Institut for Matematik og Datalogi
Syddansk Universitet, Odense

26. marts 2020

DM507 — Algoritmer og datastrukturer

Eksaminatorie-timer uge 13, Forår 2020

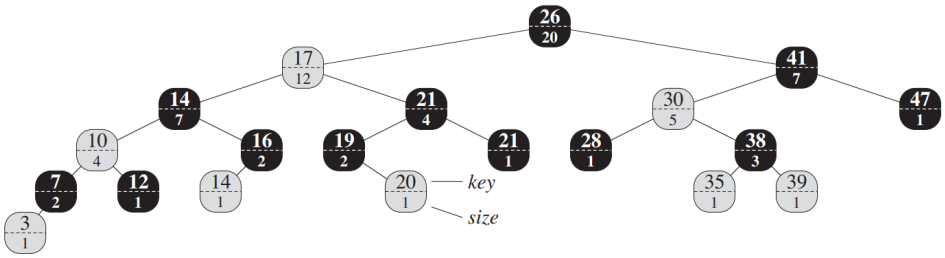
Instruktorerne for DM507

Indhold

Eksaminatorier I

1 Opgave 1 - Cormen et al. øvelse 14.1-1

Vis hvordan OS-SELECT(T.root, 10) virker på følgende rød-sort træ:



Figur 1

OS-SELECT's pseudokode i bogen [1, p. 341] er:

```
OS-SELECT(x, i)
1  r = x.left.size + 1
2  if i == r
3        return x
4  elseif i < r
5        return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i − r)
```

Figur 2

OS-SELECT på roden, med $i = 10$, udføres.

Først skal r udregnes. Da roden 26, x, har venstre barn 17, så skal r være lig med 17's size + 1. Altså er $r = 12+1 = 13$. Da $10 < 13$, $i < r$, så skal OS-SELECT(x.left, i) udregnes.

Da x.left er 17 og i er 10, så er knuden 17 den nye x og den "nye" i er 10.
Det venstre barn af den nye x, 17, er 14, som har size 7. Altså er $r = 8$. Da $10 > 8$, $i > r$, så skal OS-SELECT(x.right, $i - r$) udregnes.

Da x.right er 21, så er knuden 21 den nye x, og den nye i er $10 - 8 = 2$.
Det venstre barn af den nye x, 21, er 19, som har size 2. Altså er $r = 3$. Da $2 < 3$, $i < r$, så skal OS-SELECT(x.left, i) udregnes.

Da x.left er 19 og i er 2, så er knuden 19 den nye x og den "nye" i er 2.
Det venstre barn af den nye x, 21, er nil, da 21 ikke har et venstre barn. nil size er 0. Altså er $r = 1$. Da $2 > 1$, $i > r$, så skal OS-SELECT(x.right, $i - r$) udregnes.

Da x.right er 20, så er knuden 20 den nye x, og den nye i er $2 - 1 = 1$.
Det venstre barn af den nye x, 20, er nil, da 20 ikke har et venstre barn. nil size er 0. Altså er $r = 1$. Da $1 = 1$, $i = r$, så er x den knude, som skal findes.
Knude 20 bliver returneret, og dermed har OS-SELECT fundet knuden i træet med rank 10.

2 Opgave 2 - Cormen et al. øvelse 14.1-2

Vi skal vise hvordan `OS-RANK(T,x)` bliver udført på træet i figur 1 når x.key = 35. Pseudokoden og træet kan også findes på slide 4 i [3]. `OS-RANK` finder x's rang i træet T, dvs. har man alle knuder sorteret i rækkefølge efter deres nøgle, så er x's rang dens placering heri (hvis vores rækkefølge er 1-indexeret).

I `OS-RANK` starter vi først med at initialisere `r` til at være `x.left.size + 1`, hvilket i figur 1 er $0 + 1 = 1$, da x er et blad (her stopper `T.nil` os fra at få en fejl). Herefter sætter vi `y = x` inden vi går ind i while-løkken.

I while-løkken tjekker vi først at `y` ikke er roden, hvilket den ikke er så vi kan fortsætte ind i løkken. Da `y` ikke er sin forældres højre barn ændrer vi ikke `r`, hvorefter vi følge samme proces med `y's` forældre.

`y` er nu knuden med nøglen 38, som er sin forældres højrebarn. Vi skal derfor nu opdatere `r` til at være sin nuværende værdi, samt forældrens venstrebarns size + 1. I koden er det den her del: `r = r + y.p.left.size + 1`. Venstrebarnet til 38's forældre er knuden med nøglen 28 som har size 1, dvs. at nu bliver `r = 1 + 1 + 1 = 3`. Så sætter vi `y` til at være knuden med nøglen 30.

Da 30 ikke er højrebarn til 41 sker der ikke mere end at vi sætter `y` til at være knuden med nøglen 41. 41 er højre barn til roden, så `r` bliver nu til `r = 3 + 12 + 1 = 16`. Til sidst bliver `y` nu til knuden med nøglen 26, hvilket er roden i træet, så vi går ud af while-løkken og returnerer `r = 16`.

3 Opgave 3 - Cormen et al. øvelse 14.1-5

I denne opgave er vi givet et order-statistic tree[1], T, en reference til en knude x i træet, samt naturligt tal i. Vores opgave er at lave en algoritme, som kan returnere den i'te successor til x (dvs. hvis man lavede et in-order gennemløb af træet, så den knude som er den i'te knude efter x). Algoritmen må højst bruge $O(\lg n)$ tid, hvor n er antal elementer i hele træet. Vi husker på, at et order-statistic tree er et rød-sort træ, som i alle knuder også opbevarer noget ekstra information: størrelsen af undertræet (inklusiv knuden selv, og vi tæller ikke "NIL" blade med).

Det er her vigtigt at huske på, at vores algoritme højst må bruge $O(\lg n)$ tid. Hvis vi ikke

[1] Afhængigt af, hvordan man læser opgaveteksten, er det ikke sikkert, at vi er givet en reference til træet som x er opbevaret i. Dette er dog ikke noget problem, da vi blot kan starte med at følge stien fra x til roden af træet for at få en reference til roden (og dermed træet). Dette tager højst $O(\lg n)$ tid.

havde en tidsbegrænsning, kunne vi f.eks. bare have gået gennem træet in-order, noteret hvornår vi fandt x, og så talt op indtil vi var nået til x's i'te successor. Dette ville tage $O(n)$ tid. Intuitivt, så det der driller os lidt her er, at vores algoritme ikke kan lave noget arbejde som er lineært i i, da vi f.eks. kunne risikere, at $i = \frac{n}{2}$, og så ville vores algoritme tage $\Omega(n)$ tid.

At vi må bruge $O(\lg n)$ tid kan måske inspirere os lidt: da et order-statistic tree er et rød-sort træ med en smule ekstra information, har vi stadigvæk, at træet har højde $O(\lg n)$. Så dette betyder, at vi gerne må gå fra rod-blad et konstant antal gange, da køretiden stadigvæk vil være $O(\lg n)$. Vi kan faktisk se i Rolf's slides [3], at der allerede er lavet to algoritmer til order-statistic trees, som vi kan gøre brug af: `OS-RANK(T,x)` og `OS-SELECT(x,i)`. Begge disse har køretider på $O(\lg n)$, da de i virkeligheden bare følger en sti op eller ned gennem træet, og denne sti er højst $O(\lg n)$ lang. `OS-RANK(T,x)` giver os "rank" for x, dvs. dens placering i et in-order gennemløb af T. `OS-SELECT(x,i)` giver os knuden med rank i i x's undertræ.

Vi kan nu lave en ny algoritme, som gør brug af disse to algoritmer: vi kan starte med at finde x's rank i T, ved at kalde `OS-RANK(T,x)`. Lad x's rank være r. Siden vi gerne vil have den knude som er i knuder efter x, skal vi finde en knude med rank $r + i$. Dette kan vi gøre ved at kalde `OS-SELECT(T.root,r+i)`. Dette er sådan set algoritmen; vi har brugt to andre algoritmer som hver især bruger $O(\lg n)$ tid, og vi har kun lavet ét kald til hver, dvs. vi bruger også selv $O(\lg n)$ tid. Korrektheden følger direkte: vi bruger to algoritmer som vi allerede ved fungerer korrekt, og vi får netop den knude som har en rank i større end x.

