AD Noter

Alle eksempler og materiale er baseret på tidligere eksamenssæt/afleveringer

April 7, 2024

Contents

1 Grundlæggende Teori

1.1 Tidskompleksitet/Asymptotisk Notation

1.1.1 Typer af Asymptotisk Notation

Der findes tre hovedtyper af asymptotisk notation/køretider:

- **Store Theta** Θ
  Begrænser funktioner både nedefra og oppefra, "Average case"
  Beskrives med $=$
- **Store O** O
  Begrænser funktioner oppefra, "Worst case"
  Beskrives med $\leq$
- **Store Omega** Ω
  Begrænser funktioner nedefra, "Best case"
  Beskrives med $\geq$

Udover disse, er der også to øvrige ikke-asymptotisk notation:

- **Lille o** o
  Begrænser funktioner oppefra, beskrives med $<$
- **Lille omega** ω
  Begrænser funktioner nedefra, beskrives med $>$

1.1.2 Typer af Køretider

Der findes 7 primære former for køretider, rangeret fra bedst til værst:

- Konstant tid: $O(1)$
- Log n tid: $O(logn)$
- Lineær tid: $O(n)$
- Logaritmisk tid: $O(nlogn)$
- Kvadratisk tid: $O(n^2)$
- Eksponentiel tid: $O(2^n)$
- Fakulteret tid: $O(n!)$

Gode Regneregler:

For $n, m > 0$ og $a, b \in \mathbb{R}$:

- $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$ for $n \in \mathbb{N}$
- $n^a \cdot n^b = n^{a+b}$
- $1/n^a = n^{-a}$
- $\log(n \cdot m) = \log n + \log m$
- $\log(n^a) = a \log n$

De 10 Regler for Asymptotisk Notation:

Theorem 5. Let $f, g, h, p : \mathbb{R}^+ \to \mathbb{R}$ be asymptotically positive functions.

(R1) "Overall constant factors can be ignored"
If $c > 0$ is a constant then $cf(x)$ is $\Theta(f(x))$.

(R2) "For polynomials only the highest-order term matters"
If $p(x)$ is a polynomial of degree d, then $p(x)$ is $\Theta\left(x^d\right)$.

(R3) "The fastest growing term determines the growth rate"
If $f(x)$ is $o(g(x))$ then $c_1 g(x) + c_2 f(x)$ is $\Theta(g(x))$, where $c_1 > 0$ and $c_2 \in \mathbb{R}$ are constants.

(R4) "Logarithms grow faster than constants"
If $c > 0$ is a constant then c is $o\left(\log_a(x)\right)$ for all $a > 1$.

(R5) "Powers (and polynomials) grow faster than logarithms"
$\log_a(x)$ is $o\left(x^b\right)$ for all $a > 1$ and $b > 0$.

(R6) "Exponentials grow faster than powers (and polynomials)"
x^a is $o\left(b^x\right)$ for all a and all $b > 1$.

(R7) "Larger powers grow faster"
x^a is $o\left(x^b\right)$ if $a < b$.

(R8) "Exponentials with a bigger base grow faster"
a^x is $o\left(b^x\right)$ if $0 < a < b$.

Theorem 7. Let $f, g, h : \mathbb{R}^+ \to \mathbb{R}$ be asymptotically positive functions.

(R9) Transitivity
If $f(x)$ is $O(g(x))$ and $g(x)$ is $O(h(x))$ then $f(x)$ is $O(h(x))$.
If $f(x)$ is $o(g(x))$ and $g(x)$ is $o(h(x))$ then $f(x)$ is $o(h(x))$.
If $f(x)$ is $\Theta(g(x))$ and $g(x)$ is $\Theta(h(x))$ then $f(x)$ is $\Theta(h(x))$.

(R10) Chaining
If $f(x)$ is $O(g(x))$ and $g(x)$ is $\Theta(h(x))$ then $f(x)$ is $O(h(x))$.
If $f(x)$ is $\Theta(g(x))$ and $g(x)$ is $O(h(x))$ then $f(x)$ is $O(h(x))$.
If $f(x)$ is $o(g(x))$ and $g(x)$ is $O(h(x))$ then $f(x)$ is $o(h(x))$.
If $f(x)$ is $O(g(x))$ and $g(x)$ is $o(h(x))$ then $f(x)$ is $o(h(x))$.
If $f(x)$ is $o(g(x))$ and $g(x)$ is $\Theta(h(x))$ then $f(x)$ is $o(h(x))$.
If $f(x)$ is $\Theta(g(x))$ and $g(x)$ is $o(h(x))$ then $f(x)$ is $o(h(x))$.

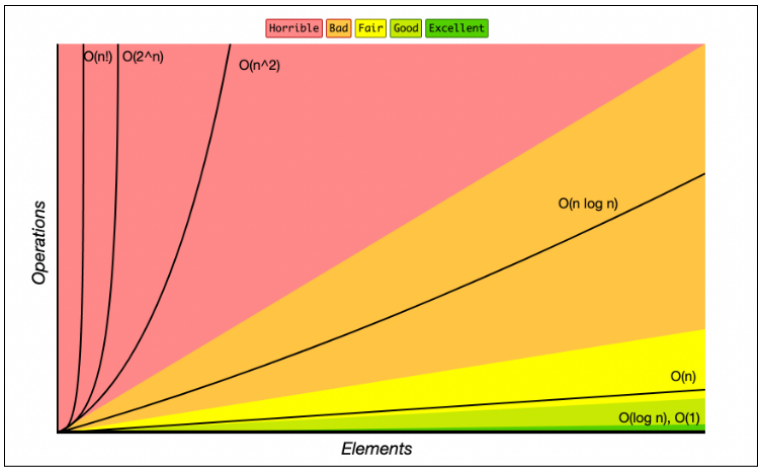


Figure 1: Graf over Store-O Notation.

1.2 Vigtige Algoritmer

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Figure 2: Oversigt over køretider af vigtige Dataalgoritmer.

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figure 3: Oversigt over køretider af vigtige Sorteringsalgoritmer.

1.3 Beviser & Invarianter

1.3.1 Bevistyper

Indenfor AD er der fem vigtige typer af beviser:

- Direkte bevis
- Kontrapositivt bevis
- Modstridsbevis
- Induktionsbevis
- Invariantsbevis

Induktion- og Invariantsbeviser er vigtigst.

1.3.2 Induktion

Består af to skridt:

- Basis: Bevis for $n = 0$, eller mindste mulige værdi for n
- Induktion: Hvis Basis holder, så bevis for $n + 1$

1.3.3 Invarianter

Invariant: En påstand som altid gælder (specifikke steder) i algoritmen

Termineringsbetingelse: En påstand som gælder når algoritmen terminerer

De to vigtigste ting når man beviser invarianter: De gælder fra starten og opretholdes gennem programmets operationer

1.4 Del & Hersk

Deler problem op i delproblemer med samme type
Løser hvert delproblem rekursivt
Kombinerer løsninger af delproblemer til løsning af oprindelige problem

1.4.1 Merge Sort

Mest almindelige del-og-hersk algoritme (Relevant for eksamen)
Deler array op og sorterer i mindre bidder via. rekursion
Antal elementer der sorteres: $n = r - p + 1$
Samlede antal kald: $2n - 1$

Køretid: $O(n)$ samt to rekursive problemer af størrelse $[n/2]$ og $[n/2]$

Worst-case rekursion: $T(n) = 2T(n/2) + \Theta(n)$

1.5 Rekursionsligninger

1.5.1 Master Theorem

Giver løsning til de hyppigst forekommende rekursionsligninger, på formen:

$$T(n) = aT(n/b) + \Theta(f(n))$$

hvor n er input størrelse, a er antal subproblemer, n/b er størrelsen af hvert subproblem og f(n) er kosten af operationer uden for rekursionen. (dele og merge løsninger)

Hvis en konstant $\epsilon > 0$ findes og $f(n)$ ikke er aftagende, så gælder der tre regler:

1. $f(n) = O\left(n^{\log_b(a)-\varepsilon}\right) \Longrightarrow T(n) = \Theta\left(n^{\log_b a}\right)$ (hvis kosten stiger per. niveau)
2. $f(n) = \Theta\left(n^{\log_b(a)}\right) \Longrightarrow T(n) = \Theta\left(n^{\log_b a} \log n\right)$ (hvis kosten er næsten den samme per. niveau)
3. $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right) \Longrightarrow T(n) = \Theta(f(n))$ (hvis kosten falder per. niveau)

Karatsuba overholder regel 1. Merge Sort overholder regel 2.

1.5.2 Substitutionsmetoden

Bruges til at bekræfte løsninger på rekursionsligninger

Består af tre trin:

- Gæt på en løsning, eks. $T(n) = O(f(n))$
- Antag at $T(n) \leq 1$ når $n \leq k$
- Brug induktion til at vise at $T(n) \leq cf(n)$ for konstant c og $n > k$

Eksempel for $T(n) \leq 2T(n/2) + cn$ hvor $c \geq 1$:

- Gæt på løsningen $T(n) = O(nlogn)$
- Antag at $T(1) = 1$
- Vis at $T(n) \leq cnlog(n) + cn$ (cn bliver lagt til for at holde ulighed i basiscase)
- Udregn Basis: $T(1) \leq c1log(1) + c1$; Sandt
- Induktionshypotese: $T(n') \leq cn'log(n') + cn'$ for $n' < n$
- Udregn induktion:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2(c(n/2)\lg(n/2) + c(n/2)) + cn \\ &= cn\lg(n) + cn \end{aligned}$$

1.5.3 Rekursionstræer

Måde at illustrere rekursionsligninger og få visuel over kosten af hvert niveau.

