Diskret Matematik, Algoritmer & Data Strukturer

Indholdsfortegnelse

Algoritmer & Data Strukturer

Rekursion

Master Theorem

Master Theorem formen:

$$T(n) = a \cdot T(n/b\,) + f(n)$$

For at finde ud af hvilken case der er tale om, sammenligner man f(n) med $n^{\log_b a}$
Først finder man ud af værdien og kigger på formen af f(n).

Vi siger at:

$$T(n) = a \cdot T(n/b\,) + n^k$$

$$\mathrm{d} = \log_b a$$

Så sammenligner vi k med d
Derfor regner vi først $\log_b a$ ud og derefter sammenligner med k

Her kan reglen bruges at:

Sammenligning	Case	Resultat
$k < d$	1	$T(n) = \Theta(n^d)$
$k = d$	2	$T(n) = \Theta(n^d \cdot \log n)$
$k > d$	3	$T(n) = \Theta(n^k)$, hvis regularity condition opfyldt

Hvis a = b, kan man barre kigge på om k er større, mindre eller lig med 1.
Hvis n er en konstant så bruger man n^0

📘 Skema: $\log_a(b)$ for a, b ∈ {1..9}, a ≠ 1

$\log_a(b)$	a=2	a=3	a=4	a=5	a=6	a=7	a=8	
b=1	0	0	0	0	0	0	0	0
b=2	1	~0.63	0.5	~0.43	~0.39	~0.36	~0.33	~0.32
b=3	~1.58	1	~0.79	~0.68	~0.61	~0.56	~0.53	~0.5
b=4	2	~1.26	1	~0.86	~0.77	~0.71	~0.66	~0.63
b=5	~2.32	~1.46	~1.16	1	~0.9	~0.83	~0.77	~0.74
b=6	~2.58	~1.63	~1.29	~1.11	1	~0.92	~0.84	~0.8
b=7	~2.81	~1.77	~1.4	~1.21	~1.09	1	~0.9	~0.85
b=8	3	~1.89	~1.5	~1.29	~1.16	~1.07	1	~0.9
b=9	~3.17	2	~1.58	~1.37	~1.23	~1.1	~0.95	1

De tre cases i Master Theorem:

Case	Form på f(n)	Sammenligning med $n^{\log_b a}$	Resultat for T(n)	Betingelser
1	$f(n) = O(n^{\log_b a - \varepsilon})$	f(n) vokser **langsommere** end $n^{\log_b a}$	$T(n) = \Theta(n^{\log_b a})$	For en konstant $\varepsilon > 0$
2	$f(n) = \Theta(n^{\log_b a})$	f(n) vokser **lige så hurtigt** som $n^{\log_b a}$	$T(n) = \Theta(n^{\log_b a} \cdot \log n)$	—
3	$f(n) = \Omega(n^{\log_b a + \varepsilon})$	f(n) vokser **hurtigere** end $n^{\log_b a}$	$T(n) = \Theta(f(n))$	Skal også opfylde *regularity condition*: $a \cdot f(n/b) \leq c \cdot f(n)$ for $c < 1$ og stor nok n

Noter:

- a = antal rekursive kald
- b = hvor meget input reduceres hver gang
- f(n) = arbejdet uden for rekursive kald
- $\log_b a$ = grænsen, vi sammenligner f(n) med
- Θ = theta = "vokser lige så hurtigt som"
- Ω = Omega = "vokser mindst som"
- O = "vokser højst som"
- $\varepsilon = epsilon =$ en lille konstant der bruges til at vise vækshastighed.
  - "En lille smule" eller "Noget der er lidt mindre eller lidt større end noget andet"

🔷 Hvad betyder det i praksis?

I Master Theorem bruges ε til at afgøre:

Situation	Tolkning
$f(n) = O(n^{\log_b a - \varepsilon})$	f(n) vokser **langsomt**
$f(n) = \Theta(n^{\log_b a})$	f(n) vokser **lige så hurtigt**
$f(n) = \Omega(n^{\log_b a + \varepsilon})$	f(n) vokser **hurtigt**

Hvis det **ikke** er en ren n^k-form:

Så kan du nogle gange stadig skrive det **som en kombination**, fx:

f(n)	Kan omskrives til
$\sqrt{n}$	$n^{1/2}$
$n \log n$	$n^1 \cdot \log n$
$\log n$	Ikke n^k, men stadig case 1*

(*: her kan du bruge en **udvidelse** af Master Theorem, men den er sjældent påkrævet i eksamen)

Big-O notation

Det går ud på at finde ud af hvor hurtigt tingene vokser.

Spørgsmålet: $n\ er\ O(\sqrt{n})$ er ikke om de er ens, men om venstre siden er lavere eller lig højresiden

Ved spørgsmål om udsagn i Big-O:
Hvis funktionen til højre vokser **lige så hurtigt eller hurtigere**, så er udsagnet **sandt**.
Hvis funktionen til venstre vokser **hurtigere**, så er udsagnet **falsk**.

Først kigger vi på venstre siden: lad os antage at venstre siden er n+n, så er det 2n.
Så omregner vi det til Big-O notation ved at bruge theta.
her bruger vi ikke konstanter, da det ikke ændrer noget. Så her vil venstre siden blive theta(n)
O betyder = vokser højst som. Altså venstre siden må højst vokse som højre siden, men aldrig højere.

Her kan man kigge på følgende tabel:

1. Skema: Vækstordener fra langsom til hurtig

Funktion	Navn
1	Konstant tid
$\log(n^{1/3})$	Logaritmisk (svag)
$\log n$	Logaritmisk
$\log(n^3)$	Logaritmisk (stærk)
$(\log n)^2$	Logaritmisk kvadrat
$(\log n)^3$	Logaritmisk kubik
$\sqrt{n}$	Kvadratrod
$n^{1/7}$	Sub-lineær potens
$n^k, 0 < k < 1$	Sub-lineær potens
$x + n$	Lineær
$n + n$	Lineær
n	Lineær
$n \log n$	Lineært-logaritmisk
$n^{1.5}$	Potens (mellem n og n^2)
n^2	Kvadratisk
n^3	Kubisk
2^n	Eksponentiel
3^n	Eksponentiel
$2^n n$	Eksponentiel × lineær
$n!$	Faktoriél

Small-O notation

Fungerer lidt på samme måde som Big-O, her kigger man bare på at ved small-o SKAL venstre siden være langsommere end højre siden.

For at løse dette kig på ovenstående tabel:

Brug denne:
Ved spørgsmål om udsagn i small-O
Hvis funktionen til højre vokser **hurtigere**, så er udsagnet **sandt**.
Hvis funktionen til venstre vokser **lige så hurtigt eller hurtigere**, så er udsagnet **falsk**.

Andre former for notation:

✅ Når du skal vurdere et udsagn:

Notation	Venstre vs. højre	Udsagnet er sand hvis...
O	$f(n) \leq g(n)$	Venstre vokser **lige så hurtigt eller langsommere**
o	$f(n) < g(n)$	Venstre vokser **strengt langsommere**
Ω	$f(n) \geq g(n)$	Venstre vokser **lige så hurtigt eller hurtigere**
ω	$f(n) > g(n)$	Venstre vokser **strengt hurtigere**
Θ	$f(n) = g(n)$	De vokser **ens** (samme tempo)

Sorteringsalgortimer

Køretider for algoritmer:

Køretider for O:

✅ Big-O kompleksitetstabel – eksamensversion (øvre grænser)

Algoritme	Best case	Average case	Worst case	Sorteret input	Omvendt input	Ens input
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$*	$O(n^2)$*	$O(n^2)$	$O(n^2)$*
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	✅ $O(n)$
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(nk)$	$O(nk)$	$O(nk)$
Tim Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	$O(n \log n)$	$O(n)$

* Gælder ved dårligt pivotvalg (f.eks. første eller sidste element uden optimering).

Heap Operationer:

🔷 Liste over heap-operationer

Operation	Hvad den gør
`Min-Heap-Insert(A, x)`	Indsætter tallet `x` i heapen `A` og opretholder heap-egenskaben
`Heap-Extract-Min(A)`	Fjerner det mindste element (øverste) og genskaber heapen
`Heap-Minimum(A)`	Returnerer det mindste element (øverst) uden at fjerne det
`Heap-Decrease-Key(A, i, k)`	Sænker værdien i index `i` til `k` og flytter det op hvis nødvendigt
`Heapify(A, i)`	Genskaber heap-strukturen fra index `i` og nedad
`Build-Min-Heap(A)`	Bygger en min-heap fra en vilkårlig array `A`
`Is-Min-Heap(A)`	Tjekker om arrayet `A` overholder min-heap-egenskaben
`Heap-Delete(A, i)`	Fjerner elementet på index `i` fra heapen (bruger Decrease-Key + Extract)
`Heap-Increase-Key(A, i, k)`	Øger værdien i index `i` til `k` og flytter det ned hvis nødvendigt

Generelle regler

-

Heap-Extract-Min(A)

Der star typisk Heap-Extract-Min(A)
Der kan også stå A,1 - Det er det samme

Step by step guide

1. Tag den sidste plads og indsæt på A1 (den første)
2. Fjern den sidste plads
3. Lav et træ startende fra 1, med altid 2 børn.
4. Byt rundt så det mindste tal kommer op, og børnene bliver større end forældren.
5. Bliv ved indtil man er i bund.

Eksempel:

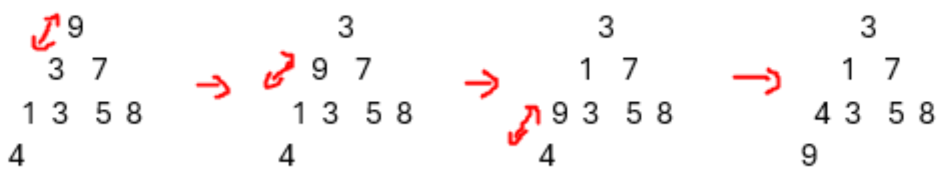
[6 3 7 1 3 5 8 4 9]

1. Tag den sidste plads og indsæt på A1 [9 3 7 1 3 5 8 4 9]

2. Fjern den sidste plads: [9 3 7 1 3 5 8 4]

3. Lav et træ

4. +5 byt rundt så det mindste tal kommer op og bliv ved



Resultat: [3 1 7 4 3 5 8 9]

Min-Heap-Insert(A,x)

Der står typisk Min-Heap-Insert(A, x) – det betyder, at tallet **x skal tilføjes i heapen A**.
Du bygger heapen **nedefra og op**.

