06. Dynamisk programmering

Forelesning 6

På sett og vis en generalisering av *splitt og hersk*, der delprobler kan overlappe: I stedet for et *tre* av delproblem-avhengigheter har vi en *rettet asyklisk graf*. Vi finner og lagrer del-løsninger i en rekkefølge som stemmer med avhengighetene.

Pensum

- [ ] Innledningen til del IV
- [ ] Kap. 14. Dynamic programming: Innl. og 14.1–14.4
- [ ] Oppgave 15.2-2 med løsning (0-1 knapsack)
- [ ] Appendiks D i pensumheftet

Læringsmål

- [F_1] Forstå *delinstanser*
- [F_2] Forstå *dynamisk programmering*
- [F_3] Forstå løsning ved *memoisering* (*top-down*)
- [F_4] Forstå løsning ved *iterasjon* (*bottom-up*)
- [F_5] Forstå hvordan man *rekonstruerer* en løsning fra lagrede beslutninger
- [F_6] Forstå hva *optimal delstruktur* er
- [F_7] Forstå hva *overlappende delproblemer* er
- [F_8] Forstå *stavkutting*, *matrisekjede-multiplikasjon* og *LCS*
- [F_9] Forstå løsningen på *det binære ryggsekkproblemet* (se appendiks D i pensumheftet)

1

📚Optimal delstruktur

Optimal delstruktur er en egenskap et problem kan ha, og er nødvendig for at problemet kan ha en løsning med dynamisk programmering. I tillegg må problemet ha overlappende delinstanser. Når et problem har overlappende delinstanser betyr det at flere oppdelinger av problemet inneholder noen av de samme delinstansene.

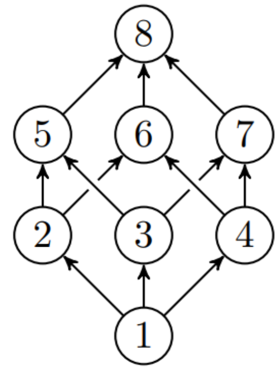
Oppskrift fra boka

1. **Characterize the structure of an optimal solution**
2. **Recursively define the value of an optimal solution**
3. **Compute the value of an optimal solution**
4. **Construct an optimal solution from computed information**

At et problem har optimal delstruktur betyr at det løsningen til problemet kan konstrueres fra optimal løsninger på delinstanser av problemet.

Det vil dermed være optimalt å cache eller lagre svaret på delinstansene slik at man slipper å regne ut løsningen når problemet oppstår igjen.

Man kan generelt se på dynamisk programmering som et “Time-memory tradeoff” hvor vi ofrer minne for å lagre delløsninger, men i gjengjeld kan vi spare mye tid.



Overlappende og delte delproblemer . . .

Nyttig **når vi har overlappende delproblemer**

Korrekt **når vi har optimal substruktur**

💻Rod Cutting Problem

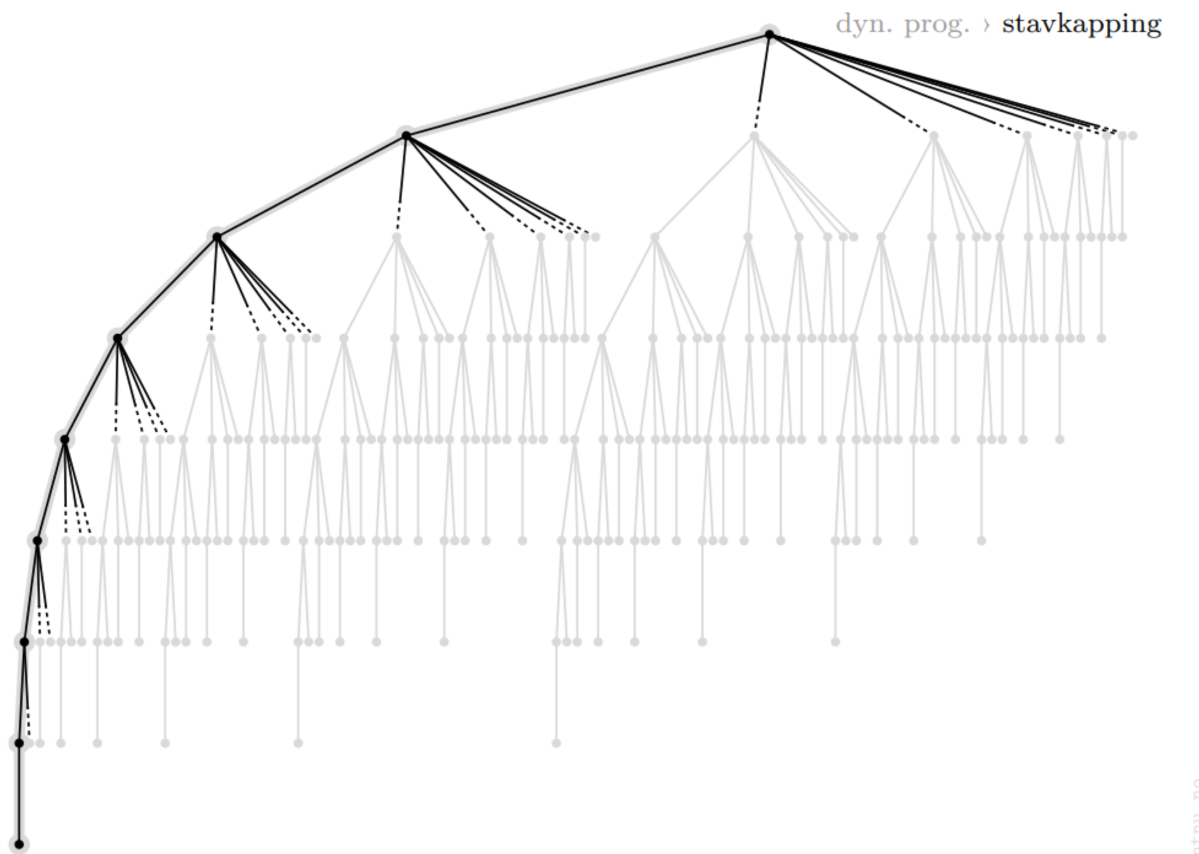
Stavkapproblemet er et eksempel på et problem som kan løses optimal ved hjelp av dynamisk programmering.

