

## Opgave 6: Backtracking

### Leerdoelen

Na afloop van deze opdracht kun je backtrack algoritmen implementeren in Java.

Backtracking is een veel gebruikte zoek strategie in de informatica om depth-first een boom van mogelijkheden te onderzoeken. In deze opgave laten we je oefenen met twee voorbeelden.

### 1 Product

Gegeven een rijtje getallen, bijvoorbeeld  $\{-1, 1, 2, 3, 4, 5, 6, 7, 11, 13\}$ , dan kunnen andere getallen al of niet geschreven worden als een product van getallen uit deze rij. Getallen uit deze rij mogen hooguit één keer gebruikt worden in het product. We kunnen het getal 24 bijvoorbeeld maken als:

$$24 = 1 \cdot 2 \cdot 3 \cdot 4$$

$$24 = 1 \cdot 4 \cdot 6$$

$$24 = 2 \cdot 3 \cdot 4$$

$$24 = 4 \cdot 6$$

Maar er is geen enkele manier om 17 te maken als product van de gegeven getallen.

```
public class Product
{
    private int getallen [ ];

    /**
     * Constructor bepaalt de rij met getallen
     * @param rij
     */
    public Product (int [ ] rij)
    {
        getallen = rij;
    }

    /**
     * genereert een String die laat zien hoe we doel kunnen maken als product van de getallen
     * @param doel
     * @return de string met alle mogelijkheden
     */
    public String zoek (int doel)
    {
        return zoek (1, 0, doel, "");
    }

    /**
     * Het eigenlijke backtrack algoritme
     * @param huidig: het product van alle getallen tot nu toe
     * @param index: van het volgende kandidaat getal in de reeks
     * @param doel: het getal dat we willen maken
     * @param trace: de string die aangeeft wat we al vermenigvuldigd hebben
     * @return de tstring die alle oplossingen bevat (met een \n achter elke oplossing)
     */
    private String zoek (int huidig, int index, int doel, String trace)
    {
        // moet je invullen
    }
}
```

```

public class Main
{
    /**
     * @param args the command line arguments
     */
    public static void main(String[ ] args)
    {
        new Main();
    }

    /**
     * vult een rij met getallen en laat gebruiker in een loopje doelen opgeven
     * programma laat zien hoe het doel gemaakt kan worden als product
     * programma stopt als het gegeven doel de waarde 0 heeft.
     */
    public Main()
    {
        Product product = new Product (new int [ ] {-1, 1, 2, 3, 4, 5, 6, 7, 11, 13});
        Scanner scan = new Scanner (System.in);
        for (int doel = scan.nextInt(); doel != 0; doel = scan.nextInt())
            System.out.print(product.zoek(doel));
    }
}

```

## 1.1 Opdracht 1

Implementeer de methode `zoek` van `product` en test deze. Voor het doel 24 moet bijvoorbeeld de boven gegeven uitvoer gegenereerd worden.

## 1.2 Extra uitdaging: *dit onderdeel hoef je niet te maken*

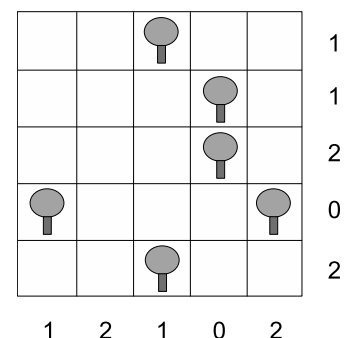
Mensen die behoefte hebben aan extra uitdaging kunnen het algoritme uitbreiden met extra operaties (optellen, aftrekken, delen).

# 2 Tentje-Boompje

De tweede backtrack opdracht is iets interessanter. Deze opdracht is een bewerking van een oude tentamenopdracht. Het grootste verschil met de tentamenopdracht is dat alle voorgegeven code toen in C++ was en hier in Java.

Een rechthoekig ( $N$  bij  $M$ ) kampeerterrein is verdeeld in  $N \times M$  standplaatsen. Op een aantal van deze plaatsen staat een boom; op de overige plekken kunnen tenten worden geplaatst. Hierbij moet echter aan de volgende 3 eisen worden voldaan:

1. Elke tent dient direct (horizontaal of verticaal) naast een boom te staan.
2. De tenten raken elkaar niet, ook niet diagonaal.
3. Het aantal tenten per rij en kolom moet exact overeenkomen met de aantallen die zijn aangegeven naast en onder het terrein. In deze opgave gaan we er van uit dat dit aantal altijd tussen de 0 en de 9 is (inclusief). Het aantal tentjes kun je dus met één cijfer weergeven.



Bekijk het gegeven terrein met afmetingen  $5 \times 5$ .













We duiden de standplaats in de linkerbovenhoek met de coördinaten  $(0,0)$  aan. Met wat redeneren kun je de puzzel vaak (gedeeltelijk) oplossen. Het kost bijvoorbeeld weinig moeite om voor het (enige) tentje op de 0<sup>de</sup> rij af te leiden dat dit in kolom 1 moet komen te staan. Immers, kolom 2 is bezet door een boom, daar mag nooit een tent staan. Regel 1 zegt dat een tent direct naast een boom moet staan.

Dit sluit kolom 0 en 4 meteen uit. We houden alleen kolom 1 en 3 als mogelijke plaats voor het tentje op de bovenste rij over. Bovendien mag er geen tentje in kolom 3 staan omdat onderaan deze kolom een 0 staat aangegeven (regel 3). Blijft alleen kolom 1 over voor het tentje op de eerste rij.