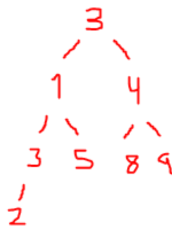
Step by step guide

1. Tilføj tallet på **nederste ledige plads** (bunden af heapen)
2. Lav et træ startende fra det nye element
3. Byt opad, så forældre altid er mindre end barnet
4. Stop når forælder er mindre eller du når toppen

Eksempel: Min-Heap-Insert(A, 2)

Input:
A = [3 1 4 3 5 8 9]
Indsæt tallet 2

1. Sæt 2 i bunden: [3 1 4 3 5 8 9 2]
2. Lav træ (nyt tal er nederst)



3. Byt op (bubble up)
- Man bytter fra bunden og op indtil at tallet ikke længere er større. Fx her vil man bytte 2 med 3, og derefter er 2 tallet større end 1 tallet og man stopper.

Resultat: [3 1 4 2 5 8 9 3]

Heap-Minimum(A)

Bruges til at slå op hvad det mindste element i en Min-Heap er, uden at fjerne det:
- Med mindste element menes det første element i Arrayet, altså A1.

Step-by-step guide

1. Find roden i heapen
2. Returnér værdien på **A[1]** (det første element i arrayet)
3. Du ændrer ikke noget i heapen

Heap-Decrease-Key(A,i,k)

Bruges til at **sænke værdien** i A[i] til en mindre værdi k, og genskabe heapen.

Step-by-step guide

1. Sæt A[i] = k
2. “Bubble up” (byt med forælder), så forælder ≤ barn
3. Stop når forælder er mindre, eller du når toppen

Regel:

Du **må ikke øge værdien**, kun sænke den – ellers bryder du Min-Heap-egenskaben.

1. **Eksempel:** A = [2 4 3 7 7 5 6 8 9]
   Kald: Heap-Decrease-Key(A, 5, 1)
   (dvs. sænk A[5] fra 7 til 1)

A[5] = 1
→ Nu: [2 4 3 7 1 5 6 8 9]

2. Bubble up: (Hvis i tvivl så kig i min-heap-insert)

- Forælder til index 5 er A[2] = 4
  → 1 < 4 → byt

→ [2 1 3 7 4 5 6 8 9]

3. Ny index = 2
   → Forælder = A[1] = 2
   → 1 < 2 → byt

→ [1 2 3 7 4 5 6 8 9]

4. Forælder = top → stop

Resultat: A = [1 2 3 7 4 5 6 8 9]

Heapify(A,i)

Bruges til at **genoprette heap-strukturen** fra index i og nedad.
Typisk brugt i Extract-Min og Build-Heap.

Step-by-step guide

1. Kig på node A[i] og dens to børn (2i og 2i+1)
2. Find den **mindste** af de tre
3. Hvis A[i] ikke er den mindste → byt med den mindste
4. Kald `Heapify` **rekursivt** på det sted du byttede til
5. Stop når A[i] ≤ børn **eller** du er i bunden

Build-Min-Heap(A)

Bruges til at **lave en gyldig Min-Heap** ud fra et helt almindeligt array.
→ Genskaber heapen **nedefra og op**.

🔷 **Step-by-step guide**

1. Find sidste forælder:

$$i = \left\lfloor \frac{n}{2} \right\rfloor$$

2. Gå baglæns fra `i` ned til 1
3. Kør `Heapify(A, i)` for hvert index
4. Når du når A[1], er heapen færdig

Eksempel: A = [5 3 6 7 2 4]

n = 6 → sidste forælder = $\lfloor 6/2 \rfloor = 3$
Start fra i = 3
Kør nu heapify fra A3

Is-Min-Heap(A)

Bruges til at **tjekke om et array A overholder Min-Heap-egenskaben**.

🔷 **Step-by-step guide**

1. Gå gennem alle forældre (index 1 til $\lfloor n/2 \rfloor$)
2. For hver forælder $A[i]$:
   - Tjek:
     - Venstre barn $A[2i] \geq A[i]$
     - Højre barn $A[2i+1] \geq A[i]$ (hvis det findes)
3. Hvis én regel brydes → returnér **false**
4. Hvis ingen bryder reglen → returnér **true**

🧠 **Husk:**

- I en **Min-Heap** gælder:

  > **Forælder ≤ hvert barn**

❌ **Eksempel på fejl:**

```text
A = [2 4 1 7 7 5 6]
```

- A[1] = 2 → børn: 4 og **1** ❌
  → 1 < 2 → **reglen brudt** → returnér **false**

🧪 **Eksempel:**

```text
A = [2 4 3 7 7 5 6]
```

- A[1] = 2 → børn: 4 og 3 → OK
- A[2] = 4 → børn: 7 og 7 → OK
- A[3] = 3 → børn: 5 og 6 → OK
  → ✅ Er en Min-Heap → returnér **true**

Heap-Delete(A,i)

Bruges til at **fjerne elementet på index i** i en Min-Heap.

🔹 Step-by-step guide

1. Sæt `A[i] = -∞` (eller en meget lille værdi)
2. Bubble up fra index `i` til toppen
3. Kør `Heap-Extract-Min(A)` for at fjerne det (nu øverste) element

Eksempel:

```text
A = [2 4 3 7 7 5 6]
```

Kald: `Heap-Delete(A, 4)` (A[4] = 7)

1. Sæt A[4] = -∞
   → `[2 4 3 -∞ 7 5 6]`
2. Bubble up:

- A[4] vs A[2] → -∞ < 4 → byt
  → `[2 -∞ 3 4 7 5 6]`
- -∞ < 2 → byt
  → `[-∞ 2 3 4 7 5 6]`

3. Kør `Extract-Min` → fjerner -∞
   → `A = [2 4 3 6 7 5]` *(efter heapify)*

✅ Resultat:

Elementet i index 4 er fjernet, og heap-strukturen er bevaret.

Heap.Increase-Key(A,I,k)

Bruges til at **øge værdien i A[i] til k**, og genskabe heapen (kun relevant i Min-Heap hvis du har opdateret et element forkert).

🔹 Step-by-step guide

1. Sæt `A[i] = k`
2. "Heapify ned" fra index `i`
   (fordi værdien nu kan være **større end børnene**)
3. Stop når A[i] ≤ sine børn, eller du når bunden

Kun gyldig hvis:

- `k ≥ A[i]` – ellers skal du bruge `Heap-Decrease-Key`

Eksempel:

```text
A = [2 3 4 7 5 6]
```

Kald: `Heap-Increase-Key(A, 2, 8)`
→ A[2] går fra 3 → 8
→ `[2 8 4 7 5 6]`

Heapify ned fra index 2:

- Børn = 7 (A[4]) og 5 (A[5])
  → Mindste barn = 5 → byt med 8
  → `[2 5 4 7 8 6]`

Fortsæt: A[5] har ingen børn → færdig

✅ Resultat:

```text
A = [2 5 4 7 8 6]
```

Min-Heap A

En min-heap A

	1	2	3	4	5
A:					

indeholder følgende fem nøgler:

$$\{1, 2, 3, 4, 5\}$$

Angiv alle pladser i A, hvor det er muligt at nøglen 4 kan stå. [*Et eller flere svar.*]

Vi har en min-heap `A` med elementerne `{1, 2, 3, 4, 5}`. Opgaven beder dig angive **alle de pladser**, hvor **nøglen 4** *kan* stå i en gyldig min-heap.

🔍 Regler for en min-heap:

- Hver forælder skal være **mindre end eller lig med** sine børn.
- Repræsenteres som array, hvor:
  - Forælder ved index `i` har børn ved index `2i` og `2i+1` (hvis disse findes).

🔁 **I en *min*-heap skal roden altid være det *mindste* tal** – ikke det største. - Derfor kan A1 ikke være sand

Hashtabeller

Linear probing

✅ **Huskeregel til eksamen – linear probing:**

Hvis et tal ligger på **den plads det ønsker** (dvs. h(x) = index),
➜ så **kan det være kommet først**.

Hvis det ligger **et andet sted**,
➜ så **må det være blevet skubbet** pga. kollision,
➜ og **kan ikke være kommet først**.

Eksempel:

> 2 point
>
> For en funktion $h_1(x)$ gælder
>
> $$\begin{aligned} h_1(22) &= 6 \\ h_1(33) &= 1 \\ h_1(44) &= 4 \\ h_1(55) &= 1 \\ h_1(66) &= 6 \\ h_1(77) &= 1 \end{aligned}$$
>
> Vi ser på en hashtabel H af længde syv, der bruger linear probing med ovennævnte funktion $h_1(x)$ som auxiliary hashfunktion. Startende med en tom hashtabel er tallene $\{22, 33, 44, 55, 66, 77\}$ blevet indsat i en eller anden rækkefølge. Resultatet er blevet:
>
> H:
>
> | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
> |---|---|---|---|---|---|---|
> | 22 | 77 | 55 | 33 | 44 | | 66 |
>
> Hvilke af tallene kan være blevet indsat først (dvs. kan have stået først i indsættelsesrækkefølgen)? [*Et eller flere svar.*]

Løsning:

Vi har en **hashtabel med længde 7** og bruger **linear probing.**
Du får at vide, hvor hvert tal *gerne ville sættes ind* (hash-værdi) — og du får at vide, hvor tallene *endte* i tabellen. Du skal finde ud af, **hvilket tal der kan være kommet først.**

Trin 1: Hashværdier og deres ønskede pladser

Tal	$h_1(x)$	Ønsket plads (index)
22	6	6
33	1	1
44	4	4
55	1	1 (kollision)
66	6	6 (kollision)
77	1	1 (kollision)

Givet slutresultat i tabellen:

diff

Kopiér Rediger

```diff
Index:    0   1   2   3   4   5   6
          22  77  55  33  44  -   66
```

✅ Korrekt svar:

Tal	Ønsket plads	Endte på	Kan være kommet først?	Hvorfor?
22	6	0	❌ Nej	Plads 6 var allerede optaget
33	1	3	❌ Nej	Blev skubbet to pladser frem
44	4	4	✅ Ja	Fik ønsket plads
55	1	2	❌ Nej	Plads 1 var optaget
66	6	6	✅ Ja	Fik ønsket plads
77	1	1	✅ Ja	Fik ønsket plads

Auxiliary hashfunktioner

Hvad er `h₁(x)` og `h₂(x)`?