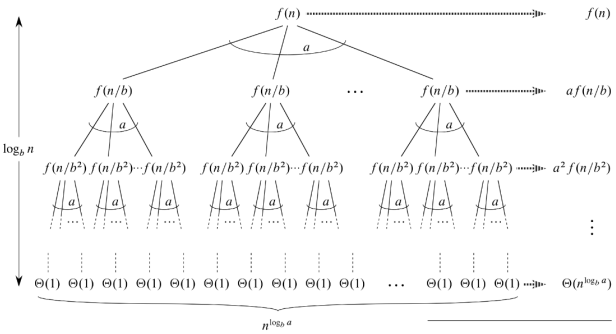


Figure 4: Rekursionstræ for regel 1.

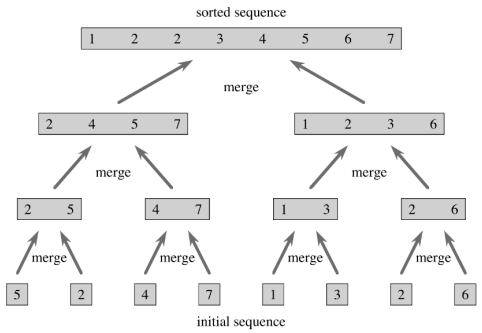


Figure 5: Rekursionstræ for Merge Sort (regel 2). Det kan ses, at kosten er den samme for hvert niveau.

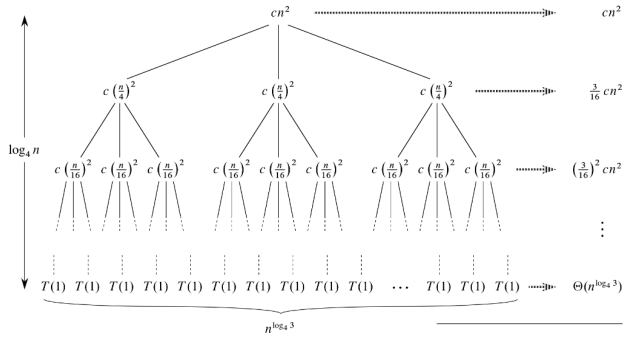


Figure 6: Rekursionstræ for regel 3.

1.6 Amortiseret Analyse

Bruges når operationers kost ikke er fordelt jævnt - nogle operationer koster meget mere end andre
"Spreder" kosten ud over andre operationer for at opnå en mere jævn overall kost
Gør worst-case tiden bedre end worst-case tiden for en specifik operation

Der er tre vigtige former af amortiseret analyse:

- Aggregeret Analyse
- Regnskab/Accounting Metoden
- Potentialemetoden

1.6.1 Aggregeret Analyse

Motivation: Vis for alle n, en sekvens med n operationer tager worst-case $T(n)$ tid.
Den amortiserede tid er $T(n)/n$

I Aggregeret Analyse findes der Stack og Binary Counter.

1.6.2 Stack Operationer

Har tre operationer:

- Push(S,x) - skubber element x på stakken S
- Pop(S) - popper og returnerer topelementet af stakken S
- MultiPop(S,k) - fjerner top k elementer af stakken S

Kost af Stack Operationer:

- Push/Pop er $O(1)$ tid individuelt og koster 1 individuelt
- n Push og Pop kald er $\Theta(n)$ tid og koster n i alt
- MultiPop er worst case $O(n)$, da stakken er størrelse n
- Total kost af Multipop er min{s,k,} og total tid er en lineær funktion af kosten
- Sekvens af n operationer koster $O(n^2)$
- Amortiseret kost er $O(1)$

1.6.3 Binary Counter/Binær Tæller

Implementer en k-bit binær tæller der tæller opad fra 0, repræsenteret med et array $A[0:k-1]$

Increment-funktionen tilføjer 1 til nuværende tal i tælleren
Hvis vi udfører n operationer på en stack der til at starte med er tom ser vi, at vi worst-case får en køretid på $O\left(n^2\right)$.
Vi beregner et upper bound $T(n)$ for alle n operationer og får derved den amortiserede cost pr. operation til $T(n)/n$.
Nu udnytter vi, at der højest kan være n Push-operationer og der kan ikke være flere Pop-operationer (inklusiv dem i MultiPop) end Push-operationer.
Så får vi at worst-case for alle n operationer er $O(n)$, og herved at den amortiserede cost pr. operaton er $O(n)/n = O(1)$.

1.6.4 Regnskab/Accounting Metoden

Motivation: Angiv forskellig kost til forskellige operationer uden at de behøver være korrekte. Den kost man angiver er dens amortiserede kost.
Den rigtige cost for den i 'te operation er c_i Den amortiserede cost for den i 'te operation er $\hat{c}_i$.

Hvis $\hat{c}_i > c_i$, så overcharger vi operationen og har på den måde noget "kredit" at bruge af. Hvis det omvendte gør sig gældende, så undercharger vi og bruger herved noget af den kredit vi har bygget op. Der skal ALTID gælde:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

Hvis vi nu kan få et upper bound på $\sum_{i=1}^{n} \hat{c}_i$ får vi også et upper bound på den faktiske omkostning.

Nu henholdsvis får/vælger vi følgende værdier:

Operation	Faktisk cost	Amortiseret cost
Push	1	2
Pop	1	0
MultiPop	$\min(s,k)$	0

Vi kan se på det på den måde, at ved Push betaler vi både for selve operationen med 1 samt lægger 1 kredit på elementet.
Pop bliver undercharged med 1, men denne cost betaler vi med den kredit vi har lagt på elementet. Vi ser at vi aldrig får en kredit der er negativ.

Vi ser, at for enhver sekvens af n operationer har vi:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

og herved bliver den gennemsnitlige cost ≤ 2, hvorved vi får en gennemsnitlig køretid på $O(1)$.

1.6.5 Potentialemetoden

Motivation: Minder om regnskabsmetoden bortset fra at kreditten ikke er gemt på enkelte elementer men i stedet i "banken". Mængden af kredit i banken er udtrykt ved en potentialefunktion Φ.

Hvis vi udfører n operationer på en datastruktur hvor D_i er strukturen efter den i 'te operation for $i = 1, \ldots, n$, så skriver vi $\Phi\left(D_i\right)$ som symbol for kreditten der er gemt med den nuværende struktur. Vi definerer den amortiserede cost $\hat{c}_i$ som den faktiske cost c_i plus forskellen mellem hvad der er i potentialet nu og hvad der var lige før.

$$\hat{c}_i = c_i + \Phi\left(D_i\right) - \Phi\left(D_{i-1}\right)$$

Hvis $\Phi(D_i) - \Phi(D_{i-1}) > 0$ kan man sige at vi putter kredit i banken, og hvis det er mindre tager vi kredit fra banken. Samme egenskab som for accounting method skal gælde:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Vi får summen af den amortiserede cost til at blive:

$$\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\
&= c_1 + \Phi(D_1) - \Phi(D_0) \\
&+ c_2 + \Phi(D_2) - \Phi(D_1) \\
&+ \vdots \\
&+ c_n + \Phi(D_n) - \Phi(D_{n-1}) \\
&= \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0)
\end{aligned}$$

Da vi ser at alle potentiale-led pånær det første og sidste går ud med hinanden i summen. Hvis vi sørger for at $\Phi(D_n) \geq \Phi(D_0)$ ser vi, at vi vil opfylde egenskaben fra før.

Kosten af n Increment-operationer er i worst-case $O(n)$

1.7 Søgetræer (LSM)

Log-Structured Merge Trees
Alternativ til Binær Søgetræ, brugt i praksis

Motivation: Sorterede lister L_i for $i = 0, 1, 2, \ldots$
Liste L_i indeholder 0 el. 2^i elementer
Største liste indeholder $N \leq n$ elementer
Binær søgning udføres i alle lister

Køretid af søgning: $O((logn)^2)$

Insertion(x): Hvis liste j er første tomme liste, lav en ny liste L_j med sorterede elementer $\{x\} \cup L_0 \cup L_1 \cup \ldots \cup L_{j-1}$.
Merge kan flette listerne sammen parvis

Køretid af insert: $O(n)$
Amortiseret køretid: $c * log(N)$

1.8 Prioritetskøer (Hobe)

Indeholder følgende operationer:

- Make-Heap() - laver en tom hobe.
- Insert(H,x) - indsætter element x med en eksisterende key i hobe H.
- Minimum(H) - returnerer pointer til element med min. key i hobe H.
- Extract-Min(H) - fjerner element med min. key i hobe H, returnerer pointer til elementet.
- Union(H_1, H_2) - laver en ny hobe med elementerne fra hobe H_1 og H_2. De gamle hobe bliver slettet ved union.

- Decrease-Key(H,x,k) - giver en ny key-værdi k til element x i hobe H, som er mindre el. lig den forrige key-værdi.
- Delete(H,x) - sletter element x fra hobe H.

1.8.1 Binære Hobe

Repræsenteret som binær, sorteret array/søgetræ
Perfekt balance
Knuder på nederste niveau er placeret til venstre
Værdierne er ikke-aftagende på stier fra børn til roden

Hver værdi har en nøgleværdi k, k stiger jo længere man går væk fra roden

Motivation: Reparer hoben lokalt hvis invarianten overtrædes

Langsom køretid; n Insert + n Extract-Min operationer tager $\Omega(nlogn)$ tid.

