INF2220: Algoritmer og datastrukturer
Pensumsammendrag

Mathias Lohne
mathialo@student.matnat.uio.no

Høsten, 2015

Noen kommentarer

Dette notatet har i hovedsak fungert som min egne lille gummiand i eksamenstreninga, men kan være nyttig til repetisjon og som notater under eksamen. Vær oppmerksom på at det sikkert inneholder feil, mangler og alt mulig sånt. Finner du noen teite blemmer er det kult om du sender meg en mail: `mathialo@student.matnat.uio.no`

Innhold og eksempler er basert på stoff fra læreboka, forelesninger, tidligere eksamener, mine egne obliger og et tilsvarende kompendium laget av Veronika Heimsbakk[1].

Innhold

[1] http://folk.uio.no/veronahe/

1 Kompleksitet og tidsanalyse

1.1 Terminologi og begreper

1.1.1 Alfabeter og språk

Når vi bruker begrepene alfabet og språk snakker vi som regel ikke om språk som engelsk, norsk eller Java (selv om disse også er språk i formell forstand), men om en samling strenger av tegn (tenk: ord). Hvilke tegn vi kan bruke avhenger av hvilket alfabet vi har.

Et **alfabet** er en ikke-tom mengde av tegn (også kalt symboler og bokstaver). Vi betegner ofte et alfabet med Σ. Eksempler på alfabeter kan være det binære alfabetet $\{0, 1\}$ eller det norske alfabetet {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, æ, ø, å}.

Gitt et alfabet Σ bruker vi Σ^* for å betegne mengden av alle mulige kombinasjoner (strenger) av tegn fra Σ av en endelig lengde. En mengde av strenger, med eller uten noen bestemte regler, kalles et **språk**. Vi konstruerer språk enten ved å liste opp alle ordene i språket, eller ved å gi noen regler som ordene i språket må følge. Under følger noen eksempler på språk, og hvordan de defineres:

- $L = \{$*"a"*, *"b"*, *"ab"*, *"ba"*$\}$ (eksempel på et endelig språk)
- $L = \Sigma^*$ (alle ordene over Σ) (eksempel på et uendelig språk)
- $L = \{i : M$ stopper på input $i\}$ (Den inputen som gjør at en turingmaskin stopper. En variant av haltingproblemet, se 8.3.1)

Eksempel. Gitt alfabetet $\{0, 1, ‘,’\}$, hva er det formelle språket som tilsvarer sorteringsproblemet over dette alfabetet?

Svar:

$$L = \{(0), (1), ..., (0, 1), (0, 10), (1, 10), ...\}$$

1.1.2 Turingmaskinen

Turingmaskinen er en teoretisk maskin. Den eksisterer ikke i virkeligheten, men er noe vi bruker for å bevise teoremer. Selvfølgelig finnes det ingen formell definisjon på *hva* en Turingmaskin egentlig er, så alle læreverk definerer den litt forskjellig. Det høres kanskje helt Texas ut, men alle definisjonene er tilnærmet ekvivalente.

Definisjon 1. Litt forenklet kan vi si at en Turingmaskin $M = (\Sigma, \Gamma, Q, \delta)$ består av fire komponenter:

- Et inputalfabet Σ. Alle mulige tegn Turingmaskinen kan forstå.
- Et teipalfabet Γ. Alle mulige tegn som kan finnes på teipen. Σ er alltid inneholdt i Γ. Γ inneholder også et blankt symbol (mellomrom), og kan inneholde andre symboler.

- En liste Q over mulige statuser for Turingmaskinen.
- En liste δ over 'responser' på input. For eksempel: *"Hvis jeg er i status 7 og leser en 'a' skal jeg gå til status 21"*.

Hvis vi skal prøve å se for oss en Turingmaskin intuitivt kan vi se for oss et uendelig lang papirremse (formelt kalt teip). På denne papirremsa står det symboler (fra Σ). Dette er inputen til Turingmaskinen. Maskinen leser ett og ett symbol, og reagerer på symbolet (etter hva δ sier den skal gjøre). Det kan deretter la symbolet stå, viske det vekk eller erstatte det med et nytt symbol (fra Γ).

1.2 Kompleksitetsklasser

Noen problemer kan løses veldig enkelt, for eksempel som å søke i et binært søketre, andre problemer er mye mer kompliserte, og noen er uløselige. I dette kurset skiller vi i hovedsak mellom P, NP og uløselige problemer, selv om kompleksitetsklasser er mye finere oppdelt enn det.

Hvorfor kompleksitetsklasser? - Si noe om computaility - problemer i samme klasse kan ha lignende løsning

P (polynomial time) P er mengden av alle problemene som kan løses i polynomisk tid, altså har tid på formen $O(n^k)$ for en $k \in \mathbb{N}$

NP (nondeterministic polynomial time) NP er mengden av alle problemer hvor vi ikke vet hvor lang tid det vil ta i forkant å finne en løsning, men som blir gjort i polynomisk tid. NP-problemer er vanskelige å regne ut, men en gitt løsning enkelt lar seg sjekke. Hvis jeg for eksempel ber deg faktorisere et stort tall n, vil det ta lang tid for deg å gjøre det, men det er lett for meg å gange sammen de tallene du påstår er faktorene til n og se om jeg får n. Hele P er inneholdt i NP, og det er usikkert hvorvidt det er en reell forskjell på N og NP.

Vi har en undermengde av NP kalt NP-komplett. Denne mengden består av de vanskeligste problemene i NP, og hvis man har en løsning på et av disse problemene vil man, ved reduksjon, ha en løsning på alle andre problemer i NP, inkludert P.

EXPTIME (exponetial time) EXPTIME er mengden av alle problemer som løses og sjekkes i eksponentiell tid. Hvis jeg for eksempel ber deg fortelle meg hva det beste trekket jeg burde gjøre i et parti sjakk er, er det vanskelig for deg å regne ut, men også vanskelig for meg å sjekke om svaret ditt stemmer. Vi kaller problemer i denne klassen *"intractable"*. I mange tilfeller er brute force den eneste muligheten vi har.

Uløselige problemer En del problemer har ikke en mulig løsning. Et eksempel på et slikt problem er halting-problemet. Dette problemet går ut på om en turingmaskin kan vite om den noen gang vil gi et resultat (det vil si stoppe, engelsk: halt), eller om den vil gå i evig loop. Et bevis for hvorfor haltingproblemet er uløselig finnes i 8.3.1.