Ekstra

Man kunne også forestille sig en anden løsning, hvor man lidt mere manuelt bruger informationen i knuderne, og på en måde fik sat de to algoritmer vi bruger sammen. Dette kunne måske være en fordel i praksis hvis man lavede noget kode der var *meget* tidskritisk, da man kunne forestille sig, at man kunne få lavere konstanter. Den asymptotiske køretid for den nye algoritme og den vi lige har fundet på ville være den samme, men i praksis kunne man forestille sig en lille fordel til den nye.

En fordel med den løsning vi har fundet her, er at vi bruger to algoritmer som der allerede findes korrekthedsargumenter for, og med den måde vi sætter dem sammen på og bruger i vores algoritme, følger korrektheden for vores algoritme stortset direkte af fremgangsmåden af algoritmen. Hvis man selv havde fundet på en ny algoritme, skulle man til at lave korrekthedsargumenter for denne. Hvis man skulle implementere algoritmen i praksis, kunne man også håbe på, at der allerede var andre som havde lavet gode implementationer af de algoritmer som vi bruger i vores algoritme, og dermed kunne vi undgå at skulle implementere dem selv (og undgå potentielle fejl). Generelt er det en ret god idé at genbruge eksisterende algoritmer/datastrukturer så vidt muligt (enten genbruge det hele, eller lave små justeringer), da man netop allerede har korrekthedsargumenter (og analyse af køretid), og man kan derfor både spare tid (som i den tid det tager for os at finde på algoritmen), undgå fejl, samt gøre ens algoritme nemmere at forstå. Det er den samme tankegang vi bruger når vi dekorerer rød-sorte træer med ekstra information (f.eks. for at få order-statistic trees).

4 Opgave 4 - Cormen et al. øvelse 14.1-7

Vi kigger på måder at tælle antal af inversioner i $O(n \cdot \log_2(n))$ asymptotisk tid.

Genopfriskning:

- En inversion er to index, *i, j*, i et array, A, hvor det gælder at $i < j$ og $A[i] > A[j]$.
- En knudes rang (rank) er et udtryk for hvor mange knuder, der ligger før knuden i en inorder-tree-walk plus 1 for knuden selv.

Vi kan ved brug af et dekoreret rød-sort træ ([1] afsnit 14 side 339, eller [3]) have adgang til en funktion der udregner rang i $\mathrm{O}(\log_2(\mathrm{n}))$ tid.

Ide til algoritme til at tælle inversioner:

Input: Array A hvor vi skal tælle inversioner. Dekoreret rød-sort træ T.

Forløb: Vi sætter elementer fra A's mindste index til højeste, ind i T. Når et element er blevet indsat udregnes rang for elementet. Herefter trækkes elementets rang fra størrelsen af træet og resultatet lægges til antallet af inversioner. Til sidst returneres antallet af inversioner.

Når vi indsætter et tal i træet, så har vi allerede indsat alle tal til venstre for dette tal i arrayet. Dvs. alle de tal som nu er i træet og som har en værdi, der er større end det tal, vi er ved at indsætte, skal stå til højre for vores værdi i et sorteret array. I det array vi er givet står de dog til venstre for vores tal, dvs. vi skal tælle en inversion for hver af disse tal. Med rang får vi antallet af elementer i træet der er mindre end vores element plus en (for det element vi lige har indsat). Derfor trækker vi rang fra størrelsen af træet for at få antallet af inversioner.

For tidskompleksitet ved vi at insert og rank tager $\mathrm{O}(\log_2(n))$ tid. Vi udfører begge operationer en gang per element i A, det giver os $O(n \cdot \log_2(n))$ tidskompleksitet.

5 Opgave 5 - Eksamen januar 2008, opgave 3

a) Hvad er `MinAbove(10)` for punkterne $(5, 9), (5, 17), (19, 6), (23, 10), (25, 15), (40, 7)$.

   Man kan udelukke punkterne $(5, 9)$, $(19, 6)$ og $(40, 7)$, da disse har et y-koordinat, som er strengt mindre end $t = 10$. Blandt de restende punkter $(5, 17)$, $(23, 10)$ og $(25, 15)$ vælges det punkt, hvis x-koordinat er mindst; altså, `MinAbove(10)` returnerer $(5, 17)$.

b) Beskriv, hvordan et udvidet rød-sort træ kan vedligeholdes under indsættelse og sletning af punkter.[2]

   En fordel ved dekoration af rød-sort træer er, at der er tale om en meget generisk metode. Overordnet handler det om, at man udvider knuder med noget ekstra information, hvor en given knudes information udregnes i $O(1)$ tid ud fra sine børns information. At dette kan gøres er en tilstrækkelig betingelse for, at værdierne under indsættelse og sletninger kan vedligeholdes uden ændring af $O(\lg n)$ køretiden. [3, s. 5]

[2] I delopgave b) og c), prøv da at glem hvad vi skal bruge den ekstra information til (altså glem `MinAbove(t)`. Tænk din opgave bare handler om at proppe $y_{\max}$ ind i et rød-sort træ. Om dette kan bruges til noget fornuftigt er en andens problem.

I dette tilfælde er den ekstra information $y_{\max}$. Givet knude `x` med venstre barn `x.left` og højre barn `x.right`, kan $y_{\max}$ for `x` udregnes i $O(1)$ tid ud fra børnenes $y_{\max}$ værdier? Ja, da $y_{\max}$ for `x` kan bestemmes som

$$y_{\max} = \max\{\texttt{x.y},\ \texttt{x.left.y}_{\max},\ \texttt{x.right.y}_{\max}\} \tag{1}$$

i $O(1)$ tid. I tilfælde af et barn af `x` er `NIL`, så vil det tilsvarende argument falde fra i (1) (eller betragt det som værende 0). Altså

- hvis `x` kun har et barn z, så vil $y_{\max} = \max\{\texttt{x.y},\ \texttt{z.y}_{\max}\}$
- hvis `x` er et blad, så vil $y_{\max} = \max\{\texttt{x.y}\} = \texttt{x.y}$

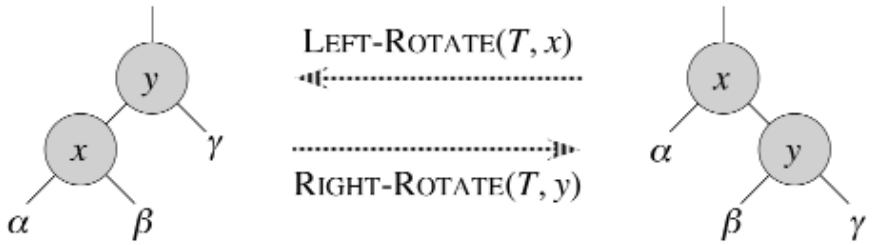
Observer at bladenes $y_{\max}$ beregnes i $O(1)$ tid.

På nuværende tilpunkt har vi argumenteret for, at en arbitrær knudes $y_{\max}$ værdi kan udregnes i $O(1)$ tid ud fra sine børns $y_{\max}$ værdier. Det følger heraf, at $y_{\max}$ under indsættelse og sletninger kan vedligeholdes uden ændring af $O(\lg n)$ køretid [3, s. 5].[3]

På nuværende tidspunkt har vi argumenteret for et rød-sort træ med $y_{\max}$ information kan vedligeholdes under indsættelser og sletninger. Hvordan dette mere specifikt gøres har vi ikke berørt endnu. Det er dog i grove træk det samme uanset informationen - den store forskel ligger i logikken bag hvordan informationen skal opdateres. Med udgangspunkt i denne opgave, så vil en måde at vedligeholde det udvidede rød-sort træ under indsættelser og sletninger være

- Indsættelser: Består af to faser
  1. Indsæt punktet (x_1, y_1) som ved et almindeligt rød-sort træ, hvor nøglen er lig x_1, men stop inden nogen rebalanceringer/omfarvninger foretages. Lad den indsatte knude være `v`, hvor $\texttt{v.y}_{\texttt{max}} = y_1$. Benyt (1) til at opdatere $\texttt{v.p.y}_{\texttt{max}}$. Hvis $\texttt{v.p.y}_{\texttt{max}}$ ændres, sæt da `v = v.p` og gentag. Stop når `v.p == NIL` eller $\texttt{v.p.y}_{\texttt{max}}$ ikke ændres. Efter første fase er $y_{\max}$ værdierne korrekte igennem hele træet.
  2. Fortsæt den almindelige indsættelsesprocedure for rød-sorte træer; altså, foretag evt. rebalanceringer/omfarvninger. Omfarvninger ændrer ikke på $y_{\max}$ værdierne for en given knude, men det gør rotationer. Derfor skal man ved en rotation benytte (1) til at opdatere $y_{\max}$ for de to knuder - kaldet x og y på Figur 3 - der får nye undertræer. For en højre rotation skal y opdateres før x, da x børn skal have korrekte $y_{\max}$ værdier før dens $y_{\max}$ værdi opdateres. Det omvendte gør sig gældende for en venstre rotation.