1.8.2 Fibonacci Hobe

Er flere hobe; rodknuder i en usorteret hægtet liste
Nøgleværdierne stiger når man går mod roden
Forældre kan have mere end to børn i et træ

Motivation: Bruger amortisering - hurtigere end binære hobe
Consolidate - sørger for der ikke findes to rodknuder med lige mange børn
Implementeres i $O(t(H) + D(n))$ tid, hvor t(H) er længden af rodlisten og D(n) er øvre grænse for antal børn på en knude

Køretider for Hobe:

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

1.8.3 Amortiseret Analyse af Fib Hobe

Motivation: Insert betaler for fremtidige Consolidate - omkostning er t(H)
Decrease-Key betaler for fremtidige Cut - omkostning er m(H) - antal markerede knuder

Potentialefunktionen: $\Phi(H) = t(H) + 2m(H)$

Omkostning ved Insert: $\hat{c} = c + \Phi(H') - \Phi(H) = O(1) + 1$

Omkostning ved Cascading-Cut: $\hat{c} = c + \Phi(H') - \Phi(H)$

Omkostning ved Extract-Min: $c = (t(H) + D(n))$ (faktisk omkostning), ellers $\hat{c} = c + \Phi(H') - \Phi(H)$

1.9 Dynamisk Programmering

Minder om Del og Hersk - deler problemer op i subproblemer og løser individuelt
I DP er der optimal delstruktur - delproblemer overlapper - delproblemer har samme "deldelproblemer"
To vigtige typer af DP til eksamen:

- Rod-Cutting

- LCS

DP kan huske og genbruge tidligere beregninger - memoisering

1.9.1 Rod-Cutting

Givet en rod af længde n, find den bedst mulige måde at få "lige udbytte" på ved at dele roden op - jo større sub-rode jo bedre
Relevant for eksamen

Rod-Cutting formlen er givet ved:

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}.$$

En rod kan deles op på 2^{n-1} måder

Køretid af Rod-Cutting: $O(2^n)$, kan gå ned til $\Theta(n^2)$ med DP

1.9.2 LCS

Længste fælles delsekvens (Longest Common Subsequence)
Givet to sekvenser X, Y, find den længste sekvens som er en delsekvens af både X og Y
Relevant for eksamen

LCS-formlen er givet ved:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Køretid af LCS: $O(mn)$

1.10 Grådige Algoritmer

Minder om DP, men laver "midlertidige" optimale løsninger der på nuværende tidspunkt i en algoritme ser bedst ud
Motivationen er at lave en lokal optimal løsning, som fører til en global optimal løsning

Der er to primære egenskaber i grådige algoritmer:

- **Greedy Choice Property:**
  Man samler en global optimal løsning ved at lave lokale optimale grådige valg.
  Man tager det valg som er bedst for nuværende problem uden at kigge på løsninger til delproblemer
  Opnår altid et lignende, men mindre delproblem
- **Optimal Delstruktur:**
  Hvis der findes en optimal løsning som indeholder det grådige valg, så består den optimale løsning af det grådige valg + løsningen til delproblemet som er tilbage efter det grådige valg er lavet.

1.10.1 Aktivitetsudvælgelse

Givet et sæt S med aktiviteter, en aktivitet a_i har en start- og en sluttid; s_i og f_i hhv. Løsningen er ikke optimal hvis to aktiviteters tidsforløb overlapper hinanden.
Motivationen er at finde det højeste antal ikke-overlappende aktiviteter.

Der er n delproblemer. Hvert delproblem er $O(n)$ tid, i alt $O(n^2)$ tid.

Formel for aktivitetsudvælgelse:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\{c[i,k] + c[k,j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

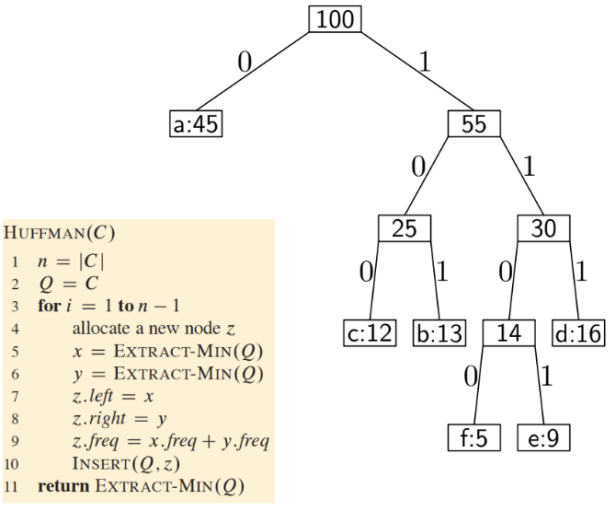
hvor S_{ij} er et set af aktiviteter der starter efter aktivitet a_i og før aktivitet a_j

1.10.2 Huffman Koder

Grådig datakompressions-algoritme, data er en sekvens af tegn
Motivation: Bruger frekvens af hvert tegn til at opbygge en optimal måde at repræsentere hvert tegn som en binær string.

Køretid: $O(nlogn)$; $O(n)$ for min-hob, $O(logn)$ for selve processen.

Eksempel på Huffman:




```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       x = EXTRACT-MIN(Q)
6       y = EXTRACT-MIN(Q)
7       z.left = x
8       z.right = y
9       z.freq = x.freq + y.freq
10      INSERT(Q, z)
11 return EXTRACT-MIN(Q)
```

Figure 7: Eksempel og Pseudokode for Huffman koder.

Vi har 6 tegn; a, b, c, d, e, f, hvert med et tal, som er deres frekvens.
Begynder med de mindste; f, e. $f = 5$ og $e = 9$; 5+9 = 14. 14 bliver forældren til f og e.
Gå til nye mindste; c, b. $c = 12$, $b = 13$. 12+13 = 25. 25 bliver forældren til c og b.
Gå til nye mindste; d. $d = 16$. Bliver lagt ved siden af forældren til f og e. 14 + 16 = 30. 30 bliver forældren til d og 14.
Læg 25 og 30 sammen; forældren af dem er 25 + 30 = 55.
Kig på a. $a = 45$, a er mindre end 55, derfor må der være en forælder til a og 55. 45 + 55 = 100, 100 bliver forældren til a og 55, træet er komplet.

Hvert led i træet har en binær værdi, alle på venstre led har 0 og alle på højre har 1. Ved at følge bladende i træet finder man en given binær værdi for et tegn.
Eks. binær værdien for f er 1, 1, 0, 0 = 1100.

1.10.3 Properties af Huffman Koder

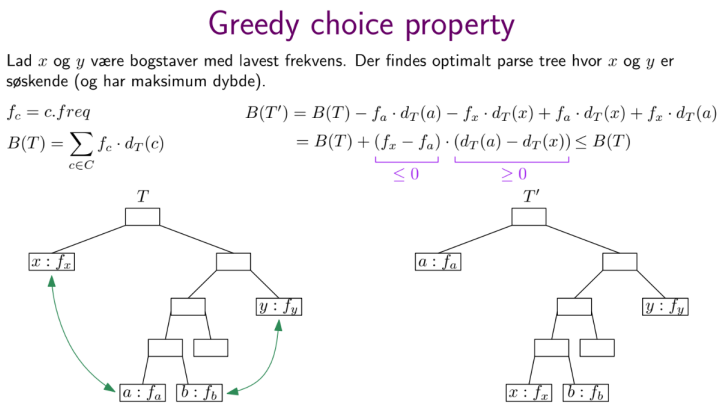


Figure 8: Greedy Choice Property af Huffman kode.

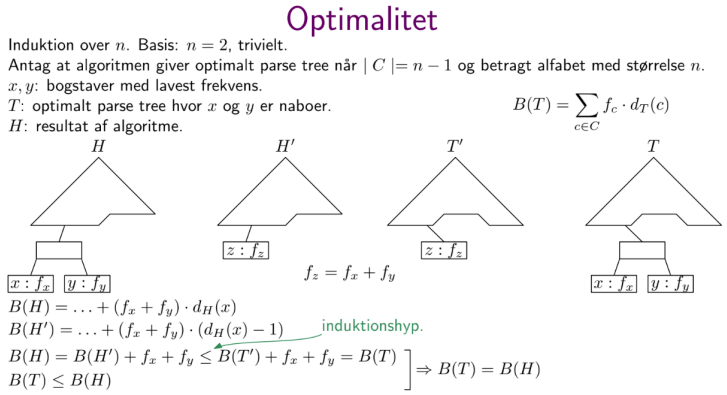


Figure 9: Optimal delstruktur af Huffman kode.

1.11 Binære Søgetræer

1.11.1 Binær Søgetræ

Rekursiv natur; in-order-traversel (venstre node → forældrenode → højre node) Hver node i venstre barn er mindre end forældren
Hver node i højre barn er større end forældren

Søgning:
Begynd ved træets rod (toppen)
Hvis værdien < nuværende node, gå venstre; hvis værdien > nuværende node, gå højre

Køretid: $O(n)$ el. $O(h)$ hvor h er højden af træet

Insertion:
Nye noder indsættes altid i bunden af træet
De samme regler gælder; hvis ny node er mindre, så er den venstre, og hvis større så er den til højre

Køretid: $O(n)$ el. $O(h)$ hvor h er højden af træet

Deletion:
Når en node slettes, så skal træet beholde sin order. Der er tre måder man kan slette en node på:

Hvis noden er et barn uden at være forælder, kan den blot slettes

Hvis noden er forælder til et barn, rykkes barnet ind i positionen hvor den slettede node var før

Hvis noden er forælder til to børn, findes in-order-predecessor (IOP) - den største node i nodens venstre sub-træ.
In-order-predecessor er altid et barn, findes ved at starte i venstre sub-træs node og gå til højre.
Noden der slettes bliver udskiftet med IOP-noden og fjerne den, siden den nu er et barn

Køretid: $O(n)$ el. $O(h)$ hvor h er højden af træet

Samlet køretid: $O(n)$ el. $O(h)$ i alle operationer, $O(logn)$ i best case hvis det er et AVL-træ (højde er det samme som køretid)

1.11.2 Rød-Sort Søgetræ

Binært søgetræ med farvekode; enten rød eller sort
Roden er altid sort
Hvert blad er sort (NIL)
Hvis en forælder er rød, er begge børn sorte

Køretid: $O(logn)$ for alle operationer

Røde-sorte træer er selvbalancerende
Bruges ofte pga. bedre køretid og effektivitet

1.12 Disjunkte Mængder

Sæt er disjunkte når de ikke deler elementer mellem hinanden.

