# Leertaak 3: Algoritmische opgaven

## Doel

Het doel van deze leertaak is:

* Leren werken met een aantal klassieke algortimen en datastructuren (recursie, backtracking, trees, ...).

## Backtracking

**Speelkaartenprobleem**

Rangschik 8 speelkaarten in onderstaand patroon, zodanig dat wordt voldaan aan de volgende eisen:

* Er zijn 2 azen, 2 heren, 2 vrouwen en 2 boeren.
* Elke aas grenst (horizontaal of verticaal) aan (tenminste) een heer.
* Elke heer grenst aan een vrouw.
* Elke vrouw grenst aan een boer.
* Er zijn geen kaarten van eenzelfde soort die aan elkaar grenzen.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. Leg uit hoe *recursie* werkt en aan welke twee voorwaarden recursie moet voldoen voor een correcte werking.

Een probleem oplossen met recursie houdt in dat het wordt opgebroken in kleinere sub-problemen. Als een sub-probleem overeenkomt met het originele probleem kan dezelfde benadering worden toegepast om het sub-probleem recursief op te lossen. Dit sub-probleem is ongeveer hetzelfde als het originele probleem maar van een kleinere aard.

Recursie moet aan de volgende 2 voorwaarden voldoen:

1. Minstens 1 base case ( de simpelste ) wordt gebruikt om de recursie te stoppen.
2. Elke recursieve call vermindert het originele probleem. Hierdoor komt het steeds dichter tot een base case tot het die case wordt.
3. Bestudeer het algemene backtracking algoritme dat gegeven is op Blackboard. Leg uit dat het *algemene backtracking algoritme* een correcte toepassing van recursie is omdat het voldoet aan de twee voorwaarden.
4. Analyseer het speelkaartenprobleem. Vertaal de begrippen uit het probleem naar de begrippen uit het algemene backtracking algoritme.

- Wat zijn je kandidaten?

De kandidaten zijn: {Aas, 2} , {Heer, 2} , {Vrouw,2} , {Boer,2}

- Hoe bewaar je de kandidaten, in welke datastructuur?

- Hoe heb je het veld gemodelleerd (datastructuur)?

- Hoe herstel je het veld als de recursieve aanroep returnt?

1. Haal de code van SVN en maak een oplossing voor het probleem waarbij je alleen de regel implementeert dat kaarten van eenzelfde soort niet aan elkaar mogen grenzen.
2. Maak daarna je programma af door de overige checks te implementeren (“Elke aas grenst aan een heer, ...”).

## Minimax

**TicTacToe**

Gegeven is een programma voor TicTacToe (Boter, Kaas en Eieren) in de SVN repository.

1. Schrijf een JUnit test om de klasse TicTacToe te testen (positionValue(), isAWin(), chooseMove()). De unittest van chooseMove moet testen dat de methode de beste zet kiest.
2. Analyseer het probleem en de gegeven code. Bestudeer het minimax principe (<http://en.wikipedia.org/wiki/Minimax>). Wat worden je nodes, je depth en de waarderingsfunctie?
3. Implementeer de benodigde methoden. Maak gebruik van backtracking om door alle mogelijkheden heen te komen. Maak gebruik van minimax om de beste zet te kiezen en zo het oordeel van een stelling te bepalen.

**N.B. 1 Je bent verplicht te werken in de op SVN aangereikte opzet. Het is niet de bedoeling dat je zelf andere mogelijke structuren gaat bedenken.**

**N.B. 2 Voor deze opgave hoef je niet veel meer dan +/- 20 regels code te schrijven.**

*Voorbeeld van het algoritme:*

|  |  |  |
| --- | --- | --- |
| X | O |  |
| O | X |  |
| X | X | O |

**Stelling 1**

* Het evalueren van deze stelling levert op dat er nog geen winnaar of remise is. Er zal dus nog dieper gezocht moeten worden.
* O is hier aan de beurt. Er zijn twee mogelijkheden om de O te plaatsen. Plaats eerst de O rechtsboven en doe dan een recursieve aanroep.

