Hogeschool van Amsterdam

Sorting & Searching Practicum 3

Guus Portegies	**IS203**	**500691990**
Koen Karsten	**IS203**	**500695520**

1. Inhoudsopgave

2. Inleiding

In dit derde practicum verslag voor Sorting & Searching gaan we onderzoek doen naar **Substring Search** en **Reguliere Expressies**. Voor het eerste onderwerp zullen we de source code induiken van twee verschillende programmeertalen, en hier vervolgens een methode in zoeken die het Substring algoritme toepast. Vervolgens zullen we gaan bekijken welke variant van het algoritme dit betreft, en hier zullen we ook wat uitleg bij geven over de werking van de methode. Ook gaan we zelf twee algoritmen implementeren die je (onder andere) kunt gebruiken voor het zoeken naar Strings in een tekst: **Knuth-Morris-Pratt** & **Boyer-Moore.** Deze methoden zullen we na het implementeren gaan testen en de prestaties die hieruit voortkomen gaan we vergelijken.

We gaan ook zelf aan de slag met reguliere expressies, hier gaan we een klasse mee schrijven die ons kan helpen bij het verifiëren van een telefoonnummer. Dit vonden we zelf een erg lastig onderdeel van de opdracht vanwege de vele regels die voor een geldig telefoonnummer gelden, maar uiteindelijk hebben we hier toch veel van geleerd en zijn we trots op het resultaat!

3. Substring implementatie in programmeertalen

Voor deze opdracht gaan we kijken naar de implementatie van de substring methode in verschillende programmeertalen. De eerste programmeertaal waarvan we de substring methode bekijken is Java, als tweede programmeertaal hebben wij C++ gekozen. De substring methoden in deze talen heten valueOf() & search().

Om te achterhalen hoe de substring methoden van deze talen werken bekijken wij de source code hiervan. De resultaten van ons onderzoek volgen in de volgende twee paragraven.

3.1 Substring implementatie in Java

Als eerste gaan de we indexOf() van Java bekijken, de hebben de source code hiervan gevonden op: http://www.docjar.com/html/api/java/lang/String.java.html

```
static int indexOf(char[] source, int sourceOffset, int sourceCount, char[]
target, int targetOffset, int targetCount, int fromIndex) {
    if (fromIndex >= sourceCount) {
        return (targetCount == 0 ? sourceCount : -1);
    }
    if (fromIndex < 0) {
        fromIndex = 0;
    }
    if (targetCount == 0) {
        return fromIndex;
    }
    char first = target[targetOffset];
    int max = sourceOffset + (sourceCount - targetCount);

    for (int i = sourceOffset + fromIndex; i <= max; i++) {
        if (source[i] != first) {
          while (++i <= max && source[i] != first);
        }
        /* Found first character, now look at the rest of v2 */
        if (i <= max) {
          int j = i + 1;
          int end = j + targetCount - 1;
          for (int k = targetOffset+1; j < end && source[j] == target[k]; j++, k++);
          if (j == end) {
            /* Found whole string. */
            return i - sourceOffset;
          }
        }
    }
    return -1;
}
```

Het algoritme dat hier wordt toegepast betreft de brute-force versie. Dit houdt in dat de methode bij ieder woord eerst controleert of het eerste karakter overeenkomt met het eerste karakter van het gezochte woord. Wanneer dit het geval is gaat het algoritme verder met het tweede karakter, dit proces blijft zich herhalen totdat het hele woord gevonden is of totdat er een letter niet overeenkomt. Als er een letter niet overeenkomt met het gezochte karakter dan gaat het algoritme het volgende woord controleren. Wanneer er geen woorden meer te controleren zijn dan returnt de methode -1. Wanneer de gezochte String gevonden is wordt de index hiervan ge-returnt.

3.2 Substring implementatie in C++

Nu gaan we de search() methode van C++ onderzoeken, onze bron hiervoor is:
http://www.cplusplus.com/reference/algorithm/search/

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 search ( ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2)
  {
    if (first2==last2) return first1;  // specified in C++11
      while (first1!=last1) {
        ForwardIterator1 it1 = first1;
        ForwardIterator2 it2 = first2;
        while (*it1==*it2) {    // or: while (pred(*it1,*it2)) for version 2