Der er tre vigtige operationer inden for disjunkte mængder:

- Make-Set(x) - et nyt sæt bliver lavet der indeholder en enkelt værdi (x)
- Union(x,y) - uniter to disjunkte sæt der indeholder x og y.
- Find-Set(x) - returnerer en pointer til sættet der indeholder værdien x.

En fjerde operation er sub-operation til Union; Link(x,y) - øger røddernes rank med 1 hvis de er identiske.

1.12.1 Køretider af Disjunkte Mængder

Operationerne har forskellig køretid baseret på hvilken type repræsentation der er tale om:

Linked List Representation:
Både Make- og Find-Set er $O(1)$ tid.
Union tager $O(n)$ tid.
Med amortisering tager det $\Theta(n)$ tid.
I alt: $O(m + nlogn)$ hvor n er antal Make-kald.

Disjoint-Set Forest:
Make-Set er $O(1)$ tid for et kald, flere kald er $O(n)$ tid.
Find-Set kan tage $O(n)$ tid uden optimiseringer.
Union er $O(n)$ tid uden optimiseringer.
Link er $O(1)$ tid.

To forbedringer kan laves i form af Union by Rank/Size og Data Compression:

- Union by Rank/Size - kigger på hhv. højde og størrelse af træet under union-fasen.
- Data Compression - komprimerer træets højde ved at indføre caching i Find.

Køretider efter Optimiseringer:
Make-Set er stadig $O(1)$ tid.
Find-Set er $O(logn)$ tid.
Union er $O(logn)$ uden compression, $\approx O(1)$ tid med compression.

Med amortisering er den samlede tid $O(\alpha(n))$, hvor $\alpha(n)$ er en invers Ackermann funktion - en meget langsom voksende funktion.

1.13 Mindste Udspændte Træer

To former for MST til eksamen:
Kruskal (standard) algoritme
Prim algoritme

Grådige algoritmer - udfører valg der er bedst på nuværende tidspunkt indtil et færdigt træ er lavet

1.13.1 Kruskal Algoritme

Mest almindelige algoritme til MST
Metode:

- Sorter alle kanter og vægte i en ikke-aftagende rækkefølge
- Vælg den mindste kant (vægt-mæssig) og check hvis den bliver integreret i det dannede træ.
- Hvis den gør, inkluder kanten, ellers udeluk den.
- Repeter for hver kant i træet indtil der er (V-1) kanter tilbage.

Køretid: $O(E * logE) or O(E * logV)$, hvor at:
Sortering af kanter tager $O(E * logE)$ tid
Iteration (find og union) tager i worst case $O(logV)$ tid.
E er antal kanter, E kan højest være $O(V^2)$, derfor er $O(logV)$ og $O(logE)$ det samme.

Plads: $O(V + E)$

1.13.2 Prim Algoritme

Motivation: Begynder med et tomt MST og vil beholde two sæt af knuder
Metode:

- Start fra en tilfældig knude i træet (som regel 1)
- Find kanter der forbinder knuder med startknuden
- Find minimum vægt mellem kanter
- Tilføj kanterne til MST
- Gentag indtil træet er fuldendt

Køretid: $O(E * logE)$, hvor E er antal kanter
Plads: $O(V^2)$ hvor V er antal knuder

1.13.3 Sikre Kanter/Snit

Betragt et subset A af et MST med et sæt kanter

Snit af $(S, V - S)$ skal respektere kantmængden A; dvs. at enten er ingen eller alle kanter i snittet inkluderet i A

Sikre kanter skal være en gyldig minimum vægt i et MST

1.14 Korteste Veje

1.14.1 Bellman-Ford/Single-Source Korteste Veje

Bestemmer korteste veje mellem knuder i en graf
Korteste veje opdateres under gennemløb af algoritmen - den første løsning er ikke altid den bedste
Bellman-Ford følger dynamisk programmerings strategi, siden vi prøver alle mulige løsninger og vælge de bedste løsninger ud fra det
Virker på alle grafer (både positive og negative kanter)

En graf med N knuder skal "relaxes" (opdateres) $N - 1$ gange for at få en fuld SSKV løsning

Køretid når grafen er forbundet:
Best case: $O(E)$
Avg. case: $O(V * E)$
Worst case: $O(V * E)$, hvor E er antal kanter og V er antal knuder

Plads: $O(V)$ hvor V er antal knuder

ALLE veje i en løsning er de korteste mulige løsninger

1.14.2 Dijkstra

Løsning til single-source korteste veje
Virker KUN på grafer med positive kanter (ingen negative kanter)

Køretid: $O(V^2)$
Plads: $O(V)$ hvor V er antal knuder

Køretid kan reduceres til $O(E * logV)$ via. binary heap (Adjacency list)

Hvis der gives en vægtet graf, skal vi finde en kant fra et startknude til alle knuderne. Vi kan vælge et hvilket som helst af knuderne som et startsknude. Da man skal finde den korteste vej, så er det et minimeringsproblem, hvilket betyder, at det er et optimeringsproblem, der kan løses ved hjælp af en Greedy algoritme.

2 Multiple-Choice

Hvis du er i tvivl, lad vær med at gætte. Der er altid chance for negative point.

Udelukkelsesmetoden er super god, siden man normalt hurtigt kan udelukke et eller flere svar.

I ikke-typeopgaverne (teori/properties af algoritmer), de korrekte svar ALTID skal gælde for en given algoritme eller situation.
F.eks. Dijkstra kan bruges til at løse single source shortest path problemer men KUN hvis grafen er negativ. Derfor er det ikke en property shortest path-løsninger altid har.

2.1 Merge Sort

Eksempel 1 fra Eksamen 2023:

Givet et kald af MergeSort på 42 elementer ($r - p + 1 = 42$), hvad er det største antal kald på en gang?

Vi dividerer 42 med 2 rekursivt indtil vi når ned på 1.
I dette tilfælde skal der divideres med ulige tal (21), her **runder vi op**, siden der er ubalance i sorteringen.

42 (det første kald før opdeling tæller med)
$42/2 = 21$
$21/2 = 11$ (rundet op)
$11/2 = 6$ (rundet op)
$6/2 = 3$
$3/2 = 2$ (rundet op
$2/2 = 1$

Dvs. at der bliver maks. udført **7** kald på en gang.

Eksempel 2 fra PrøveEksamen 2023:

Givet et kald af MergeSort på 21 elementer, hvor mange kald bliver der lavet i alt?

Her kan vi bruge formlen $2n - 1$, hvor n er antal elementer. $n = 21$:

$2 * 21 - 1 = 42 - 1 = 41$

Dvs. at der bliver udført 41 kald i alt.

Eksempel 3 fra PrøveEksamen 2023:

Givet MergeSort med 8 input: [42,33,36,43,11,38,22,66], hvor mange gange bliver element 42 sammenlignet?

Vi kan fokusere på de sorterede lister hvor 42 indgår. Der laves 1 sammenligning for at lave den sorterede liste med 2 elementer, 2 sammenligninger for at lave den sorterede liste med 4 elementer, og yderligere 2 sammenligninger for at lave den endelige sorterede liste. Dvs. totalt 5 sammenligninger.

Opsummering:
Divider rekursivt for at finde maks. kald på en gang.
Husk at runde op når man dividerer rekursivt.
Brug formlen $2n - 1$ for at finde antal total kald.
Betragt kun listerne hvor element x indgår for at tælle dets antal sammenligninger.
Brug implementationen og detaljerne om MergeSort fra kapitel 2.3 i bogen.

2.2 Køretider

Eksempel fra Eksamen 2023:

Givet pseudokoden:

```
Loops(n)
r = 0
```

```
for i = 1 to n/2
    for j = n/2 - i to n
        r = r + j
return r
```

Det antages at $n/2$ er et heltal, så den yderste for-løkke har $n/2$ iterationer. Hvad er køretiden?
1. $T(n) = O(n)$.
2. $T(n) = O(n \lg n)$.
3. $T(n) = O\left(n^2\right)$.
4. $T(n) = \Omega(n)$.
5. $T(n) = \Omega(n \lg n)$.
6. $T(n) = \Omega\left(n^2\right)$.

Som regel er udsagnene ikke afhængige - f.eks. hvis $f(n) = O(n)$ gælder, så gælder $f(n) = O(nlogn)$ også.
Average case bliver udregnet her; dvs. Store-Θ-notation. Ud fra Θ-løsningen, kan der så udpeges Store-O og Store-Omega-køretider.

For at angive køretiden, kigger vi på antallet af konditioner hvor n indgår. Her indgår n i:
Fra $1 \rightarrow n/2 = n/2$
Fra $j = n/2 - 1 \rightarrow n = n/2 * n$
$n/2 + n/2 = n$
$n * n = n^2$

Dvs. at køretiden er $\Theta(n^2)$. Da vi nu ved at average-case er $\Theta(n^2)$, SKAL Store-O (worst-case) være større el. lig average case.

Ud fra svarmulighederne kan vi se at den eneste som opfylder dette kriterie er $O(n^2)$, siden at $\Theta(n^2) \leq O(n^2)$. Hvis vi så kigger på nedre grænse, så er det blot alle værdier som opfylder $\Theta(n^2) \geq \Omega()$. Det kan vi se, at alle køretider for Ω-notationen gør.

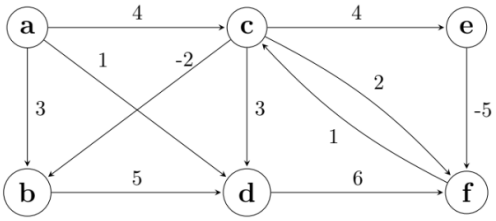
Dvs. at løsninger 3,4,5 og 6 er rigtige.