De er **hashfunktioner**, dvs. formler der omregner en værdi (`x`, fx 25) til en **plads i hashtabellen.**

`h₁(x)` – primær hashfunktion:

- **Formål:** Find den **første plads** vi prøver at indsætte værdien på.
- **Tænk på det som: "Hvor vil jeg gerne sidde?"**
- Eksempel:
  Hvis $h_1(25) = 3$, så prøver vi **først** at sætte 25 i index **3**.

`h₂(x)` – sekundær hashfunktion (kun ved kollision):

- **Formål:** Bestemmer hvor langt vi skal **springe frem**, hvis den ønskede plads (fra h_1) er **optaget**.
- **Tænk på det som: "Hvor mange skridt går jeg, når jeg ikke må sidde hvor jeg vil?"**
- Bruges i **double hashing** til at undgå kollisioner.
- Eksempel:
  - Hvis $h_2(25) = 2$, og $h_1(25) = 3$, så prøver vi:
    - Først: (3 + 0×2) % 8 = 3
    - Så: (3 + 1×2) % 8 = 5
    - Så: (3 + 2×2) % 8 = 7
    - osv., indtil vi finder en ledig plads

Opsummering:

Funktion	Navn	Betydning	Bruges til
$h_1(x)$	Primær hash	Startplads	Første forsøg
$h_2(x)$	Sekundær hash	Spring-længde (ved kollision)	Hvor langt vi springer frem

Huskeregel til eksamen:

> `h₁(x)` = hvor du *vil* sætte tallet
> `h₂(x)` = hvor langt du *går*, hvis pladsen er optaget

Double Hashing

Double Hashing – hvordan fungerer det?

Ved **double hashing** bruger man to hashfunktioner:

$h_1(x)$ og $h_2(x)$

Hvis der er kollision (pladsen fra h_1 er optaget), så prøver man:

$$\text{Index} = (h_1(x) + i \cdot h_2(x)) \mod m$$

hvor `i` = antal forsøg (0, 1, 2, ...), og `m` = tabelstørrelse ([illegible])

mod = modulo (%) - Hvis resultatet er større ellr lig m, bruger man modulo - husk i starter fra 0

Eksempel:

Vi ser på en hashtabel H, der bruger double hashing og auxiliary hashfunktioner $h_1(x)$ og $h_2(x)$. Hashtabellen ser lige nu sådan ud:

	0	1	2	3	4	5	6	7
H:	13	56		32	91		82	

Derefter indsættes 25, hvorefter hashtabellen ser sådan ud:

	0	1	2	3	4	5	6	7
H:	13	56		32	91	25	82	

Hvis $h_1(25) = 3$, hvilke af følgende værdier af $h_2(25)$ er da mulige? [*Et eller flere svar.*]

🧠 Opgaveforståelse

Vi har en hashtabel med **længde 8** (`m = 8`), og vi bruger **double hashing**, altså:

$$\text{Index} = (h_1(x) + i \cdot h_2(x)) \mod m$$

Oplysninger:

1. Før indsættelse:

```diff
Index: 0   1   2   3   4   5   6   7
       13  56  -   32  91  -   82  -
```

2. Efter at 25 er indsat:

```diff
Index: 0   1   2   3   4   5   6   7
       13  56  -   32  91  25  82  -
```

3. Opgaven fortæller:
   - `h₁(25) = 3`
   - Tallet 25 ender i **index 5**

🔍 Hvad skal vi finde?

> Hvilke **mulige værdier af** `h₂(25)` kan have givet dette resultat?

🧮 Trin-for-trin løsning:

🔢 Trin 1: Brug formlen

Vi bruger:

$$\text{Index} = (3 + i \cdot h_2(25)) \mod 8 = 5$$

Vi skal finde **alle hele tal** `h₂(25)`, hvor der findes et **heltal** `i ≥ 1`, så det lander i **index 5**.

🔁 Trin 2: Prøv alle `h₂(25)` fra 1 til 7 (fordi modulo 8)

➤ $h_2 = 1$:
- i=1 → (3 + 1) % 8 = 4 ❌
- i=2 → (3 + 2) % 8 = 5 ✅ → **gyldig**

➤ $h_2 = 2$:
- i=1 → 3 + 2 = 5 → ✅

➤ $h_2 = 3$:
- i=1 → 6 % 8 = 6 ❌
- i=2 → 3 + 6 = 9 → 9 % 8 = 1 ❌
- i=3 → 3 + 9 = 12 % 8 = 4 ❌
- i=4 → 3 + 12 = 15 % 8 = 7 ❌
- i=5 → 3 + 15 = 18 % 8 = 2 ❌
- i=6 → 3 + 18 = 21 % 8 = 5 ✅ → **gyldig**

➤ $h_2 = 4$:
- i=1 → 3 + 4 = 7 ❌
- i=2 → 11 % 8 = 3 ❌ (loop)
  → **ingen i giver 5** ❌

➤ $h_2 = 5$:
- i=1 → 3 + 5 = 8 → 0 ❌
- i=2 → 3 + 10 = 13 % 8 = 5 ✅ → **gyldig**

➤ $h_2 = 6$:
- i=1 → 3 + 6 = 9 % 8 = 1 ❌
- i=2 → 15 % 8 = 7 ❌
- i=3 → 21 % 8 = 5 ✅ → **gyldig**

➤ $h_2 = 7$:
- i=1 → 3 + 7 = 10 % 8 = 2 ❌
- i=2 → 3 + 14 = 17 % 8 = 1 ❌
- i=3 → 3 + 21 = 24 % 8 = 0 ❌
- i=4 → 3 + 28 = 31 % 8 = 7 ❌
- i=5 → 3 + 35 = 38 % 8 = 6 ❌
- i=6 → 3 + 42 = 45 % 8 = 5 ✅ → **gyldig**

🔁 Hvad betyder `%` (modulo)?

Tegnet `%` betyder **"modulo"**, og det bruges til at finde **resten**, når man **deler et tal med et andet.**

📘 Eksempler:

Udtryk	Forklaring	Resultat
`10 % 3`	10 divideret med 3 → 3 går 3 gange i 10 (3×3=9), resten er **1**	1
`17 % 5`	5 går 3 gange i 17 (3×5=15), resten er **2**	2
`8 % 8`	8 går 1 gang i 8 (1×8=8), resten er **0**	0
`15 % 8`	8 går 1 gang i 15 (1×8=8), resten er **7**	7

COUNTING-SORT (A,x,y)

Eksempel:

Nedenfor er et array af ni heltal med værdier mellem 0 og 6 (inklusive).

	1	2	3	4	5	6	7	8	9
A:	2	0	6	2	3	5	5	1	2

Vi sorterer nu tallene ved at udføre COUNTING-SORT(A,9,6).
I algoritmen bruges et array C med syv pladser (indekseret fra 0 til 6). Hver plads indeholder en tæller, dvs. et heltal.
Hvad er summen af disse syv heltal i C ved algoritmens afslutning?

🧠 Opgaveforståelse:

Du får:

- Et array A med 9 heltal mellem 0 og 6
- Du skal **køre Counting-Sort**
- Undervejs oprettes en tælle-array `C[0..6]`
- Du skal **finde summen af alle 7 tal i C, når algoritmen er færdig med at tælle**

🔁 Trin 1: Counting-Sort – tællefasen

Vi laver en tælle-array `C[0..6]`, hvor `C[i]` angiver, **hvor mange gange tallet `i` optræder i A**

Gå igennem A og tæl forekomster:

Tal	Antal
0	1
1	1
2	3
3	1
4	0
5	2
6	1

➕ Trin 2: Find summen af værdierne i C

$$\text{Sum} = 1+1+3+1+0+2+1 = \boxed{9}$$

✅ Svar:

Summen af de syv heltal i C er: 9

Huffmann-træ

Huffman-træ laves ved at kombinere de to mindste hyppigheder indtil ét træ er tilbage. Bitlængden for et tegn svarer til **hvor dybt det ender i træet (fra roden)**.

Eksempel på opgave: Kodeord

En fil indeholder nedenstående tegn med de angivne hyppigheder.

Tegn	b	c	d	f	g	h
Hyppighed	90	15	40	30	125	35

Lav et Huffman-træ på dette input. Hvor mange bits er der i kodeordet for d?

🔁 Trinvis løsning (Huffman-algoritmen)

Vi arbejder med en **prioritetskø**, hvor vi hele tiden **kombinerer de to laveste værdier.**

📦 Initial liste (sortér stigende):

```makefile
c:15, f:30, h:35, d:40, b:90, g:125
```

1. Kombiner c (15) + f (30) → ny node: 45

```yaml
Ny node: [cf]=45
Rest: h:35, d:40, b:90, g:125, cf:45
→ sorteret: h:35, d:40, cf:45, b:90, g:125
```

2. Kombiner h (35) + d (40) → ny node: 75

(**d er nu på niveau 1** i træet)

```yaml
Ny node: [hd]=75
Rest: cf:45, b:90, g:125, hd:75
→ sorteret: cf:45, hd:75, b:90, g:125
```

3. Kombiner cf (45) + hd (75) → ny node: 120

```yaml
Ny node: [cfhd]=120
Rest: b:90, g:125, cfhd:120
→ sorteret: b:90, cfhd:120, g:125
```

4. Kombiner b (90) + cfhd (120) → ny node: 210

```yaml
Ny node: [bcfhd]=210
Rest: g:125, bcfhd:210
→ sorteret: g:125, bcfhd:210
```

5. Kombiner g (125) + bcfhd (210) → root = 335

🌳 Nu kan vi aflæse dybden for `d`

Lad os se vejen d fulgte:

- d blev kombineret med h: (niveau 1)
- hd blev kombineret med cf: (niveau 2)
- cfhd blev kombineret med b: (niveau 3)
- bcfhd blev kombineret med g: (niveau 4)

✅ **Svar: d's kodeord har længde = 4 bit**

Eksempel på opgave: Hvor mange bits fylder xxx tegn?