[3] I bogen [1, s. 346] fremgår dette af **Theorem 14.1 (Augmenting a red-black tree)**, som også har det tilhørende bevis.

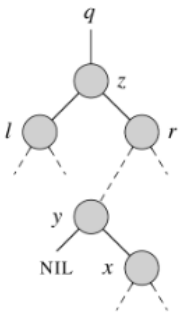


Figur 3

- Slettelser: Består af to faser

  1) Slet en given knude som ved et almindeligt rød-sort træ, men stop inden nogen rebalanceringer/omfarvninger foretages. Lad v være forældren til knuden der strukturelt fjernes fra træet - det er altså ikke knuden vi logisk fjerner fra træet (se evt. et af de mulige sletnings cases i Figur 4). Hvis $\max\{\texttt{v.y},\ \texttt{v.left.y}_{\max},\ \texttt{v.right.y}_{\max}\} < \texttt{v.y}_{\max}$, så er den nuværende $\texttt{v.y}_{\max}$ ikke bestemt af nogen y-værdi for en knude i undetræet med rod i `v` (det var den strukturelt fjerende knude som dikterede $\texttt{v.y}_{\max}$). Derfor opdateres $\texttt{v.y}_{\max}$ vha. (1), `v` sættes til `v.p`, og det gentages medmindre `v` nu er `NIL`. Hvis den strukturelt fjernede knude ikke dikterer $\texttt{v.y}_{\max}$, så har `v`, samt alle dens forfædre, den korrekte $y_{\max}$ værdi, og vi stopper. Efter første fase er $y_{\max}$ værdierne korrekte igennem hele træet.

  2) Fortsæt den almindelige sletningsprocedure for rød-sorte træer; altså, foretag evt. rebalanceringer/omfarvninger. Igen gælder der, at rotationer skal foretages, så $y_{\max}$ værdierne vedbliver med at være korrekte efter først fase - dette gøres på samme måde som for rotationer for indsættelser.

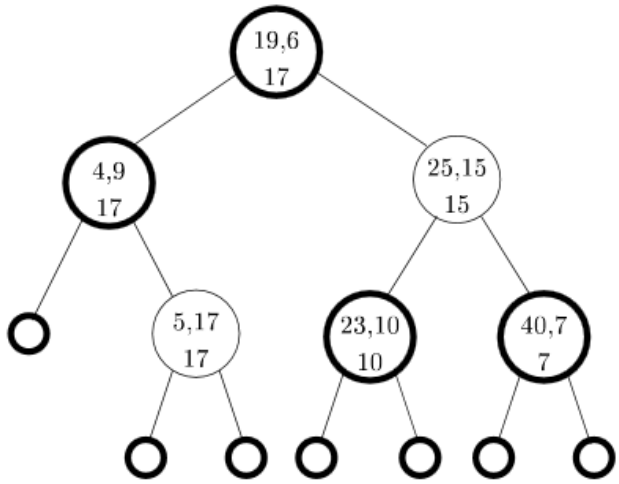


Figur 4: Her er z knuden vi logisk sletter mens y er knuden vi strukturelt sletter. Knuden v vil være y forældre.

Ovenstående beskrivelser er relativt generelle, og en implementering af det kunne evt. undlade at opdele det i to faser, så man ikke risikerer at skulle gå fra et blad

til roden af træet to gange. Her er det undladt, så opgaven ikke bliver for snørklet. Specielt skulle man gerne kunne se, at køretiden ikke har ændret sig. De eneste ekstra arbejde der foretages er, at man i worst case efter at have indsat eller slettet en knude, skal gå fra det givne sted i træet op mod roden og foretage noget konstant arbejde (opdatering af $y_{\max}$) ved hver knude på vejen. Derudover er rotationer lidt mere komplekse, men en rotation tager forsat $O(1)$ tid.

c) Illustrer en del af svaret fra spørgsmål b ved at indsætte punktet $(30, 11)$ i det udvidede rød-sort træ, som er gengivet i Figur 5.



Figur 5

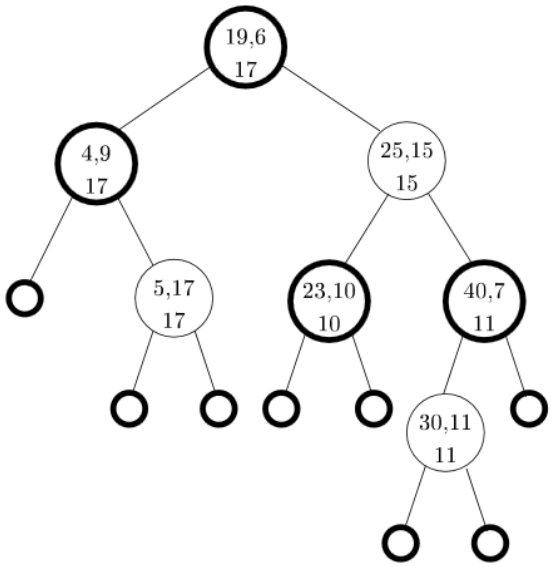
Når et punkt indsættes, så bruges x-koordinatet som nøglen. Da 30 er større end 19, så skal den nye knude være i rodens højre undertræ. Da 30 er større end 25, så skal den nye knude være i 25 knudens højre undertræ. Da 30 er mindre end 40, så skal den nye knude være i 40 knudens venstre undertræ, som er et `NIL` undertræ, og dermed er indsættelsespositionen fundet. Den nye knude har y-koordinat 11, og da en nyindsat knude - inden rebalanceringer - altid er et blad, så er $y_{\max}$ trivielt lig 11.

Nu kan 40 knudens $y_{\max}$ værdi opdateres om nødvendigt (vigtigt: begge dens børn har korrekte $y_{\max}$ værdier). I dette tilfælde vil $y_{\max}$ blive opdateret, da 40 knudens venstre barn (30 knuden) har en større $y_{\max}$ værdi end 40 knudens egen y værdi (det højre barn er `NIL`, så dens $y_{\max}$ kan betragtes som lig 0). Dermed bliver 40 knudens $y_{\max}$ værdi opdateret til 11.

Nu kan 25 knudens $y_{\max}$ værdi opdateres om nødvendigt (vigtigt: begge dens børn har korrekte $y_{\max}$ værdier). I dette tilfælde vil $y_{\max}$ ikke blive opdateret, da dens egen y værdi er større end $y_{\max}$ for begge dens børn. Da der ikke foretages en opdatering, så kan vi stoppe - der er ingen grund til at opdatere $y_{\max}$ for de øvrige forfædre.

I dette tilfælde skal der ikke foretages nogen rebalanceringer/omfarvninger, da den indsatte knude har en sort forældre (der er altså ingen rød-rød overtrædelser som skal fikses).

Det endelige udvidede rød-sort træ efter indsættelsen er illustreret i Figur 6.



Figur 6

d) Beskriv, hvordan `MinAbove(t)` kan udføres i tid $O(\lg n)$.

