Portfolio

AP – AA

9-1-2016

HAN

Remco van Alen (507276) en Bas van Summeren (479334)

# Inhoudsopgave

[Inhoudsopgave 1](#_Toc440122066)

[1 Inleiding 2](#_Toc440122067)

[2 Weekopdrachten 3](#_Toc440122068)

[2.1 Week 1 – Partition 3](#_Toc440122069)

[2.2 Week 2 – Balanced Tree 4](#_Toc440122070)

[2.3 Week 3 – Brute Force 5](#_Toc440122071)

[2.4 Week 4 – Graph algorithm 7](#_Toc440122072)

[2.5 Week 5 – Backtracking 8](#_Toc440122073)

[2.6 Week 6 – Approximation 10](#_Toc440122074)

[3 Eindopdrachten Remco 11](#_Toc440122075)

[3.1 Slotprobleem 2a – Geld Teruggeven 11](#_Toc440122076)

[3.2 Slotprobleem 10 - Anagrammen 11](#_Toc440122077)

[4 Eindopdrachten Bas 12](#_Toc440122078)

[4.1 Slotprobleem 12](#_Toc440122079)

[4.2 Slotprobleem 12](#_Toc440122080)

# Inleiding

# Weekopdrachten

## Week 1 – Partition

### De opdracht

### De uitvoering

### Het resultaat

## Week 2 – Balanced Tree

### De opdracht

Voor week 2 moest een boom structuur geprogrammeerd worden om data in op te slaan. De opdracht omschrijft een gebalanceerde boom.  
In deze opdracht is gekozen om een AVL-Tree op te bouwen. Deze Tree is een boom structuur die zelf balancerend is. Dit wil zeggen dat aan alle kanten van de boom, ongeveer evenveel elementen hangen. Hierdoor kunnen functies in O(log n) tijd worden uitgevoerd.

### De uitvoering

De boom moet een aantal taken kunnen uitvoeren deze zijn hieronder geschreven in de volgorde van uitvoeren:

#### Zoeken

Het opzoeken van een sleutel is nodig om de gegevens op te halen. Echter kan deze ook gebruikt worden om de locatie voor een nieuw object uit te rekenen. Hiervoor is de functie:

**protected** AVLNode<K, V> getNextNode(K key)  
{  
 **int** comparison = **value**.compareTo(key);  
 **if** (comparison > 0)  
 {  
 **return leftValue**;  
 } **else if** (comparison < 0)  
 {  
 **return rightValue**;  
 } **else** {  
 **return this**;  
 }  
}

Deze functie vergelijkt de inkomende sleutel tegen de eigen waarde. Wanneer de sleutel kleiner is wordt er doorverwezen naar de linker sub-boom. Wanneer deze groter is zal doorverwezen worden naar de rechter sub-boom. Door deze functie herhaaldelijk uit te voeren kunnen 2 resultaten ontstaan:

* Er wordt null geretourneerd. De laatst gevonden node heeft geen kind op de kant waarna verwezen word. Wanneer er gezocht wordt naar een sleutel betekend dit dat de sleutel niet aanwezig is. Wanneer er een locatie wordt gezocht voor een nieuw element, zal dit onder de laatst gevonden node zijn.
* Er wordt meerdere malen dezelfde node geretourneerd. De laatst gevonden node heeft de sleutel in zich. Bij het opzoeken van een sleutel kan deze geretourneerd worden. Wanneer er ruimte wordt gezocht om nieuwe gegevens te plaatsen zal deze nieuwe data in de laatst gevonden node geplaatst worden. Dit voorkomt dubbele sleutels

#### Insertie

Om de boom bruikbaar te maken moeten er gegevens in geplaatst kunnen worden. Om gedaan te krijgen is een functie die met de bovenstaande functie de laatste node zoekt. Wanneer de gevonden node de sleutel bevat zal deze worden overschreven.

Wanneer de sleutel niet is gevonden zal een nieuwe node worden toegevoegd aan de laatst gevonden node. Hierna wordt de boom gebalanceerd.

#### Balanceren

Om te balanceren moet eerst uitgerekend worden welke kant op gebalanceerd moet worden. Hiervoor worden de hoeveelheid elementen aan beide kanten vergleken. De balanceer factor wordt uitgerekend met: (“Elementen links” – “Elementen rechts”). Wanneer deze waarde groter dan 1 is, zal de linkerzijde zwaarder zijn, en wanneer deze negatief is zal de rechterzijde zwaarder zijn.