Opsummering:
Vi løser for average-case; Store-Θ-notationen.
Alle worst-case løsninger; Store-O skal opfylde kriteriet $\Theta() \leq O()$
Alle bedst-case løsninger; Store-Ω skal opfylde kriteriet $\Theta() \geq \Omega()$
Kig på løkkerne og antallet af gange n optræder i en kapacitet.
Læg/Gang antallet sammen til sidst og opnå svarene.
Husk at referere til de 10 regler for at udregne køretider (konstanter ignoreres, det største led tager prioritet, osv.)

2.3 Korteste Veje

Eksempel fra Eksamen 2023:

Givet et korteste-veje problem med startknude a, hvilke kanter er med i korteste-veje træet?



Her kigger vi på to ting; hvad vægterne er på hver kant og hvilken vej pilene går mellem knuderne. Bemærk at der optræder negative kanter i dette eksempel, derfor skal Bellman-Ford algoritmen bruges.

Vi starter i knude a og ser at den går til knuder b, c, d. Ingen kanter går til a.
$a, b = 3, a, c = 4, a, d = 1$.
Vi kigger så rekursivt på hver knude efter a og finder løsningerne. Dette kan som regel gøres ved blot at aflæse grafen og så skrive vægtene ned for at skabe bedre overblik.
Det skal dog huskes, at det er vægten fra startknude a til en given knude som skal udregnes. F.eks. er vægten fra f til c kun 1, men fordi at vi skal gå gennem knuderne c, e, f for at nå derned, bliver vægten mere end blot vægten fra a til c.

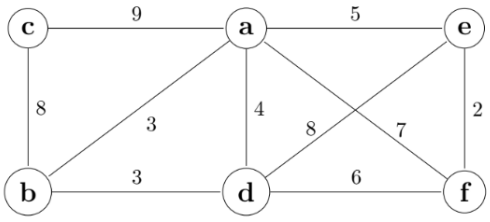
Efter at have gjort dette, kan vi se at kanterne der skal inkluderes i korteste-veje træet er:
$\{a, c\}, \{a, d\}, \{c, b\}, \{c, e\}, \{e, f\}$.

Opsummering:
Start i den givede startknude.
Kig på vægtene af kanterne og hvor hver knude går hen til
Husk at det ikke er den enkelte vægt af kanten mellem knuder, men summen af kanterne fra startknuden til den givede knude.
Hvis det er svært at holde overblik over, så skriv det ned og opdater løsningerne dynamisk.

2.4 Prim Algoritmen

Eksempel fra Eksamen 2023:

Givet en vægtet, uorienteret graf:



Hvis Prim's algoritme køres på denne graf med start i knude a, i hvilken rækkefølge bliver knuderne inkluderet i det mindste udspændende træ? (Med andre ord, i hvilken rækkefølge bliver de taget ud af prioritetskøen?)
Vælg præcis ét svar.

1. a, b, d, e, f, c
2. a, b, e, d, f, c
3. a, b, d, c, f, e
4. a, b, e, f, d, c

Vi starter i knude a, og kigger på vægten mellem naboknuderne. Vi ser at vægten er 3 til b, 9 til c, 4 til d, 5 til e og 7 til f.
Den første knude der inkluderes (udover a) er b, siden den har den laveste vægt.
Den næste knude der tilføjes er d, siden den har den laveste vægt på 4 fra knude a. Dermed kan vi også eliminere svar 2 og 4.
Den næste knude der tilføjes er e, siden den har den laveste vægt på 5 fra knude a. Vi kan så også eliminere svar 3.
De sidste knuder der tilføjes er f og c med vægte på hhv. 7 og 9.

Dvs. svaret er nr. 1.

Dette er et trivielt (lmao) eksempel, husk at der potentielt kan være mindre vægte hvis kanter bliver lagt sammen. (F.eks. hvis vægten var 7 til c fra a, så ville b til d være en bedre løsning, da $3 + 3 = 6 < 7$).

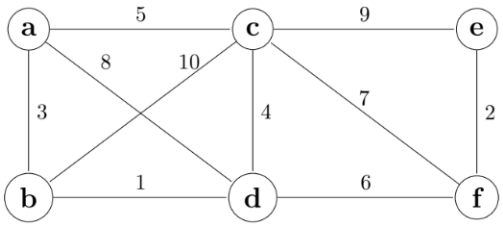
Opsummering: Det er nemt at eliminere svarmuligheder hurtigt.
Knuderne skal tilføjes i rækkefølge fra mindste kant til største kant.

Husk at lægge kanter sammen i tilfælde hvor det er relevant.
Hvis det er svært at holde overskud, så skriv det ned og opdater løsningen over tid.

2.5 Mindst Udspændende Træer

Eksempel fra Eksamen 2023:

Givet et MST, hvilke kanter er med i løsningen?



Da der ikke er angivet nogen startknude, vælger vi enten at starte i knuden a, da det er nemmest at danne overblik (og den er tilfældigvis i top-venstre hjørne af grafen), eller i den mindste kant mellem to knuder (som i dette tilfælde er b,d med en kant på 1).
Vi kigger igen på vægtene af kanterne gennem grafen, og finder de mindste vægte som så danner løsningen. Løsningerne skal alle være naboer til hinanden, og der kigges ikke på summen af tidligere kanter for at bestemme løsningen. Det ses at fra a til b er kanten 3, derfor er a,b en løsning.
Fra a til d er kanten 8, men fra b til d er kanten blot 1. Derfor er b,d også en løsning.
Fra a til c er kanten 5, fra d til c er kanten 4. Da vi ikke kigger på summen af tidligere kanter, så er c,d også en løsning.
Dette gentages for alle knuder i træet indtil løsningen er opnået.

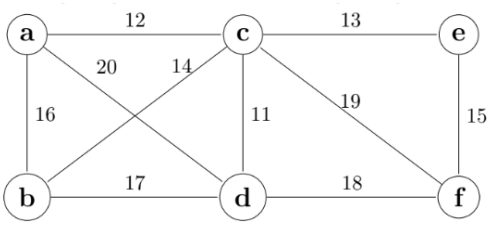
Efter processen er færdig, har vi fundet frem til løsningen af MST'et: $\{a, b\}, \{b, d\}, \{c, d\}, \{d, f\}, \{e, f\}$.

Opsummering:
Start enten i en selvangivet startknude eller den mindste kant.
Alle løsninger skal være naboer til hinanden.
Summen af tidligere kanter tages ikke i betragtning, kun den individuelle kant mellem to knuder.
Hvis det er svært at holde overskud, så skriv det ned og opdater løsningen over tid.

2.6 Kanter og Snit af MST

Eksempel fra Eksamen 2023:

Betragt følgende uorienterede, vægtede graf med knudemængde $V = \{a, b, c, d, e, f\}$:



Betragt kantmængden $A = \{\{a, c\}, \{c, d\}\}$, som er en delmængde af et minimalt udspændende træ (eng. minimum spanning tree). Vi er interesserede i snit (eng. cuts), sikre kanter (eng. safe edges) for A, og snit der respekterer (eng. respects) en kantmængde A. Hvilke af følgende udsagn er sande? Vælg ét eller flere korrekte svar.

1. Lad $S_1 = \{a, b, c, d\}$. Snittet $(S_1, V - S_1)$ respekterer kantmængden A.
2. Lad $S_2 = \{a, c, e\}$. Snittet $(S_2, V - S_2)$ respekterer kantmængden A.

3. Lad $S_3 = \{b, e, f\}$. Snittet $(S_3, V - S_3)$ respekterer kantmængden A.
4. $\{d, f\}$ er en sikker kant for A.
5. $\{c, e\}$ er en sikker kant for A.
6. $\{a, d\}$ er en sikker kant for A.

Vi ved, at kantmængden indeholder knuderne a,c,d. (Vigtigt!).
Egenskaberne for et snit som respekterer kantmængden er, at enten alle eller ingen kanter i snittet skal inkluderes i snittet før at det respekterer kantmængden.

Hvis vi kigger på svar 1, så er a, c og d med i snittet (b er overfladisk, da den ikke indgår i kantmængden). Siden alle kanter i mængden er med i snittet, så bliver det respekteret. Dermed er svar 1 korrekt.
I svar 2 er a og c med sammen med e (overfladisk). Da alle knuder i A ikke er med, så respekterer det ikke kantmængden, og svar 2 er ikke korrekt.
I svar 3 er b, e og f med. Da ingen af knuderne inkluderet i A er med i snittet, så respekterer det kantmængden. Svar 3 er derfor korrekt.

Hvis vi kigger på sikre kante, så er egenskaben for en sikker kant, at den skal være en gyldig minimum vægt i MST'et.
Vi betragter kanten d,f; vægten er 18. Da der er en mindre vægt til f (e,f), så er det ikke en sikker kant.
Kanten c,e har en vægt på 13. Den eneste anden vægt er på 15 (e,f), derfor er c,e en sikker kant, da den har en minimum vægt.
Kanten a,d har en vægt på 20, alle andre kanter til d er på mindre end 20, derfor er a,d ikke en sikker kant.

Vi kan nu konkludere at svar 1, 3 og 5 er korrekte.

Opsummering:
Betragt kun kanterne inkluderet i kantmængden når der udføres snit.
Hvis et snit skal respektere kantmængden, skal enten alle eller ingen kanter i snittet være en del af mængden.
En sikker kant skal være en gyldig minimum vægt i MST'et.

2.7 Amortiseret Analyse

Eksempel fra Eksamen 2023
I denne opgave betragter vi potentialfunktionen der bruges til at analysere en binær tæller i CLRS sektion 17.3. Lad $\Phi(D_i)$ betegne potentialfunktionens værdi efter i INCREMENT operationer. Hvilke egenskaber har $\Phi(D_i)$? (Læg mærke til at udsagnene ikke er uafhængige — hvis der gælder $f(i) = \Omega(i)$ så gælder også $f(i) = \Omega(\lg i)$, osv.) Vælg ét eller flere korrekte svar.