          ++it1; ++it2;
          if (it2==last2) return first1;
          if (it1==last1) return last1;
        }
        ++first1;
    }
    return last1;
  }
```

Dit betreft ook een brute-force implementatie van het substring algoritme, alleen worden hier direct de eerste twee letters vergeleken i.p.v. enkel de eerste. Wanneer deze implementatie een match vindt dan wordt de index hiervan ge-returnt. C++ biedt ook een alternatief aan dat juist de laatste match returnt: find_end() .

4. Vergelijken van algoritmes KMP & BM

4.1 Knuth-Morris-Pratt

De uitwerking van dit algoritme hebben we gebaseerd op de uitwerking die te vinden is in het boek *Algorithms - Sedgewick & Wayne, 2011*. Wel hebben we deze uitwerking aangepast naar onze eigen wensen. Zo returnt de klasse nu het aantal keer dat een substring is gevonden, in plaats van de index van de eerste match. Ook wordt het aantal vergelijkingen bijgehouden dat nodig is om de hele tekst te doorlopen. Onze implementatie van het Knuth-Morris-Pratt algoritme volgt nu:

```
private int[][] dfa;
private int matches;
private int comparisons;
final private String haystack;
private String needle;

public KMP(String haystack) {
    this.haystack = haystack; // Set the haystak for future reference
}

public int search(int position) {
    int i, j, N = haystack.length(), M = needle.length();
    for (i = position, j = 0; i < N && j < M; i++) {
        comparisons++; // Increment the amount of comparisons
        j = dfa[haystack.charAt(i)][j];
    }
    if (j == M) {
        return i - M; // found (hit end of pattern)
    } else {
        return N; // not found (hit end of text)
    }
}
public int searchForWord(String word) {
    // Step 1: Initialize some default values for future use
    this.matches = 0;
    this.comparisons = 0;
    this.needle = word; // The word were looking for
    int M = needle.length(); // Used for detecting end of needle (succes)
    int R = haystack.length(); // Used for detecting haystack end (failure)

    // Step 2: Initialize and alter dfa Array for use in Search() method
    dfa = new int[R][M];
    dfa[needle.charAt(0)][0] = 1;
    for (int X = 0, j = 1; j < M; j++) {  // Compute dfa[][j].
        for (int c = 0; c < R; c++) {
            dfa[c][j] = dfa[c][X];
        }
        dfa[needle.charAt(j)][j] = j + 1;
        X = dfa[needle.charAt(j)][X];

    }

    // Step 3: Start looking for the substring by looping the haystack
    int position = 0;
    for (int i = 0; i < haystack.length(); i++) {
        position = search(position + 1);
        if (position >= haystack.length()) {
            return this.matches;
        } else {
            matches++;
        }
    }

    // Return the amount of matches
    return this.getMatches();
}
```

Wanner je onze KMP klasse aanroept dien je direct de String mee te geven waarin je de Substring wilt gaan zoeken, deze schrijven we dan weg in de variabele *haystack*.

Vervolgens kun je met behulp van de methode *seachForWord(String word)* een woord gaan zoeken in de haystack. Hier worden eerst wat variabelen gereset naar bijvoorbeeld 0 of de lengte van de *needle* zodat je de functie meerdere keren kunt aanroepen zonder statistieken door elkaar te halen. Hierna volgen wat loops om de *dfa* Array te initializen, deze zullen we later in het programma gebruiken om de *needle* te zoeken in de *haystack*. Tot slot begint er een loop te lopen welke de gehele *haystack* doorloopt, opzoek naar de *needle*. Dit gebeurt met behulp van de *search(in position)* methode, deze roepen we continu opnieuw aan tot dat we de hele *haystack* doorlopen hebben.

De *Search()* blijft lopen totdat we een match hebben gevonden in de *haystack*, wanneer dit gebeurt wordt de *matches* variabele met 1 opgehoogd. Ook wordt er bij iedere vergelijking de *comparisons* variabele opgehoogd, wanneer het programma klaar is met zoeken kunnen de matches en comparisons opgehaald worden met *getMathces() & getComparisons()*.

4.2 Boyer-Moore

```
final private String haystack;
final private int[] right;
private String needle;

private int matches = 0;
private int comparisons = 0;