Vi ser stadig på et Huffman-træ for en fil med nedenstående hyppigheder. Der er $90 + 15 + 40 + 30 + 125 + 35 = 335$ tegn i filen.

Tegn	b	c	d	f	g	h
Hyppighed	90	15	40	30	125	35

Hvor mange bits fylder disse 335 tegn i alt hvis de kodes med Huffman-koder?

Trinvis løsning: Lav Huffman-træ

Vi gør det præcis som før – og **følger med i, hvor dybt hvert tegn ender i træet** (altså: kodeordslængden = antal bits for det tegn).

Trin 1: Sorter hyppighederne

makefile   Kopiér   Rediger

```makefile
c:15, f:30, h:35, d:40, b:90, g:125
```

Trin 2: Kombiner to laveste (c + f):
15 + 30 = 45 → ny node cf
Rest: h:35, d:40, cf:45, b:90, g:125

Trin 3: Kombiner h + d:
35 + 40 = 75 → ny node hd
Rest: cf:45, hd:75, b:90, g:125

Trin 4: Kombiner cf + hd:
45 + 75 = 120 → ny node cfhd
Rest: b:90, cfhd:120, g:125

Trin 5: Kombiner b + cfhd:
90 + 120 = 210 → ny node bcfhd
Rest: g:125, bcfhd:210

Trin 6: Kombiner g + bcfhd:
125 + 210 = 335 → root

Nu: Find kodeordslængde (dybde i træet) for hver tegn

- **g:** blev koblet i sidste trin → **kodeordslængde = 1**
- **bcfhd:** blev koblet i sidste trin → kodeord +1
  - **b:** indgik i bcfhd i trin 4 → +1 → **kodeordslængde = 2**
  - **cfhd:**
    - cf (c og f) og hd (h og d) blev koblet ind i cfhd → +1
      - **c, f:** kodeordslængde = 4
      - **h, d:** kodeordslængde = 3

Så vi har:

Tegn	Hyppighed	Kodeordslængde	Bidrag i bits
g	125	1	125
b	90	2	180
h	35	3	105
d	40	3	120
c	15	4	60
f	30	4	120

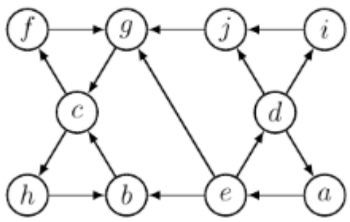
Svar = Alle bidrag i bits lagt sammen
- Dette eksempel: 125+180+105+120+60+120=710 bits

Bredde-Først / Breadth-First Search (BFS(G, a))

d(x) er **afstanden (antal kanter)** fra startknuden til knuden x

Eksempel:

Udfør bredde-først søgning BFS(G, a) på grafen nedenfor med start i knuden a.
For BFS afhænger udførelsen af ordningen af knuders nabolister. Du skal i dette eksamenssæt antage, at en knudes naboliste er sorteret i alfabetisk orden efter naboknudernes navne.



Hvilken knude er den første, som får tildelt d-værdien 4?

📘 Opgaven kort:

- Vi skal udføre **BFS (Bredde-først søgning)** fra knuden a
- Nabolister er **alfabetisk sorterede**
- Vi skal finde:

> **Hvilken knude får først tildelt d-værdien 4?**

🧠 BFS: Hvordan virker det?

- BFS starter i startknuden (a)
- Derefter besøges alle naboer (niveau 1), så naboers naboer (niveau 2), osv.
- Hver gang en knude besøges for første gang, får den en **d-værdi**, som er dens **afstand (antal kanter) fra startknuden**
- Vi bruger en **kø** og behandler knuder i rækkefølge

Lille obs: man kigger på den første som er 4 kanter væk fra a. Og der står det er sorteret i alfabetisk rækkefølge; derfor:
a -> e (ingen andre valgmuligheder)
a -> b (alfabetisk)
b -> c (ingen andre valgmuligheder)
c -> f (alfabetisk rækkefølge)

Hvis man sidder fast på et tidspunkt så går man tilbage til sidst man var unstuck og prøver igen.

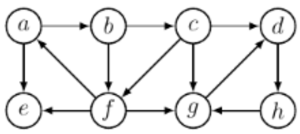
Dybde-Først / Depth-First Search (DFS(G, a))

Når du i DFS rammer et **endepunkt** (en knude uden nye naboer),
så **går du tilbage** ("backtracker") til den seneste knude, hvor du **ikke har besøgt alle naboer endnu**.

Højste discovery time:

Eksempel:

Udfør dybde-først søgning DFS-VISIT(G, a) på grafen nedenfor med start i knuden a. Læs også næste spørgsmål, inden at du udfører algoritmen.
For DFS afhænger udførelsen af ordningen af knuders nabolister. Du skal i dette eksamenssæt antage, at en knudes naboliste er sorteret i alfabetisk orden efter naboknudernes navne.



Hvilken af nedenstående knuder er den sidste, som opdages (dvs. som får den højeste discovery time).

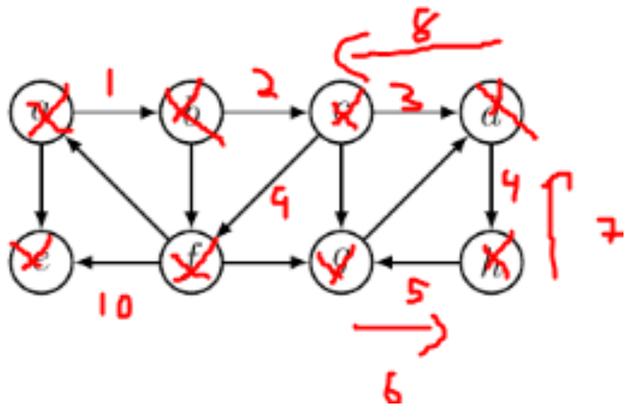
📘 Opgaven kort:

- Vi udfører **DFS-VISIT(G, a)** (altså dybde-først søgning fra a)
- Grafen er **rettet**, og nabolister er i **alfabetisk rækkefølge**
- Du skal finde:

> **Hvilken knude bliver opdaget (discovery) sidst?**
> (Altså: Hvem får **højeste discovery time?**)

🧠 DFS: Hvordan virker det?

- DFS går **dybt** ned ad én sti, før den backtracker
- Hver knude får en **discovery time** (når vi ankommer), og en **finish time** (når vi er færdige med alle dens naboer)
- Du **går i dybden**, og bruger en **stak (implicit i rekursion)**



Når du i DFS rammer et **endepunkt** (en knude uden nye naboer),
så **går du tilbage** ("backtracker") til den seneste knude, hvor du **ikke har besøgt alle naboer endnu**.

Derfor kommer vi sidst til e, og svaret er derfor e

Forward edge:

🧠 Først: Hvad er en **forward edge**?

I DFS betyder:

- En **forward edge** er en kant fra en **knude** `u` **til en efterkommer** `v`,
  hvor `v` **allerede er blevet besøgt** under DFS, men ikke er et direkte barn i DFS-træet.

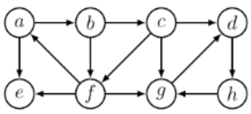
→ Det er altså en **"spring fremad" kant** til en knude, vi ville være nået til **senere alligevel**

🧠 Tabel: Hvad betyder ordene i forklaringen?

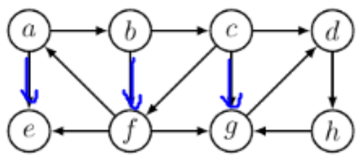
Ord	Forklaring – simpelt sprog
knude	Et punkt i grafen – fx `a`, `b`, `c` ...
kant	En pil fra én knude til en anden – fx `a → b`
`u`	Den **knude du står i**, hvor pilen (kanten) starter
`v`	Den **knude pilen peger hen til**
efterkommer	En knude du ville komme til **ved at følge stien i DFS-træet** fra `u`
barn	Den **næste knude** du gik til direkte fra `u` i DFS
DFS-træet	Det "træ" der bliver bygget under DFS – hvem fandt hvem

Eksempel:

Vi ser stadig på dybde-først søgning DFS-VISIT(G, a) på grafen nedenfor med start i knuden a.



Hvor mange forward edges findes i grafen under denne søgning?



Der er 3 forward edges fordi alle 3 kanter er veje til en knude, som allerede er opdaget.

Husk i forward edge skal det være i samme "træ" - ellers bliver det en cross edge, fordi den er fra et andet "træ" (kig under cross edge)

På figuren til højre ses et **andet eksempel:**
e kan opdages 3 gange, men første gang den opdages er via f. Derfor er f forældre til e.
Da vi så senere kommer til d->e er e allerede opdaget og vi er stadig i samme gren, derfor er d-e en forward edge.

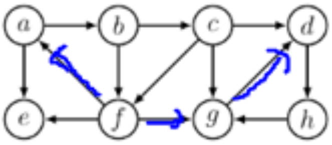
```
a
├─ b
│   └─ d
│       ├─ f
│       │   └─ e
│       └─ e
└─ c
    └─ e
```

Back edge:

🧠 Først: Hvad er en back edge?

> En **back edge** er en kant fra en knude `u` til en **forfader** `v` i DFS-træet.
> Det betyder: DFS **allerede har besøgt** `v`, og `v` ligger **højere oppe** i den sti, du kom fra.

→ Det er altså en **"spring tilbage"** kant til et sted, du **kom fra før** i samme gren.



De blå er back edges fordi det er pile til steder vi besøgte tidligere - altså pile der går tilbage / opad i træet

Cross edge:

🧠 Hvad er en cross edge?

> En **cross edge** er en kant fra `u` til `v`,
> hvor `v` **allerede er besøgt**, men ligger i **en anden gren** af DFS-træet.