|  |  |  |
| --- | --- | --- |
| X | O | O |
| O | X |  |
| X | X | O |

**Stelling 2**

* Er is nog steeds geen winnaar of remise. Plaats de X en doe een recursieve aanroep.

|  |  |  |
| --- | --- | --- |
| X | O | O |
| O | X | X |
| X | X | O |

**Stelling 3**

* Er is een oordeel, stelling 3 is remise. We hoeven dus geen verdere zetten meer te doen (dat was ook niet gelukt want het bord is vol). Return ‘remise’.

|  |  |  |
| --- | --- | --- |
| X | O | O |
| O | X |  |
| X | X | O |

* Dat betekent dat stelling 2 remise is, want er zijn geen andere zetten.
* Dat betekent dat O in stelling 1 dus minstens remise heeft. Maar er was nog een tweede mogelijkheid dus, plaats nu de O rechtsmidden.

|  |  |  |
| --- | --- | --- |
| X | O |  |
| O | X | O |
| X | X | O |

**Stelling 4**

* Met een vergelijkbare redenatie komt hier ‘Verlies’ uit voor O.
* O heeft in stelling 1 dus de keuze uit twee zetten. Eén zet leidt tot remise en de ander tot verlies. Het minimax algoritme zegt dan dus dat O de beste kiest die tot remise leidt.

|  |  |  |
| --- | --- | --- |
| X | O |  |
| O | X |  |
| X |  | O |

**Stelling 0**

* In stelling 0 kan X er dus op rekenen dat middenonder hem remise oplevert. Daarna kan hij de andere opties nog onderzoeken. Etc.

Bewaar je programma voor het project (leertaak 6).

## Dynamisch Programmeren

Lees BB🡪Ondersteunend materiaal🡪SE🡪Dynamic Programming🡪DynProg-Steenbergen.pdf.   
Lees daarna <http://en.wikipedia.org/wiki/Dynamic_programming>. Het checkerboard voorbeeld laat zien hoe je door “bottom up” te werken een lastig probleem toch lineair kunt uitwerken. (N.B. De nummering van de rijen op wikipedia is verwarrend. Eerst is rij 1 de onderste rij, in de minCost functie staat er opeens i=5)

**Probleemomschrijving**

Gegeven is een array met n getallen groter dan nul en een getal B. Bepaal of je (een deel van) de getallen uit het array kunt optellen tot B. Met andere woorden, bestaat er een deelverzameling waarvan de som B is?

Bijvoorbeeld: { 3, 5, 7, 9, 11} en B = 17. Hier bestaat zo’n deelverzameling, 3 + 5 + 9 = 17.

Stel je probeert het probleem op te lossen met recursie.

* Base case: n = 0, het array is leeg. Het antwoord is *false*, tenzij B = 0. Want met een leeg array kun je de som 0 maken.
* Recursie stap: Het antwoord is *true* wanneer:  
  - Je met de n-1e getallen de som B kunt maken. In dat geval zit het ne getal dus niet in de oplossing. In ons voorbeeld kun je met { 3, 5, 7, 9, 11} de som 17 maken omdat je met { 3, 5, 7, 9 } de som 17 kunt maken.  
  - Óf je kunt met de n-1e getallen de som (B – het ne getal) maken. In dat geval zit het ne getal wél in de oplossing. In ons voorbeeld geldt dat { 3, 5, 7, 9 } voor de som 17 true moet opleveren omdat je met { 3, 5, 7 } de som 8 (= 17 – 9) kunt maken. Maar dat laatste feit ontdek je natuurlijk pas aan het eind van de recursie wanneer je op de base case uitkomt die true teruggeeft.

**Voor een uitgebreidere uitleg: zie het addendum op BlackBoard.**

1. Wat is de orde (efficiency) van deze aanpak als je het naïef recursief uitprogrammeert? (Vergelijk met de naïeve fibonacci implementatie van fib(n) = fib(n-1) + fib(n-2)). Geef een toelichting.

We kunnen het probleem ook slimmer oplossen met behulp van dynamisch programmeren. Definieer een matrix (= tabel = 2D array) M(i,j) met n rijen en B kolommen. Definieer M(i,j) = true als je met een deelverzameling van de eerste i getallen de som j kunt maken.