BoyerMoore(String haystack) {
    // Set haystack for future usage
    this.haystack = haystack;
    // Compute the amount of chars to skip after any mismatch
    right = new int[haystack.length()];
}

private int search(int position) {  // Search for pattern in txt.
    int N = this.haystack.length(); // Used for detecting haystack end (failure)
    int M = needle.length(); // Used for detecting end of needle (succes)
    int skip;
    for (int i = position; i <= N - M; i += skip) {  // Does the pattern match th
        skip = 0;
        for (int j = M - 1; j >= 0; j--) {
            if (needle.charAt(j) != this.haystack.charAt(i + j)) {
                skip = j - right[this.haystack.charAt(i + j)];
                if (skip < 1) {
                    skip = 1;
                }
                break;
            }
        }
        if (skip == 0) {
            matches++;
            return i; // found (hit end of pattern)
        }
        comparisons++;
    }
    return N; // not found
}

public int searchForWord(String word) {
    // Step 1: Initialize some default values for future use
    this.matches = 0;
    this.comparisons = 0;
    this.needle = word; // The word were looking for
    int M = needle.length(); // Used for detecting end of needle (succes)
    int R = haystack.length(); // Used for detecting haystack end (failure)

    // Step 2: Initialize and alter the right Array for use in search()
    for (int c = 0; c < R; c++) { right[c] = -1; }
    for (int j = 0; j < M; j++) { right[needle.charAt(j)] = j; }

    // Step 3: Start looking for the substring by looping the haystack
    int position = 0;
    for (int i = 0; i < haystack.length(); i++) {
        position = search(position + 1);
        if (position >= haystack.length()) {
            return this.getMatches();
        }
    }

    return this.getMatches();
}
```

Bij het aanroepen van de Boyer-Moore klasse initializeren we wederom de *haystack* variabele, maar we maken ook de integer array *right* aan die even groot is als de lengte van *haystack.*

Vervolgens kan er weer met behulp van de *SearchForWord()* methode gezocht worden naar een Substring binnen de *haystack*. Ook deze begint door enkele variabelen te resetten, en past vervolgens de right array aan voor gebruik in de *search()* methode. Hierna begint de methode door de haystack te lopen, en maakt hier gebruik van de *search()* methode.

Binnen *Search*() wordt het aantal matches opgehoogd wanneer deze gevonden worden, en het aantal comparisons wanneer deze plaatsvinden.

4.3 De algoritmen met elkaar vergelijken

Voor het vergelijken van de twee algoritmen hebben we de volgende code in de Main geplaatst, zodat we beide klassen goed kunnen testen:

```
// Step 1: Convert the text file to search in to a String
// String gedicht = fileToString("/Users/guus_portegies/Documents/School/HvA/2014-2015/Searching & Sorting/Opdracht 3/practicum3/src/pract
String gedicht = fileToString("/Users/koen_karsten/Documents/Hva/S&S/Sorting-Searching-assignment-3/practicum3/src/practicum3/gedicht.txt"

// Step 2: load both the algorithms, and pass the testFile into it
BoyerMoore boyerMoore = new BoyerMoore(gedicht);
KMP kmp = new KMP(gedicht);
RegularExpressions regExp = new RegularExpressions();

// Step 3: Search for 10 different words with both the 2 algorithms
System.out.println("==========BM algorithm==========");
System.out.println("de:  FOUND:" + boyerMoore.searchForWord("de") +" COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("het:  FOUND:" + boyerMoore.searchForWord("het") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("voor:  FOUND:" + boyerMoore.searchForWord("voor") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("welke:  FOUND:" + boyerMoore.searchForWord("welke") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("mei:  FOUND:" + boyerMoore.searchForWord("mei") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("droefenis:  FOUND:" + boyerMoore.searchForWord("droefenis") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("ziternietin:  FOUND:" + boyerMoore.searchForWord("ziternietin") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("haar:  FOUND:" + boyerMoore.searchForWord("haar") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("zitten:  FOUND:" + boyerMoore.searchForWord("zitten") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("golven:  FOUND:" + boyerMoore.searchForWord("golven") + " COMPARISONS: " + boyerMoore.getComparisons());
System.out.println("==========KMP algorithm==========");
System.out.println("de:  FOUND:" + kmp.searchForWord("de") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("het:  FOUND:" + kmp.searchForWord("het") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("voor:  FOUND:" + kmp.searchForWord("voor") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("welke:  FOUND:" + kmp.searchForWord("welke") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("mei:  FOUND:" + kmp.searchForWord("mei") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("droefenis:  FOUND:" + kmp.searchForWord("droefenis") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("ziternietin:  FOUND:" + kmp.searchForWord("ziternietin") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("haar:  FOUND:" + kmp.searchForWord("haar") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("zitten:  FOUND:" + kmp.searchForWord("zitten") + " COMPARISONS: " + kmp.getComparisons());
System.out.println("golven:  FOUND:" + kmp.searchForWord("golven") + " COMPARISONS: " + kmp.getComparisons());