En dum løsning på dette problemet kan oppnås ved å prøve alle kombinasjonene rekursivt, og til slutt oppnå en kjøretid på $O(2^n)$. Dette er ikke bra nok!

Input: En lengde n og priser p_i for lengder $i = 1, \ldots, n$.

Output: Lengder $\ell_1, \ldots, \ell_k$ der summen av lengder $\ell_1 + \cdots + \ell_k$ er n og totalprisen $r_n = p_{\ell_1} + \cdots + p_{\ell_k}$ er maksimal.

En optimal løsning vil lagre løsningene på de forrige verdiene, og det blir dermed mulig å slå opp løsningen på tidligere delproblemer. Grafen nedenfor illustrerer hvor mange kalkulasjoner som spares på denne måten.



📚Bottom-up iterasjon vs top-down memoisering

Det finnes to hovedmetoder for å programmere en løsning som utnytter dynamisk programmering; bottom-up og top-down. Forskjellen på metodene handler om hvordan man designer løsningen til å utføre kallene på delproblemene.

- Top-down: du begynner på hovedproblemet og finner ut hvilke delløsninger som er mulige for å komme seg dit. Her er det vanlig å bruke rekursjon og memoisering av funksjoner.
- Botton-up: du finner ut av hvilke delløsninger som kommer til å måtte løses, og løser disse. Deretter jobber du deg opp mot hovedproblemet og bruker de tidligere løsningene. Denne metoden kalles ofte tabulation fordi du fyller gjerne inn delløsningene i en tabell.

💻Matrisekjedemultiplikasjon

Matrisekjedemultiplikasjon er en et

problem hvor vi ønsker å finne den optimale måten å gange sammen en kjede (liste) med matriser av ulik størrelse $\langle A_1, A_2, \ldots, A_n \rangle$. =

Hvis vi har fire matriser, finnes det fem unike måter å gange disse sammen på, og basert på størrelsen på hver av matrisene kan antallet kalkulasjoner variere stort.

Hvis vi prøver å gange sammen matrisene slik som vi vanligvis hadde gjort, kan det hende vi gjør unødvendig mange kalkulasjoner.

Input: En sekvens $\langle p_0, p_1, \ldots, p_n \rangle$, der p_{i-1} og p_i er antall rader og kolonner i en matrise A_i.

Output: En parentessetting av produktet $\mathrm{A}_{1:n} = \mathrm{A}_1\mathrm{A}_2 \cdots \mathrm{A}_n$ som minimerer antall skalarprodukt.

$$
\begin{aligned}
&(A_1(A_2(A_3A_4)))\ , \\
&(A_1((A_2A_3)A_4))\ , \\
&((A_1A_2)(A_3A_4))\ , \\
&((A_1(A_2A_3))A_4)\ , \\
&(((A_1A_2)A_3)A_4)\ .
\end{aligned}
$$

Fire matriser har 5 unike måter å ganges sammen

💻Longest common subsequence

Gitt to sekvenser ønsker vi å finne den lengste subsekvensen som er lik i begge sekvensene.

Hvis vi prøver å løse problemet ved hjelp av brute-force finner vi fort ut at vi får et eksponensielt stort problem som vil være ubrukelig ved store n.

Input: To sekvenser, $\mathrm{X} = \langle x_1, \ldots, x_m \rangle$ og $\mathrm{Y} = \langle y_1, \ldots, y_n \rangle$.

Output: En sekvens $\mathrm{Z} = \langle z_1, \ldots, z_k \rangle$ og indekser $i_1 \leqslant \cdots \leqslant i_k$ og $\ell_1 \leqslant \cdots \leqslant \ell_k$ der $\mathrm{Z}[j] = \mathrm{X}[i_j] = \mathrm{Y}[\ell_j]$ for $j = 1 \ldots k$, der Z har er lengst mulig.

💻Binære ryggsekkproblemet

Det binære rykksekkproblemet handler om at vi skal velge et sett med ting av forskjellig verdi og vekt slik at vi utnytter kapasiteten vår W så godt som mulig.

Input: Verdier $v_1, \ldots, v_n$, vekter $w_1, \ldots, w_n$ og en kapasitet W.

Output: Indekser $i_1, \ldots, i_k$ slik at $w_{i_1} + \cdots + w_{i_k} \leq \mathrm{W}$ og totalverdien $v_{i_1} + \cdots + v_{i_k}$ er maksimal.