→ Altså: `v` er **hverken forfader eller efterkommer**, men **bare en anden gren**

Der er ingen cross edges i tidligere graf, derfor tager vi nyt eksempel:

```
a
├— b
|  └— d
|     └— f
|        └— e (kant til e)
└— c
   └— e (også har en kant til e)
```

Her er e en efterkommer af b-d-f grenen og ikke c grennen fordi den FØRST er fundet i b-d-f

🧠 Hvad sker der i DFS?

- DFS starter i `a`
- Går til `b → d → f → e`
  → ✅ `e` opdages her – altså `f` **er forælder til** `e` **i DFS-træet**
- Når DFS senere kommer til `c → e`
  → ❌ `e` **er allerede besøgt**, så kanten **bruges ikke til at opdage** `e`

✅ Konklusion:

> `e` **er en efterkommer af** `f` **(og dermed** `d`, `b`, `a`)
> `e` **er IKKE en efterkommer af** `c`
> → `c → e` er derfor en **cross edge** ✅

Topologisk sortering

🔍 Hvad er topologisk sortering?

> En topologisk sortering af en rettet, acyklisk graf (DAG) er en rækkefølge af knuderne, så **hver kant** `u → v` kommer med `u` **før** `v`.

→ Altså: **ingen knude kommer før dens forældre**

Kort forklaret:
Man sortere efter:
Hvor mange pile går der ind i knuden fra start.
Når du har fundet dem med 0, så vælger du dem, derefter fjerner du dem og deres udadgående pile og gør det igen, de næste kommer så i rækkefølgen derefter

Guide:

1. **"Tæl hvor mange pile går ind i hver knude (indegree)"**
2. **"Start med dem med 0"**
3. **"Vælg én af dem og sæt den i rækkefølgen"**
4. **"Fjern den knude og dens udadgående pile"**
5. **"Opdater indegree for alle påvirkede knuder"**
6. **"Gentag til alle knuder er brugt"**

💡 Ekstra tips til eksamen:

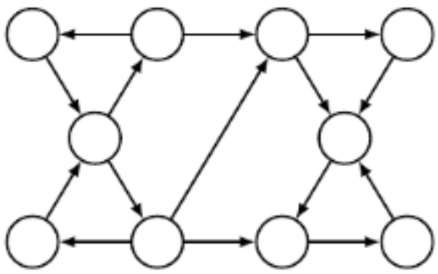
- Hvis der er **flere knuder med 0 indegree**, må du vælge dem i **vilkårlig rækkefølge**
  → Det er derfor, der kan være **flere gyldige løsninger**
- Hvis du ender med en knude med **indegree > 0 og intet at vælge,**
  → så har du en **cyklus**, og grafen **kan ikke topologisk sorteres**

Stærk sammenhængskomponent (SCC)

🔍 Husk: Hvad er en **stærk sammenhængskomponent?**

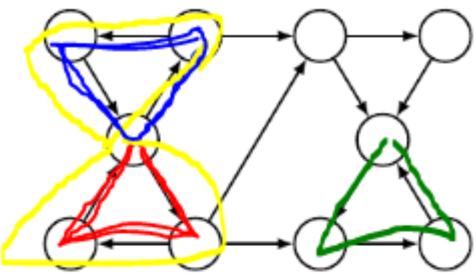
> En **stærk sammenhængskomponent (SCC)** er en **maksimal gruppe af knuder**, hvor der findes en vej **fra alle til alle** – altså:
>
> - **"Man kan komme fra A til B"**
> - **"og fra B til A"**

Hvor mange stærke sammenhængskomponenter har grafen nedenfor?



Jeg har nedenfor tegnet de 4 "cirkler" der findes. Man skal forstå det som cirkler hvor man kan køre rundt i ring. Altså hvis man danner en ring, og alle punkter i ringen kan nå sig selv. Så er det en SCC

ammenhængskomponenter har gra



Hvilke af disse algoritmer kan finde korteste vej:

> **Alle algoritmerne KAN finde korteste vej** – men **kun i nogle bestemte situationer.**

Derfor er spørgsmålet ikke:

👉 *"Kan de finde korteste vej?"*

Men i stedet:

👉 *"Er de **lovlige at bruge** i denne graf?"*

Brug dette skema:

🧠 Eksamenstjek – Hvilken algoritme må jeg bruge?

Spørgsmål	Svar	Hvad gør du?
Er grafen vægtet?	❌ Nej	Fjern: Dijkstra, Bellman-Ford, Floyd, DAG-SP
	✅ Ja	Fortsæt
Er der negative vægte?	✅ Ja	Fjern: Dijkstra, BFS
	❌ Nej	Fortsæt
Er der negative cykler?	✅ Ja	Fjern: Dijkstra, Bellman-Ford, Floyd, DAG-SP
	❌ Nej	Fortsæt
Er grafen en DAG?	❌ Nej	Fjern: DAG-Shortest-Paths
	✅ Ja	Fortsæt
Er opgaven 'én til alle'?	✅ Ja	Fjern: Floyd-Warshall
	❌ Nej	Fjern: Dijkstra, DAG-Shortest-Paths

Hvornår er en graf så hvad:

✅ Hvornår er grafen vægtet?

Spørgsmål	Hvad du skal kigge efter
Står der tal på kanterne?	Der er vægte mellem knuderne (fx 2, 5, -1)
Opgaven siger: "afstand", "pris" eller "vægt"?	Det er en vægtet graf

Uvægtet graf = Der **står ingen tal på kanterne**
Eller: **Alle forbindelser koster det samme** (typisk "1 hop")

Vægtet graf = Der står **et tal (pris, længde, afstand, tid)** mellem to knuder

✅ Hvornår har grafen **negative vægte**?

Spørgsmål	Hvad du skal kigge efter
Står der tal som **-1, -2, -3** på kanterne?	Ja → Grafen har negative vægte
Er der **ingen vægte**, eller kun positive tal?	Grafen har ikke negative vægte

✅ Hvornår har grafen **negative cykler**?

Spørgsmål	Hvad du skal kigge efter
Står der **en rute i ring** hvor summen af vægtene er negativ?	F.eks. `a → b → c → a` med vægte `-2, -3, +4` = -1
Står der direkte i opgaven: "Grafen indeholder en negativ cykel"?	Så: Stop – intet kan bruges

🧠 1. Hvornår er en graf en DAG?

(DAG = **Directed Acyclic Graph** → rettet graf uden cykler)

🔍 Du ved grafen er en DAG, når:

- Der er **pile** (altså grafen er *rettet*) ✅
- Der er **ingen cykler** = du **kan ikke komme tilbage til dig selv** via pile ❌ 🔁

🪄 Sådan spotter du det i opgaven:

- Alle pile går "én vej" (ingen rundkørsler)
- Du kan ikke finde en sti hvor du fx går `a → b → c → a`
- Opgaven **kan direkte nævne "acyklisk" eller "DAG"** – så er det 100 % sikkert

✅ Hvornår skal du finde **én til alle**?

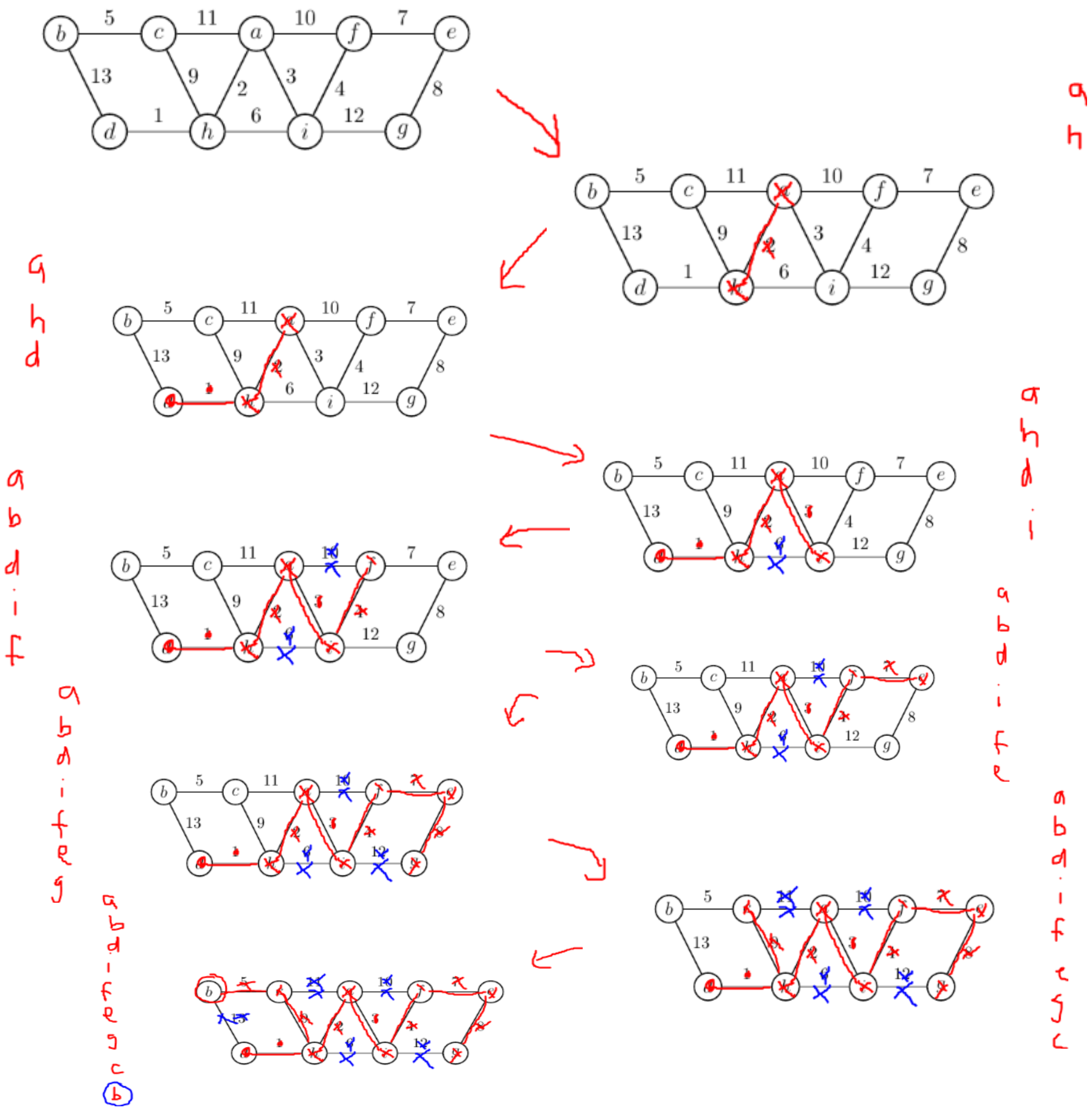
✅ Hvornår skal du finde **alle til alle**?