Vi skal se på noen eksempler på problemer:

Eksempel. Traveling salesperson (TSP)

Et av de mest kjente og studerte optimeringsproblemene kalles *Traveling salesperson*. Problemet går slik: En handelsmann har en liste med byer han må innom. Han vil reise innom hver by én gang, og vil bruke minst mulig penger på turen. Han vet prisen det vil koste å reise mellom hver by. Hvilken rekkefølge burde handelsmannen besøke byene i for å betale minst mulig i reisepenger?

Dette problemet er av eksponentiell karakter. Vi kan ikke si noe om hvilken vei som blir billigst uten å sjekke alle muligheter. Problemet er NP-komplett.

Eksempel. Subset sum

Problemet er slik: Gitt en mengde M av tall, skal vi avjøre om det finnes en delmengde $M' \subset M$ slik at

$$\sum_{m \in M'} m = 0$$

Mengden $A = \{-2, -3, 1, 4, 6\}$ er en slik mengde, siden $1 + 4 + (-2) + (-3) = 0$. Mengden $B = \{-6, -3, 1, 4, 7\}$ er ikke en slik mengde, siden det ikke er mulig å plukke ut noen elementer slik at summen av elementene blir 0.

Å løse dette problemet er ganske vanskelig siden det er NP-komplett. Vi må (slik som i TSP) prøve oss fram med forskjellige kombinasjoner.

1.3 O-notasjon

Når vi skal analysere kjøretid er vi sjeldent opptatt av et nøyaktig svar, men mer opptatt av hva slags størrelsesorden kjøretiden befinner seg i. Dette er litt av motivasjonen for O-notasjon. Formelt kan vi definere det slik:

Hvorfor bigO? vil vite noe om generell tid. faktisk tid vil avhenge av maskin etc.

Definisjon 2. La f og g være to funksjoner $f, g : \mathbb{N} \to \mathbb{R}$. Vi sier da at $f(n) = O(g(n))$ hvis det eksisterer positive heltall c og N slik at for hvert heltall $n \geq N$ er $f(n) \leq c\, g(n)$

$O(g(n))$ blir dermed en øvre skranke for kjøretid.

Når vi i denne sammenhengen bruker O-notasjon vil vi bruke det som et mål på hvordan kjøretiden øker med inputen. Vi ser på et eksempel:

Eksempel. Gitt et tall n skal vi finne alle mulige heltall som n er delelig med. En veldig enkel algoritme for å løse dette er å forsøke å dele på alle tallene fra 1 til n.

Hvis $n = 5$ vil dette gå raskt, da vi bare må teste 5 mulige utfall ($n/k, k \in \{1, 2, 3, 4, 5\}$). Hvis $n = 139\,823$ blir problemet mer komplisert, og algoritmen vil måtte gjøre mange flere tester. Generelt må vi gjøre n tester for et inputtall n.

Vi ser at kjøretiden øker lineært med størrelsen på input. Vi sier derfor at algoritmen bruker $O(n)$ tid.

Som nevnt tidligere er vi mer opptatt av størrelsesorden enn den konkrete kjøretiden. Vi bryr oss derfor ikke om konstanter. Hvis vi i eksempelet hadde måttet gjøre 2 tester for hver input hadde vi fortsatt hatt $O(n)$ tid, selv om kjøretiden hadde vært $T(n) = 2n$. Vi kan sette opp noen regneregler for O-notasjon:

Teorem 1. Regneregler for O-notasjon:

i $O(k\,g(n)) = O(g(n)) \quad$ for $k = 1, 2, ...$

ii La f og g være to funksjoner, og anta at det finnes et tall N slik at $f(n) > g(n)$ for alle $n > N$. Da er $O(f(n) + g(n)) = O(f(n))$

Bevis. Teorem 1, del i

La f og g være to funksjoner $f, g : \mathbb{N} \to \mathbb{R}$ og la k være et tall $\in \mathbb{N}$. Videre antar vi at $f(n) = O(g(n))$. Da har vi fra definisjon 2 at $f(n) \leq c\,g(n)$ for en konstant c. Vi setter $g'(n) = k\,g(n)$ for et positivt heltall k. Da har vi at

$$f(n) \leq c\,g(n) \leq c\,k\,g(n) = c\,g'(n)$$

Dermed er $f(n) = O(g'(n))$, og fra antagelsen har vi at $f(n) = O(g(n))$. Vi har derfor at $O(k\,g(n)) = O(g(n))$ □

Beviset for del ii følger samme strategi. Det teorem 1 del ii egentlig sier er at vi kun bryr oss om den 'største' funksjonen. Har vi for eksempel kjøretid lik $T(n) = n^2 + n$ har vi kun $O(n^2)$.

Eksempel. Beregning av kjøretid. Vi har gitt følgende program

```
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        // Do something simple...
    }
}

for (int i=0; i<n-3; i++) {
    // Do something else...
}
```

og skal beregne worst case kjøretid til programmet. Vi ser at den indre for-løkka i den øverste for-løkka starter på `i`, og ikke på 0. Fra teorem 1 del i har vi at det

ikke har noe å si. Vi regner med den løkka. Vi ser også at det er en enkel for-løkke etterpå, men fra teorem 1 del ii har vi at vi kan se bort fra den. Kjøretiden blir altså $O(n^2)$.

Eksempel. Finn kjøretid for følgende program: (Ex14 1b)

```
for (i=n; i >= 1; i = i/2) {
    for (j=1; j<n; j++) {
        // do something
    }
}
```

Vi ser at den indre løkka vil gå n ganger. Den ytre løkka halverer telleren hver gang, det vil si at den kjører $\log_2 n$ ganger. Kjøretiden blir derfor $O(n \log_2 n)$.

Eksempel. Finn kjøretid for følgende program: (Ex11 2b)

```
float foo(A) {
    n = A.length;

    if (n==1) {
        return A[0];
    }

    // let A1, A2, A3 and A4 be arrays of length n/2

    for (i=0; i<=n/2; i++) {
        for (j=0; j<=n/2; j++) {
            A1[i] = A[i];
            A2[i] = A[i+j];
            A3[i] = A[n/2 + j];
            A4[i] = A[j];
        }
    }

    b1 = foo(A1);
    b2 = foo(A2);
    b3 = foo(A3);
    b4 = foo(A4);
}
```

Her kan vi ikke like lett se løsninga siden funksjonen er rekursiv. Vi går sakte gjennom hva programmet gjør og forsøker å sette opp en funksjon for kjøretiden.