De maximale verschil in balans dat een AVL boom mag hebben is 1. Dit wil zeggen dat wanneer de linker of rechterzijde van een node 2 elementen meer heeft dan de andere zijde, zal deze gebalanceerd moeten worden.

Dit is gedaan in de volgende functie:

**public void** rebalance()  
{**while** (getRoot().topDownBalance()) ;}

**private boolean** topDownBalance(){  
 **int** balanceFactor = balanceFactor();  
 **boolean** found = **false**;  
 **if** (balanceFactor > 1){  
 rotateRight();  
 **return true**;  
 } **else if** (balanceFactor < - 1){  
 rotateLeft();  
 **return true**;  
 }**if** (**leftValue** != **null**){  
 found = found || **leftValue**.topDownBalance();  
 }**if** (**rightValue** != **null**) {  
 found = found || **rightValue**.topDownBalance();  
 }  
 **return** found;  
}

Hiermee wordt vanuit de root geoptimaliseerd om eerst de gehele boom te balanceren om vervolgens steeds verder naar onder te werken. Ook zal de functie doorgaan totdat het geheel in balans is.

Hierbij wordt de balansfactor vergeleken en gecontroleerd of er een linker of rechter rotatie uitgevoerd moet worden.

Een rotatie wordt uitgevoerd met de code:

**private void** rotateLeft()  
{  
 AVLNode<K, V> oldRight = **this**.**rightValue**;  
 **if** (oldRight != **null**)  
 {  
 AVLNode<K, V> oldRightsLeft = oldRight.**leftValue**;  
 **if** (oldRightsLeft != **null**)  
 {  
  
 oldRight.rotateRight();  
 oldRight = **this**.**rightValue**;  
 oldRightsLeft = oldRight.**leftValue**;  
 }  
 **rightValue** = oldRightsLeft;  
 **if** (**rightValue** != **null**)  
 {  
 **rightValue**.**parent** = **this**;  
 }  
 **if** (**leftValue** != **null**)  
 {  
 **leftValue**.**parent** = **this**;  
 }  
 oldRight.**leftValue** = **this**;  
 oldRight.**parent** = **this**.**parent**;  
 **if** (**parent** != **null**)  
 {  
 **if** (**parent**.**leftValue** != **null** && equals(**parent**.**leftValue**))  
 {  
 **parent**.**leftValue** = oldRight;  
 } **else if** (**parent**.**rightValue** != **null** && equals(**parent**.**rightValue**))  
 {  
 **parent**.**rightValue** = oldRight;  
 }  
 }  
 **parent** = oldRight;  
 }  
}

Hierbij wordt eerst gecontroleerd of de te balanceren node ook elementen onder zich heeft, wat zal resulteren in een dubbele rotatie. Vervolgens worden de posities omgedraaid en de kinderen en ouders verwisseld.

### Het resultaat

Na insertie van 65, 32 en 54:

Current tree:

- 54 : "54"

- 32 : "32"

- 65 : "65"

Na insertie van: 5 en 8:

- 32 : "32"

- 5 : "5"

- 8 : "8"

- 54 : "54"

- 65 : "65"

Na insertie van: 123, 456 en 323

- 54 : "54"

- 8 : "8"

- 5 : "5"

- 32 : "32"

- 123 : "123"

- 65 : "65"

- 456 : "456"

- 323 : "323"

Na insertie van: 243, 165

- 65 : "65"

- 8 : "8"

- 5 : "5"

- 54 : "54"

- 32 : "32"

- 243 : "243"

- 123 : "123"

- 165 : "165"

- 323 : "323"

- 456 : "456"

## Week 3 – Brute Force

### De opdracht

De gekozen opgave is “Sudoku problem by brute force”. Dit houdt in dat een sudoku doormiddel van het brute force algorithme moet worden opgelost.

### De uitvoering

Voor het uitvoeren van deze opdracht is eerst het spelbord gerealiseerd. 9 rijen van 9 vakjes, 9 kolommen van 9 vakjes en 9 blokken van 3x3 vakjes. Door het gebruikt van pointers worden er in totaal maar 81 vakjes gemaakt. Dit geeft het onderstaande resultaat:



Daarna zijn de spelregels toegepast. Er is een functie geschreven die controleert of een bepaald getal al voorkomt in een bepaald rij. Dit is ook gedaan voor kolommen en blokken van 3x3.