- 🔸 **Hvis opgaven nævner en startknude** → du skal finde **én til alle**
- 🔸 **Hvis opgaven ikke nævner nogen startknude** → du skal finde **alle til alle**

Prims Algoritme

🧠 Hvad er Prim's algoritme?

> Du **starter i én knude** (her: `a`), og så **bygger du langsomt et træ** ved **altid at tage den billigste kant**, der forbinder **en ny knude**.



Vi slutter altså her ved b, da der ikke er flere mulige veje nu.

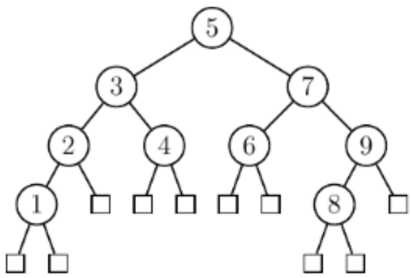
Rød Sort træer

✅ Rød-sorte træ-regler (vigtigste):

1. **Rodknuden er sort**
2. **Ingen rød knude har en rød barn** (→ altså: ingen to røde i træk)
3. Hver vej fra rod til blad har samme antal sorte knuder (men det er kun relevant, hvis vi ændrer strukturen – ~~her gør vi ikke det)~~

Delmængder af knuder

Hvilke af nedenstående delmængder af knuder vil gøre træet til et lovligt rød-sort træ, hvis netop denne delmænge af knuder farves røde (og resten farves sorte). [*Et eller flere svar.*]



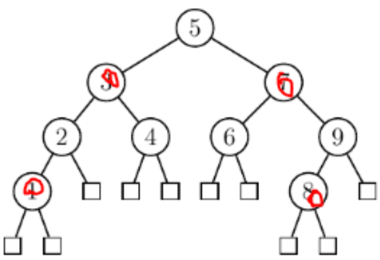
📌 I denne opgave:

- De **røde knuder** er dem, der vælges i delmængden.
- **Resten er sorte.**
- Vi skal tjekke om der **forekommer to røde i træk** (regel 2).

Nemmeste måde at løse denne opgave på:
Tegn træet og "farvelæg" for hver svarmulighed og tjek om den overholder regel 1 og 2

fx her: (Hvor svarmuligheden er 1,3,7,8)

☐ $1, 3, 7, 8$



Den er lovlig fordi roden er sort, og der kommer ingen røde efter røde

Indsæt nøgle

🧠 Rød-sort træ-regler (vigtigst her):

1. Ny indsættelse sker som **rød knude**
2. En **rød knude må ikke have en rød forælder**
3. Hvis der sker konflikt, laver vi:
   - **Recoloring** (hvis onkel er rød)
   - **Rotation + farveændring** (hvis onkel er sort eller mangler)

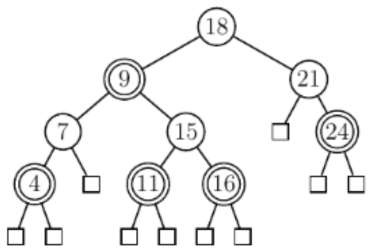
Når du **indsætter** en værdi i et **binært søgetræ** (som et rød-sort træ også er), så følger du denne regel:

📐 Reglen for binært søgetræ:

- Hvis den nye værdi er **mindre** end den aktuelle knude → **gå til venstre**
- Hvis den nye værdi er **større** → **gå til højre**

Eksempel:

Indsæt nøglen 25 i nedenstående rød-sorte træ (dobbeltcirkler angiver røde knuder).



📌 Step 1: Indsæt 25

Vi følger BST-regler:

- 25 > 18 → højre
- 25 > 21 → højre
- 25 > 24 → højre
  → **25 bliver barn af 24**

🟥 25 bliver **rød** (som altid ved indsættelse)

📌 Step 2: Kig på forældre og onkel

- 25 er rød
- **Forælder = 24** → også 🔴 rød
- **Onkel = null / mangler** (da 24 er højre barn af 21)
  → **Rød far + sort onkel = brud på regler** → vi skal rette

📌 Step 3: Fix med **venstre rotation omkring 21**

Vi har en **rød højre barn af en rød højre barn** (24 → 25)

→ Vi laver **venstre rotation omkring 21**, og skifter farver:

- 24 bliver ny far
- 21 bliver venstre barn
- 25 forbliver højre barn af 24
- 24 bliver sort, 21 og 25 bliver røde

✅ Det endelige træ:

- 18 (sort)
  - venstre: 9
  - højre: **24 (sort)**
    - venstre: **21 (rød)**
    - højre: **25 (rød)**

Køretider på algoritmer spørgsmål (Θ - notation)

✅ Oversigt: Eksempler med svar i Θ-notation

#	Kodeeksempel	Beskrivelse	Svar
1	`i = 1; while i <= n: i = i + i`	`i` fordobles hver gang	$\Theta(\log n)$
2	`i = 1; while i <= n: i = i + 1`	`i` vokser lineært	$\Theta(n)$
3	`for i in range(n): pass`	Én løkke over `n`	$\Theta(n)$
4	`for i in range(n): for j in range(n): pass`	To løkker over `n`	$\Theta(n^2)$
5	`for i in range(n): for j in range(i): pass`	Summen: 1+2+...+n	$\Theta(n^2)$
6	`for i in range(n): for j in range(log n): pass`	Ydre = n, indre = log n	$\Theta(n \log n)$
7	`while True: break`	Kører én gang	$\Theta(1)$
8	`for i in range(n): while k < n: k *= 2`	Indre = log n, ydre = n	$\Theta(n \log n)$
9	`for i in range(n): for j in range(n): for k in range(n): pass`	Tre uafhængige løkker	$\Theta(n^3)$
10	`for i in range(n): for j in range(i, n): pass`	Halv trekantsum	$\Theta(n^2)$
11	`i = 1; while i ≤ n: j = n; while j > 1: j -= 1; i *= 2`	Ydre = log n, indre = n	$\Theta(n \log n)$
12	`i = 1; j = 1; while i ≤ n: i += 5; while j < i: j += 1`	`j` tæller op på tværs – samlet n gange	$\Theta(n)$
13	`i = 1; j = n; while i ≤ j: i *= 4; j *= 2`	`i` og `j` vokser eksponentielt	$\Theta(\log n)$

Sortere n heltal med COUNTING-SORT

Vi ser i denne opgave på at sortere n heltal med værdier i intervallet $[0\,;n^3[$.
Hvad er worst case køretiden for COUNTINGSORT på denne type input?

🟩 **Endeligt svar: Θ(n³)**

Hvis vi bruger **Counting Sort** til at sortere `n` heltal i intervallet **[0 ; n²)**, så gælder:

🟩 **Endeligt svar: Θ(n²)**

Sortere n heltal med RADIX-SORT

Vi ser stadig på at sortere n heltal med værdier i intervallet $[0\,;n^3[$.
Hvad er worst case køretiden for RADIXSORT på denne type input, når heltal betragtes som bestående af tre digits med værdier i intervallet $[0\,;n[$?

📘 Opgaven siger:

- Vi har `n` heltal i intervallet $[0;n^3[$
- Vi skal bruge **Radix Sort**
- Hver heltal **består af 3 digits**, og **hver digit er i intervallet** $[0;n[$

🧠 Radix Sort teori:

Radix Sort har køretid:

$$\Theta(d \cdot (n+k))$$

hvor:

- `d` = antal cifre (digits)
- `n` = antal elementer
- `k` = antal mulige værdier pr. digit (radix/base)

📊 I vores opgave:

- `d = 3` (der står: "bestående af tre digits")
- `k = n` (fordi hver digit er i $[0;n[$)
- `n` = antal heltal

✅ **Beregning:**

$$\Theta(3 \cdot (n+n)) = \Theta(3 \cdot 2n) = \Theta(n)$$

Konstantfaktorer som `3` og `2` droppes i Θ-notation.

🟩 **Endeligt svar:**

$$\Theta(n)$$

Sortere n heltal med QUICK-SORT

💡 **Trick til eksamen:**

Hvis der står "worst case" for QuickSort → svar altid

👉 $\boxed{\Theta(n^2)}$

(uanset hvad inputtet er)

Eksempel:

Vi ser stadig på at sortere n heltal med værdier i intervallet $[0\,;n^3[$.
Hvad er worst case køretiden for QUICKSORT på denne type input?

📘 **Opgaven siger:**

- Vi skal sortere `n` heltal
- Værdierne ligger i intervallet $[0; n^3[$
- Vi bruger **QuickSort**
- Spørgsmålet gælder **worst case køretid**

🧠 **QuickSort teori:**

- **Best case (og gennemsnit):** $\Theta(n \log n)$
- **Worst case:**
  Hvis pivots vælges dårligt (f.eks. altid det største eller mindste element), bliver listen opdelt ekstremt skævt:

$$\Theta(n^2)$$

🤔 **Gælder input-intervallet $[0; n^3[$ for worst case?**

Nej. Det har **ingen betydning** for QuickSort's **worst case**, fordi QuickSort ikke er afhængig af værdi-intervallet – kun af hvordan elementerne er fordelt og hvordan pivots vælges.