`MinAbove(t)` kan implementeres ved at gøre følgende

i) Hvis træet er tomt eller rodens $\texttt{y}_{\texttt{max}}$ værdi er strengt mindre end t, så stopper vi og meddeler et sådant punkt ikke findes.

ii) Lad `v` være en knude som i starten er initialiseret til roden.

iii) Hvis $\texttt{v.left} \neq \texttt{NIL}$ og $\texttt{v.left.y}_{\texttt{max}} \geq t$, sæt $\texttt{v} = \texttt{v.left}$ og gentag iii). Hvis $\texttt{v.y}_{\texttt{max}} \geq t$, returner da `(v.x, v.y)`. Ellers sættes $\texttt{v} = \texttt{v.right}$ og iii) gentages.

Observer at rodens $\texttt{y}_{\texttt{max}}$ værdi hurtigt kan afgøre, om der eksisterer en knude med en tilstrækkeligt høj y værdi.

Intuitionen er, at man prøver at gå så langt til venstre som muligt, men man går ikke ned i et venstre undertræ hvis rod har en $\texttt{y}_{\texttt{max}}$ værdi strengt mindre end t. Hvis det venstre undertræ var en nitte, så tjekker man om `v.y` er mindst t, da den i så fald vil være knuden med det mindste x-koordinat og et tilstrækkelig y-koordinat. Ellers fortsætter man i højre undertræ, hvor den nødvendigvis vil være (det følger af tjekket i i)).

Køretiden er klart $O(\lg n)$, da højden af træet er $O(\lg n)$ og i værste fald gennemløbes en sti fra roden til et blad.

6 Opgave 6 - Cormen et al. øvelse 2.1-3

Ved brug af løkke-invariant, bevis at algoritmen Linear-Search er korrekt. Invarianten skal opfylde de tre nødvendige egenskaber. Pseudokoden for LinearSearch, som bruges her, er:

```
1    LinearSearch(A, v):
2        i = 0
3        while i<A.length && A[i]!=v
4            i = i+1;
5        if i==A.length
6            return Nil
7        else
8            return i
```

Figur 7: pseudokode til LinearSearch. A er 0-indekseret

Andre pseudokoder kan også bruges, dog kan det være, at forklaringerne skal ændres lidt. De tre nødvendige egenskaber for en invariant er:

1. Initialization. Holder invarianten i starten?
2. Maintenance. Hvis invarianten holder før en iteration af while-løkken, holder den så efter iterationen af while-løkken (og dermed før den næste iteration)?
3. Termination. Kan invarianten bruges til, at vise at algoritmen afslutter korrekt?

Invarianten her vælges til at være: "v er ikke i $A[0..(i-1)]$".

Vi beviser nu, at invarianten har hver af de 3 egenskaber.

Initialization. Før while-løkken er $i = 0$. $A[0..(0-1)] = A[0..-1]$. $A[0..-1]$ indeholder intet, da 0 til -1 er et tomt interval. Derfor er v ikke i $A[0..(i-1)]$, og invarianten er sand før den første iteration.

Maintenance. Der skal vises, at hvis invarianten holder før en iteration, så holder invarianten efter iterationen. i bruges til at annotere indholdet af variablen i før iterationen, og i' bruges til at annotere indholdet efter iterationen.

Der observeres at, hvis while-løkken kører en iteration, så må det gælde, at $i < A.length$, og $A[i] \neq v$, da dette er guarden i while-løkken.

Antag at v ikke er i $A[0..(i-1)]$. Under iterationen lægges en til variablen i, dvs. at $i' = i+1$. Da v ikke er i $A[0..(i-1)]$, og v ikke er $A[i]$ (for så var vi slet ikke gået ind i while-løkken), så må det gælde, at v ikke er i $A[0..i]$. Eftersom $i' = i+1 \iff i = i'-1$, så betyder det, at v ikke er i $A[0..(i'-1)]$ Vi har dermed vist, at hvis invarianten holder før en iteration, så holder den også efter iterationen.

Termination. Der skal vises, at algoritmen giver det rigtige resultat, når den terminerer.

While-løkken terminerer, hvis $i = A.length$, eller hvis $A[i] = v$.

Hvis while-løkken blev termineret, fordi $i = A.length$, så kan vi bruges vores invariant, som siger at v ikke er i $A[0..(i-1)] = A[0..(A.length-1)]$, dvs. v findes ikke i arrayet. Vores algoritme returnerer altså korrekt nil hvis v ikke er i A.

Hvis while-løkken ikke blev termineret, fordi $i = A.length$, så må while-løkken have termineret, fordi $A[i] = v$. Dvs. vi fandt faktisk v i A, og vi ved at v blev fundet på plads i. Dvs. vi returnerer korrekt i.

Vi har nu ved hjælp af en løkke-invariant bevist, at vores algoritme for linear search er korrekt.

7 Opgave 7 - Cormen et al. opgave 2-2

I denne opgave skal vi kigge på sorteringsalgoritmen `Bubblesort` som der eksisterer mange forskellige versioner af. Vores version virker ved at vi har en ydre for-løkke som med variablen `i` tæller fra 1 til `A.length-1`. Den variabel bliver brugt i en indre for-løkke til at gennemgå arrayet fra enden til lige før `i` og bytte rundt på elementer som ikke er i den rigtige orden. Prøver man at køre algoritmen i hånden vil man kunne se at det i'te mindste element er blevet "bubblet"hen på sin rigtige plads efter den i'te udførsel af den indre for-løkke.

Spørgsmål a

For at være sikre på at `Bubblesort` er korrekt, skal vi vise at den sætter elementer i ikkefaldene orden (se eq. 2.3 i opgaven i [1]) og vi skal være sikre på at algoritmen kun bytter rundt på elementer. Altså, må den ikke kunne fjerne, ændre eller tilføje flere elementer.

I `Bubblesort` kan man hurtigt se at der ikke bliver ændret eller fjernet elementer, da algoritmen kun bytter rundt på dem.

Spørgsmål b

Her skal vi opstille en invariant for den indre for-løkke og vise at invarianten holder.

En invariant for den indre for-løkke kunne være følgende:

$$\texttt{A[j]} = \min\{A[k] : \texttt{j} \leq k \leq \texttt{A.length}\}$$

Med andre ord `A[j]` indeholder det mindste element fra `A[j..A.length]`.

For at vise at invarianten holder, skal vi vise at denne holder inden første iteration af forløkken (**initialization**), og at hvis invarianten holder før en iteration så skal den også holde før den næste iteration (**maintenance**). Til sidst skal vi bruge invarianten og grunden til at for-løkken stoppede til at konkludere noget brugbart om for-løkken (**termination**).

Initialization: Til at starte med er `j = A.length` og det er derfor trivielt sandt, da det mindste af et element bare er det ene element.

Maintenance: Vi skal nu vise at hvis invarianten holder før en iteration så skal den også holde før den næste iteration. Det kan man se ved at kigge på hvad der sker inde i forløkken, her sørger `if-statementet` i linje 3 for at `A[j-1]` indeholder det mindste element af `A[j-1]` og `A[j]`. Dvs. når j bliver dekrementeret til `j' = j-1`, så indeholder `A[j']` stadig det mindste element fra `A[j'..A.length]` og vores invariant holder.

Termination: Når vi kommer ud af for-løkken, så må det være fordi j blev dekrementeret, og efter dekrementationen var j ikke længere mindst $i+1$, dvs. j må have været i. Vi kan nu bruge vores invariant, og indsætte $j = i$ i den, og vi får at "$A[j] = A[i]$ indeholder det mindste element fra $A[j..A.length] = A[i..A.length]$". Dvs. vi har brugt vores invariant til at bevise, at den indre for-løkke placerer det mindste element fra $A[i..A.length]$ i $A[i]$.

Spørgsmål c

Nu skal vi opstille en invariant som hjælper os med at vise at `Bubblesort` er en korrekt sorteringsalgoritme (under brug af den netop viste invariant).

Invariant Lad invarianten være følgende: "Før en iteration er alle tal i `A[1..i - 1]` i sorteret rækkefølge, og alle mindre end tallene i `A[i..A.length]`".

Initialization Før den første iteration er $i = 1$, og vores invariant siger at `A[1..0]` er i sorteret rækkefølge og alle mindre end tallene i `A[1..A.length]`. Dette er klart sandt, da der slet ikke er nogen tal i `A[1..0]`.

Maintenance Lad i være indholdet af variablen i før den nuværende iteration og i' indholdet før den næste iteration.