Het algoritme bevat geen intelligentie. Er wordt eerst op het eerste vakje een ‘1’ geplaatst. Als dit vakje al vooraf is ingevuld zal deze worden overgeslagen. Anders wordt er gecontroleerd of deze ‘1’ aan de spelregels voldoet. Als dit een valide set is wordt de ‘1’ geplaatst en anders wordt het volgende getal geprobeerd. Net zo lang totdat het bord vol is. Get kan natuurlijk dat dit niet uitkomt, bijvoorbeeld als er geen valide mogelijkheid is voor een leeg vakje. Dan wordt er doormiddel van backtracking een stap terug gedaan, omdat alle volgende mogelijkheden kunnen worden uitgesloten van een goede oplossing.

Wanneer er een zet terug gedaan zal dat nummer worden opgehoogd. Als dit tot en met het getal ‘9’ ook niet meer kan wordt er nog een stap terug gezet totdat er een getal kan worden verandert. Daarna zal de richting weer vooruitgaan. Wanneer er bij het eerste vakje geen mogelijkheid meer is kan er worden geconcludeerd dat de sudoku niet kan worden opgelost.

### Het resultaat

Dit is de opgeloste sudoku:



Het algoritme:

Er is geen gebruik gemaakt van recursie, maar wanneer dit wel was toegepast had dit tot forse vermindering van code kunnen leiden.



**for** (**int** rowI = 0; rowI < size; rowI++) *// Per rij*  
{  
 **for** (**int** columnI = 0; columnI < size; columnI++) *// Per kolom*  
 {  
 **if** (! bord.isDefaultNumber(rowI, columnI)) *// Wanneer er geen default getal is ingevuld*  
 {  
 hasNewValue = **false**;  
 goBack = **false**;  
 **if** (bord.getValue(rowI, columnI) != **null**)  
 {  
 oldValue = bord.getValue(rowI, columnI);  
 } **else** {  
 oldValue = 0;  
 }  
 **while** (hasNewValue == **false**) *// Zolang er geen nieuwe waarde is*  
 {  
 counterSteps ++;  
 **if** (oldValue < 9) *// Als het huidige getal nog niet maximaal is*  
 {  
 newValue = oldValue + 1;  
 **if** (bord.isValidValue(rowI, columnI, newValue)) *// Controle van zet* {

bord.setValue(rowI, columnI, newValue, **false**);  
 hasNewValue = **true**;  
  
 } **else** {  
 oldValue = newValue;  
 }  
 }**else if** (columnI >= 1) *Wanneer het niet de eerste kolom is*  
 {  
 bord.setValue(rowI, columnI, 0, **false**);  
 columnI = columnI - 2;  
 goBack = **true**;  
 **break**;  
 } **else if** (rowI >= 1) *Wanneer het niet de eerste rij is*  
 {  
 bord.setValue(rowI, columnI, 0, **false**);  
 rowI = rowI - 1;  
 columnI = size - 2;  
 goBack = **true**;  
 **break**;  
 }  
 }  
 } **else if** (goBack == **true**) *// Als er een zet achteruit wordt gemaakt*  
 {  
 **if** (columnI >= 1) *// Wanneer het niet de eerste kolom is*  
 {  
 columnI = columnI - 2; *// -2 omdat de for-loop dit weer met 1 ophoogt*  
 goBack = **true**;  
 } **else if** (rowI >= 1) *// Wanneer het niet de eerste rij is*  
 {  
 rowI = rowI - 1;  
 columnI = size - 2; *-2 omdat de for-loop dit weer met 1 ophoogt*  
 goBack = **true**;  
 } **else**{ *// Als het de eerste kolom van de eerste rij is*  
 rowI = size;  
 columnI = size;  
 System.***out***.println(**"This puzzle cannot be solved!!"**);  
 **break**;  
 }

## Week 4 – Graph algorithm

### De opdracht

Voor week 4 moest een probleem worden opgelost worden met een Greedy Algortime. Hiervoor is gekozen om de minimum spanning tree op te lossen.

De minimum spanning tree is een graaf probleem. Hierin moeten alle punten verbonden worden met de minste afstand op de Edges tussen de vertices. Om dit probleem op te lossen is gekozen voor de Reverse-Delete methode. Deze methode is de omgekeerde van de Kruskal Methode.

De Kruskal methode voegt de kortst mogelijke edges toe aan de oplossing zonder een circulaire constructie te maken. Reverse-Delete verwijderd de langst mogelijke edges zonder de punten los te koppelen van elkaar. Het voordeel van deze methode is dat deze ook correct zal werken voor al losgekoppelde onderdelen van een graaf, wat zal resulteren in een Minimum Spanning Forest.

