# Leertaak 3: Chris de Windt & Serkan Orhan

## Opdracht 1

Recursie werkt doordat een methode zichzelf aanroept en bij iedere aanroep wordt het probleem gesplits in subproblemen totdat de subproblemen simple genoeg zijn om op te lossen. Twee voorwaarden bij recursie zijn:

* Er moet een stop criteria zijn
* De probleem moet opgesplits worden in kleinere subproblemen

## Opdracht 2

Het algemene backtraicking algoritme voldoet aan de twee voorwaarden omdat je een stop criteria hebt, namelijk “if solution complete”. De andere voorwaarde is dat de problemen steeds kleiner wordt. Dit wordt gedaan door “move candidate from candidates to solution”.

## Opdracht 3

- Wat zijn je kandidaten?

De speelkaarten   
- Hoe bewaar je de kandidaten, in welke datastructuur?

In een stack  
- Hoe heb je het veld gemodelleerd (datastructuur)?

Met een multidimensional array   
- Hoe herstel je het veld als de recursieve aanroep returnt?

Door null in het veld te platen

## Opdracht 4

Zie leertaak3Backtracking

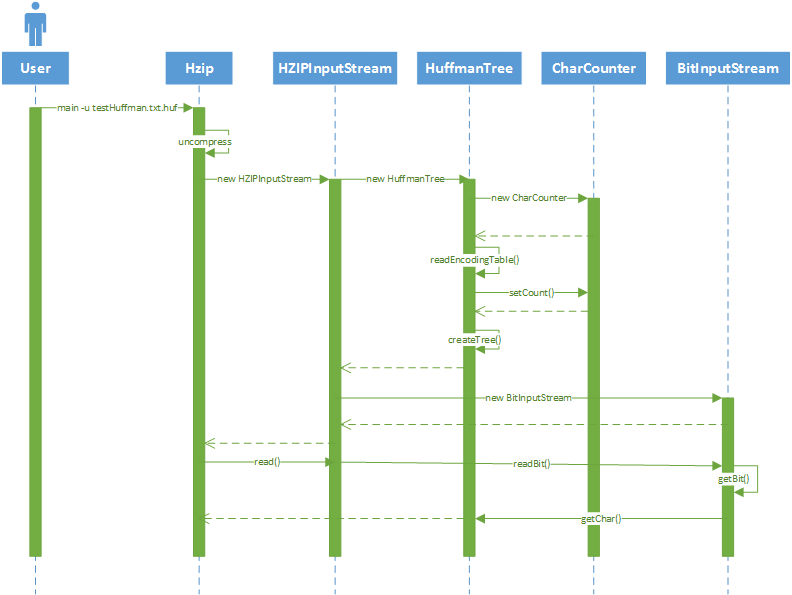
## Opdracht 5

Zie leertaak3Backtracking

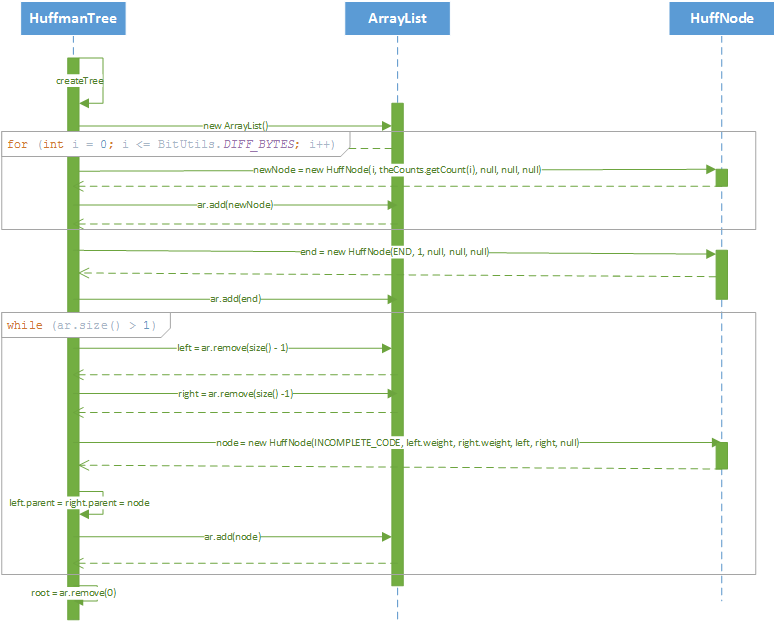
## Opdracht 15

|  |
| --- |
| public class TestHuffman extends TestCase {   public void testHuffman(){   File original = new File("data/TestA.dat");  File uncompressed = new File("data/TestA.dat.uc");  *assertEquals*(fileToString(original), fileToString(uncompressed));  }    private String fileToString(File file){  String string = null;  try {  BufferedReader br = new BufferedReader(new FileReader(file));  StringBuilder sb = new StringBuilder();  String line = br.readLine();   while (line != null) {  sb.append(line);  sb.append(System.*lineSeparator*());  line = br.readLine();  }  string = sb.toString();  br.close();  } catch (Exception ex) {  ex.printStackTrace();   }  return string;  } } |

## Opdracht 16



## Opdracht 17



## Opdracht 18

getChart(String code)

|  |
| --- |
| public int getChar(String code) {   HuffNode node = root;  for (int i = 0; node != null && i < code.length(); i++)  if (code.charAt(i) == '0')  node = node.left;  else  node = node.right;   if (node == null)  return *ERROR*;   return node.value; } |

createTree()

|  |
| --- |
| private void createTree() {  ArrayList<HuffNode> ar = new ArrayList<HuffNode>();   for (int i = 0; i <= BitUtils.*DIFF\_BYTES*; i++)  if (theCounts.getCount(i) > 0) {  HuffNode newNode = new HuffNode(i, theCounts.getCount(i), null, null, null);  theNodes[i] = newNode;  ar.add(newNode);  }   theNodes[*END*] = new HuffNode(*END*, 1, null, null, null);  ar.add(theNodes[*END*]);   while (ar.size() > 1) {  HuffNode left = ar.remove(ar.size() - 1);  HuffNode right = ar.remove(ar.size() - 1);   HuffNode node = new HuffNode(*INCOMPLETE\_CODE*, left.weight + right.weight, left, right, null);   left.parent = right.parent = node;   ar.add(node);  }   root = ar.remove(0); } |

## Opdracht 19

De unittest uit de problem analysis slaagt maar de gegeven unittesten werken niet.

## Opdracht 20

Ongeveer 38%. Bij sommige files heb je zelfs verlies

## Opdracht 21

De groote van de file. Als het niet een groot genoeg file is dan heeft het niet veel zin om de file te comprimeren. Hoe vaak de letter voorkomen speelt ook een rol. Als de letters ongeveer even vaak voorkomen, heb je weinig winst