

SEARCHING AND SORTING

PRACTICUM 3

DAAN HOOGLAND MAARTEN PETIT

Contents

Inleiding.....	2
Implementatie van substring in programmeertalen	3
Implementatie substring algoritme in Java	3
Implementatie substring algoritme in C++	3
Source code Java methode indexOf()	4
Substring Search Algoritmes	5
Knuth-Morris-Pratt.....	5
Boyer-Moore.....	5
Resultaten	6
Reguliere Expressies.....	7
Unittests.....	7
Tabel met testgevallen.....	8

Inleiding

Practicum opdracht drie van het vak Sorting & Searching bestaat uit drie onderdelen.

De eerste twee onderdelen gaan beide over substring algoritmes. We gaan van de programmeertalen Java & C++ de implementatie van het substring algoritme bekijken.

Bij het tweede onderdeel gaan we twee substring algoritmes vergelijken. De twee algoritmes die we gaan vergelijken zijn Knuth-Morris-Pratt en Boyer-Moore.

Het laatste onderdeel gaat over reguliere expressies. We gaan een methode maken in Java die controleert of een telefoonnummer voldoet aan een aantal gestelde eisen.

Implementatie van substring in programmeertalen

In dit onderdeel gaan we bekijken hoe het substring algoritme is geïmplementeerd in de programmeertalen Java & C++. Door de sourcecode te bekijken gaan we na om welke variant van het substring algoritme het gaat.

Implementatie substring algoritme in Java

De Source code van het substring algoritme voor Java staat op de volgende pagina.

Bij de implementatie van het substring algoritme in Java gaat het om de brute force variant. De eerste letter van de meegegeven string wordt vergeleken met de eerste letter van de source string. Komt deze overeen dan wordt de tweede letter vergeleken. Komen alle letters overeen dan is de string aanwezig en returend de methode de index hiervan. Komt de letter niet overeen. Dan wordt de eerste letter vergeleken met de volgende letter in de source String. Wanneer het einde van de string wordt bereikt is de string ook niet aanwezig en returned de methode -1.

Implementatie substring algoritme in C++

De source code van het substring algoritme in C++ (methodenaam: search)

Return value: An iterator to the first element of the first occurrence of [first2,last2) in [first1,last1).

If the sequence is not found, the function returns last1.

```
1 template<class ForwardIterator1, class ForwardIterator2>
2   ForwardIterator1 search ( ForwardIterator1 first1, ForwardIterator1 last1,
3                           ForwardIterator2 first2, ForwardIterator2 last2)
4 {
5   if (first2==last2) return first1; // specified in C++11
6
7   while (first1!=last1)
8   {
9     ForwardIterator1 it1 = first1;
10    ForwardIterator2 it2 = first2;
11    while (*it1==*it2) { // or: while (pred(*it1,*it2)) for version 2
12      ++it1; ++it2;
13      if (it2==last2) return first1;
14      if (it1==last1) return last1;
15    }
16    ++first1;
17  }
18  return last1;
19 }
```

Bron: <http://www.cplusplus.com/reference/algorithm/search/>

Bij de implementatie van het substring algoritme in C++ wordt ook gebruik gemaakt van de brute force methode. De methode begint bij de eerste letters van de strings. Zijn deze twee letter gelijk dan wordt de volgende letter vergeleken. Wanneer de laatste letter ook gelijk is stopt de loop en returned de methode de iterators naar de plek van de string als eerste voorkomt. Wanneer de letters niet overeenkomen gaat de methode terug naar de eerste vergeleken letter van de huidige reeks. Vanaf hier gaat methode weer opnieuw verder.

Source code Java methode indexOf()

```
/**
 * Code shared by String and StringBuffer to do searches. The
 * source is the character array being searched, and the target
 * is the string being searched for.
 *
 * @param source the characters being searched.
 * @param sourceOffset offset of the source string.
 * @param sourceCount count of the source string.
 * @param target the characters being searched for.
 * @param targetOffset offset of the target string.
 * @param targetCount count of the target string.
 * @param fromIndex the index to begin searching from.
 */
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++) {
        /* Look for first character. */
        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;
}
```

Substring Search Algoritmes

Als 2e deel van de opdracht moeten wij 2 substring search algoritmes met elkaar vergelijken. Deze 2 zijn Knuth-Morris-Pratt en Boyer-Moore. In de rest van dit hoofdstuk word naar deze 2 algoritmes verwezen met hun afkortingen, KMP en BM. Deze stukken code komen direct uit het boek, maar zijn aangepast om niet te stoppen bij het eerste woord, maar om door te gaan en de hoeveelheid worden te tellen.