1. $\Phi(D_i) = \Omega(\lg i)$.
2. $\Phi(D_i) = \Omega(i)$.
3. $\Phi(D_i) = O(1)$.
4. $\Phi(D_i) = O(\lg i)$.
5. $\Phi(D_i) = O(i)$.

Siden at den amortiserede værdi per. operationer er $O(1)$, (fra CLRS) vil potentialfunktionens værdi efter i Increment operationer være $\Phi(D_i) = O(i)$. Vi bruger opgavens antagelse; udsagnene ikke er uafhængige; hvis der gælder $f(i) = \Omega(i)$ så gælder også $f(i) = \Omega(lg\ i)$, osv. Derfor vil det også gælde at $\underline{\Phi(D_i) = O(i) = O(lg\ i)}$.

Opsummering:
Referer til køretider af amortiserede operationer fra CLRS.
Brug antagelsen givet i opgaven.
Husk at svaret skal gælde for alle situationer.

2.8 Rekursionsligninger

Master Theorem for Dividing Functions:
Given a recurrence of the form

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

where

- $a \geq 1$
- $b > 1$
- $f(n) = \Theta(n^k \log^p n)$

We have the following cases:

Case 1: If $\log_b a > k$ then $T(n) = \Theta\left(n^{\log_b a}\right)$

Case 2: If $\log_b a = k$

- if $p > -1$ then $T(n) = \Theta\left(n^k \log^{p+1} n\right)$
- if $p = -1$ then $T(n) = \Theta\left(n^k \log\log n\right)$
- if $p < -1$ then $T(n) = \Theta\left(n^k\right)$

Case 3: If $\log_b a < k$

- if $p \geq 0$ then $T(n) = \Theta\left(n^k \log^p n\right)$
- if $p < 0$ then $T(n) = \Theta\left(n^k\right)$

Master Theorem for Decreasing Functions
Given a recurrence of the form

$$T(n) = aT(n-b) + f(n)$$

where

- $a > 0$
- $b > 0$
- $f(n) = O(n^k)$ where $k \geq 0$

We have the following cases:

Case 1: If $a < 1$

- $T(n) = \Theta(n^k)$
- $T(n) = \Theta(f(n))$

Case 2: If $a = 1$

- $T(n) = \Theta(n^{k+1})$
- $T(n) = \Theta(n \cdot f(n))$

Case 3: If $a > 1$

- $T(n) = \Theta(n^k \cdot a^{n/b})$
- $T(n) = \Theta(f(n) \cdot a^{n/b})$

Svar til Exam 2023

1. $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$.
   We have $log_2 2 = 1 < k = 2$ and we also have $p = 0$, for this we have **case 3**. $\Theta\left(n^k \log^p n\right) = \Theta\left(n^2 \log^0 n\right) = \Theta\left(n^2\right)$

2. $T(n) = 2T(\lfloor n/2 \rfloor) + 3n$.
   We have $log_2 2 = 1 = k = 1$ and we also have $p = 0$, for this we have **case 2**. $\Theta\left(n^k \log^{p+1} n\right) = \Theta\left(n^1 \log^{0+1} n\right) = \Theta\left(n \log n\right)$

3. $T(n) = 2T(\lfloor n/3 \rfloor) + 2n$.
   We have $log_3 2 = 0. < k = 1$ and we also have $p = 0$, for this we have **case 3**. $\Theta\left(n^k \log^p n\right) = \Theta\left(n^1 \log^0 n\right) = \Theta\left(n\right)$

4. $T(n) = 3T(\lfloor n/2 \rfloor) + n \lg n$.
   We have $log_2 3 = 1.58. > k = 1$ and we also have $p = 1$, for this we have **case 1**. $\Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 3}\right) = \Theta\left(n^{1.58}\right)$

5. $T(n) = 3T(\lfloor n/3 \rfloor) + n$.
   We have $log_3 3 = 1 = k = 1$ and we also have $p = 0$, for this we have **case 2**. $\Theta\left(n^k \log^{p+1} n\right) = \Theta\left(n^1 \log^{0+1} n\right) = \Theta\left(n \log n\right)$

6. $T(n) = T(n-1) + \lg n$. Master Theorem cannot be used here, Substituion Method must be used instead.

Opsummering:
Brug eksempler og formlerne til at udregne med Master metoderne.
Ikke gæt på substitutionsmetoden, medmindre du er skrap i den.

2.9 Del og Hersk

Eksempel fra Eksamen 2023:

Antag at du har en rekursiv algoritme A, og lad n betegne størrelsen på algoritmens input X. Hvis $n = 1$ beregnes resultatet $A(X)$ i konstant tid. For $n > 1$ bruger algoritmen først $O(n \lg n)$ tid på at beregne fire inputs X_1, X_2, X_3, X_4, hver af størrelse $\lfloor n/2 \rfloor$, og $A\left(X_1\right), A\left(X_2\right), A\left(X_3\right), A\left(X_4\right)$ beregnes rekursivt. Endelig kombineres de rekursive svar til et output $A(X)$ i tid $O\left(n^2\right)$.

Hvilken af disse rekursionsligninger beskriver køretiden $T(n)$ af A på input X ? Vælg præcis ét svar.
1. $T(n) = 4T(\lfloor n/2 \rfloor) + O(n \lg n)$.
2. $T(n) = 2T(\lfloor n/4 \rfloor) + O(n \lg n)$.
3. $T(n) = 4T(\lfloor n/2 \rfloor) + O\left(n^2\right)$.
4. $T(n) = 2T(\lfloor n/4 \rfloor) + O\left(n^2\right)$.

Vi ved at der er 4 inputs af størrelse $n/2$, og at der bliver brugt $O(nlogn)$ tid på at beregne inputsne, og derefter bliver de sat sammen i $O(n^2)$ tid.

Fra at analysere ligningerne, kan vi se at 2 af dem, (1 og 3), indgår størrelsen $n/2$. Dermed kan vi udelukke svar 2 og 4. Vi kan også se at i svar 1 og 3 bliver der udregnet 4 delproblemer frem for 2. Til sidst kigger vi på køretiderne. Vi ved at $O(nlogn) \leq O(n^2)$, derfor må køretiden være $O(n^2)$ siden det er det største led.

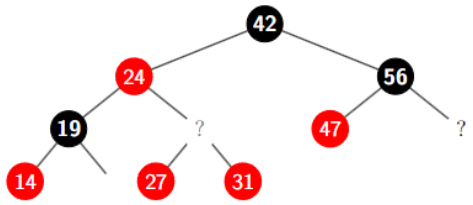
Dvs. at ligning nr. 3 beskriver køretiden.

Opsummering:
Put fokus på antal inputs/delproblemer, størrelsen af delproblemerne og den største nævnte køretid.
aT angiver antal delproblemer a, n/b angiver størrelsen af delproblemerne og køretiden er den største individuelle køretid af en operation.

2.10 Binære/Rød-Sorte Søgetræer

Eksempler fra Eksamen 2023:

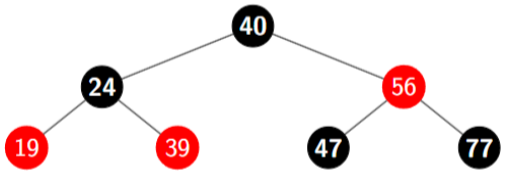
Betragt et sort-rødt søgetræ: Hvilke knuder kan indsættes på felterne markeret med ?.



Givet egenskaberne af et rødt-sort søgetræ, ved vi at:
Børn til en forælder skal altid dele samme farve
Barnet til venstre skal være mindre end forældren, og barnet til højre skal være større end forældren.

Ud fra dette, kan vi se at på den venstre gren skal knuden være sort og have en værdi $27 < x < 31$.
På den højre gren skal knuden være rød og have en værdi $47 < 56 < x$.

Betragt et sort-rødt søgetræ: Hvorhenne kan knuden 42 indsættes? Funktionen RB-INSERT bruges.



Vi kan se at knuden skal være sort på venstre side eller rød på højre side. Knuder i rød-sorte søgetræer indsættes på samme måde som i binære søgetræer, dvs. de bliver altid sat ind i **bunden** for at holde styr på træet. Derfor så ved vi, at knuden skal være i bunden.

Ud fra denne information ved vi så at knuden kun kan blive indsat i bunden og bliver enten et sort barn til 39 eller rødt barn til 47.

Opsummering:
Næsten altid trivielt (haha)
Husk egenskaberne for insertion
Alle operationer i rød-sorte søgetræer kører i $O(logn)$ tid.

2.11 Disjunkte Mængder

Eksempel fra Eksamen 2023:

Vi betragter trærepræsentationer af disjunkte mængder (eng. disjoint sets), som beskrevet i CLRS sektion 21.3, ved brug af heuristikkerne union by rank og path compression. Antag at MAKE-SET er kørt n gange og lad k være en variabel. Hvilke af følgende udsagn er altid sande? Vælg ét eller flere korrekte svar.

1. Tiden for et kald til UniON(x, y) er $O(\lg n)$.
2. k kald til FIND-SET(42) lige efter hinanden tager tid $O(k + \lg n)$.
3. Kaldene Make-Set $(x_1), \ldots,$ MaKe-Set (x_k) tager total tid $O(k)$.
4. En LinK(x, y) operation øger den maksimale værdi af rank i træerne for x og y med 1 .
5. I værste fald tager en FIND-SET operation $\Omega(n)$ tid.
6. UnION(x, x) tager konstant tid.