### De uitvoering

De eerste stap van deze reverse-delete methode is de edges sorteren op basis van hun lengte. De langste edge’s zullen als eerste verwijderd moeten worden. Bij een snel sorteer algoritme kan dit uitgevoerd worden in O(E log E), waarbij E voor de hoeveelheid edges staat. E is maximaal V² groot, waarbij V de hoeveelheid Vertices is.

De belangrijkste stap is het controleren of een Edge verwijderd mag worden. Hiervoor is de volgende functie geïmplementeerd:

**boolean** safeToDelete(Edge toRemove)  
{  
 **if** (toRemove.getFirst().amountOfEdges() <= 1 ||

toRemove.getSecond().amountOfEdges() <= 1)  
 {  
 **return false**;  
 } **else if** (hasPathWithout(toRemove.getFirst(), toRemove.getSecond(),   
 toRemove))  
 {  
 **return true**;  
 }  
 **else** {  
 **return false**;  
 }  
}

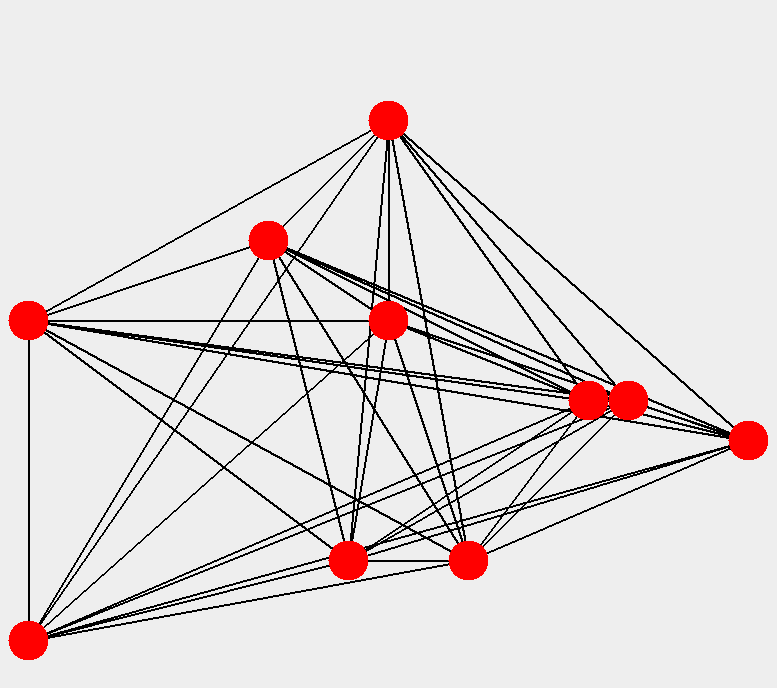
Hierin wordt vooraf gecontroleerd of dit de enige edge is die aan de Vertices verbonden zijn. In het geval dat een van de twee Vertcies maar een enkele Edge heeft mag de Edge niet verwijderd worden. Vervolgens moet gecontroleerd worden of er een ander pad tussen de locaties bestaat. Het controleren van het pad gebeurt met behulp van een greedy Nearest neighbor algoritme. Door vooraf de edges van een Node te sorteren op lengte zal dit sneller verlopen.

Deze controle wordt uitgevoerd voor iedere Edge die beschikbaar is. Hiermee behaald deze stap een O(E) in complexiteit. Wanneer iedere Edge behandeld is zal het algoritme klaar zijn.