Antag nu, at invarianten er sand, dvs. alle tal i `A[1..i - 1]` er i sorteret rækkefølge, og alle mindre end tallene i `A[i..A.length]`".

Vi ser nu, hvad der sker når vi laver en iteration. I iterationen går vi gennem den indre for-løkke. Den har vi allerede bevist en invariant om, som siger at efter den indre for-løkke, så er `A[i]` det mindste tal blandt `A[i..A.length]`. Dvs. vi ved nu, at `A[1..i]` alle er mindre end `A[i + 1..A.length]` (per vores antagelse ved vi at `A[1..i - 1]` var mindre, men nu ved vi også at `A[i]` er mindre).

Vi ved også, at `A[1..i]` er sorteret, fordi per antagelse ved vi at `A[1..i - 1]` er sorteret, og da `A[1..i - 1]` alle var mindre end `A[i..A.length]`, så var de alle også mindre end `A[i]`, dvs. `A[i]` er det største blandt dem, og står derfor korrekt i den sorterede rækkefølge.

Inden den næste iteration bliver i inkrementeret, dvs. indholdet af variablen i før den næste iteration (i') er $i' = i + 1$, som er det samme som $i = i' - 1$. Hvis vi indsætter det i stedet for i i det vi lige er kommet frem til, får vi, at "`A[1..i]` = `A[1..i' - 1]` er sorteret og alle er mindre end `A[i + 1..A.length]` = `A[i'..A.length]`".

Dvs. vi har nu vist, at hvis invarianten er sand før en iteration, så er den også sand før den næste iteration.

Termination Vi kommer ud af for-løkken fordi i ikke længere var højst `A.length - 1`, dvs. det må gælde at i = `A.length`. Hvis vi indsætter dette i vores invariant som vi lige har bevist er sand, så får vi: "Alle tal i `A[1..i - 1]` = `A[1..A.length - 1]` er i sorteret rækkefølge, og alle mindre end tallene i `A[i..A.length]` = `[A.length..A.length]`". Med andre ord: arrayet fra den første til den anden sidste plads er sorteret, og alle tallene er mindre end tallet på den sidste plads. Med det kan vi konkludere at den sidste plads indeholder det største tal, og står dermed også på den korrekte plads i en sorteret rækkefølge, og hele `A` står dermed i sorteret rækkefølge.

Til slut kan vi se, at den eneste måde algoritmen manipulerer arrayet på er ved bytte rundt på to tal, dvs. vi kan aldrig fjerne eller tilføje ekstra tal, så de tal der står i `A` må være de

originale, og derfor må algoritmen sortere `A` korrekt.

Spørgsmål d

Her skal vi først finde worst-case køretiden for `Bubblesort` og derefter sammenligne den med køretiden for `Insertionsort`. Lad $n =$ `A.length` i resten af analysen.

For at finde worst-case køretiden for `Bubblesort` skal vi finde et udtryk for antallet af sammenligninger som algoritmen foretager. Først kan man se at alle sammenligninger sker i den indre for-løkke. Den laver en sammenligning for hver iteration og den laver $n-(i+1)+1 = n-i$ iterationer. Den ydre for-løkke laver $n-1$ iterationer, dvs. vi får følgende sum:

$$\sum_{i=1}^{n-1} n - i$$

Som vi kan skrive om til følgende ved at trække i ud af summen:

$$\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

Nu kan man se at i den første sum der summer vi n sammen $n-1$ gange og man kan derfor omskrive den til $n \cdot (n-1)$. Den anden sum kan skrives ud som $1+2+3+...+(n-1)$, hvilket vi kan omskrive til $\frac{(n)\cdot(n-1)}{2}$ lidt ligesom i analysen for `Insertionsort`. Vi kan derfor omskrive hele summen til følgende:

$$\begin{aligned}\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i &= n\cdot(n-1) - \frac{(n)\cdot(n-1)}{2} \\ &= n\cdot(n-1) - \frac{(n)\cdot(n-1)}{2} \\ &= \frac{(n)\cdot(n-1)}{2}\end{aligned}$$

Som man måske kan huske fra analysen af `Insertionsort` er denne sum $\Theta(n^2)$, og vi kan konkludere at denne version af `Bubblesort` altid bruger $\Theta(n^2)$ tid, da vi ikke har antaget noget om tallene som algoritmen skal sortere.

`Insertionsort` har også worst-case køretiden $\Theta(n^2)$, men dette er kun for bestemet input (f.eks. omvendt sorteret) og køretiden bliver linær $\Theta(n)$ i best-case (sorteret input). Altså er `Insertionsort` mere effektiv når input er sorteret eller næsten sorteret, og kun i worst-case er `Insertionsort` ligeså langsom som `Bubblesort`.

8 Opgave 8 - Eksamen juni 2013, opgave 6

For at et udsagn er en løkke-invariant for algoritmen, skal det gælde at udsagnet altid er sandt før en iteration. Dette er det samme som 1) at udsagnet er sandt før første iteration, og 2) at hvis vi antager at udsagnet er sandt før en vilkårlig iteration, så er det også sandt efter iterationen (og dermed sandt før den næste iteration). Hvis begge krav er overholdt,

er udsagnet en løkke-invariant, og hvis bare én dem ikke er overholdt, er udsagnet ikke en løkke-invariant.

i)

1) Da i bliver sat til n, og vi er givet at $n \geq 1$, er udsagnet sandt før den første iteration.

2) Antag, at udsagnet gælder før den k'te iteration, dvs. $i \geq 1$. Lad værdien af i efter iterationen være i'. Fordi vi laver en iteration ved vi at $i > 1$ (det er while-condition, så hvis dette ikke var sandt havde vi slet ikke lavet en iteration). I iterationen sætter vi $i' = i - 1$, og da vi ved at $i > 1$, så er i mindst 2, dvs. det laveste i' kan blive er 1. Dvs. efter iterationen må det gælde at $i' \geq 1$, og vi har dermed vist, at udsagnet også er sandt efter iterationen.

Dvs. udsagnet er en løkke-invariant.

ii)

1) Til at starte med er $i = n$ og $r = 1$. Hvis input f.eks. er $n = 2$, har vi at $i = 2$, $r = 1$, og da $i! = 2 \neq r$, så gælder udsagnet ikke inden første iteration.

Dvs. udsagnet er ikke en løkke-invariant.

iii)

1) Til at starte med er $i = n$ og $r = 1$. Derfor har vi at $r! \cdot i! = 1! \cdot n! = n!$, og udsagnet er derfor sandt før den første iteration.

2) Lad os se på indholdet af i og r efter den første iteration. Her må det gælde at $r = n$ og $i = n - 1$. Ifølge vores udsagn må det derfor gælde at $r! \cdot i! = n!$, dvs. det må gælde at $n! \cdot (n-1)! = n!$, men dette er klart falsk. Da udsagnet ikke holder efter den første iteration, er det dermed ikke en løkke-invariant.

Dvs. udsagnet er ikke en løkke-invariant.

iv)

1) Til at starte med er $i = n$ og $r = 1$. Hvis vi indsætter det i udsagnet får vi at $r = \frac{n!}{i!} = \frac{n!}{n!} = 1$, som er sandt. Udsagnet er derfor sandt før den første iteration.

2) Antag, at udsagnet gælder før den k'te iteration. Lad indholdet af variablerne før den k'te iteration være i og r, og efter den k'te iteration være i' og r'. Vi kan se, at efter iterationen må indholdet være $i' = i - 1$ og $r' = r \cdot i$ (det er det som body i løkken sætter variablerne til). Lad os se om udsagnet stadig gælder, ved at indsætte værdierne for i' og r' udtrykt vha. de gamle værdier (som vi ved at udsagnet gælder for):

$$r' = \frac{n!}{i'!} \iff r \cdot i = \frac{n!}{(i-1)!} \iff r = \frac{n!}{(i-1)! \cdot i} = \frac{n!}{i!}$$

Vi kan se, at vi kommer frem til, at udsagnet også er sandt for variablerne efter den k'te iteration. (Fordi vi gik ind i løkken ved vi at $i > 1$, dvs. vi kommer ikke til at dividere med 0, og man kan også se, at r og i altid vil være positive, dvs. at tage fakultet af dem er veldefineret).