Her får vi oplyst, at algoritmetypen er "disjoint set forests" fra bogen, derfor behøver vi kun at fokusere på den type og dens køretider.
Vi ved at træets højde er $O(logn)$, derfor så er køretiden af både Find og Union også $O(logn)$, derfor er svar 1 korrekt.
Det første kald til Find vil altid være $O(logn)$ tid, men siden at trin i Find-algoritmen bliver skippet hvis der bliver kaldt på den samme flere gange; dvs. tiden bliver nu $O(k)$. Vi kan så sige at svar 2 er korrekt, og svar 5 er forkert, siden worst-case tiden er $O(logn)$.
Vi ved at 1 kald til Make tager $O(1)$ tid og at n antal kald til Make tager $O(n)$ tid; derfor er svar 3 også korrekt, siden at Make bliver kaldt for k antal elementer i træet.
Et kald til Link-operationen øger kun knudernes rank hvis de er de samme rank, derfor er svar 4 forkert.
I implementeringen er Find kaldt 2 gange under Union, og vi ved fra før at køretiden for Find er $O(logn)$ første gang og $O(k)$ anden gang. Derfor er svar 6 forkert.

Vi kan så konkludere, at svar 1,2 og 3 er korrekte.

Opsummering:
Fokuser kun på den datatype der er givet i opgaven.
Kig på pseudokode-implementationerne i bogen og brug slides til at finde køretiderne.
Husk at de korrekte svar ALTID skal gælde.

2.12 Invarianter

Eksempel fra Eksamen 2023:

Betragt følgende funktion, i pseudo-kode notationen fra CLRS, hvor input A er en tabel af heltal og n er længden af A.

```
Second-Smallest(A, n)
    m = A[1]
    r = infinity
    for i = 2 to n
        if A[i] < m
            r = m
            m = A[i]
        elseif A[i] < r
            r = A[i]
    return r
```

Hvilke udsagn gyldige løkke-invarianter ved starten af hver iteration i for-løkken? Vælg ét eller flere korrekte svar. 1. $m = \min(A[1], \ldots, A[i-1])$.
2. $m = \min(A[1], \ldots, A[n])$.
3. $r \geq m$.
4. $m \geq 0$.
5. Mindst 1 element i $A[1], \ldots, A[i-1]$ er mindre end eller lig med r.
6. Højst 2 elementer i $A[1], \ldots, A[i-1]$ er mindre end eller lig med r.

Her kigger vi på konditionerne af løkkerne.
Vi kan med det samme se, at invariant 3 gælder, da m aldrig sættes til at være 0, r sættes lig uendelig, og invarianten siger at r skal være $\geq$ m, som den er. Vi kan så også udelukke svar 4.
Vi kan så se, at m bliver initialiseret som $A[1]$ og i til 2. i vil så være base case (dvs. invarianterne skal gælde for i = 2).
Det kan ses, at min. i tabellen A skal være = 1, ellers gælder $m = A[1]$ ikke. Derfor kan svar 2 udelukkes og svar 1 gælder. n kan også være mindre end 2, som heller ikke overholder invarianten.
Til sidst er der ingen begrænsning på hvor mange elementer kan have samme værdi som r, derfor gælder svar 6 ikke, men svar 5 gælder.

Dvs. at svar 1, 3 og 5 er korrekte.

Opsummering:
Håndkør koden i flere iterationer, og brug en base case hvis i tvivl.
Kig godt på løkkerne og deres konditioner for invarianter.

2.13 Valg af Datastruktur

Eksempel fra Eksamen 2023:

Antag at vi gerne vil skabe en datastruktur VENTETID, der indeholder en mængde af n afgangstidspunkter (ugedag og klokkeslet, fx for en bus). For enkelheds skyld skal der blot holdes styr på én rute og ét stoppested, så den eneste information er afgangstidspunkter. Datastrukturen skal understøtte to operationer: INSERT (x), der indsætter et nyt afgangstidspunkt, og HvORLÆNGE (x) der returnerer antal minutter fra tidspunkt x til den næste afgang.

Hvad er det mest passende valg af datastruktur til løsning af dette problem blandt nedenstående, og hvilken worst-case (amortiseret) køretid for HvORLÆNGE (x) bliver resultatet af dette valg? Udtryk svaret som funktion af n. Vælg præcis ét svar.

1. Disjunkte mængder (eng. disjoint sets), tid $O(1)$.
2. Disjunkte mængder (eng. disjoint sets), tid $O(\lg n)$.
3. Fibonacci hob (eng. Fibonacci heap), tid $O(\lg n)$.
4. Fibonacci hob (eng. Fibonacci heap), tid $O(1)$.
5. Rød-sort binært søgetræ (eng. red-black binary search tree), tid $O(\lg n)$.
6. Rød-sort binært søgetræ (eng. red-black binary search tree), tid $O(1)$.

Her bliver der kigget på operationerne givet i intro til chapter 10 i bogen. Vi har med dynamic set datastrukturer at gøre.

Vi får givet to operationer: INSERT og HVORLÆNGE. INSERT er præcis den samme fra bogen, HVORLÆNGE ligner mest en FIND-operation.
Vi ved at køretiden for rød-sort træ altid er $O(logn)$, så svar 6 kan hurtigt elimineres.
Disjunkte sæt kan elimineres (svar 1 og 2), siden at de ikke sorterer elementer, men kun uniter dem, og er dermed upraktiske til denne type databehandling.
Vi ved, at i Fib hobe, så tager det kun $O(logn)$ tid at fjerne og slette elementer, som vi ikke skal gøre. Derfor kan vi udelukke svar 3.
Så har vi svar 4 og 5 tilbage. Vi ved at behøver at kunne indsætte og returnere et sammenhængende element, samt potentielt slette elementer hvis tiden skal fjernes senere. Dette er gjort nemt ved rød-sorte træer, da der altid nemt kan skabes orden i træet igen.

Derfor vil det korrekte svar være svar 5.

Opsummering:
Håb på det bedste.
Husk køretiderne for operationer i de angivede datastrukturer, samt egenskaberne for de givede datastrukturer.
Brug oversigten fra bogen til at finde ligheder blandt operationerne.

2.14 DP

Eksempel fra Eksamen 2023:

En stigende delsekvens af en tabel A, der indeholder heltal $A[1], \ldots, A[n]$, er en delmængde af indekser $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ således at $A[i_1] < A[i_2] < \cdots < A[i_k]$. Længden af delsekvensen er antal indekser, k. En ikke-aftagende delsekvens defineres på samme måde bortset fra at kravet er $A[i_1] \leq A[i_2] \leq \cdots \leq A[i_k]$. Betragt følgende funktion, i pseudo-kode notationen fra CLRS, hvor input A er en tabel af heltal.

```
MaximumMystery(A, n)
    let r[1 . . . n] be a new array
    r[1] = 1
```

```
for j = 2 to n
    q = 1
    for i = 1 to j - 1
        if not A[j] < A[i]
            q = max(q, r[i] + 1)
    r[j] = q
return r[n]
```

Hvad beregner proceduren MAXIMUMMYSTERY? Vælg præcis ét svar.

1. Indekserne i en længste stigende delsekvens.
2. Indekserne i en længste ikke-aftagende delsekvens.
3. Længden af en længste stigende delsekvens.
4. Længden af en længste ikke-aftagende delsekvens.

Det antages, at længden af tabellen er givet ved input n, og at der kun bliver returneret en værdi, $r[n]$, som er længden. Derfor kan vi eliminere svar 1 og 2, siden de vil returnere alle indekser.
Vi ved fra egenskaberne, at en stigende delsekvens altid skal have det næste element være større end det forrige, mens en ikke-aftagende kun skal være større el. lig det forrige element. Vi kan se at løkken gælder for hvis ikke element j er direkte mindre end element i. Derfor så må det være en ikke-aftagende delsekvens.

Dvs. at svar 4 er korrekt.

Opsummering:
Kig på løkkernes conditions.
Som regel kan svar elimineres hurtigt, dette gør processen nemmere.
Håb på det bedste.

3 Skriftlige Opgaver

Ingen minuspoint så giv kvalificerede bud anyway (kan give partial point)

Tager længere tid end multiple choice, så vent til senere medmindre du er skarp i det

3.1 Huffmann

Eksempel fra Prøveeksamen 2023:

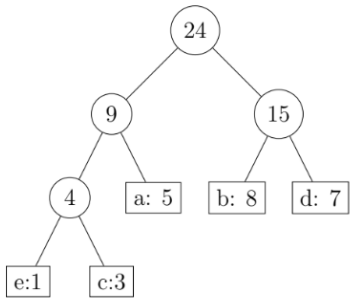
Vi betragter algoritmen til konstruktion af Huffman koder i CLRS sektion 16.3. a) Tegn Huffman træet som algoritmen beregner for alfabetet $C = \{a, b, c, d, e\}$ med frekvenser $f_a = 5, f_b = 8, f_c = 3, f_d = 7, f_e = 1$. Det er ikke nødvendigt at angive labels (0 og 1) på kanterne, men angiv frekvenser i blade såvel som indre knuder (se eksempel i CLRS figur 16.4).

Fra processen af konstruering af et Huffman træ, så tager det form som et binær træ, hvor frekvenserne er nøglerne til knuderne. Egenskaberne for et Huffman træ er:
Knuderne indsættes fra bunden. De to mindste knuder indsættes først.
Forældren til de to knuder er summen af deres nøgleværdier.
Hvis en nøgleværdi er større end en forælder, så bliver den højre nabo, og de to tilsammen danner en ny forælder som så er summen af de to nøgler.
Processen gentages indtil træet er færdigt.

(I dette tilfælde er det ligegyldigt at angive binære værdier, men hvis man skal, så er venstre barn altid 0 og højre barn 1).

Vi starter med de to mindste nøgler, e = 1 og c = 3. De bliver så hhv. venstre og højre barn i bunden. af træet.
De to nøgler bliver lagt sammen og giver forældren 4.
Den næste nøgle indsættes, som er a = 5. Den indsættes som nabo til 4.
De to nøgler bliver lagt sammen og giver forældren 9.