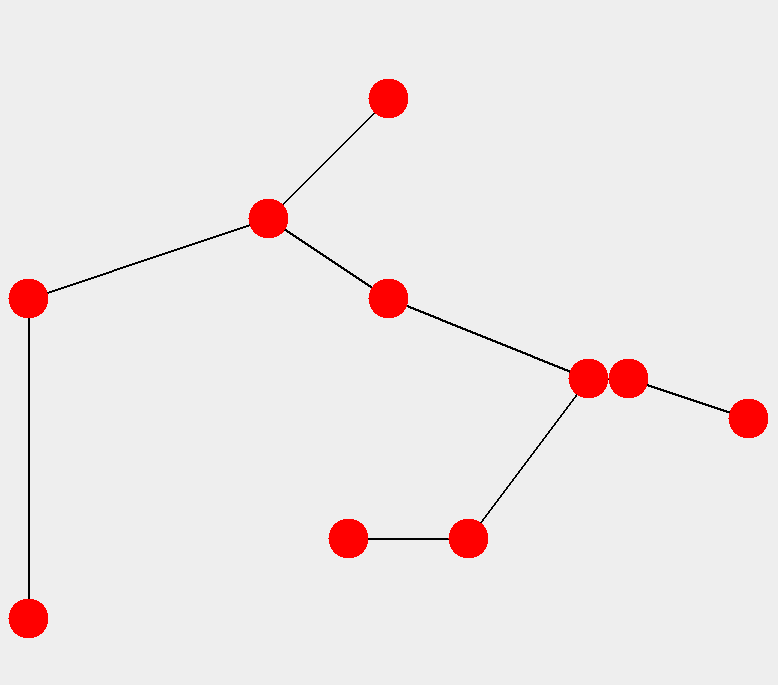
Dit kan gecontroleerd worden door de bovenstaande functie te herhalen voor alle overgebleven edges.

### Het resultaat

Voor:



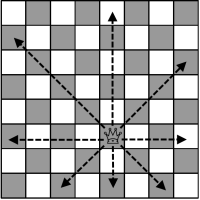
Na:



## Week 5 – Backtracking

### De opdracht

De gekozen opgave is “n-Queens-problem”. Voor deze puzzel worden er N queens om een schaakbord geplaatst van N\*N groot waarbij de queens elkaar niet kunnen slaan. Op de onderstaande afbeelding staat een queen met de mogelijk staprichtingen weergegeven.