(Hvis man ikke er så fortrolig med ovenstående: først indsætter vi værdierne for i' or r' udtrykt vha. i og r, dvs. den første ligning er ækvivalent med den anden ligning. Herefter dividerer vi med i på begge sider, dvs. den anden ligning er ækvivalent med den tredje ligning, og den tredje ligning ved vi er sand, da $(i-1)! \cdot i = i!$ og per vores antagelse, dvs. den første ligning som er udsagnet omkring variablerne efter den k'te iteration er ækvivalent med noget der er sandt, dvs. den ligning er også sand).

Dvs. udsagnet er en løkke-invariant.

v)

1) Til at starte med er $r = 1$, og for f.eks. $n = 2$, har vi at $r = 1 \neq n! = 2! = 2$, dvs. udsagnet er ikke sandt før den første iteration.

Dvs. udsagnet er ikke en løkke-invariant.

9 Opgave 9 - Eksamen juni 2009, opgave 2

Vi kigger på pseudokoden i figur 8.

```
1   Kvadratrod(n):
2       i = 0
3       s = 0
4       while s <= n
5           s = s + 2 * i + 1
6           i = i + 1
7       r = i - 1
8       return r
```

Figur 8: psudokode for estimering af kvadratrod for n

Spørgsmål a

Vi skal vise korrekthed for følgende invariant for while-løkken: i starten af hver while-løkke er $s = i^2$.

Antagelse: I starten af hver while-løkke er $s = i^2$.

Basis skridt: Før første gennemløb er $i = s = 0$. Invarianten holder trivielt.

Induktions skridt: Lad i' og s' være værdierne for i og s i starten af næste iteration. Vi ved at $i' = i + 1$. Vi antager nu at invarianten holder for i og s og vil vise at det så også holder for i' og s':

$$\begin{aligned} s' &= s + 2i + 1 \\ &= i^2 + 2i + 1 && \text{(omskrivning af } s \text{ via antagelse } s = i^2) \\ &= (i+1)^2 && \text{(omskrivning via kvadratsætning)} \\ &= i'^2 && \text{(omskrivning via } i' = i + 1) \end{aligned}$$

Invarianten holder ved starten af hvert gennemløb af while-løkken.

Spørgsmål b

Vi skal argumentere for at r er det største heltal mindre end eller lig med $\sqrt{n}$.

Vi lader i' og s' være værdierne for i og s fra starten af næste gennemløb af while-løkken i pseudokoden. Så ved vi at:

$$i' = i + 1$$

$$i = i' - 1$$

Lad while-løkken stoppe ved næste iteration, så bliver

$$i'^2 = s' > n$$

fordi while-løkken stopper netop når s overstiger n. Værdierne for forrige gennemløb er så (her stoppede vi ikke, så derfor må $s \leq n$):

$$i^2 = s \leq n \Leftrightarrow (i' - 1)^2 \leq n$$

Hvis vi tager kvadratroden af det samlede udtryk, så får vi.

$$(i' - 1)^2 \leq n < i'^2$$

$$\sqrt{(i' - 1)^2} \leq \sqrt{n} < \sqrt{i'^2}$$

$$(i' - 1) \leq \sqrt{n} < i'$$

I den andensidste iteration vil $i \leq \sqrt{n}$. I det sidste gennemløb af while-løkken er $\sqrt{n} < i'$. Da der kun bliver lagt en til i for hvert gennemløb vil i og i' ligge rundt om $\sqrt{n}$. I slutningen af algoritmen sætter vi $r = i' - 1$ - vi har altså at $r \leq \sqrt{n} < r + 1$.

Eksaminatorier II

10 Opgave 1 - Cormen et al. øvelse 4.4-1

Brug rekursionstræmetoden til at bestemme en asymptotisk øvre grænse for

$$T(n) = 3 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

og benyt Master Theorem som tjek.

Som beskrevet af [1, s. 67], så ignorerer man typisk *floor* og *ceiling* funktioner i rekursionsligninger, da de normalt ikke har betydning for løsningen.[4] Derfor betragter vi i stedet bare

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + n$$

hvor fanout er 3, delproblemerne halveres ved hvert rekursivt kald, og det lokale arbejde er linært (altså $f(n) = n$).

I Figur 9 er rekursionstræet for rekursionsligningen illustreret. Ud fra figuren observerer vi, at arbejdet i det i'te lag er

$$3^i \cdot \frac{n}{2^i} = \left(\frac{3}{2}\right)^i \cdot n$$

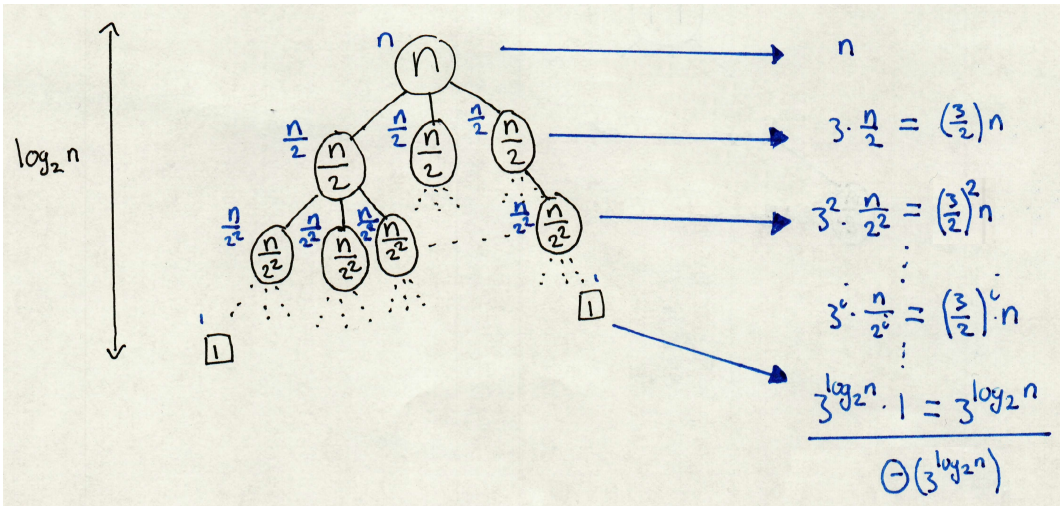
Da $\frac{3}{2} > 1$, så stiger arbejdet eksponentielt ned gennem lagene, og det er det sidste lag som dominerer. Da fanout er 3, så er der 3^h knuder i det sidste lag, hvor h er højden. Da vi halverer problemstørrelsen ved hvert rekursivt kald, så er højden givet ved at bestemme, hvor mange gange man skal halvere den originale problemstørrelse før man rammer basistilfældet. Altså, man skal løse

$$\frac{n}{2^i} = 1 \Leftrightarrow n = 2^i \Leftrightarrow i = \log_2 n$$

Der er altså $3^{\log_2 n}$ knuder i det sidste lag, hvor hver knude repræsenterer noget arbejde, som kan løses i konstant tid. Derfor er løsnignen til rekursionsligningen

$$T(n) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.5849\ldots})$$

[4] I dette tilfælde er vi kun interesseret i at finde en øvre grænse for rekursionsligningen. Lad $T'(n) = \Theta(g(n))$ være løsningen til rekursionsligningen, hvor *floor* funktionen ignoreres. Da $\lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$, så må $T(n) \leq T'(n)$ og endvidere må det følge at $T(n) = O(g(n))$. Vi er altså på den "sikre side", hvis det nu viser sig *floor* funktionen havde betydning, da vi kun ønsker at finde en asymptotisk øvre grænse.



Figur 9

Som tjek benytter vi Master Theorem. Vi starter med at se, om vi kan løse rekursionsligningen med Master Theorem. Umiddelbart ser det ud til at vi kan, da rekursionsligningen har den rigtige form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

hvor $a = 3$, $b = 2$ og $f(n) = n$.

Vi kan nu udregne α som $\alpha = \log_b a = \log_2 3$.