De overige tenten kunnen voor dit simpele voorbeeld ook allemaal vrij snel neergezet worden, hetgeen uiteindelijk tot de getekende toewijzing van tenten aan standplaatsen leidt.

De eenvoud van het voorbeeld is bedrieglijk: naarmate het kampeerterrein groter wordt blijkt het voor een menselijke puzzelaar stukken moeilijker te worden om een oplossing te vinden. De bedoeling van deze opdracht is om de puzzelaar met de computer te hulp te staan door een backtrack-algoritme te schrijven dat voor een gegeven puzzel de oplossingen bepaalt.

Om je wat op weg te helpen geven we wat Java-code voor die je in je oplossing mag gebruiken. Hierin wordt het terrein als een tweedimensionale rij van plaatsen gedeclareerd, waarbij iedere plaats **Gras**, **Boom** of **Tent** kan zijn. We gebruiken hiervoor het enum type **Veld**:

					1
					1
					2
					0
					2
1	2	1	0	2	

```
public enum Veld
{
    Tent, Boom, Gras;

    @Override
    public String toString()
    {
        switch (this)
        {
            case Tent: return "T";
            case Boom: return "B";
            case Gras: return ".";
        }
        return "The Java compiler needs this";
    }
}
```

De camping en het backtrack-algoritme implementeren we in de klasse **Camping**. We houden hier in de afmetingen van de camping bij (**breedte** en **lengte**), en de inhoud van de velden (matrix **veld**). Het toegestane aantal tenten per rij en per kolom wordt bijgehouden in de **int**-rijen **aantalH**, resp. **aantalV**.

De constructor **Camping(String fileName)** leest de gegevens over een camping uit een file en vult de attributen van camping. Deze constructor gooit een **IOException** als er iets fout gaat bij het lezen van de file. Op Bb kun je een implementatie vinden.

```
public class Camping
{
    private int breedte, lengte;
    private Veld [] [] veld;
    private int [] aantalH, aantalV;

    /**
     * constructor reads camping from given file
     * format:
     * lengte breedte
     * lengte lines with 'b' or 'B' for Boom and anything else for empty + count (one digit)
     * line with vertical counts (each count is one digit)
     * @param fileName
     */
    public Camping(String fileName) throws IOException
    {
        // given, see Bb
    }

    /**
```

```

    * produces a multi-line string describing the Camping
    * @return the string
    */
    @Override
    public String toString()
    {
        // given, see Bb
    }

    private boolean opCamping(int x, int y)
    {
        return 0 <= x && x < breedte && 0 <= y && y < lengte;
    }
}

```

Voor het implementeren van de regels gebruiken we een aantal *predicaten* (methodes met een boolean resultaat). Naast de gegeven methode `opCamping` zijn bijvoorbeeld `boolean naastBoom (int x, int y)` en `boolean geenBuren (int x, int y)` nuttige predicaten.

## 2.1 De methode naastBoom

Definieer de methode `naastBoom` die nagaat of aan regel 1 is voldaan, d.w.z. voor de plek met coördinaten `x` en `y` wordt gekeken of op de direct aangrenzende plaatsen horizontaal of vertikaal een boom staat.

## 2.2 De methode geenBuren

Definieer de methode `geenBuren` die nagaat of aan regel 2 is voldaan, d.w.z. voor de plek met coördinaten `x` en `y` wordt er gekeken of op de direct aangrenzende plaatsen (ook diagonaal) geen tent staat.

## 2.3 De methode plaatsTenten

Definieer de methode `plaatsTenten` die met behulp van backtracking het kampeerterrein probeert te vullen. De als parameter meegegeven coördinaten `x` en `y` geven het veld aan van waaraf verder gevuld dient te worden. Initieel wordt `plaatsTenten` dus met de coördinaten van de linkerbovenhoek (dus (0,0)) aangeroepen. Maak hier een aparte methode `plaatsTenten ()` voor.

In het recursieve geval heb je voor de aangegeven positie 2 mogelijkheden: wel of geen tent plaatsen. Uiteraard wordt dit mede bepaald door de 3 regels waaraan een oplossing moet voldoen. Zelfs als je volgens de regels een tent mag plaatsen hoeft je dat niet altijd te doen om een oplossing te vinden.

Indien gewenst mag je natuurlijk zo veel hulpmethoden en extra attributen introduceren als je nodig hebt. Het kan bijvoorbeeld handig zijn om het aantal tentjes in een rij of kolom te tellen, of te administreren.

Op Bb staan ook een aantal voorbeeld campings als txt file. Plaats deze in de root folder van je netBeans project. De gegeven main klasse zal deze allemaal proberen op te lossen.

## Extra uitdaging: *dit onderdeel hoeft je niet te maken*

Voor mensen die behoefte hebben aan extra uitdaging kan dit algoritme verfijnd worden met heuristieken. Voor de voorbeeld campings is dat niet nodig, maar voor grotere campings heb je echt last van de complexiteit van backtracking. Met slimme heuristieken kun je van een aantal plaatsen bepalen waar een tent moet komen, of juist nooit een tent moet komen. Deze heuristieken kunnen de zoekruimte aanzienlijk inperken. Zie voor ideeën de uitleg aan het begin van deze opgave.

## Inleveren van je producten

Lever vóór maandag 28 maart 8.30 uur via Blackboard jullie Java code in. Neem de uitvoer voor c4.txt als commentaar op in je programma.