Vi ser at `foo` har to nestede for-løkker, hver av dem går $n/2$ ganger. Vi har derfor at hver gang vi kommer til denne løkka blir kjøretiden $T_{\text{løkke}}(m) = (m/2)^2$. Mot slutten av programmet har vi fire rekursive kall. Alle kallene kjører `foo` med input av lengde $n/2$. Vi kan dermed sette opp en funksjon for kjøretiden:

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

der C er en konstant. Vi kan sette inn for $T(n/2)$:

$$T(n) = C + 4\left(\frac{n}{2}\right)^2 + 4\left(C + 4\left(\frac{n/2}{2}\right)^2 + 4\,T\left(\frac{n/2}{2}\right)\right)$$

Igjen kan vi sette inn for $T(n/2)$, og slik kan vi fortsette. Siden vi halverer n hver gang ser vi at vi må gjøre dette $\log_2(n)$ ganger før vi får at $n = 1$, og rekursjonen stoppes av den øverste if-testen. Vi går over til O-notasjon slik at vi kan droppe konstanter. Vi har dermed at kjøretiden er

$$O\left(n^2 \log_2 n + n^2\right) = O\left(n^2 \log_2 n\right)$$

2 Paradigmer for algoritmedesign

2.1 Splitt og hersk

Splitt og hersk er en teknikk som i stor grad benytter seg av rekursjon. Vi deler problemet opp i mindre delproblemer, deler de delproblemene opp i mindre deldelproblemer også videre. Slik fortsetter vi helt til problemene er så små at løsningen er triviell. Deretter setter vi sammen løsningen på småproblemene til en løsning på hele problemet.

Eksempler på algoritmer som bruker denne teknikken er søking i binære trær (se 3.1) og Quicksort (7.2.4).

2.2 Grådige algoritmer

Grådige algoritmer er algoritmer som løser optimeringsproblemer. En grådig algoritme vil gå steg for steg gjennom problemet, og gjøre det som ser best ut på hvert tidspunkt.

Eksempler på grådige algoritmer er Dijkstras algoritme (se 4.2.2), Prims algoritme (4.2.3), Kruskals algoritme (4.2.4) og Huffmankoding (6.3).

2.3 Dynamisk programmering

Dynamisk programmering er en designteknikk som går ut på å forsøke å gjøre komplekse optimeringsproblemer enklere ved å dele problemet opp i mindre delproblemer, og løse dem hver for seg. Vi lagrer løsningene, og bruker resultatet fra dem til å konstruere med en endelig løsning. Prinsippet går ut på at en optimal løsning på hele problemet vil være et resultat av optimale løsninger på delproblemene. Det er ikke alltid tilfelle, men når det er det kan dynamisk programmering forbedre kjøretiden dramatisk.

For illustrere tankegangen skal vi se på et eksempel. Fibonaccitallene er definert rekursivt slik:

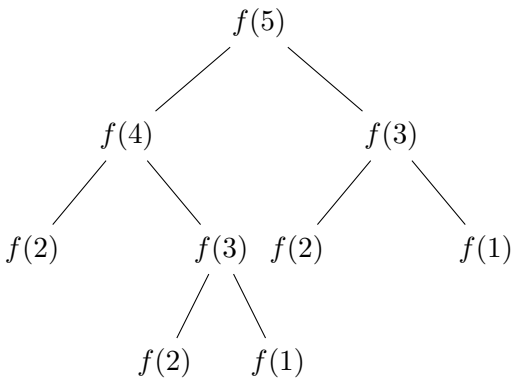
$$f(n) = \begin{cases} 1 & \text{for } n \in \{1, 2\} \\ f(n-1) + f(n-2) & \text{ellers} \end{cases}$$

Vi skal programmere en funksjon som regner ut $f(n)$. Det er fristende å gjøre det helt likt som definisjonen:

```
public int recursiveFib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    else {
        return recursiveFib(n-1) + recursiveFib(n-2);
    }
}
```

Dette er en grei og oversiktlig implementasjon, i den forstand at den finner tallet vi ber den om, men la oss foreta en liten tidsanalyse. For hvert tall vi ber den om må den regne ut to tall. For hver av disse to tallene må vi igjen regne ut to tall. Slik baller det på seg. Vi kan tegne opp et tre over funksjonskallene:

Figur 1: Funksjonskall for $f(5)$



Vi ser at vi vil regne ut $f(3)$ to ganger, det virker litt overflødig. Generelt vil denne algoritmen bruke $O(2^n)$ tid, som er veldig dårlig. Hvis vi prøver å tenke dynamisk kan vi løse problemet mye bedre. I stedet for å programmere fibonaccifunksjonen vår rekursivt vil vi gjøre det iterativt, og hvor vi lagrer løsningene underveis. Da kan vi, i stedet for å regne ut de foregående tallene, finne dem i en tabell.

```
public int dynamicFib(int n) {
    int last = 1, lastlast = 1, fibNum = 1;

    for (int i=3; i<=n; i++) {
        fibNum = last + lastlast;
        lastlast = last;
        last = fibNum;
    }

    return fibNum;
}
```

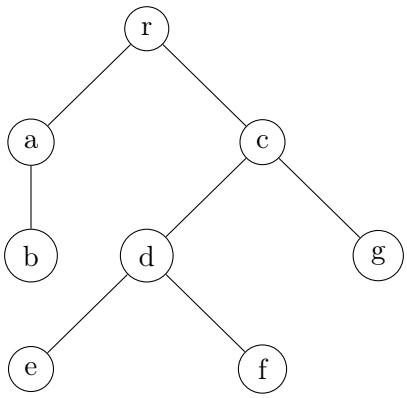
Denne koden er kanskje litt mindre intuitiv enn den forrige, men den er mye raskere! Her ser vi at algoritmen kun har én loop, og kjøretiden er opplagt $O(n)$.

Et eksempel på en algoritme som bruker denne teknikken er Floyds algoritme (se 4.2.5).

