# Overzicht algoritmes: dynamisch programmeren tot einde cursus

Compact schema vanaf het deel dynamisch programmeren tot en met SAT solving.

Notatie: n = aantal items/knopen/variabelen volgens context, `m` = tweede invoerlengte of aantal clauses volgens context.

## 1. Dynamisch programmeren: algemeen

| Concept | Betekenis | Hoe herken je het? | Strategie |
|---|---|---|---|
| Optimalisatieprobleem | Je zoekt minimum of maximum kost/opbrengst. | Er zijn veel kandidaatoplossingen met een score. | Definieer `opt(...)`: beste waarde voor een deelprobleem. |
| Optimale substructuur | Delen van een optimale oplossing zijn zelf optimaal. | Als een betere deeloplossing de totale oplossing zou verbeteren, heb je DP-kans. | Bewijs vaak uit het ongerijmde. |
| Overlappende deelproblemen | Dezelfde deelproblemen komen vaak terug. | Naieve recursie berekent dezelfde states opnieuw. | Sla waarden op in tabel of memo. |
| Top-down met memoizatie | Recursief oplossen, maar reeds berekende states bewaren. | Natuurlijk als de recursieve formule makkelijk is. | Cache `opt(state)`; elke state hoogstens een keer echt berekend. |
| Bottom-up tabulatie | Tabel in volgorde opvullen van klein naar groot. | Natuurlijk als je afhankelijkheden duidelijk kan ordenen. | Basisgevallen invullen, daarna grotere states. |

DP-checklist:

1. Kies de state: welke parameters beschrijven een deelprobleem?
2. Vul basisgevallen in.
3. Schrijf de recurrence: welke keuze probeer je?
4. Bepaal vulvolgorde: welke states moeten eerst gekend zijn?
5. Bewaar extra keuze-info als je de oplossing zelf moet reconstrueren.
6. Complexiteit = aantal states maal werk per state.

## 2. Klassieke DP-problemen

| Probleem | Waarvoor? | State en basis | Recurrence | Complexiteit |
|---|---|---|---|---|
| Rod cutting | Staaf van lengte n snijden om opbrengst te maximaliseren. | `opt(j)` = beste opbrengst voor lengte `j`; `opt(0)=0`. | `opt(j)=max_{1<=i<=j}(p_i + opt(j-i))`. | Naief exponentieel; memo/bottom-up `O(n^2)` tijd, `O(n)` geheugen. |
| Matrix-chain multiplication | Beste parenthesering voor `A_1 ... A_n` zodat aantal scalaire vermenigvuldigingen minimaal is. | `m[i,j]` = min kost voor `A_i ... A_j`; `m[i,i]=0`. | `m[i,j]=min_{i<=k<j}(m[i,k]+m[k+1,j]+p_{i-1} p_k p_j)`. | `O(n^3)` tijd, `O(n^2)` geheugen. |
| Longest common subsequence | Langste gemeenschappelijke deelsequentie van `X[1..m]` en `Y[1..n]`. | `c[i,j]` = LCS-lengte van prefixen; `c[0,j]=c[i,0]=0`. | Als `x_i=y_j`: `c[i,j]=c[i-1,j-1]+1`; anders `max(c[i-1,j], c[i,j-1])`. | `O(mn)` tijd, `O(mn)` geheugen; enkel lengte kan met 2 rijen. |
| Optimal binary search tree | Binaire zoekboom met minimale verwachte zoekkost bij kansen `p_i` en `q_i`. | `e[i,j]` = min verwachte kost voor keys `k_i..k_j`; `e[i,i-1]=q_{i-1}`. | `e[i,j]=w(i,j)+min_{i<=r<=j}(e[i,r-1]+e[r+1,j])`, met `w(i,j)=w(i,j-1)+p_j+q_j`. | `O(n^3)` tijd, `O(n^2)` geheugen. |
| Floyd-Warshall als DP | Alle kortste paden tussen alle paren knopen. | `D_k[i,j]` = kortste pad van `i` naar `j` met enkel tussenknopen uit `{1..k}`. | `D_k[i,j]=min(D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j])`. | `O(n^3)` tijd, `O(n^2)` geheugen. |

## 3. DP-oplossing reconstrueren

| Probleem | Wat extra bewaren? | Hoe reconstrueren? |
|---|---|---|
| Rod cutting | `s[j]` = eerste snede die de max haalt. | Begin bij `j=n`, neem `s[j]`, ga verder met `j-s[j]`. |
| Matrix-chain multiplication | `s[i,j]` = beste split k. | Recursief print: `(A_i...A_k)(A_{k+1}...A_j)`. |
| LCS | Pijltjes, of waarden opnieuw vergelijken. | Diagonaal bij match, anders naar buur met grootste waarde. |
| Optimal BST | `root[i,j]` = beste wortel `r`. | Wortel `r`, links `i..r-1`, rechts `r+1..j`. |