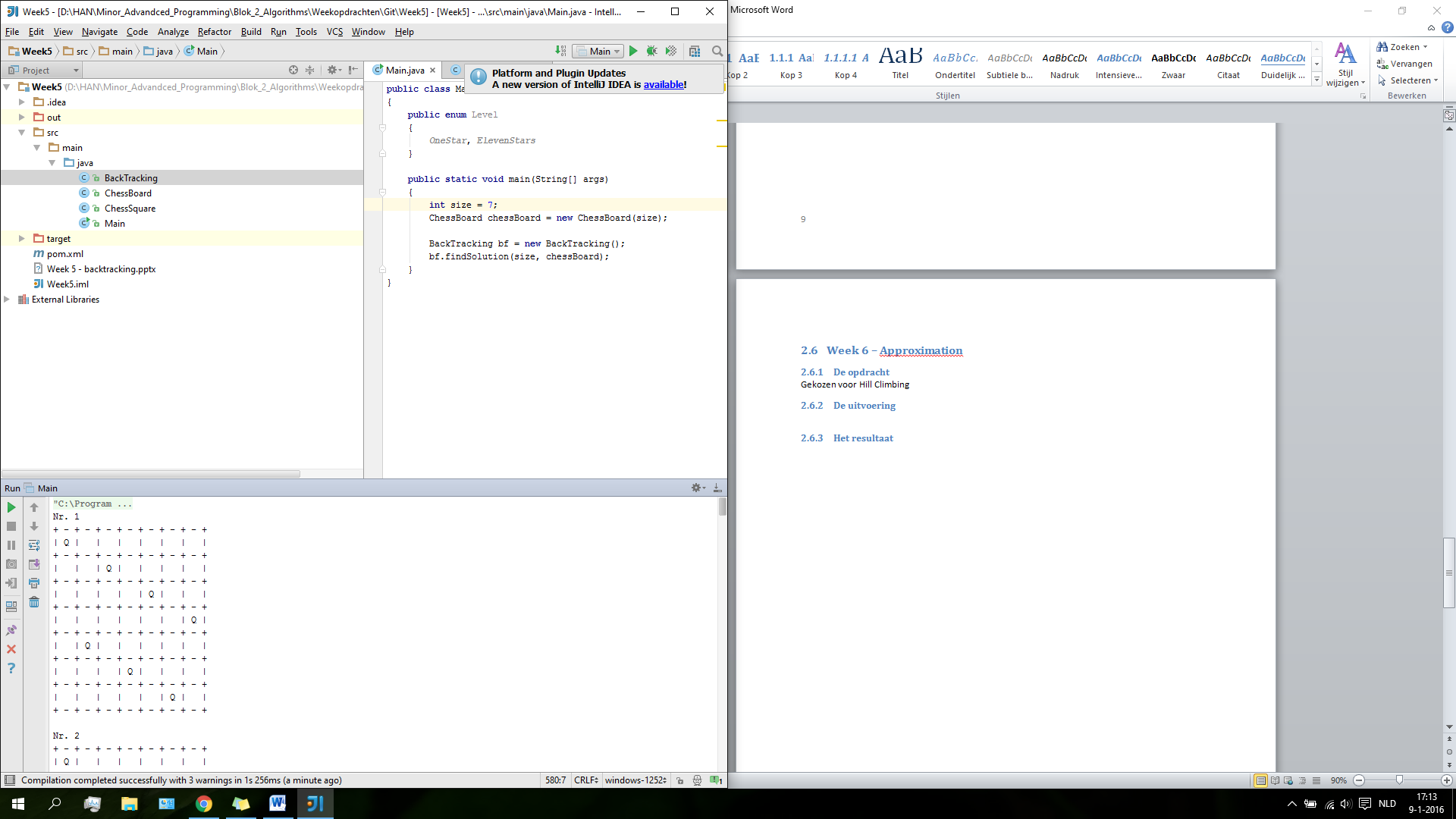
### De uitvoering

Er is gebruik gemaakt van de sudoku opdracht. Daardoor was het spelbord al af. De spelregels zijn wel aangepast. Er moet nog steeds een horizontale en verticale controle plaatsen vinden, maar niet een blok van 3 x 3. Daarentegen komt er wel een extra controle voor de diagonalen.

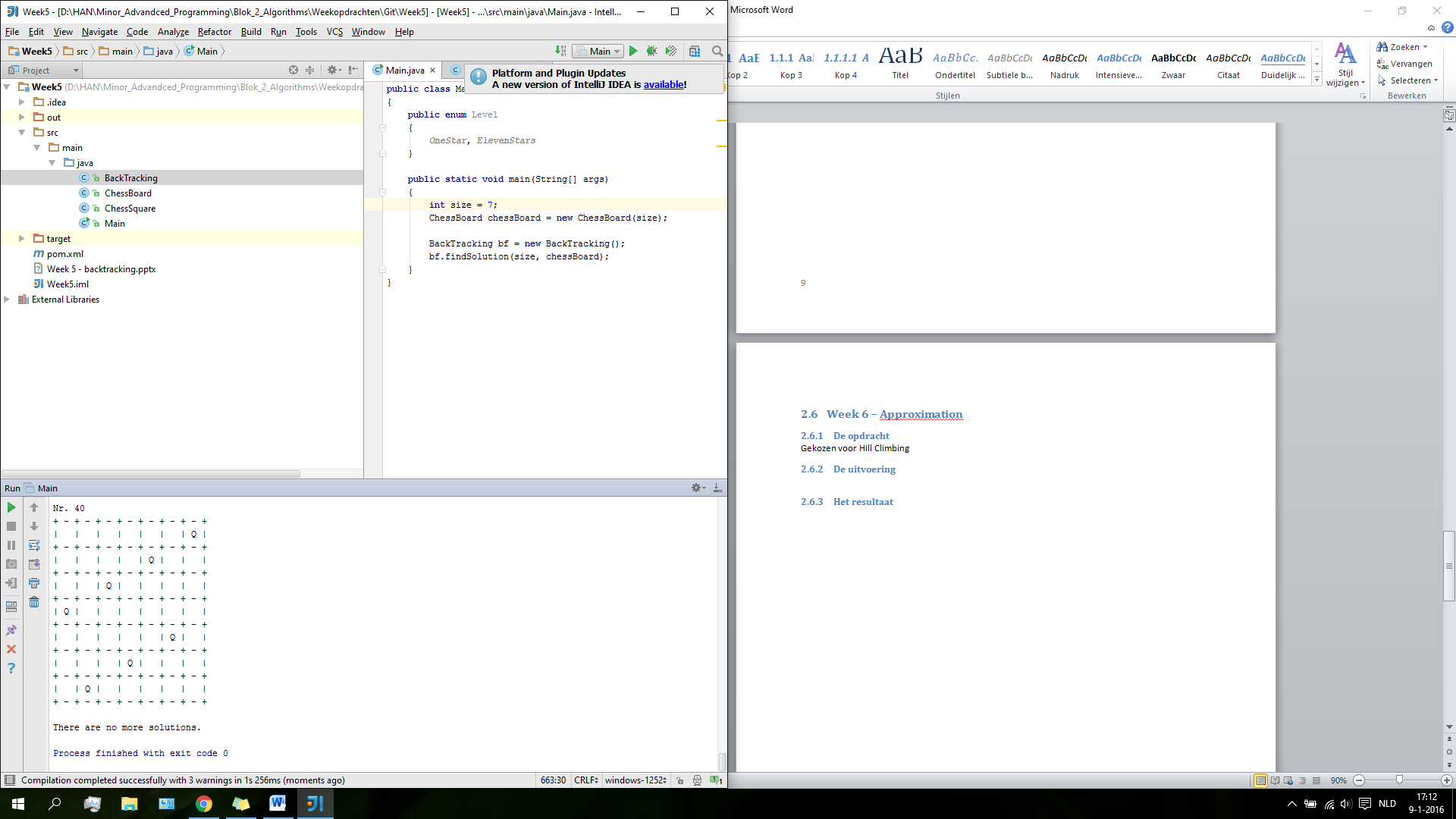
Wanneer er een queen wordt geplaatst wordt de rij niet afgemaakt, maar wordt er gelijk naar de volgende rij gekeken. Als bij het laatst vakje blijkt dat de rij geen queen bevat zal backtracking worden toegepast. Dit houdt in dat er terug wordt gekeken en dat de laatst geplaatste queen wordt weggehaald. Daarna zal er ergens anders op die rij alsnog een queen worden geplaatst. Als er nergens op die rij een nieuwe queen kan worden geplaatst zal er nog een stap terug worden gezet. Het kan voorkomen dat er bij het eerste vakje van de eerste kolom een stap terug met worden gezet en dan kan geconcludeerd worden dat de puzzel niet op telossen is. Dit kan alleen voorkomen bij een spelbord van 2x2 en 3x3 groot.