3 Trær

Et tre en et spesielt tilfelle av en rettet graf der hver node har inngrad 1 (med unntak av rota i treet). Vi ser på et eksempel:

Figur 2: Et lite binært tre



Terminologi

Vi skal se litt på ord og uttrykk for trær. Gjennomgående bruker vi treet i figur 2 som eksempel

I figuren blir nodene tegnet som rundinger. For å betegne relasjonen mellom nodene bruker vi ofte familierelasjoner. Vi sier at e og f er **søsken**, d er **forelder** til e, og e er **barn** av d. Vi kan også si at g er **onkel** til f, men dette er mindre vanlig, da vi sjeldent har bruk for å snakke om *"onkelnoder"*.

Nodene r, a, c og d kalles **indre noder**, det vil si at disse nodene har barn. Noder som ikke har noen barn kalles **løvnoder**

I figur 2 er r **rotnoden**. Rota i treet er den eneste noden uten noen foreldre. Rota er derfor et naturlig startpunkt når vi skal søke eller traversere gjennom treet.

3.1 Binære søketrær

Binære søketrær er trær med noen spesielle krav. Hver node kan ikke ha mer enn to barn, vi kaller dem ofte venstre og høyre barn. Venstre barn er alltid mindre enn noden selv, og høyre barn er alltid større enn noden. Dette gjør binære søketrær meget godt egnet for søking.

Teorem 2. Å sette inn, fjerne eller søke etter noder i et binært søketre har

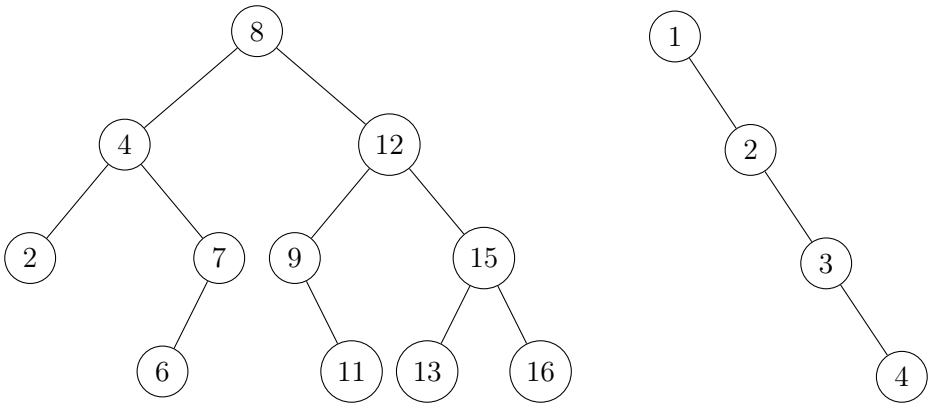
i I beste fall $O(\log n)$ tid

ii I verste fall $O(n)$ tid

Vi skal ikke bevise teorem 2 her[2], men vi kan se på et eksempel som viser ytterpunktene. I figur 3 ser vi to forskjellige binære søketrær. I treet til venstre har vi et fint, balansert tre. Det er lett å se at høyden (og dermed antall operasjoner vi må gjøre for å komme til bunn i treet) er lik $\lceil \log_2 n \rceil$ der n er antall noder i treet.

I treet til høyre har hver node kun ett barn. Vi har i praksis en lenkeliste. Hvis vi skal søke etter 4, må vi tråkke gjennom alle de andre nodene for å komme dit.

Figur 3: To eksempler på binære søketrær



Innsetting

Når vi skal sette inn en node i et binært søketre starter vi i rota. Vi sammenligner verdien vi skal sette inn med verdien i rota. Hvis verdien vi setter inn er mindre enn rota går vil til venstre, er den større går vi til høyre. Hva som skjer ved likhet er opp til oss å bestemme, men vi må være konsekvente. Når vi kommer til en nullpeker kan vi sette denne pekeren til å peke på noden vi setter inn.

Vi kan implementere denne funksjonen rekursivt. I en ytre klasse kan vi skrive en skallfunksjon som kaller på rotas `insert`-metode. Vi har en indre `Node`-klasse med en rekursiv insertmetode. Den kan implementeres slik:

```
public void insert(T otherElement) {
    int comparison = element.compareTo(otherElement);

    if (comparison > 0) {
        // This element is bigger than the one we're
        // inserting => left subtree
        if (leftChild == null) {

            // nothing there, insert
            leftChild = new Node(otherElement);
            return;
        } else {
```

[2]Det kan vises ved induksjon at høyden til et binært tre i beste fall er $\log_2 n$

```
            // occupied, move down
            leftChild.insert(otherElement);
        }

    } else if (comparison < 0) {
        // The element we're inserting is bigger than
        // this one => right subtree
        if (rightChild == null) {

            // nothing there, insert
            rightChild = new Node(otherElement);
            return;
        } else {

            // occupied, mode down
            rightChild.insert(otherElement);
        }
    } else {
        // The two elements are the same. Since this is a tree
        // for searching, we'll just forget about it
        return;
    }
}
```

Fjerning

Når vi skal fjerne en node fra et binært søketre har vi tre forskjellige situasjoner som vi må se på.

Noden har ingen barn (løvnode). Denne situasjonen er ganske grei. Siden noden ikke har noen barn å forholde seg til er det bare å fjerne den fra treet.

Noden har ett barn. For å fjerne en node a med ett barn kan vi ganske enkelt flytte pekeren fra foreldernoden til a, til barnet til a.

Noden har to barn. Hvis noden vi skal fjerne har to barn

```
public void remove() {
    if (leftChild == null && rightChild == null) {
        // no children, just unlink
        if (parent.leftChild == this)
            parent.leftChild = null;
        else
            parent.rightChild = null;

    } else if (leftChild == null || rightChild == null) {

        // one child: unlink, and set parents link to this' child
        if (parent.leftChild == this)
            parent.leftChild = this.leftChild;
        else
            parent.rightChild = this.rightChild;

    } else {
        // two children: take the smallest element to the right and
        // replace with this.
```

```
            Node replacement = rightChild.findSmallestChild();

            // I will edit the content of this and delete the
            // replacement node, instead of deleting this and edit the
            // pointers of the replacement node.
            this.element = replacement.getElement();
            replacement.remove();

        }

        // all hail the mighty garbage collector!
    }
```

Søking

kodeeksempel og sånt

3.2 Rød-svarte trær

Rødsvarte trær er binære søketrær med noen spesielle strukturkrav. Kravene er designet for å motkjempe skjevhet (som illustrert i figur 3), og dermed forbedre kjøretid.