## 4. SAT: basis

| Concept | Betekenis | Belangrijk |
|---|---|---|
| P | Problemen oplosbaar in polynomiale tijd. | Meeste klassieke algoritmen uit de cursus zitten hier. |
| NP | Kandidaatoplossing kan in polynomiale tijd gecontroleerd worden. | SAT zit in NP: interpretatie invullen en formule checken. |
| NP-compleet | Elk NP-probleem kan in polynomiale tijd naar dit probleem worden gereduceerd. | SAT is NP-compleet. |
| SAT | Bestaat er een interpretatie die de formule waar maakt? | Decision problem: ja/nee. |
| Literal | Variabele of negatie van variabele. | Voorbeeld: `x`, `not x`. |
| Clause | OR van literals. | Voorbeeld: `(x OR not y OR z)`. |
| CNF | AND van clauses. | SAT-solvers werken typisch op CNF. |

CNF-omzetting:

1. Vervang `phi <=> psi` door `(phi => psi) AND (psi => phi)`.
2. Vervang `phi => psi` door `(not phi OR psi)`.
3. Duw negaties naar binnen met De Morgan.
4. Gebruik distributiviteit om AND van OR-clauses te krijgen.

## 5. Problemen encoderen als SAT

| Patroon | CNF-idee |
|---|---|
| Variabelekeuze | Maak Booleaanse variabelen, bv. `x_{i,j}` = toezichter `i` werkt in slot `j`. |
| Minstens een | `(x1 OR x2 OR ... OR xk)`. |
| Hoogstens een | Voor elk paar: `(not xi OR not xj)`. |
| Precies een | Minstens een + hoogstens een. |
| Geen conflict | Voeg clause toe die verboden combinatie uitsluit, bv. `(not x_i OR not x_j)`. |
| Reductie naar SAT | Encodeer constraints als clauses en laat solver een model zoeken. |

Voorbeelden uit de slides: n-koninginnen, kaartkleuring, scheduling.

## 6. SAT-algoritmen en optimalisaties

| Techniek | Idee | Kernregel | Rol |
|---|---|---|---|
| Naieve backtracking | Bouw interpretatie op met DFS in zoekboom. | Kies onbepaalde variabele, probeer `true`, anders `false`. | Simpel, maar worst-case exponentieel. |
| Unit propagation | Als een clause nog maar een mogelijke literal heeft, moet die waar worden. | Clause `(l)` forceert `l=true`; lege clause = conflict. | Vermindert zoekboom sterk. |
| DPLL | Backtracking + unit propagation + vaak pure literals. | Eerst deterministische gevolgen afleiden, dan pas branch kiezen. | Basis van klassieke SAT-solvers. |
| Pure literal rule | Literal komt maar in een polariteit voor. | Als x enkel positief voorkomt, zet `x=true`. | Bewaart satisfiability, maar niet noodzakelijk alle modellen. |
| Two watched literals | Per clause twee niet-false literals volgen. | Clause pas bekijken als watched literal false wordt. | Maakt unit propagation veel efficienter. |
| Resolutie | Leid nieuwe clause af uit twee clauses met tegengestelde literal. | `(A OR l)` en `(B OR not l)` geven `(A OR B)`. | Basis voor conflictanalyse. |
| CDCL / clause learning | Analyseer conflict, leer clause, spring terug. | Geleerde clause voorkomt dezelfde fout later. | Moderne SAT-solvers. |
| Backjumping | Niet gewoon een niveau terug, maar naar relevante keuze. | Spring naar de oorzaak van conflict. | Niet-chronologische backtracking. |

## 7. Varianten van SAT

| Variant | Vraag | Methode/idee |
|---|---|---|
| #SAT | Hoeveel modellen heeft de formule? | CDCL kan verder zoeken; optimisaties moeten modelaantallen bewaren. BDDs zijn een specifieke methode. |
| BDD | Beslissingsdiagram voor een formule. | Kies variabelevolgorde, splits op `true/false`, reduceer gelijke knopen en tel paden naar `true`. |
| Pseudo-Boolean constraints | Lineaire constraints over 0/1-variabelen. | Clause `p OR not q OR r` wordt `p + (1-q) + r >= 1`. |
| SAT naar PBC | Clauses als somconstraints schrijven. | Handiger voor tellen zoals `sum x_i = 1` of `sum x_i <= k`. |
| PBC naar SAT | Niet meer expressief dan SAT, wel vaak compacter. | Kan via BDD naar SAT worden omgezet. |
| MaxSAT | Zo goed mogelijk model als niet alle wensen tegelijk kunnen. | Harde constraints moeten waar zijn; maximaliseer aantal zachte clauses. |
| Gewogen MaxSAT | Zachte clauses hebben gewicht. | Maximaliseer totaal gewicht van voldane zachte clauses. |
| Blocking variable transform | Soft constraint s krijgt blocking variable `b_s`. | Maak harde constraint `b_s -> s`; optimaliseer over `b_s`. |

## 8. Snelle keuzehulp

| Als je ziet... | Denk aan... |
|---|---|
| Minimum/maximum met overlappende deelproblemen | Dynamisch programmeren. |
| Prefixen van twee strings | LCS-achtige 2D DP. |
| Interval `i..j` met keuze van split/wortel | Matrix-chain of optimal BST. |
| Veel constraints en je wil weten of er een oplossing bestaat | SAT-encoding + DPLL/CDCL. |
| "Precies een" of "hoogstens k" | SAT-clauses of PBC. |
| Niet alle constraints kunnen tegelijk | MaxSAT. |
| Aantal oplossingen nodig | #SAT of BDD. |