Vi indsætter så de sidste to nøgler d = 7 og b = 8 i bunden af højre gren af træet, siden at de begge er mindre end 9.
De to nøgler lægges sammen og giver forældren 15.
Da der ikke er flere nøgler tilbage, så bliver de to øverste knuder i træet lagt sammen for at danne roden. 15 + 9 = 24.
Roden i træet er så 24, og træet er færdigt. Det endelige træ ser således ud:



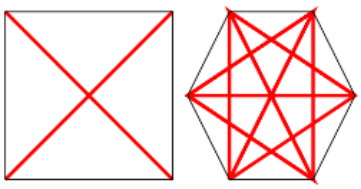
Opsummering:
Start med de mindste nøgler.
Sum nøgler sammen for at danne nye noder.
Hvis en nøgle er større end summen, bliver den nabo og lægges sammen for at danne en ny forælder. Ellers lægges den over i højre gren af træet.
Venstre barn har altid binær værdi 0, højre barn har altid binær værdi 1.

3.2 Induktionsbeviser

Kræver altid bevis for både Basis og Induktionscase

Eksempel fra Prøveeksamen 2023:

I denne opgave betragter vi n-kanter som disse eksempler ($n = 4$ og $n = 6$) :



Vi er interesserede i antal diagonaler, dvs. linjer der forbinder to hjørner i n-kanterne og er helt indeholdt i figuren. En 3-kant har ingen diagonaler da alle hjørner er naboer, mens vi for $n = 4$ og $n = 6$ har henholdsvis 2 og 9 diagonaler.

En diagonal deler en n-kant en n_1-kant og en n_2 kant, der deler en kant og hvor $n_1 + n_2 = n + 2$. For eksempel deler 4 -kantens diagonal den i to 3 -kanter. a) Bevis ved induktion at antal diagonaler d_n i en n-kant er lig med $n(n-3)/2$ for $n \geq 3$. Der lægges vægt på, at argumentationen er klar og koncis.

Her får vi angivet, at $n \geq 3$. Det er vigtigt i forhold til Basis case, siden at n kan minimum være lig 3.

Vi udregner først base case, hvor $n = 3$. Det er givet at antal diagonaler d_n i en n-kant er lig med $n(n-3)/2$; derfor kan vi blot indsætte 3 på n's plads og udregne:

$3 * (3-3)/2 = 3 * 0/2 = 0$

Da det er oplyst at en trekant ikke har nogen diagonaler, så holder Basis case.

For at bevise induktionscasen, så vil jeg overlade ordet til facit fra eksamen (jeg er røv til induktionsbeviser):

Antag nu at $n > 3$, at udsagnet er korrekt for n'-kanter med $n' < n$, og betragt en n-kant. Tag tre hjørner v_1, v_2, v_3 der ligger ved siden af hinanden, dvs. hvor v_1 og v_3 er forbundet til v_2. Fra v_2 kan der laves diagonaler til hver af de andre $n-3$ hjørner og desuden kan der laves en diagonal mellem v_1 og v_3. De resterende diagonaler ligger i den $(n-1)$-kant, der fås ved at udelade v_2 (og forbinde v_1 og v_3 direkte). Ifølge induktionshypotesen har vi $d_{n-1} = (n-1)(n-4)/2$ diagonaler i denne del, dvs.

$$d_n = n - 3 + 1 + d_{n-1} = n - 2 + (n-1)(n-4)/2 = n(n-3)/2.$$

Opsummering:
Håb på det bedste.
Lav Basis case (som regel nemt) for at få partial point.
Husk at holde øje med min. gyldige værdi af n, hvis n ikke har en, så er Basis case blot $n = 0$.

3.3 Dynamisk Programmering

Ligger som regel op til brug af LCS- eller Rod-Cutting formlerne.
Meget lidt skal ændres (samme køretid, samme muligheder for greedy choice).

Eksempel 1 fra Eksamen 2023:

Vi betragter følgende variant af længste fælles delsekvens (eng. longest common subsequence) problemet fra CLRS sektion 15.4. Input er to vektorer $x = \langle x_1, \ldots, x_m \rangle \in \Sigma^m$ og $y = \langle y_1, \ldots, y_n \rangle \in \Sigma^n$. I lighed med længste fælles delsekvens er målet at finde en optimal vektor $z = \langle z_1, \ldots, z_k \rangle$, men i stedet for at maksimere længden af z skal vagten af tegnene i z maksimeres. Vi antager at vægtene er ikke-negative. Vægten er givet ved en datastruktur w, således at vægten af $z_i \in \Sigma$, der betegnes med $w[z_i]$, kan findes i konstant tid. Vægten af z er summen af vægte, $\sum_{i=1}^{k} w[z_i]$, og målet er et finde en fælles delsekvens med størst mulig vægt.

Eksempel. Antag at vi har $x = \langle A, H, A, A \rangle, y = \langle A, A, R, G, H \rangle$, og $w[A] = 2$, $w[G] = 8, w[R] = 10, w[H] = 5$. Delsekvensen $\langle A, A \rangle$ med vægt $2 + 2 = 4$ findes i både x og y. Delsekvensen $\langle A, H \rangle$ med vægt $2 + 4 = 7$ findes også i både x og y, og er den delsekvens, der har størst vægt.

a) Skriv en rekursionsligning for den optimale vægt af en fælles delsekvens af x og y.

Det er oplyst, at varianten af algoritmen er fra LCS. Derfor kan formlen til LCS genbruges. Det eneste som skal ændres er at der skal tages højde for vægten w:

$$c[i,j] = \begin{cases} 0 & \text{hvis } i = 0 \text{ eller } j = 0 \\ c[i-1, j-1] + w[\alpha] & \text{hvis } i, j > 0 \text{ og } x_i = y_j = \alpha \\ \max(c[i-1, j], c[i, j-1]) & \text{hvis } i, j > 0 \text{ og } x_i \neq y_j \end{cases}$$

b) Lav en analyse af køretiden af den algoritme, der bruger rekursionsligningen og dynamisk programmering til at beregne den optimale vægt. Analysen skal give en øvre grænse på køretiden som funktion af n og m i store- O notation, der er så præcis som mulig.

Siden at vi blot har genbrugt LCS-formlen, kan køretiden af formlen også genbruges. Det oplyses at tiden for at finde en vægt er konstant, så den behøves ikke betragtes med i køretiden. Køretiden for LCS i store-O er $O(mn)$, derfor er køretiden af denne algoritme også $O(mn)$.

c) Beskriv hvordan algoritmen kan modificeres til at håndtere det tilfælde hvor vægte kan være negative uden at påvirke den asymptotiske køretid. Argumentér for køretid og korrekthed af din løsning.

Siden at LCS har Greedy Choice Property og Optimal Delstruktur, så kan negative værdier aldrig være med i en optimal løsning. (Facit siger noget andet):

Tegn med negative vægte vil aldrig være del af en optimal løsning, hvilket kan ses med et modstridsargument: Hvis z indeholder et tegn med negativ vægt, så har delsekvensen der udelader dette element højere vægt. Tegn med negativ vægt kan derfor fjernes fra input i tid $O(n + m)$, hvorefter det resterende problem er uden negative vægte. Den samlede tid er $O(n + m + nm)$, hvilket er $O(nm)$ som inden.

Eksempel 2 fra Prøveeksamen 2023:

Vi betragter følgende variant af rod cutting problemet fra CLRS sektion 15.3. Input er et heltal n og en vektor p hvor p_i er fortjenesten på en stav af længde n. I lighed med rod cutting skal en stav (eng. rod) af længde n deles op i kortere stave, alle af heltalslængde, således at den totale fortjeneste maksimeres. I modsætning til det originale rod cutting problem er der dog en omkostning på 1 for hvert snit, som skal modregnes i fortjenesten.

Eksempel. Antag at vi har en stav af længde $n = 5$, hvor $p = (p_1, p_2, p_3, p_4, p_5) = (1, 3, 5, 6, 6)$. Hvis vi deler staven i to dele af størrelse henholdsvis 2 og 3 , så bliver fortjenesten $p_2 + p_3 - 1 = 7$, hvor vi trækker 1 fra fordi der er ét snit. Hvis vi i stedet delte i 5 dele af størrelse 1 ville fortjenesten blive $5p_1 - 4 = 1$.

a) Skriv en rekursionsligning for fortjenesten (eng. revenue) r_n for en stav af længde n.

Det er oplyst at varianten af algoritmen er fra rod cutting. Derfor kan formlen til rod cutting genbruges. Da længden af roden n godt kan være $= 0$, så har vi at $r_0 = 0$ og at:

$$r_n = \max_{0 \leq i \leq n} (p_i - r_{n-i} - 1)$$

b) Lav en analyse af køretiden af algoritme, der bruger rekursionsligningen og dynamisk programmering til at beregne fortjenesten ved en stav af længde n. Analysen skal give en øvre grænse på køretiden som funktion af n i store- O notation, der er så præcis som mulig.

Siden vi har genbrugt formlen for rod cutting, så kan køretiden igen genbruges. Da køretiden for rod cutting i store-O er $O(n^2)$, så er køretiden for algoritmen også $O(n^2)$.

Opsummering: Tager som regel udgangspunkt i en eksisterende DP-algoritme eller formel.
Strukturen kan genbruges, ekstra cases skal blot tilføjes til algoritmen.
Køretid kan genbruges fra algoritmerne, medmindre at der specifikt står en operation kræver ekstra tid udover det sædvandelige.

3.4 LCS-Konstruktion

Tager 100 år med tablet. Ingen hurtig måde at lave den på.
Lav den kun hvis du har al tiden i verden.

3.5 Grådige Algoritmer

Betragter som regel en opgave fra multiple choice-delen.
Nej.

3.6 Korrekthed og Køretider

Don't bother.