Vi deler nodene opp i to kategorier, røde og svarte. Vi følger noen bestemte regler på hvordan vi skal farge nodene, og fargen på nodene avgjør som vi må benytte oss av rotasjon eller ikke (se 3.4). Fargen til en node er **ikke** statisk, den kan endres.

Definisjon 3. Et rød-svart tre er et binært søketre der hver node er farget enten rød eller svart slik at:

i Roten er svart.

ii Hvis en node er rød, må barna være svarte.

iii Enhver vei fra en node til en null-peker må inneholde samme antall svarte noder.

Insetting

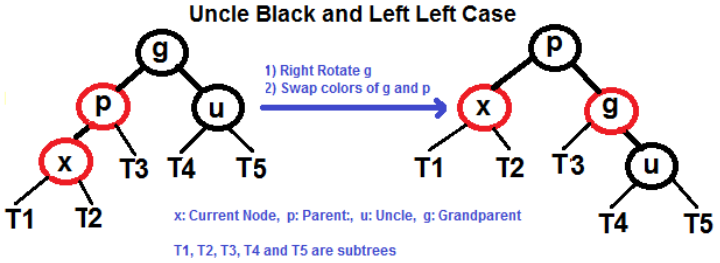
For å sette inn en node i rød-svarte trær kan vi bruke algoritmen i teorem 3

Teorem 3. Innsetting i rød-svarte trær.

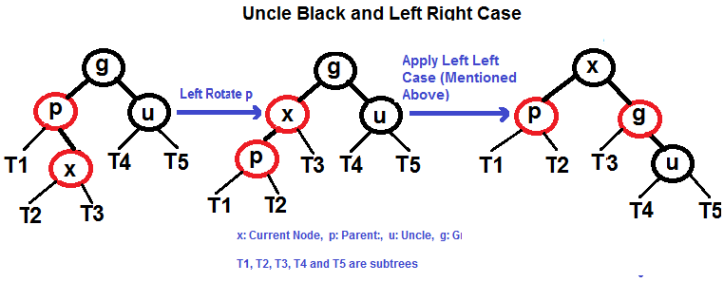
La N være noden vi ønsker å sette inn i treet. La P være forelder, G besteforelder, og U onkel til N. Følgende algoritme vil da gi en korrekt innsetting i et rød-svart tre:

1. Gjør innsetting som i vanlig binært søketre, der N farges rød.

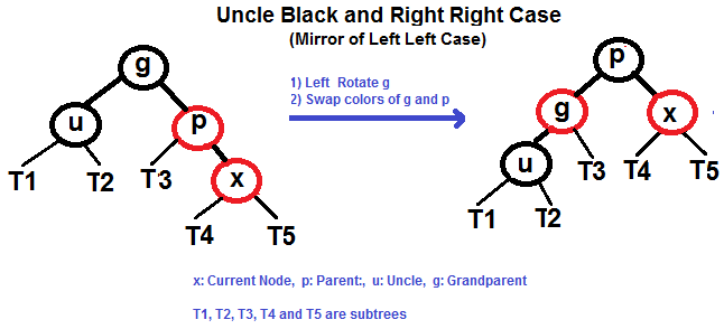
2. Hvis N er rota i treet: farg N svart.
3. Hvis P er svart: alt OK. Innsetting ferdig.
4. Hvis P er rød og N ikke er rot:
   (a) Hvis U er svart:
      i. Farg P svart
      ii. Farg G rød
      iii. Repeter fra pt. 2 med G som N
   (b) Hvis U er rød:
      i. N er venstre barn av P, som er venstre barn av G:



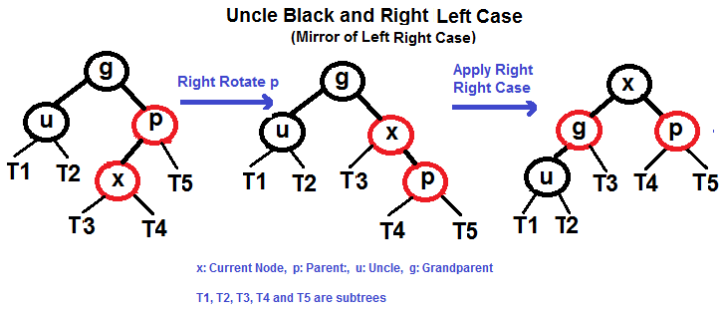
      ii. N er høyre barn av P, som er venstre barn av G:



      iii. N er høyre barn av P, som er høyre barn av G (speilvendt av i):



      iv. N er venstre barn av P, som er høyre barn av G (speilvendt av ii):



3.3 B-trær

B-trær er konstruert for å effektivisere antall disklesninger, og gir mening å bruke hvis vi har et tre som er så stort at det må lagres på gammeldagse spinnedisker, og ikke i RAM. Vi lagrer dataene i blokker, og leser en og en blokk av gangen. Alle dataene er lagret i løvnodene, mens de indre nodene brukes for søking. B-trær er ikke binære, dvs at de kan ha flere enn 2 barn.

Definisjon 4. La M angi antall mulige nøkler i hver indre node, og L angi maksimalt antall dataelementer i hver løvnode. B-trær er søketrær der følgende kriterier er oppfylt:

i Alle dataene er lagret i løvnodene

ii Interne noder lagrer inntil $M-1$ nøkler for søking: nøkkel i angir den minste verdien i subtre $i+1$.

iii Roten er enten en løvnode, eller har mellom 2 og M barn.

iv Alle andre indre noder har mellom $\lceil M/2 \rceil$ og M barn.

v Alle løvnoder har samme dybde.

vi Alle løvnoder har mellom $\lceil L/2 \rceil$ og L dataelementer

Innsetting

3.4 Rotasjon

3.4.1 Zig

3.4.2 Zig-zag

4 Grafer

En graf er en samling av kanter og noder. Vi kaller mengden av alle nodene V (verticies), og mengden av alle kantene E (edges). Grafen G blir da en samling av disse to mengdene. Formelt kan vi definere en graf slik:

> **Definisjon 5.** En graf G er et par (V, E), der V er en ikke-tom mengde noder og E en mengde nodepar $\{v_1, v_2\}$; $v_1, v_2 \in V$ der $\{v_1, v_2\}$ angir at grafen inneholder en kant fra v_1 til v_2.

Vi bruker $|E|$ for å betegne antall kanter og $|V|$ for å betegne antall noder.

