find modeksempler.] Opgaven bruger notationen $f(n) \in O(g(n))$, hvilket betyder det samme som $f(n) = O(g(n))$. Se evt. diskussion i Cormen et al., 4. udgave, side 55 [Cormen et al., 3. udgave: side 45].

3. Implementer Mergesort i Java eller Python ud fra bogens pseudokode (Cormen et al., 4. udgave, side 36 og 39 [Cormen et al., 3. udgave: side 31 og 34]). Lad input være et array (Java) eller en liste (Python) af heltal.

   For Cormen et al., 3. udgave: I denne udgave optræder værdien ∞ i pseudokoden på side 31. Brug her et stort tal, og sørg for at kun heltal under denne værdi optræder i input.[1] Alternativt (og bedre, da man undgår at skulle tage hensyn til værdien valgt som ∞), brug til MERGE pseudokoden fra Cormen et al., 4. udgave, som kan findes på side 10 i slides om sortering. Denne pseudokode anvender ikke ∞.

   Test at din kode fungerer ved at generere små arrays/lister med forskelligt indhold (f.eks. stigende, faldende, tilfældige), sortere dem og checke resultatet.

4. ($*$) Cormen et al., 4. udgave, opgave 2-4 (side 47) [Cormen et al., 3. udgave: opgave 2-4 (side 41)], spørgsmål **d**.

B: Løses hjemme inden øvelsestimerne i uge 9

1. Eksamen januar 2007, opgave 3. Opgaven bruger notationen $f(n) \in O(g(n))$, hvilket betyder det samme som $f(n) = O(g(n))$.

[1] Faktisk har både Java og Python en indbygget værdi for ∞ (i Java `Double.POSITIVE_INFINITY`, i Python `float("inf")`). Disse er teknisk set kommatal, men fungerer efter hensigten, hvis de sammenlignes med heltal. I Java kan et kommatal og et heltal dog ikke optræde i samme array, så bogens pseudokode kan ikke implementeres vha. denne værdi. I Python kan et kommatal og et heltal godt optræde i samme liste, så der kan bogens pseudokode godt implementeres vha. denne værdi.

2. Fortsæt opgaven ovenfor med implementation af Mergesort på denne måde:

   Tilføj tidtagning af din kode. Du skal kun tage tid på selve sorteringen, ikke den del af programmet som genererer array'ets/listens indhold. Kør derefter din kode med input, som er tilfældige heltal. Gør dette for mindst 5 forskellige værdier af n (antal elementer at sortere), vælg værdier som får programmet til at bruge fra ca. 100 til ca. 5000 millisekunder. Gentag hver enkelt kørsel tre gange og find gennemsnittet af antal millisekunder brugt ved de tre kørsler. Divider de fremkomne tal med $n \log_2 n$, og check derved, hvor godt analysen passer med praksis – de resulterende tal burde ifølge analysen være konstante.

   [Hint: Se seddel med opgaver til uge 7 for informationer om bogens indeksering i forhold til Javas/Pythons, samt om metoder i Java og Python til tidstagning og til generering af tilfældige heltal. For at beregne logaritmer, brug i Java metoden `java.lang.Math.log(x)`, som beregner $\log_e(x)$, dvs. logaritmen med grundtal e. For at beregne $\log_2(x)$ kan man bruge at $\log_2(x) = \log_e(x)/\log_e(2)$ (jvf. Cormen et al., 4. udgave, formel 3.19 side 66 [Cormen et al., 3. udgave: formel 3.15 side 56]). I Python kan man bruge metoden `math.log(x,2)` til at beregne $\log_2(x)$ (husk at importere `math` modulet).]