// Helper function to convert the .txt to a String
// Based upon http://stackoverflow.com/questions/326390/how-t
static String fileToString(String path) throws IOException {
    byte[] encoded = Files.readAllBytes(Paths.get(path));
    return new String(encoded, StandardCharsets.UTF_8);
}
```

Bij de eerste stap laden we het gedicht “mei” in via een tekstbestand dat we gemaakt hebben genaamd “gedicht.txt”. Hierbij maken we gebruik van de *fileToString(String path)* methode, deze hebben we gevonden op http://stackoverflow.com/questions/326390/how-to-create-a-java-string-from-the-contents-of-a-file - answer-326440 en naar onze eigen wensen aangepast.

Wanner het gedicht geladen is geven we deze door aan onze BoyerMoore & KMP klassen, hierna kunnen we beginnen met het zoeken naar woorden in het gedicht. Eerst zoeken we tien zelf uitgekozen woorden met behulp van het Boyer-Moore algoritme, daarna herhalen we dit met het Knuth-Morris-Pratt algoritme. Per woord & algoritme combinatie printen we het aantal matches en het aantal comparisons dat hiervoor nodig was. Zo kunnen we dit nu dus gaan verwerken in een tabel.

4.4 De uitkomst

Hier volgt een tabel met de matches & comparisons:

Woord	matches Knuth-Morris-Pratt	matches Boyer-Moore	comparisons Knuth-Morris-Pratt	comparisons Boyer-Moore
de	3090	3090	181.051	94.830
het	552	552	179.065	66.829
voor	183	183	178.510	47.582
welke	1	1	177.965	39.016
mei	22	22	178.005	66.637
droefenis	6	6	178.009	25.704
ziternietin	0	0	177.961	20.952
haar	419	419	179.218	48.193
zitten	9	9	178.006	37.013
golven	27	27	178.096	37.353

Wat direct opvalt uit deze tabel is dat beide algoritmes hetzelfde aantal matches hebben, dit is een goed teken want dit betekent dat ze functioneren zoals we verwacht hadden. Wat daarna opvalt is dat het aantal comparisons bij KMP aanzienlijk hoger ligt, dan bij het Boyer-Moore algoritme. Wat een grote reden hiervoor is, is het feit dat het Boyer-Moore algoritme het hele woord overslaat wanneer er een letter niet matcht. Het KMP algoritme gaat in plaats hiervan door bij de volgende letter, dit kost dus aanzienlijk meer comparisons. Bij een laag aantal matches is dit verschil dus het grootste, want hier heb je juist de meeste mismatches.

Deze resultaten liggen ruwweg in lijn met wat in het boek op pagina 779 vermeld wordt, dit is wederom een bevestiging voor ons dat de algoritmen naar behoren werken.

5. Reguliere expressies

5.1 RegularExpression Class

Voor deze opdracht hebben wij een methode, checkNummer(String telNr), geschreven om te controleren of een telefoonnummer voldoet aan de in de opdracht omschreven regels.

```
public boolean checkNummer(String telNr) {

    // Remove all spaces from the telephone number
    String str = telNr.replaceAll("\\s","");

    // Alle nummers beginnen met 031 of +31 of 0031 of 0.
    // ^(((\\+31)|(0031)|(0))

    // Als mobiel nummer
    // EXTRA abonneenummer begint nooit met een 0
    // ((6)[-]?[1-9]{1}(\\d{7}))

    // Als nummer met gebiedscode van 3 cijfers zoals 020 / 033
    // EXTRA 2e letter nooit een 0, dus geen: 002
    // EXTRA kan ook met streepje: 020-
    // (([1-5|7-9])[0-9][-]?[1-9](\\d{6}))

    // Als nummer met gebiedscode van 4 cijfers zoals 0228 / 0521
    // EXTRA kan ook met streepje: 0228-
    // ([1-9]{2}\\d{1}[- ]?[0-9](\\d{6}))

    // Uitzonderingen korte servicenummers zoals 112 / 0900 / 0800
    // EXTRA ook 0906 en 0909 netnummers toegevoegd: http://nl.wikipedia.org/wiki/Lijst_van_Nederlandse_netnummers
    // ((0)((800)|(90)[0,6,9])[-]?(\\d{4}))|(112)