Vi sier at en node er **nabo** med en annen node hvis de har en kant mellom seg. I figur 4 er B og A naboer, men A og C er ikke naboer.

En **sti** eller **vei** er en sekvens av noder (og kantene mellom dem) fra en node til en annen. En sti fra A til G i figuren under kan for eksempel være A, D, C, G. (siden grafen inneholder løkker er ikke stien entydig)

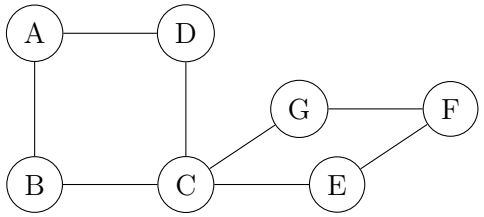
En graf er **rettet** hvis kantene har en spesiell retning, og **urettet** hvis vi ikke bryr oss om retningen på kantene. I en rettet graf vil kanten $\{v_1, v_2\}$ bety at det går en kant fra v_1 til v_2, men ikke nødvendigvis fra v_2 til v_1. Grafen i figur 2 er urettet. Hvis en graf er rettet kaller vi ofte naboene for **etterfølgere**.

En graf er **vektet** hvis kantene har en verdi knyttet til seg, ofte kalt *kosten* til kanten. I en **uvektet** graf har ikke kantene noen spesiell verdi. Vi kan tenke på en uvektet graf som en vektet graf der alle kantene har kost = 1.

Grafen er **syklisk** hvis den inneholder løkker, og **asyklisk** hvis den ikke inneholder løkker. I figur 4 ser vi at A, B, C, D danner en løkke (det gjør også C, G, F, E). Grafen er derfor syklisk.

En type grafer vi jobber mye med er **DAG**er. DAG står for *Directed Asyclic Graph*, på norsk: en rettet, asyklisk graf.

Figur 4: Eksempel på en urettet, uvektet graf

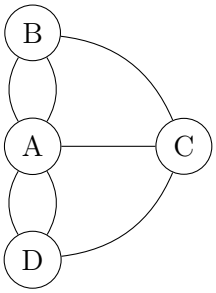


4.1 Terminologi

4.1.1 Hamiltonsk sti

En hamiltonsk sti er en sti som besøker alle nodene én gang. Dette kan virke som en triviell sak, men i en del tilfeller er det faktisk ikke mulig å finne en hamiltonsk sti. I grafen i figur 4 er B, A, D, C, G, F, E en hamiltonsk sti (det finnes fler).

Et berømt eksempel på en graf *uten* en hamiltonsk sti er broene i Königsberg:



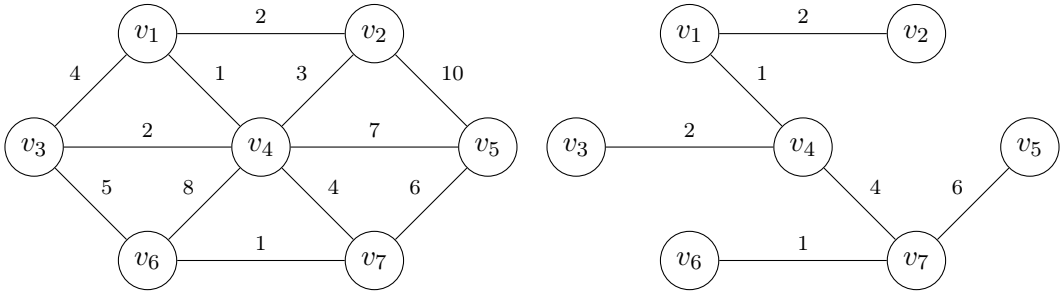
4.1.2 Spenntrær

Vi begynner med en definisjon:

Definisjon 6. Et **spenntre** for en urettet graf G er et tre med kanter fra grafen slik at alle nodene i G er forbundet. Spesielt er et **minimalt spenntre** de(t) spenntre(ene) med lavest total kostnad.

Vi tegner opp et eksempel:

Figur 5: En graf (venstre) og det minimale spenntreet for grafen (høyre)

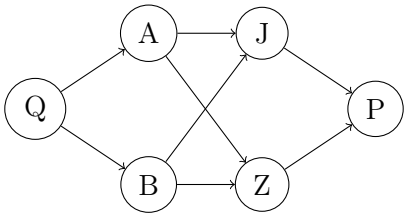


For å finne et minimalt spenntre til en graf kan vi bruke Prims algoritme (4.2.3) eller Kruskals algoritme (4.2.4)

4.1.3 Topologisk sortering

Topologisk sortering er en ordning av noder i en DAG slik at dersom det finnes en vei fra v_i til v_j i grafen kommer v_i før v_j i sorteringa. En topologisk sortering er ikke nødvendigvis entydig bestemt, ofte finnes det veldig mange. Hvis en graf er syklisk eksisterer det ikke en topologisk sortering av grafen. Vi ser på et eksempel:

Eksempel. Gitt følgende graf, finn alle topologiske sorteringer.



Vi ser at Q har inngrad 0, det er derfor et logisk sted å begynne. Etter Q kommer A og B. Vi ser at vi kan ikke gå videre fra A før uten å ha vært innom B, siden både J og Z er avhengige av B. Tilsvarende kan vi ikke gå videre fra B før vi har vært innom A. Til slutt ser vi at P er avhengig av både J og Z. Begge de to må derfor komme før P i sorteringa. Vi har da dekt alle mulige utfall. Vi kan liste opp alle mulige sorteringer:

- Q, A, B, J, Z, P
- Q, B, A, J, Z, P
- Q, A, B, Z, J, P
- Q, B, A, Z, J, P

Vi kan ta med et moteksempel også: Q, A, J, B, Z, P er ikke en gyldig topologisk sortering siden J kommer før B i sorteringa, men i grafen ser vi at J er avhengig av B (det går en kant fra B til J).

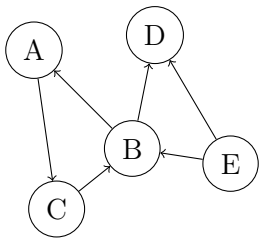
4.1.4 Strongly connected components (SCC)

Vi ser på definisjonen av strongly connected components (heretter SCC):

Definisjon 7. Anta at vi har en rettet graf $G = (V, E)$. Vi har da at en **SCC** av G er et maksimalt sett av noder $U \subseteq V$ slik at for alle u_1, $u_2 \in U$ har vi at $u_1 \rightsquigarrow u_2$ og $u_2 \rightsquigarrow u_1$.