Vi ser på de 3 cases, og ser at $f(n)$ er asymptotisk mindre end n^α, da

$$f(n) = n = O(n^{\alpha-\epsilon}) = O(n^{\log_2 3 - 0.01}),$$

hvor $\epsilon = 0.01$. Derfor kan Case 1 af Master Theorem benyttes, og det følger rekursionsligningen har løsningen

$$T(n) = \Theta(n^\alpha) = \Theta(n^{1.5849\ldots})$$

Dette passer med resultatet fundet vha. rekursionstræmetoden.

Vi var egentlig interesseret i at finde en asymptotisk øvre grænse for $T(n)$, men da $T(n) = \Theta(n^{1.5849\ldots})$, så følger det at $T(n) = O(n^{1.5849\ldots})$ (se evt.[2, s. 16]).

11 Opgave 2 - Cormen et al. øvelse 4.4-2

Brug rekursionstræmetoden til at bestemme en asymptotisk øvre grænse for

$$T(n) = T\left(\frac{n}{2}\right) + n^2$$

og benyt Master Theorem som tjek.
Fanout er 1, delproblemerne halveres ved hvert rekursivt kald, og det lokale arbejde er kvadratisk (altså $f(n) = n^2$).
Da fanout kun er 1, så består hvert lag i rekursionstræet kun af en knude. Arbejdet for de første 5 lag er:

1. n^2
2. $(n/2)^2$
3. $(n/4)^2$
4. $(n/8)^2$
5. $(n/16)^2$

Altså er arbejdet for det i'te lag:

$$\left(\frac{n}{2^i}\right)^2 = \left(\frac{n^2}{4^i}\right) = n^2 \cdot \left(\frac{1}{4^i}\right) = n^2 \cdot \left(\frac{1}{4}\right)^i$$

Fordi arbejdet bliver exponentialt mindre for hvert lag man går ned, så må det øverste lag dominere køretiden. Da køretiden for det øverste lag er $n^2 = O(n^2)$, så må dette også være den asymptotiske øvre grænse for rekursionen.

Master Theorem bliver brugt som tjek. Der tjekkes om ligning er på den rigtig form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

hvor $a = 1$, $b = 2$ og $f(n) = n^2$.
Vi kan nu udregne α, $\alpha = \log_b a = \log_2 1 = 0$.
Hvis der kigges på de tre cases [4, p.19], så rammes case 3, da

$$f(n) = \Omega(n^{\alpha+\epsilon})$$

$$n^2 = \Omega(n^{0+\epsilon})$$

Hvis epsilon ϵ vælges til at være 1.

$$n^2 = \Omega(n^{0+1}) = \Omega(n)$$

Der skal dog også vises, at der findes et $c < 1$ og et n_0 således at

$$a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$$

for alle n større eller lig n_0.
Med $a = 1$, $b = 2$ og $f(n) = n^2$,

$$\left(\frac{n}{2}\right)^2 \leq c \cdot n^2$$

Hvis c vælges til at være $\frac{1}{4}$:

$$\left(\frac{n}{2}\right)^2 \leq \frac{1}{4} \cdot n^2$$

$$\frac{n^2}{4} \leq \frac{1}{4} \cdot n^2$$
$$n^2 \cdot \frac{1}{4} \leq \frac{1}{4} \cdot n^2$$

Da dette må gælde for alle n værdier, så må case 3 gælde for denne rekursionsligning.
I case 3 gælder $T(n) = \Theta(f(n)) = \Theta(n^2)$.
Altså bliver svaret bekræftet af Master Theorem.

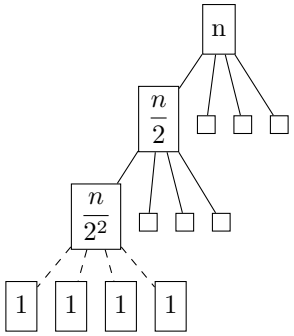
12 Opgave 3 - Cormen et al. øvelse 4.4-7

Tegn rekursionstræet og bestem en asymptotisk øvre grænse for

$$T(n) = 4 \cdot T(\left\lfloor \frac{n}{2} \right\rfloor) + n$$

Verificer resultatet med Master theorem (se ugeseddel).

I stedet for at tegne hele træet er venstrestien vist i figur 10, alle de tomme firkanter har også børn lige som deres søskend længst til venstre. Højden, h, er i træet $\log_2(n)$, da n bliver halveret for hvert lag.



Figur 10: Rekursiontræet

For at finde køretiden kigger vi på arbejdet der bliver lavet i hvert lag:

1. lag: Her er kun roden, vi laver derfor n arbejde i det her lag
2. lag: Roden har 4 børn som alle laver $\frac{n}{2}$ arbejde, det giver $4 \cdot \frac{n}{2} = (4/2)n$ arbejde i det her lag
3. lag: Rodens 4 børn har alle 4 børn, det giver 16 børn som alle laver $\frac{n}{2^2}$ arbejde, det giver $4^2 \cdot \frac{n}{2^2} = (4/2)^2 n$ arbejde i det her lag
4. lag: Der er nu 4^3 knuder i dette lag som alle laver $\frac{n}{2^3}$ arbejde, det giver $4^3 \cdot \frac{n}{2^3} = (4/2)^3 n$ arbejde i det her lag

5. I det h'ende (sidste) lag er der 4^h blade som alle laver konstant arbejde, da n på dette tidspunkt er 1. Det giver $4^h \cdot 1 = 4^{\log_2(n)}$ arbejde (h er højden).

Da arbejdet i hvert lag vokser eksponentielt ned gennem træet, så dominerer det sidste lag (se slide 18 i [4]). Vi har fundet arbejdet for det nederste lag, hvilket var $4^{\log_2(n)}$ og det kan vi omskrive til følgende ved brug af fact 2 på slide 15 i [4]:

$$4^{\log_2(n)} = n^{\log_2(4)} = n^2$$

Vi kan derfor konkludere at $T(n) = \Theta(n^2)$ for den givne rekursionsligning.

Nu mangler vi bare at verificere dette via Master Theorem. Kigger man på slide 19 i [4] udregner vi først $\alpha = \log_b(a) = \log_2(4) = 2$. Kigger man på første case skal vi se om $f(n) = O(n^{\alpha-\epsilon})$. I vores rekursion ligning er $f(n) = n$, og vælger vi f.eks. $\epsilon = 0.1$ kan man hurtigt se at $n = O(n^{1.9})$. Derfor gælder det at $T(n) = \Theta(n^2)$, hvilket også var det vi fandt ud af med vores analyse af rekursionstræet.

13 Opgave 4 - Eksamen juni 2010, opgave 1a

Vi er givet rekursionsligningen:

$$T(n) = 16 \cdot T\left(\frac{n}{2}\right) + n^4 + n^2$$

og vi skal løse den med Master Theorem. Rolf præsenterer Master Theorem på slide 19 [4].

Vi starter med at se, om vi kan løse rekursionsligningen med Master Theorem. Umiddelbart ser det ud til at vi kan, da rekursionsligningen har den rigtige form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

hvor $a = 16$, $b = 2$ og $f(n) = n^4 + n^2$.

Vi kan nu udregne α som $\alpha = \log_b a = \log_2 16 = 4$.

Vi ser på de 3 cases, og ser at $f(n)$ asymptotisk vokser med samme hastighed som n^α, altså

$$f(n) = n^4 + n^2 = \Theta(n^\alpha) = \Theta(n^4)$$

dvs. vi er i case 2 og kan bruge Master Theorem, og rekursionsligningen har dermed løsningen

$$T(n) = \Theta(n^\alpha \log n) = \Theta(n^4 \log n)$$

14 Opgave 5 - Eksamen januar 2006, opgave 1c

Vi kigger på følgende rekursions-ligning og vil finde dens asymptotiske løsning.

$$T(n) = 3 \cdot T\left(\frac{n}{3}\right) + n^2$$

Vi kan bruge Master Theorem fra [1] side 94 til at finde denne løsning. Vi indsætter værdierne i Master Theorem's formen

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

og ser at $a = 3,\ b = 3,\ f(n) = n^2$.

Vi kan nu udregne $\log_b(a) = \log_3(3) = 1$ og bruge det til at finde ud af hvilket case vi er i.