✅ **Svar:**

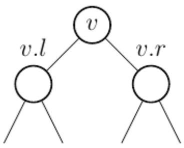
$$\boxed{\Theta(n^2)}$$

v.maxS

Vi har en række tal $x_1 < x_2 < x_3 < \cdots < x_n$, hvor hvert tal enten er hvidt eller sort. En *sort streak* er en sammenhængende delrække $x_i, x_{i+1}, \ldots, x_j$ af sorte tal.
Vi ønsker at kunne indsætte og slette tal i $O(\log n)$ tid og at kunne finde længden af længste sorte streak i $O(1)$ tid.
Vi bruger derfor et rød-sort træ, hvor tallene selv er nøgler, og hvor hver knude v gemmer følgende ekstra information:

v.maxS:	Længden af længste sorte streak i $T(v)$.
v.maxLS:	Længden af længste sorte streak i $T(v)$, som starter ved det mindste tal i $T(v)$.
v.maxRS:	Længden af længste sorte streak i $T(v)$, som ender ved det største tal i $T(v)$.
v.hasWhite:	En boolean, som angiver, om $T(v)$ indeholder et hvidt tal.

Her angiver $T(v)$ rækken af tal, der findes i undertræet med v som rod. Bemærk, at længderne ovenfor godt kan være nul.
Vi ønsker at kunne beregne informationerne i en knude v ud fra informationerne i dens børn $v.l$ og $v.r$.



Vi starter med at se på beregningen af *v.maxS*. Hvilket af følgende svar angiver, hvordan denne kan beregnes ud fra informationen i børnene (vi ser kun på det tilfælde, hvor disse børn begge eksisterer)?

aldrig +1 hvis vi ikke kender rodens farve

✅ Hovedregel:

Når du **IKKE ved**, om roden `v` selv er sort, må du **IKKE bruge** `+1` i formlen.

🎯 Derfor:

I opgaven står fx:

> *"Vi starter med at se på beregningen af v.maxS ..."*

Men der står intet om **roden** `v` **selv** er sort. Derfor:

- Du kan **ikke** sige "nu fortsætter streaken op i `v`, så vi lægger +1 til".
- Du **kan kun** samle oplysninger fra børnene!

→ Det betyder:

```text
v.maxS = max(v.l.maxS, v.r.maxS, v.l.maxRS + v.r.maxLS)
```

v.maxLS

🤔 Hvordan regner man `v.maxLS` ud?

Der er to muligheder:

Situation	Hvad sker der
Hvis `v` er **hvid** eller `v.l` indeholder **hvide** noder	Så stopper streaken – brug bare `v.l.maxLS`
Hvis både `v` og venstre side er **sorte**	Så kan du tage hele `v.l.maxS`, lægge `+1` til (for `v` selv), og lægge `v.r.maxLS` til (for højre side)

Brug ovenstående!

❗ Vi ser på `v.maxLS`, altså: **længste sorte streak der starter ved det mindste tal i** `T(v)`

→ Det betyder: streaken **starter i venstre barn** og **kan forlænges op i** `v`
Men **kun hvis** `v` **selv er sort**, og **venstre barn ikke indeholder hvide tal.**

📌 Regeltip til eksamen:

> Når du udregner `maxLS`, så spørg dig selv:
>
> ✅ Er `v` sort?
> ✅ Er `v.l` helt sort (`v.l.hasWhite == false`)?
>
> Hvis **begge er ja** → du må bruge `+1 + v.r.maxLS`
> Ellers → bare brug `v.l.maxLS`

r.maxS

Strategi til eksamen:

Vi **starter i roden** `v` og ved, at `v.maxS` er længden af den længste sorte streak.

Vi har 3 muligheder:

1. Den streak ligger **kun i venstre** → gå til `v.l`
2. Den streak ligger **kun i højre** → gå til `v.r`
3. Den streak går **gennem v** (dvs. kombinerer `v.l.maxRS + 1 + v.r.maxLS`) → `v` er en del af streaken ✅

Hvad skal du så gøre i eksamen?

Bare **sammenlign:**

plaintext Kopiér Rediger

```plaintext
Hvis v.maxS > v.l.maxS og v.maxS > v.r.maxS  → returner v
Ellers hvis v.maxS == v.l.maxS              → rekursivt søg i v.l
Ellers                                      → rekursivt søg i v.r
```

Logiske udsagn

☐ Hvis q er sand, og r er falsk, kan p tildeles en sandhedsværdi, sådan at udsagnet $(\neg p \wedge q) \vee (p \wedge r)$ er sandt.

Forstå dette spørgsmål som at de spørger om det lange udsagn kan blive sandt, de er ligeglade med p.

Sandt falsk:

✅ Trin-for-trin skabelon til logiske udsagn

1. **Lær standard-omskrivninger udenad**
   Brug dem til hurtigt at vurdere ækvivalens:

Udsagn	Ækvivalent med
$p \Rightarrow q$	$\neg p \vee q$
$\neg(p \vee q)$	$\neg p \wedge \neg q$
$\neg(p \wedge q)$	$\neg p \vee \neg q$
$p \Leftrightarrow q$	$(p \Rightarrow q) \wedge (q \Rightarrow p)$
$\neg(p \Rightarrow q)$	$p \wedge \neg q$

2. **Tjek om påstandene matcher dem**
   - Brug omskrivningerne til at forenkle udtrykkene.
   - Er begge sider ens (logisk ækvivalente)? Ja = sandt ✅, nej = falsk ❌
3. **Er du i tvivl? Brug en sandhedstabel**
   Lav én med de mulige værdier for p, q, r og se om begge sider altid giver samme resultat.

Sandhedstabel:

🧠 Hvad ER en sandhedstabel?

En sandhedstabel bruges til at **afprøve logiske udsagn.** Tabellen viser, om et udsagn er **sand eller falsk** afhængigt af, hvilke værdier de enkelte dele (p, q, r) har (enten **sand (S)** eller **falsk (F)**).

✅ Liste over logiske symboler og deres betydning

Symbol	Navn	Betyder kort	Eksempel	Læses som
¬	Negation	**Ikke**	¬p	"Ikke p"
∧	Konjunktion	**Og**	p ∧ q	"p og q er begge sande"
∨	Disjunktion	**Eller**	p ∨ q	"mindst én af p eller q er sand"
⇒	Implikation	**Hvis ... så**	p ⇒ q	"Hvis p er sand, så er q også"
⇔	Bi-implikation	**Hvis og kun hvis**	p ⇔ q	"p og q har samme sandhedsværdi"
⊕	Eksklusiv eller	**Enten eller**	p ⊕ q	"Enten p eller q, men ikke begge"
⊤	Sandhedsværdi	**Sand**	—	Konstant "sand"
⊥	Sandhedsværdi	**Falsk**	—	Konstant "falsk"
/ eller ≠	Forskellig fra	**Ikke lig med**	p ≠ q	"p er ikke lig med q"

Ækvivalent =

To udsagn er **ækvivalente**, hvis de **altid har samme sandhedsværdi –** uanset hvad p, q, r osv. er.

✅ Sandhedstabel-skabelon (3 udsagn: p, q, r)

p	q	r	¬p	¬q	¬r	p ∧ q	p ∨ q	p ⇒ q	p ⇔ q	(p ∧ q) ⇒ r	(p ⇒ r) ∧ (q ⇒
T	T	T	F	F	F	T	T	T	T	T	T
T	T	F	F	F	T	T	T	T	T	F	F
T	F	T	F	T	F	F	T	F	F	T	T
T	F	F	F	T	T	F	T	F	F	T	F
F	T	T	T	F	F	F	T	T	F	T	T
F	T	F	T	F	T	F	T	T	F	T	F
F	F	T	T	T	F	F	F	T	T	T	T
F	F	F	T	T	T	F	F	T	T	T	T

🟥 1. Sandhedstabel for **p og q**

p	q	¬p	¬q	p ∧ q	p ∨ q	p ⇒ q
T	T	F	F	T	T	T
T	F	F	T	F	T	F
F	T	T	F	F	T	T
F	F	T	T	F	F	T

For at tjekke antal rækker i en sandhedstabel:

🧠 Du har ret:

Hvis du skal lave en sandhedstabel for $p \Rightarrow (q \wedge r)$, så er der **8 rækker**, fordi:

> 3 udsagn ⇒ 2^3 = **8 kombinationer**

2 fordi true false, 3 fordi 3 udsagn.

Tautologi:

🔁 Simpelt forklaret:

> En tautologi er ligesom et "logisk fakta", der **ikke kan være falsk.**

✅ Eksempler på tautologier:

1. $p \vee \neg p$
   (Enten er p sand – eller også er det ikke)
   → **Altid sand**
2. $(p \Rightarrow q) \vee (q \Rightarrow p)$
   (Uanset hvad, så "en af dem følger af den anden")
   → **Altid sand**
3. $(p \wedge q) \Rightarrow p$
   (Hvis både p og q er sande, så er p i hvert fald sand)
   → **Altid sand**

Logiske tegn der er mærkelige

🧠 Guide: Hvad betyder de logiske og matematiske tegn?

Tegn	Kaldes	Betyder på dansk	Eksempel
$\forall$	"For alle"	Det gælder for **alle** værdier	$\forall x \in \mathbb{Z} : x^2 > 0$ betyder "for alle heltal er $x^2 > 0$"
$\exists$	"Der findes"	Der findes **mindst én** værdi	$\exists x \in \mathbb{Z} : x^2 = 4$ betyder "der findes et heltal med kvadratet 4"
$\neg$	"Ikke"	Det **modsatte** (negation)	$\neg p$: "det er ikke sandt at p"
$\wedge$	"Og" (konjunktion)	Begge ting skal være sande	$p \wedge q$: "p og q er sande"
$\vee$	"Eller" (disjunktion)	Mindst én af dem er sand	$p \vee q$: "p eller q (eller begge) er sande"
$\Rightarrow$	"Implikation"	Hvis p er sand, så er q også sand	$p \Rightarrow q$: "hvis p, så q"
$\Leftrightarrow$	"Ækvivalens"	De er **logisk ens** – altid samme sandhedsværdi	$p \Leftrightarrow q$: "p og q er ækvivalente"
$\in$	"Tilhører"	Værdien er **med i mængden**	$x \in \mathbb{Z}$: "x er et heltal"
$\mathbb{Z}$	"Heltal"	Alle positive og negative heltal + 0	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{N}$	"Naturlige tal"	Typisk $\{0, 1, 2, 3, \ldots\}$	Bruges i mængde-udsagn
$\iff$	"Hvis og kun hvis" (ækvivalens)	Betyder præcis det samme som $\Leftrightarrow$	Ofte brugt i logiske sammenligninger

✅ Guide: Forstå kvantorer og udsagn (på dansk!)