Knuth-Morris-Pratt

```
/**
 * Searches the given text for the String pattern given in the constructor.
 *
 * @param txt Text to be searched for pattern.
 * @return Number of occurrences of pattern within the String txt
 */
public int search(String txt) {
    int count = 0, characterCompares = 0;
    int i, j, N = txt.length(), M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++) {
        characterCompares++;
        j = dfa[txt.charAt(i)][j];
        if (j == M) {
            count++;
            i++;
            j = 0;
        }
    }
    System.out.println("Character comparisons: " + characterCompares);
    System.out.println("Word count: " + count);
    return count;
}
```

In de bovenstaande code wordt gezocht in de String txt naar het patroon dat aangegeven is in de constructor. De count variabele houdt de hoeveelheid woorden die het algoritme tegenkomt die overeen komen met het patroon bij. Normaal zou het programma hier stoppen, maar omdat j weer naar 0 gezet word, gaat het verder.

Om het aantal vergelijkingen te tellen staat er een counter binnen de loop, boven de vergelijkingen. Hier valt niet veel over te zeggen omdat deze simpelweg het aantal vergelijkingen bijhoudt dat nodig is om door de hele tekst heen te lopen.

Boyer-Moore

```
/**
 * Searches the given text for the String pattern given in the constructor.
 *
 * @param txt Text to be searched for pattern.
 * @return Number of occurrences of pattern within the String txt
 */
public int search(String txt) {
    int count = 0, characterCompares = 0;
    int M = pat.length();
    int N = txt.length();
    int skip;
    for (int i = 0; i <= N - M; i += skip) {
        skip = 0;
        for (int j = M - 1; j >= 0; j--) {
            characterCompares++;
            if (pat.charAt(j) != txt.charAt(i + j)) {
                skip = Math.max(1, j - right[txt.charAt(i + j)]);
                break;
            }
        }
    }
}
```

```

    }
    if (skip == 0) {
        count++;
        skip = 1;
    }
}
System.out.println("Character comparisons: " + characterCompares);
System.out.println("Word count: " + count);
return count;
}

```

Net zoals bij het KMP algoritme hierboven moest de code worden aangepast om de woorden en het aantal vergelijkingen te tellen. Hierbij worden gebruik gemaakt van dezelfde 2 variabelen.

Als skip gelijk is aan 0 betekend dat er een match gevonden is, waardoor de count variabelen 1 opgehoogd moet worden.

Bij de characterCompare variabele moet elke keer als er een karakter vergeleken word 1 erbij komen. Deze staat in de binnenste for loop. Deze variabele zou hierdoor dus minder voorkomen omdat de buitenste for loop veel grotere stappen maakt.

Resultaten

Voor de resultaten hebben wij 10 woorden genomen en gecheckt hoe vaak deze voorkwamen in de applicatie. Daarnaast hebben wij bijgehouden hoeveel vergelijkingen elk algoritme doet om bij dit nummer te komen. Hieronder in de tabel is dit te zien:

	Knuth-Morris-Pratt		Boyer-Moore	
	Aantal	Vergelijkingen	Aantal	Vergelijkingen
Uit	278	182259	278	67219
Rammen	1	182536	1	45169
Zee	90	182447	90	73502
Raamkozijn	1	182536	1	23651
Konijn	0	182537	0	36752
Eifeltoren	0	182537	0	30931
Een	1102	181425	1102	83858
Weinig	7	182530	7	36728
Toen	131	182406	131	65497
Computer	0	182537	0	30945

Wat al duidelijk was is dat de aantallen altijd gelijk zijn, maar de vergelijkingen verschillen enorm van elkaar. Bij KMP zijn de vergelijkingen ongeveer altijd gelijk, maar bij BM verschillen ze erg.