    String regex = ""
        + "^(((\\+31)|(0031)|(0))"                      // Alle nummers beginnen met 031 of +31 of 0031 of 0.
        + "(((6)[-]?[1-9](\\d{7}))$"                    // Als mobiel nummer dus begint met 06
        + "|(([1-5|7-9])[0-9][-]?[1-9](\\d{6}))$"       // Als nummer met gebiedscode van 3 cijfers zoals 020 / 033
        + "|([1-9]{2}\\d{1}[-]?[1-9](\\d{5}))$)"        // Als nummer met gebiedscode van 4 cijfers zoals 0228 / 0521
        + "|((0)((800)|(90)[0,6,9])[-]?(\\d{4}))|(112))";   // Uitzonderingen korte servicenummers zoals 112 / 0900 / 0800

    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(str);

    return matcher.matches();
}
```

De methode checkNummer ontvangt als parameter een string met het telefoonnummer. Voor alle zekerheid verwijderen we eerst alle spaties uit de string. Vervolgens hebben we aan de hand van de opgegeven regels de reguliere expressie opgesteld. Deze bestaat uit verschillende gecombineerde checks, zoals in het commentaar naast de code is te lezen.

5.2 Unit Tests

Om te testen of de reguliere expressie naar behoren werkt hebben we een aantal unit testen geschreven. Hiervoor hebben we een nieuwe JUnit klasse aangemaakt in de map test packages. In deze klasse bevinden zich de door ons opgestelde test methodes.

```
/**
 * Test of checkNummer method, of class RegularExpression.
 * TEST MOBILE NUMBERS
 */
@Test
public void testCheckNummer1() {
    String telNr = "0634149271";
    RegularExpression instance = new RegularExpression();
    boolean expResult = true;
    boolean result = instance.checkNummer(telNr);
    System.out.println("Test checkNummer(), Phonenumber: " + telNr +"\t Expected result: " + expResult + "\t\t Result: " + result);
    assertEquals(expResult, result);
}
```

Aan de hand van de opgegeven regels hebben wij de unit testen geschreven. In de onderstaande tabel vindt u al onze testgevallen met uitleg waarom we voor deze test hebben gekozen, het verwachte resultaat en het daadwerkelijk gegeven resultaat.

Input (tel. nr.)	Verwacht resultaat	Resultaat	Uitleg
0634149271	True	True	Een mobiel telefoonnummer moet uit 10 cijfers bestaan en begint met 06 gevold door het abonneenummer.
06341492710	False	False	Een telefoonnummer heeft 10 cijfers.
06-34149271	True	True	Er mag een streepje (-) na het netnummer of de 06.
+31634149271	True	True	Telefoonnummer mag beginnen met +31 of 0031 in dat geval vervalt de startnul van het abonneenummer.
00316-34149271			Telefoonnummer mag beginnen met +31 of 0031 in dat geval vervalt de startnul van het abonneenummer en er mag een streepje na het netnummer of de 06.
0604149271	False	False	Een abonneenummer begint nooit met 0.
0201234567	True	True	Een netnummer is 3 of 4 cijfers, 020 in dit geval en een telefoonnummer bestaat uit 10 cijfers.
023-1234567	True	True	Er mag een streepje na het netnummer of de 06.
123456789	False	False	Een telefoonnummer begint met +31, 0031 of 0. (uitzondering 112)
0061234567	False	False	Een 3 cijferig netnummer kan nooit een 0 als 2e cijfer hebben.
0228512203	True	True	Een netnummer is 3 of 4 cijfers, 0228 in dit geval en een telefoonnummer bestaat uit 10 cijfers.
0228-512203	True	True	Er mag een streepje na het netnummer of de 06.
0304-123456	False	False	Een netnummer van 4 cijfers heeft nooit een 0 als 2e of 3e cijfer.
09001234	True	True	Dit is een uitzondering, kort servicenummer, mag beginnen met 0900 of 0800 gevolgd door 4 cijfers.
0900-1234	True	True	Dit is een uitzondering, kort servicenummer, mag beginnen met 0900 of 0800 gevolgd door 4 cijfers en mag een streepje bevatten na het netnummer.
0906-1234	True	True	Extra uitzondering toegevoegd voor korte servicenummers, gevonden in de lijst met netnummers op wikipedia.
112	True	True	Uitzondering kort servicenummer 112.

Onderstaand ziet u de resultaten van de opgestelde Unit testen.