1. Hoe ziet de eerste rij van de matrix M(1,...) eruit voor ons voorbeeld van { 3, 5, 7, 9, 11}? M(1, j) geeft dus aan of je met alleen het 1e getal uit het array de som j kunt maken.
2. Hoe kun je de waarde van M(i, j) bepalen op grond van de waarden in de i-1e rij?  
   (Vergelijk bijvoorbeeld met de relatie van het collegevoorbeeld. Voor het checkerboardprobleem was de relatie q(i,j) = min( q(i-1,j-1), q(i-1,j), q(i-1,j+1) ) +c(i,j). Welke relatie geldt in de huidige opdracht?)
3. Implementeer het algoritme en laat de unittest slagen. Welke cel bevat het uiteindelijke antwoord op de vraag waarin we geïnteresseerd zijn?
4. Wat is de orde van deze aanpak? Geef een toelichting.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 1 {3} |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 {3,5} |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 {3,5,7} |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 {3,5,7,9} |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 {3,5,7,9,11} |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Bij vraag 13 heb je “bottom up” gewerkt. (Ook al ben je boven in de tabel begonnen. Bottom up slaat op het feit dat je bent begonnen bij het begin en toen alle cellen hebt ingevuld totdat je bij het eind komt.) Bij dynamisch programmeren kun je ook “top down” werken. Daarbij begin je bij de eindvraag M(n, B) en ga je recursieve aanroepen plegen. Zodra je een cel M(k, B’) voor de eerste keer hebt berekend sla je het antwoord op. De volgende keer dat je de waarde M(k, B’) nodig hebt dan hoef je hem niet meer uit te rekenen. In DynProg\_Steenbergen.pdf wordt dit “memoization” genoemd.

1. Implementeer bovenstaand probleem volgens de “top down” aanpak en laat zien dat deze de unittest ook doet slagen.

## File compressie

*De volgende opdracht voor Think is alweer in beeld. Deze opdrachtgever heeft echter de gewoonte om iedereen die voor hem klussen aanneemt eerst te testen. Daartoe heeft hij een standaard opdracht: schrijf zelf een compressie programma. De opdrachtgever beoordeelt je vaardigheden op alle aspecten van de uitwerking.*

### Specification[[1]](#footnote-1)

Schrijf een programma dat ASCII-files comprimeert met behulp van het Huffman-algoritme. Het programma moet files kunnen comprimeren en gecomprimeerde files weer terug kunnen decomprimeren.

### Problem analysis

* Lees het materiaal over het Huffman algoritme op Blackboard.

1. Maak een unittest voor de compressie/decompressie klassen. Test daarin dat het decomprimeren van een gecomprimeerde TestA.dat weer de originele file oplevert.

### Architecture & planning

Het grootste deel van de code is reeds gegeven. In de eerste increment vul je de ontbrekende code aan.

**N.B. Je gaat alleen aan het werk in de klasse HuffmannTree. Het is niet nodig om je te verdiepen in bit encoding e.d.**

### Detailed analysis



Figuur 1: Sequence diagram comprimeren

1. Teken een sequencediagram voor het decomprimeren van een file vergelijkbaar met bovenstaande diagram

### Design

1. HuffmanTree.createTree() is waar de boom wordt opgebouwd volgens het algoritme. Teken een sequencediagram hoe je deze methode wilt gaan implementeren.

### Implementation

1. Implementeer de ontbrekende code in de klasse HuffmanTree. (Twee plekken: createTree() en getChar() )

### Test

1. Controleer dat je unittest uit de problemanalysis slaagt. Controleer daarna dat de gegeven unittesten (TestHuffmanTree en TestHzip) ook werken.
2. Neem een aantal inputfiles en bereken hoeveel % winst je bereikt door te comprimeren.
3. Welke kenmerken van een file bepalen of je veel of weinig winst kunt behalen? Kun je dat met testdata reproduceren?

1. Hoewel deze casus niet afkomstig is uit Brugali zullen we wel het “Brugali-proces” als stappenplan hanteren. [↑](#footnote-ref-1)