De belangrijkste reden waarom BM een kleiner aantal vergelijkingen heeft is omdat BM beter werkt op grotere lappen tekst en bij langere woorden efficiënter is (wat ook te zien is in de resultaten). Dit komt omdat als er een letter vergeleken word die helemaal niet in het patroon zit, het de gehele lengte van het word overslaat en vanaf daar verder gaat kijken.

Bij KMP vergelijkt het steeds elk karakter in de zoekstring met het patroon, waardoor hij elke keer bijna de hele tekst door moet lopen.

Onze resultaten komen ongeveer overeen met de resultatentabel uit het boek (pagina 779)

Reguliere Expressies

Bij dit onderdeel van het practicum gaan we een reguliere expressies maken om te testen of een telefoonnummer aan een aantal gestelde eisen voldoet. Om een telefoonnummer te checken hebben we de methode `checkNumber(String telNr)` gemaakt.

```
public boolean checkNumber(String telNr) {
    boolean isValid = false;

    //((\+31)|(0031)|(0)): All numbers begin with 0 or +31 or 0031
    //((6)[- ]?[1-9](\d{7})): A mobile number
    //([1-9]{2}[- ]?[1-9](\d{6})): A phone number with an areacode of 3 digits.
    //([1-9]{2}\d[- ]?[1-9](\d{5})): A phone number with an areacode of 4 digits.
    String regex = "((\\+31)|(0031)|(0))((6)[- ]?[1-9](\\d{7})|([1-9]{2}[- ]?[1-9](\\d{6}))|([1-9]{2}\\d[- ]?[1-9](\\d{5}))|(0[8-9]00(-?)\\d{4}))|(112))";

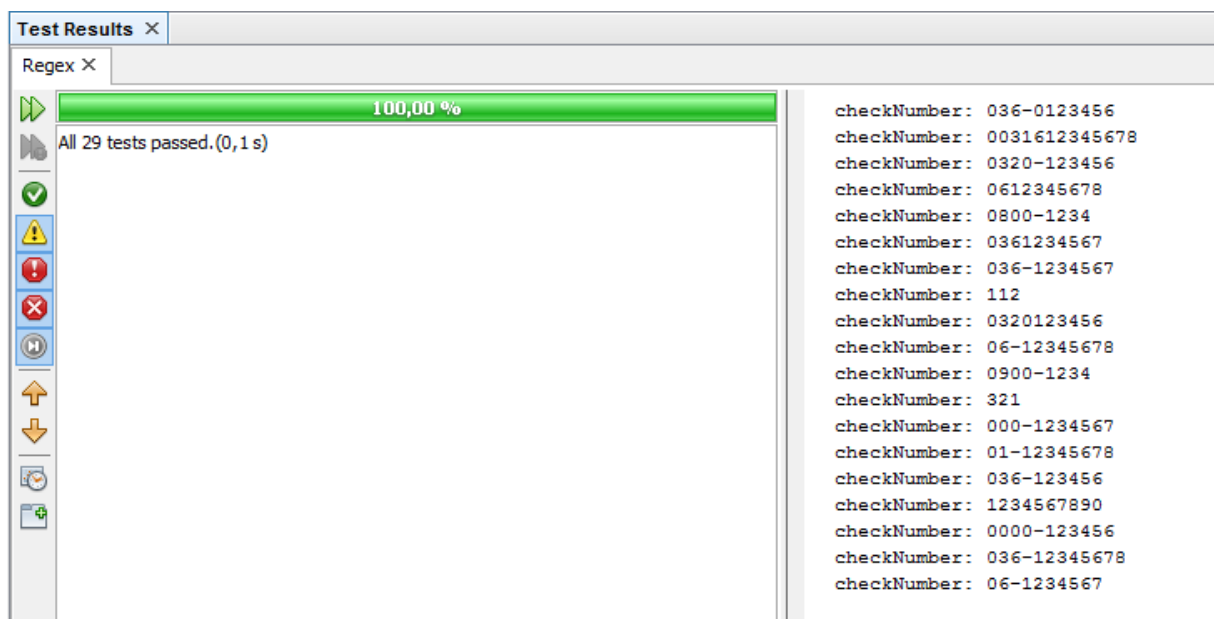
    CharSequence inputString = telNr; //Convert string to CharSequence.
    Pattern pattern = Pattern.compile(regex); //Compile regex String to pattern.
    Matcher matcher = pattern.matcher(inputString); //New matcher with telNr and pattern.
    if (matcher.matches()) { //Compare pattern and telNr.
        isValid = true; //If the telNr matches telNr is valid.
    }
    return isValid; //Return true when valid.
}
```