Med andre ord: en SCC er en del (partisjon) av grafen der alle nodene i den partisjonen kan nå alle andre noder i partisjonen. Vi ser på et eksempel:

Eksempel. Finn alle SCCer i grafen:



Vi ser at {A, B, C} danner en SCC. D har ingen kanter ut. Den må derfor være sin egen SCC. E ingen kanter inn, den må også være sin egen SCC. Vi har da at vi kan partisjonere grafen slik: {{A, B, C}, {D}, {E} }

4.1.5 Articulation points

4.2 Grafalgoritmer

4.2.1 Finne SCC

4.2.2 Dijkstras algoritme

4.2.3 Prims algoritme

4.2.4 Kruskals algoritme

4.2.5 Floyds algoritme

5 Andre datastrukturer

5.1 Prioritetskø (heap)

5.1.1 Venstreorientert heap

5.2 Hashtabeller

utvidbar hashing

5.3 Kø/stack

5.3.1 Kø (FIFO)

5.3.2 Stack (LIFO)

6 Tekstalgoritmer

6.1 Brute force

6.2 Boyer-Moore

6.2.1 Bad character shift

6.2.2 Good suffix shift

6.3 Huffmankoding

7 Sortering

7.1 Formaliteter

Før vi kan begynne å se på noen spesielle sorteringsalgoritmer må vi formalisere hva vi mener med sortering. Vi definerer sortering slik:

Definisjon 8. Anta at $\{a_i\}_{i=0}^n$ er en liste av sammenlignbare elementer. Vi sier at $\{a'_i\}_{i=0}^n$ er den tilhørende sorterte lista hvis følgende kriterier er oppfylt:

i $a'_i \leq a'_{i+1}$ for alle $i = 0, 1, ..., n-1$

ii Alle elementene i $\{a\}$ er med i $\{a'\}$

Det andre kriteriet kan virke litt snodig, men uten det ville sortering vært veldig enkelt. Vi kunne i så fall bare generert en ny liste med elementer i sortert rekkefølge, og det første kriteriet ville vært oppfylt. Vi trenger derfor bevaringskriteriet.

Med "sammenlignbare" mener vi at det finnes en måte å entydig bestemme om et element er større enn, mindre enn eller lik et annet element. Hvis vi skal sortere tall er jobben enkel: vi sammenligner numerisk verdi. Hvis vi skal sammenligne to tekststrenger er det ikke like opplagt hvordan vi skal gjøre det. Skal vi sortere alfabetisk? Etter lengde på ordet? I et sånt tilfelle er det opp til oss å velge et fornuftig sammenligningskriterie. Det er vilkårlig hvordan vi sammenligner to elementer, så lenge vi gjør det likt gjennom hele sorteringen.

7.2 Noen algoritmer

Vi skal nå se på noen konkrete sorteringsalgoritmer. Gjennomgående i alle eksempler vil vi sortere tall etter tallverdi, men som diskutert i 7.1 vil vi enkelt kunne tilpasse algoritmene til å sortere på andre kriterier.

7.2.1 Boblesortering

7.2.2 Innstikksortering

7.2.3 Tresortering

7.2.4 Quicksort

7.2.5 Radix

8 Bevisføring

8.1 Noen bevisteknikker

Bevis ved selvmotsigelse Teknikken går ut på å gjøre noen antagelser, og vise til at det fører til en motsigelse. Da må noen av antagelsene være gale, og hvis vi kun har to muligheter (for eksempel rett/galt) kan vi konkludere med at det andre alternativet må være rett. Eksempel på et slik bevis er beviset for haltingproblemet (8.3.1)

Induksjon Bevis ved induksjon går ut på å anta at påstanden holder for alle mulige tall k opp til n, og vise at det også medfører at det også holder for $k = n + 1$. Dermed har man vist at hvis det også holder for $n + 1$ vil det holde for $n + 2$, også videre. Det siste man må gjøre er å *"starte"* induksjonen, med for eksempel å vise at påstanden holder for $k = 1$ (da vil den også holde for $k = 2, 3, ...$)

Bevis ved moteksempel Skal man vise at en påstand er falsk, kan man ganske enkelt finne et eksempel som viser at den opprinnelige påstanden er falsk. Påstår jeg at alle hester er hvite, kan du motbevise den påstanden ved å vise meg en hest som er svart.

8.2 Reduksjon

Anta at vi har et problem A, og vi lurer på om problemet er løselig eller ikke. A kan være latterlig komplekst og vanskelig å forstå, men kanskje vi kan uttrykke det på en annen måte? Hvis vi kan vise at hvis A er løselig er også B løselig, og hvis B er løselig er også C løselig. Slik kan vi fortsette til vi kommer til noe kjent, for eksempel haltingproblemet (8.3.1). For matematikere:

$$A \text{ er løselig} \Rightarrow B \text{ er løselig} \Rightarrow ... \Rightarrow \text{haltingproblemet er løselig}$$

Hvis vi kan vise en slik rekke av implikasjonspiler har vi *redusert* A til haltingproblemet, og dermed vet vi at A er uløselig.

8.3 Noen beviser

8.3.1 Haltingproblemet

Kan en Turingmaskin avgjøre om en annen Turingmaskin noen sinne vil stoppe, eller om den vil gå i evig loop, om den ser på inputen til maskinen? Som nevnt i 1.1.1 kan vi definere en språk som mengden av input som vil få en maskin til å stoppe:

$$L = \{i : M \text{ stopper på input } i\}$$

Det kan vises at en Turingmaskin som løser dette problemet *ikke* kan eksistere. Det kan vises på flere måter, Alan Turing beviste det slik:

Bevis. Vi skal bevise at haltingproblemet er uløselig, og vi skal gjøre det ved selvmotsigelse.

Anta at vi har en Turingmaskin M som bestemmer om en annen Turingmaskin H vil stoppe, gitt input i. Vi kan da bygge en annen Turingmaskin M' rundt denne som tar de samme parametrene (H og i) som input, og gir resultat hvis M gir false (Altså at H ikke vil stoppe, gitt input i), og går i en evig loop hvis M gir true (at H vil stoppe, gitt i).