Udtryk	Hvad betyder det?	Hvad skal du gøre?
$\forall x \in \mathbb{Z} : P(x)$	For **alle heltal x**, gælder det, at udsagnet $P(x)$ er sand	Tjek at **ingen undtagelser findes.** Det skal virke for ALLE heltal.
$\exists x \in \mathbb{Z} : P(x)$	Der findes **mindst ét heltal x**, så udsagnet $P(x)$ er sand	Du skal kun finde **én x**, hvor det virker. Så er det sandt.
$\neg\exists x \in \mathbb{Z} : P(x)$	Der findes **ingen x**, hvor $P(x)$ er sand	Det betyder, at **udsagnet aldrig passer** for nogen x.
$\neg\forall x \in \mathbb{Z} : P(x)$	Det er **ikke sandt for alle x**	Der findes **mindst én x**, hvor det er falsk. (Samme som $\exists x : \neg P(x)$)
$\forall x \exists y : P(x, y)$	For **hver x**, findes der **mindst én y**, så $P(x, y)$ er sand	For hver x skal du kunne finde et y, der opfylder kravet
$\exists x \forall y : P(x, y)$	Der findes **ét x**, hvor **alle y** opfylder $P(x, y)$	Find én x, hvor det **altid passer – uanset y**
$\neg\exists x \forall y : P(x, y)$	Der findes **ingen x**, hvor udsagnet passer for alle y	For **alle x**, vil der være **mindst én y**, hvor det fejler
$\neg\forall x \exists y : P(x, y)$	Det er **ikke rigtigt**, at hver x har en y som opfylder P(x, y)	Der findes **mindst én x**, hvor **ingen y** får det til at virke

🔢 Udsagnet:

$$\forall n \in \mathbb{Z} : \exists k \in \mathbb{Z} : n > 2k$$

Det betyder på dansk:

> **For alle heltal n**, findes der **mindst ét heltal k**, sådan at:
> **n er større end 2 gange k**

,

🔢 Udsagnet:

$$\exists k \in \mathbb{Z} : \forall n \in \mathbb{Z} : n > 2k$$

På dansk:

> Der **findes ét heltal k**, sådan at **for alle heltal n** gælder:
> **n er større end 2·k**

For flere kig under Diskret Matematik

Betragt nedenstående relation på mængden (a,b,c)

✅ Guide: Hvad betyder egenskaberne for en relation $R \subseteq A \times A$?

Egenskab	Hvad det betyder	Tjekregel
Refleksiv	Alle elementer relaterer til sig selv	For alle $x \in A$, skal $(x,x) \in R$
Symmetrisk	Hvis x relaterer til y, så relaterer y også til x	Hvis $(x,y) \in R$, så skal $(y,x) \in R$
Antisymmetrisk	Hvis x relaterer til y og omvendt, så må $x = y$	Hvis $(x,y) \in R$ og $(y,x) \in R$, så skal $x = y$
Transitiv	Hvis $x \to y$ og $y \to z$, så gælder $x \to z$ også	Hvis $(x,y), (y,z) \in R$, så skal $(x,z) \in R$
Ækvivalensrelation	Skal være: **refleksiv**, **symmetrisk** og **transitiv**	Tjek alle tre ovenfor – hvis bare én mangler, er det ikke en ækvivalensrelation
Partiel ordning	Skal være: **refleksiv**, **antisymmetrisk** og **transitiv**	Tjek de tre relevante egenskaber – partielle ordninger er ikke nødvendigvis symmetriske

✅ Refleksiv

Definition:
En relation R på en mængde A er **refleksiv**, hvis **alle** elementer i mængden relaterer til sig selv.
Altså:

$$\text{For alle } x \in A : (x,x) \in R$$

🔍 Eksempel:

Hvis mængden er $A = \{a,b,c\}$, så skal relationen indeholde:

- (a,a)
- (b,b)
- (c,c)

Hvis bare én af dem mangler, så **er den ikke refleksiv**.

🔁 Transitiv

Definition:
En relation R er **transitiv**, hvis:

$$\text{Hvis } (a,b) \in R \text{ og } (b,c) \in R, \text{ så skal også } (a,c) \in R$$

👉 Med andre ord:
Hvis man kan gå fra a til b, og fra b til c, så skal man også kunne gå direkte fra a til c.

✅ Tjekliste:

Har du dette?	Skal dette så være med?	Hvis ikke?
(a,b), (b,c)	(a,c)	❌ Ikke transitiv

🔁 Symmetrisk

Definition:
En relation R er **symmetrisk**, hvis:

$$\text{Hvis } (a,b) \in R, \text{ så skal også } (b,a) \in R$$

Det gælder for **alle** par i relationen.

🔍 Eksempel:

Hvis der står (a,b) i relationen, så skal også (b,a) stå der — ellers er den **ikke symmetrisk**.

Transitive lukning af relation

🔁 Transitiv lukning handler om:

👉 **Hvis der findes en vej fra ét bogstav til et andet – uanset hvor mange led der er imellem – så skal vi tilføje en direkte pil.**

Eksempel:

Vi har disse pile:

- **a → b**
- **b → c**

👉 Her kan **a komme til c** gennem b.
Derfor tilføjer vi **a → c** i den *transitive lukning*.

Et eksempel med 3 led:

- **a → b**
- **b → c**
- **c → d**

Så skal vi også tilføje:

- a → c (fordi a → b → c)
- a → d (fordi a → b → c → d)
- b → d (fordi b → c → d)

Rækkefølgen for hvordan de tilføjes er ligegyldig.
Eksempel:

🔍 Opgave:

Find den transitive lukning af R
(Altså: Tilføj alle manglende "direkte pile", hvis nogen kan nå hinanden via flere skridt.)

Skriv dit bud som en liste – fx:

$$\{(a,b),(b,c),\ldots,(a,c),(d,f)\}$$

Hint: Der er **flere veje**, hvor **a er starten på flere ruter** 🧭
Tag din tid – så kigger jeg på dit svar bagefter og hjælper dig videre 😁

Vi deler den op i 2 R først for at gøre det nemmere. Vi har R1 som er (a,b),(b,c)
Her mangler der en lukning fra a->c

Så tager vi R2 som er resten
Her går den fra a-d og d-e, derfor skal vi lave en lukning fra a-e
den går også fra e-f og vi skal derfor have en lukning fra a-f, men vi skal også have fra d-f

Diskret Matematik

Udsagn	Betyder på almindeligt dansk
1a: $\exists n \in \mathbb{Z} : n^2 + 1 = 82$	Der findes et helt tal n, så $n^2 + 1 = 82$. *(Sandt, da $n = 9$)*
1b: $\exists n \in \mathbb{Z} : \exists k \in \mathbb{Z} : n + k = n - k$	Der findes to hele tal n og k, så $n + k = n - k$. *(Sandt, hvis $k = 0$)*
1c: $\forall n \in \mathbb{Z} : n^2 \in \mathbb{N}$	For alle hele tal n, er n^2 et naturligt tal. *(Falsk – $n = 0$ er ok, men $\mathbb{N}$ indeholder ikke nødvendigvis 0 i alle definitioner – og negative n giver positivt kvadrat, så typisk sandt afhængigt af definition)*
1d: $\forall n \in \mathbb{Z} : \exists k \in \mathbb{N} : \sqrt{n + k} \in \mathbb{N}$	For ethvert helt tal n, findes et naturligt tal k, så $n + k$ bliver et kvadrattal. *(Sandt – fx ved $n = -5$, vælg $k = 5 \rightarrow \sqrt{0} = 0$)*
1e: $\forall n \in \mathbb{N} : \forall k \in \mathbb{Z} : n \neq k$	For ethvert naturligt tal n, og ethvert helt tal k, gælder $n \neq k$. *(Falsk – fx $n = 1$, $k = 1$)*
1f: $\exists n \in \mathbb{N} : 1^n \neq 1$	Der findes et naturligt tal n, hvor $1^n \neq 1$. *(Falsk – $1^n = 1$ for alle n)*
1g: $\exists n \in \mathbb{Z} : n^2 = 9$	Der findes et helt tal n, så $n^2 = 9$. *(Sandt – $n = \pm 3$)*
1h: $\exists n \in \mathbb{N} : \exists k \in \mathbb{N} : n^2 + k^2 = 17$	Der findes to naturlige tal n og k, så summen af deres kvadrater er 17. *(Sandt – fx $n = 1$, $k = 4$: $1^2 + 4^2 = 1 + 16 = 17$)*

✅ Liste over logiske symboler og deres betydning

Symbol	Navn	Betyder kort	Eksempel	Læses som
¬	Negation	**Ikke**	¬p	"Ikke p"
∧	Konjunktion	**Og**	p ∧ q	"p og q er begge sande"
∨	Disjunktion	**Eller**	p ∨ q	"mindst én af p eller q er sand"
⇒	Implikation	**Hvis ... så**	p ⇒ q	"Hvis p er sand, så er q også"
⇔	Bi-implikation	**Hvis og kun hvis**	p ⇔ q	"p og q har samme sandhedsværdi"
⊕	Eksklusiv eller	**Enten eller**	p ⊕ q	"Enten p eller q, men ikke begge"
⊤	Sandhedsværdi	**Sand**	—	Konstant "sand"
⊥	Sandhedsværdi	**Falsk**	—	Konstant "falsk"
/ eller ≠	Forskellig fra	**Ikke lig med**	p ≠ q	"p er ikke lig med q"

Kig under sandhedstabeller i algoritmer

📐 De Morgan's to regler:

Originalt udsagn	Omskrivning med De Morgan
$\neg(p \lor q)$	$\equiv \neg p \land \neg q$
$\neg(p \land q)$	$\equiv \neg p \lor \neg q$

🧠 Hvad betyder det?

- Hvis **ikke** (p eller q) er sandt ⇒ så må **både p og q være falske**
- Hvis **ikke** (p og q) er sandt ⇒ så må **mindst én af p eller q være falsk**