Hvis vi skal være i case 1 skal $f(n) = O(n^{\log_b(a)-\epsilon})$ for $\epsilon > 0$. Dette kan vi ikke få til at passe da n^2 altid vil være asymptotisk større end $n^{1-\epsilon}$ ligegyldigt hvor lille vi gør ϵ.

Hvis vi skal være i case 2 skal $f(n) = \Theta(n^{\log_b(a)})$. Dette kan ikke lades at gøre da vi ikke kan finde en konstant c så $n^2 \leq c \cdot n^1$ (definition for Θ, [1] side 44).

Hvis vi skal være i case 3 skal $f(n) = \Omega(n^{1+\epsilon})$ hvor $\epsilon > 0$ og $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ hvor $c < 1$. Dette kan vi godt få til at passe med $\epsilon = 1$ og $c \geq \frac{1}{3}$.

Når $\epsilon = 1$ får vi $n^2 = \Omega(n^2)$ hvilket er det samme som $n^2 \geq k \cdot n^2$ hvor k kan være en hvilken som helst konstant mindre end eller lig med 1.

For krav nummer 2 i case 3 kan vi se at

$$\begin{aligned}
&a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n) \\
&\Downarrow \\
&3 \cdot \left(\frac{n}{3}\right)^2 \leq c \cdot n^2 \\
&\Downarrow \\
&\frac{3 \cdot n^2}{3^2} \leq c \cdot n^2 \\
&\Downarrow \\
&\frac{n^2}{3} \leq c \cdot n^2
\end{aligned}$$

Hvis vi så sætter c til at være større end eller lig med $\frac{1}{3}$ vil vi overholde krav nummer 2.

Derfor bliver den asymptotiske løsning til rekursionsligningen, $T(n) = \Theta(f(n)) = \Theta(n^2)$

15 Opgave 6 - Cormen et al. øvelse 4.4-3

Brug rekursionstræmetoden til at bestemme en asymptotisk øvre grænse for

$$T(n) = 4 \cdot T\left(\frac{n}{2} + 2\right) + n$$

og benyt Master Theorem som tjek (hvis det er muligt).

I Figur 11 er rekursionstræet for rekursionsligningen illustreret. Ud fra figuren observerer vi, at problemstørrelsen i en knude i det i'te lag er

$$\frac{n}{2^i} + \frac{2}{2^{i-1}} + \cdots + \frac{2}{2^1} + \frac{2}{2^0} = \frac{n}{2^i} + 2 \cdot \sum_{j=0}^{i-1} \left(\frac{1}{2}\right)^j$$

Dette er dog højst[5]

$$\begin{aligned}
\frac{n}{2^i} + 2 \cdot \sum_{j=0}^{i-1} \left(\frac{1}{2}\right)^j &< \frac{n}{2^i} + 2 \cdot \sum_{j=0}^{\infty} \left(\frac{1}{2}\right)^j \\
&= \frac{n}{2^i} + 2 \cdot \frac{1}{1-\frac{1}{2}} \\
&= \frac{n}{2^i} + 4
\end{aligned}$$

Højden bestemmes på samme måde som tidligere, men her er basis-tilfældet når $n \leq 5$. Højden er det mindste i hvorved

$$\frac{n}{2^i} + 4 \leq 5 \;\Leftrightarrow\; \frac{n}{2^i} \leq 1 \Leftrightarrow n \leq 2^i \;\Leftrightarrow\; \log_2 n \leq i$$

Altså er højden $\log_2 n$.

I det i'te lag er der 4^i knuder, så arbejdet er højst

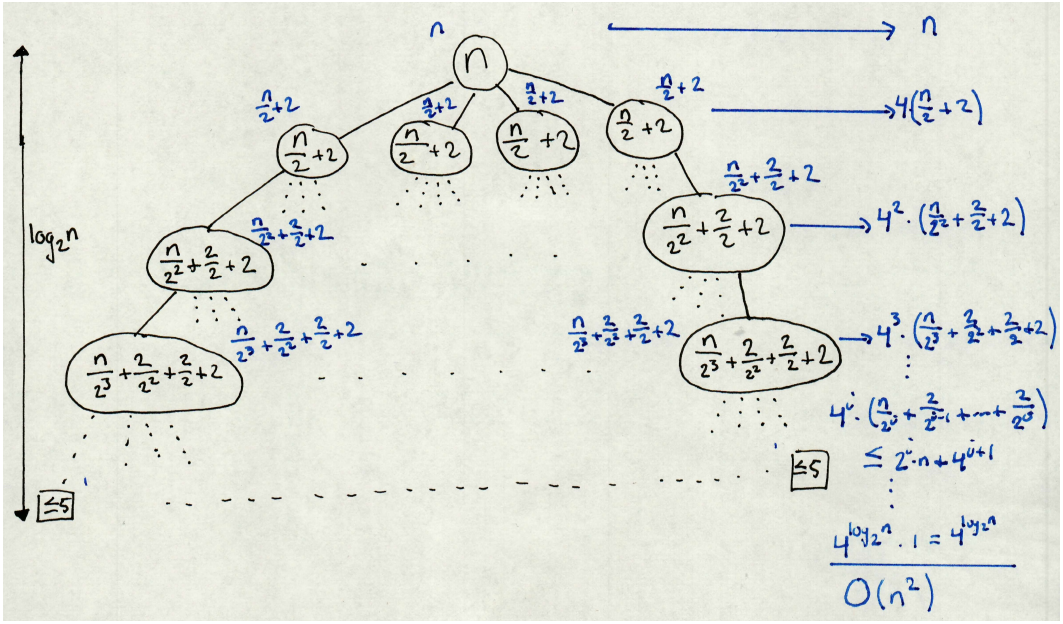
$$4^i \cdot (\frac{n}{2^i} + 4) = 2^i \cdot n + 4^{i+1}$$

Lægges arbejdet sammen for alle lagene fås

$$\begin{aligned}
\sum_{i=0}^{\log_2 n} 2^i \cdot n + 4^{i+1} &= n \cdot \sum_{i=0}^{\log_2 n} 2^i + 4 \cdot \sum_{i=0}^{\log_2 n} 4^i \\
&\overset{(*)}{=} n \cdot \frac{2^{\log_2(n)+1} - 1}{2-1} + 4 \cdot \frac{4^{\log_2(n)+1} - 1}{4-1} \\
&= n \cdot \frac{2 \cdot n^{\log_2(2)} - 1}{1} + 4 \cdot \frac{4 \cdot n^{\log_2(4)} - 1}{3} \\
&= n \cdot (2n-1) + \frac{4}{3} \cdot (4n^2 - 1) \\
&= (2 + \frac{16}{3})n^2 - n + \frac{4}{3} = \Theta(n^2)
\end{aligned}$$

I ovenstående bruger vi bruger problemstørrelsen for en knude i lag i højst er $\frac{n}{2^i} + 4$, så vi kan ikke direkte konkludere $T(n) = \Theta(n^2)$, men det følger at $T(n) = O(n^2)$.

[5] Grunden til det højst er $\frac{n}{2^i} + 4$ skyldes, at vi lader summen på mod ∞. Den brugte formel findes i Rolf's formelsamling til kurset her.



Figur 11

Master Theorem kan ikke bruges i dette tilfælde, da rekursionsligningen ikke er på formen

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Dette skyldes det $+2$ som er i rekursionsligningen.

(*): Allerede her kan man i princippet indse, at vi har at gøre med to eksponentielt voksende summer, hvor sidste led dominerer, og derfor følger det at

$$\begin{aligned} n \cdot \sum_{i=0}^{\log_2 n} 2^i + 4 \cdot \sum_{i=0}^{\log_2 n} 4^i &= n \cdot \Theta(2^{\log_2 n}) + 4 \cdot \Theta(4^{\log_2 n}) \\ &= n \cdot \Theta(n) + 4 \cdot \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

Litteratur

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] Rolf Fagerberg. Asymptotisk analyse af algoritmers køretider. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/asymptotiskAnalyseAfAlg.pdf`, 2020.

[3] Rolf Fagerberg. Bineare søgetræer med ekstra information i knuderne. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/augmentedBSTSlides.pdf`, 2020.

[4] Rolf Fagerberg. Divide-and-Conquer algoritmer. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/divideAndConquerSlides.pdf`, 2020.