### Het resultaat

Dit algoritme gaat door totdat er geen andere oplossingen meer zijn. Bij een veld van 7 x 7 worden er 40 oplossingen gevonden.



…



Het algoritme:

Ook hier is er geen gebruik gemaakt van recursie en dit had wel tot forse vermindering van code kunnen leiden.

**for** (**int** rowI = 0; rowI < size; rowI++)  
{  
 **for** (**int** columnI = 0; columnI < size; columnI++)  
 {  
 **if** (!goBack && bord.isValidValue(rowI, columnI, **true**))  
 {  
 bord.setQueenValue(rowI, columnI, **true**);  
 goBack = **false**;   
 }  
 **else if**(goBack && bord.isQueen(rowI, columnI))  
 {  
 bord.setQueenValue(rowI, columnI, **false**);  
 goBack = **false**;  
 }  
 **if**(columnI >= size - 1)  
 {  
 **if**(!bord.rowContainsQueen(rowI))  
 {

goBack = **true**;  
 }  
 }  
 **if**(goBack)  
 {  
 ….

}  
 }  
}

## Week 6 – Approximation

### De opdracht

Gekozen voor Hill Climbing

### De uitvoering

### Het resultaat

# Eindopdrachten Remco

## Slotprobleem 2a – Geld Teruggeven

## Slotprobleem 10 - Anagrammen

# Eindopdrachten Bas van Summeren

Als eindopdracht moesten 2 slotproblemen gemaakt worden. Hieronder zijn de problemen die gekozen zijn omschreven. Daarbij is ook de redernatie en het proces weergegeven.

## Slotprobleem 2a: Geld Teruggeven

De eerste gekozen opdracht is het teruggeven van wisselgeld. Dit is een veel voorkomend probleem dat nog dagelijks gebruikt wordt.

De opdracht omschreef:

*“Bij een parkeerautomaat of treinkaartjesautomaat kun je betalen met contant geld (munten). Bij sommige automaten hoef je niet met gepast geld te betalen. Deze automaten kunnen wisselgeld teruggeven.*

*Geef een algoritme voor de geldteruggave. Het algoritme heeft als invoer het bedrag van het wisselgeld en bepaalt hoeveel van welke soort munten wordt teruggegeven; liefst worden zo weinig mogelijk munten teruggeven.*

*Ga in eerste instantie uit van de muntwaarden van de bekende euromunten. Je kunt een hoger cijfer verdienen met een algoritme dat ook werkt met de munten van de oude gulden (1 cent, 5 cent, 10 cent, 25 cent, 1 gulden, 2½ gulden).”*

Om aan te passen aan de mogelijke muntstukken moesten aanvullende ontwerp stappen worden genomen echter zal het geen verandering zijn voor het algoritme.

Wel moest rekening gehouden worden met de snelheid en eenvoudigheid van het algoritme.