Hva vil skje hvis vi bruker denne Turingmaskinen på seg selv? Hvis M sier at M' vil stoppe vil M' gå i en evig loop, og følgelig vil ikke M' stoppe. Hvis M sier at M' ikke vil stoppe vil M' gi et resultat, og så stoppe. Uansett får vi en motsigelse, og derfor kan ikke Turingmaskinen M eksistere. □

Beviset er analogt med Cantors diagonaliseringsargument. kanskje du skal forklare hvorfor?

8.3.2 Cantors diagnoaliseringsargument

Vi skal vise at størrelsen av $\mathbb{R}$ er større enn $\mathbb{N}$, altså at det er flere reelle tall enn naturlige tall. Dette er et eksempel på bevis ved selvmotsigelse. Vi skal anta at vi kan vise at $\mathbb{R}$ og $\mathbb{N}$ er like store, og så vise at det fører til en motsigelse.

Vi kan begynne med noe som kanskje går litt mot intuisjonen. Det er nemlig like mange rasjonelle tall som naturlige tall. For å vise dette må vi vise at vi har en 1-1-korrespondanse mellom naturlige tall og rasjonale tall. Vi starter med å liste opp *"alle"* de naturlige og rasjonale tall. Vi setter rasjonale tall inn i en tabell:

$$\mathbb{N} = 1, 2, 3... \qquad \mathbb{Q} = \begin{array}{c|cccc} & 1 & 2 & 3 & ... \\ \hline 1 & {}^{1}/_{1} & {}^{2}/_{1} & {}^{3}/_{1} & ... \\ 2 & {}^{1}/_{2} & {}^{2}/_{2} & {}^{3}/_{2} & ... \\ 3 & {}^{1}/_{3} & {}^{2}/_{3} & {}^{3}/_{3} & ... \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Sånn rent intuitivt kan det se ut som at det er mange flere rasjonale tall enn naturlige tall, men hvis vi går på skrå i tabellen over $\mathbb{Q}$ kan vi knytte hvert rasjonalt tall til ett naturlig tall, og hvert naturlige tall til ett rasjonalt tall:

$$1 : \frac{1}{1} \qquad 2 : \frac{2}{1} \qquad 3 : \frac{1}{2} \qquad 4 : \frac{3}{1} \qquad ...$$

Vi har dermed vist at vi kan skape en 1-1-korrespondanse mellom $\mathbb{N}$ og $\mathbb{Q}$, og dermed er de like store.

Vi skal nå se på $\mathbb{N}$ og $\mathbb{R}$, altså de reelle tallene. Vi ønsker å prøve å skape en lignende 1-1-kobling som vi gjorde for de rasjonale tallene. $\mathbb{R}$ er ikke tellbart uendelig, derfor kan vi ikke liste opp alle tallene i $\mathbb{R}$ på samme måte som før med en smart tabell (dette er forøvrig grunnen til at $\mathbb{R}$ er større enn $\mathbb{N}$, vi skal nå vise *hvorfor*). En ting vi kan gjøre er å liste opp naturlige tall på en side, og tilfeldige og unike reelle tall på den andre. Det har seg faktisk slik at det er flere reelle tall mellom 0 og 1 enn det er naturlige tall tilsammen, det holder å liste opp tilfeldige reelle tall mellom 0 og 1. Vi får da en slik tabell:

$\mathbb{N}$	$\mathbb{R}$
1	0.182947...
2	0.294817...
3	0.132849...
⋮	⋮

Vi kan tenke oss at hvis vi fortsetter slik i all evighet vil vi til slutt ende opp med en 1-1-korrespondanse mellom $\mathbb{R}$ og $\mathbb{N}$. Vi vil jo aldri gå tom for naturlige eller reelle tall å ta av. Det viser seg likevel å være feil, og det er her Cantors diagonaliseringsargument kommer inn:

Vi kan nemlig lage et nytt reellt tall som ikke eksisterer i denne tabellen. Hvis vi tar første siffer fra første tall og legger til 1, andre siffer fra andre tall og legger til 1, også videre. Generelt tar vi det i-te sifferet og legger til 1. Hvis tallet er 9 kan vi gå til 0[3]. Gjør vi det for denne tabellen får vi:

$$0.203...$$

Siden dette tallet er ulikt tall nummer i i tabellen på i-te siffer vet vi at det ikke er inneholdt i tabellen, og vi har dermed vist at det ikke kan lages en 1-1 korrespondanse mellom $\mathbb{R}$ og $\mathbb{N}$. Følgelig er $\mathbb{R}$ større enn $\mathbb{N}$ □

8.3.3 Antall noder i et binært tre

Vi skal vise at antall noder n i et binært tre er begrenset slik:

$$n \leq 2^h - 1$$

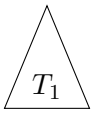
Vi skal vise dette ved induksjon.

Vi starter med å vise at formelen gjelder for et tre med høyde $h = 1$:

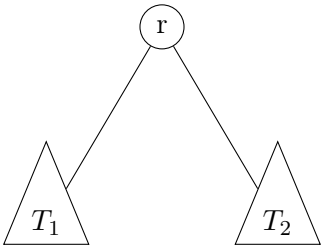
$$n = 2^1 - 1 = 1$$

som åpenbart stemmer siden et tre med høyde 1 har kun 1 node, nemlig rota.

Vi antar at formelen gjelder for alle h opp til n, og skal vise at det impliserer at den også gjelder for $n + 1$. Vi har et tre T_1 med høyde n:



Vi antar at formelen stemmer, og dermed at antall noder i T_1 maksimalt er $2^n - 1$. Vi skal nå øke høyden til $n + 1$. Vi lager en ny rot, med to fulle subtrær som barn:



Fra formelen har vi at T_1 og T_2 maksimalt har $2^n - 1$ noder hver. Legger vi sammen antall noder i treet nå får vi:

$$n \;=\; \text{noder i } T_1 + \text{noder i } T_2 + 1 \;\leq\; 2(2^n - 1) + 1 \;=\; 2^{n+1} - 1$$

Oppsummering: Vi har vist at hvis formelen gjelder for $h = n$ gjelder den også for $h = n + 1$, vi har vist at den gjelder for $h = 1$, og dermed har vi vist at den gjelder for alle $h \in \mathbb{N}$ □

[3]Formelt kan vi bruke klokkeaddisjon (moduloaddisjon): $(9 + 1) \mod 10 = 0$