De reguliere expressie: `"((\\+31)|(0031)|(0))((6)[-]?[1-9](\\d{7})|([1-9]{2}[-]?[1-9](\\d{6}))|([1-9]{2}\\d[-]?[1-9](\\d{5}))|(0[8-9]00(-?)\\d{4}))|(112))"`.

Unittests

Om de methode te testen hebben we een aantal unittests gemaakt. Op de volgende pagina staat een tabel met alle testgevallen, de correcte uitkomst en de reden waarom we het nummer hebben getest.

Output van de unittests:



The screenshot shows a 'Test Results' window with a green progress bar at 100,00 %. Below the bar, it says 'All 29 tests passed. (0,1 s)'. On the right side, there is a list of test cases, each labeled 'checkNumber:' followed by a phone number. The list includes various formats like international numbers, mobile numbers, and landline numbers with different area codes and lengths.

Test Case
checkNumber: 036-0123456
checkNumber: 0031612345678
checkNumber: 0320-123456
checkNumber: 0612345678
checkNumber: 0800-1234
checkNumber: 0361234567
checkNumber: 036-1234567
checkNumber: 112
checkNumber: 0320123456
checkNumber: 06-12345678
checkNumber: 0900-1234
checkNumber: 321
checkNumber: 000-1234567
checkNumber: 01-12345678
checkNumber: 036-123456
checkNumber: 1234567890
checkNumber: 0000-123456
checkNumber: 036-12345678
checkNumber: 06-1234567

Tabel met testgevallen

Telefoonnummer:	Correct uitkomst:	Reden
06-12345678	True	Mobiel nummer met streepje moet valid zijn.
0612345678	True	Mobiel nummer zonder streepje moet valid zijn.
036-1234567	True	Telefoonnummer met 3-cijferig netnummer met streepje moet valid zijn.
0361234567	True	Telefoonnummer met 3-cijferig netnummer zonder streepje moet valid zijn.
0320-123456	True	Telefoonnummer met 4-cijferig netnummer met streepje moet valid zijn.
0320123456	True	Telefoonnummer met 4-cijferig netnummer zonder streepje moet valid zijn.
112	True	112 is een uitzondering en moet valid zijn.
0900-1234	True	0900 nummers met streepje moeten valid zijn.
0800-1234	True	0800 nummers met streepje moeten valid zijn.
09001234	True	0900 nummers zonder streepje moeten valid zijn.
08001234	True	0900 nummers met streepje moeten valid zijn.
+31361234567	True	0800 nummers met streepje moeten valid zijn.
0031361234567	True	Telefoonnummer met landcode moet valid zijn.
+31612345678	True	Mobiel nummer met landcode moet valid zijn.
0031612345678	True	Mobiel nummer met landcode moet valid zijn.
06-01234567	False	Abonneenummer mag nooit met een 0 beginnen.
036-0123456	False	Abonneenummer mag nooit met een 0 beginnen.
0320-012345	False	Abonneenummer mag nooit met een 0 beginnen.
0601234567	False	Abonneenummer mag nooit met een 0 beginnen.
0320012345	False	Abonneenummer mag nooit met een 0 beginnen.
06-123456789	False	Een nummer heeft 10 cijfers
06-1234567	False	Een nummer heeft 10 cijfers
036-12345678	False	Een nummer heeft 10 cijfers
036-123456	False	Een nummer heeft 10 cijfers
1234567890	False	Een nummer begint altijd met 0, +31 of 0031
0000-123456	False	Een netnummer heeft nooit een 0 als 2 ^e of 3 ^e getal.
000-1234567	False	Een netnummer heeft nooit een 0 als 2 ^e of 3 ^e getal.
01-12345678	False	Een mobielnummer heeft altijd als netnummer 06
321	False	0900 nummers met streepje moeten valid zijn.