Ten eerste moest gekeken worden naar de representatie van de waarden. Om geen problemen te krijgen met decimalen zijn alle bedragen vertegenwoordigd in centen. Zo zal €0.05 voor het systeem gezien worden als 5. Dit zal een aantal berekeningen eenvoudiger maken.

Muntstukken zijn bekend aan hun waarde binnen een Valuta(Currency). Om de hoeveelheid duidelijk te koppelen en als bruikbare gegevens terug te brengen is gekozen voor een lijst van KeyValuePairs(Map). Hierin is de Key(sleutel) de waarde van het muntstuk en de Value(waarde) de hoeveelheid van deze munt die terugontvangen dient te worden.

De waarden van de muntstukken zijn uniek binnen een valuta. Hierdoor kan een Valuta weergegeven worden met een Set: een lijst van unieke waarden.

Echter zal het voor verdere berekeningen ook gemakkelijk zijn als de muntstukken aflopend geordend zijn, namelijk met de hoogste waarde eerst.

Om aan deze eisen te voldoen is de volgende representatie gekozen voor een valuta:

**public interface** Currency  
{**public static final** Comparator<Integer> ***CURRENCY\_COMPARATOR*** = **new** Comparator<Integer>()  
 {  
 **public int** compare(Integer first, Integer second)  
 {  
 **return** second - first;  
 }  
 };  
  
SortedSet<Integer> getCoins();  
}

Hierin staat de “CURRENCY\_COMPARATOR” als vergelijkingshulpmiddel om een lijst te sorteren met de hoogste waarde eerst. De valuta heeft ook een functie die zijn unieke waarden in een gesorteerde representatie terug zal geven.

Vervolgens zijn de verschillende valuta gerealiseerd. De gerealiseerde valuta zijn:

* Euro
* Euro exclusief 1 en 2 cent
* Gulden
* Gulden inclusief de speciale uitgave van 10 en 50 gulden munten.
* Euro inclusief biljetten

Deze zullen vrij verwisselbaar zijn zolang ze voldoen aan de interface.

Vervolgens moet er een algoritme gemaakt worden om het wisselgeld uit te rekenen. De meest eenvoudige is door vanaf de grootste waarde de maximale hoeveelheid munten van het totaal af te halen. Dat wil zeggen dat bij bijvoorbeeld 4.30 wisselen, eerst 2 maal 2 verwijderd moet worden. Door dit te herhalen zal bekend worden welke munten nodig zijn.

Dit is weergegeven in de volgende code:

Map<Integer, Integer> returnCoins = **new** LinkedHashMap<Integer, Integer>();  
**for** (Integer currentValue : **currentCurrency**.getCoins())  
{  
 **int** numberOfCoins = amount/((**int**)currentValue);  
 returnCoins.put(currentValue, numberOfCoins);  
 amount-=numberOfCoins\*((**int**)currentValue);  
}

Hierin word de gekozen teruggave structuur aangemaakt. Vervolgens wordt voor ieder muntstuk in de huidige valuta gekeken hoe vaak het in het bedrag past. Deze waarden wordt opgeslagen in de teruggave Map. Ten slotte worden de munten van de waarden afgetrokken en zal verder gewerkt worden met de resterende waarde. Dit zal herhaald worden voor alle muntstukken. Hiermee is de complexiteit van deze code O(c) waarin c voor het aantal muntstukken in de valuta staat. Het bedrag heeft weinig invloed op de uitvoertijd van de code.

Echter is er tot op dit punt geen rekening gehouden met de restwaarde die kan overblijven. Dit geeft problemen met het feit dat tegenwoordig de 1 en 2 euro cent munten niet vaak meer gebruikt worden. Hierdoor komt een afrondingsprobleem. Om dit probleem op te vangen is de volgende code toegepast:

**int** lastValue = **currentCurrency**.getCoins().last();  
**if**(amount>lastValue/2)  
{  
 returnCoins.put(lastValue, returnCoins.get(lastValue)+1);  
}

Als het restbedrag groter is dan de helft van de waarde van het laatste muntstuk, zal er een meer van dit muntstuk worden toegevoegd aan de collectie. Hiermee is de laatste waarde netjes afgerond naar zowel boven en onder, wanneer dit nodig is. Deze complexiteit hiervan is O(1).

Hiermee kan ik eenvoudig en snel de teruggaven van vele valuta berekenen in een complexiteit van O(c), waarin c de hoeveelheid munten in de valuta is.

## Slotprobleem 10: Anagrammen