Project 3

# 605.621 - Algorithms Project 3

## Introduction

```

Usage:

$ javac Project3.java && java Project3 x y ../input/in.txt

Example:

$ javac Project3.java && java Project3 1100 1100 ../input/in.txt

```

\* x is the first of the known signals (ie 101)

\* y is the second of the known signals (ie 001)

\* in.txt is the superposition of the two signals.

## Algorithm:

For it to be a valid interweaving of the two signals, a character must exist in s

that is the beginning character of either x or y.

Either x or y may lead, so iterating through the string twice checking first if x

leads and then if y leads will cover all cases. This will add one more iteration

of n elements to the complexity.

#### Starting with x first:

1. Find the beginning of x if it exists. If it does not exist, exit.

2. Increment to the next character in x.

3. Check if the next character in s is either the current index value of x or

the current index value of y. If not, exit.

4. If every value of s is checked, return that it is a valid interweaving.

Else go to step 3.

#### Starting with y first:

1. Find the beginning of y if it exists. If it does not exist, exit.

2. Increment to the next character in y.

3. Check if the next character in s is either the current index value of x or

the current index value of y. If not, exit. Increment the string that matched.

4. If every value of s is checked, return that it is a valid interweaving.

Else go to step 3.

#### Psuedo Code

```

boolean untangle(x,y,s) {

x\_index = 0

y\_index = 0

first\_time = true

s\_index = 0

//check if x leads

for char in s:

s\_index++

// Step 1

if first\_time == true:

if x[x\_index] == char:

first\_time = false

// Step 2

x\_index++

continue

// Step 3

if char == x[x\_index % len(x)]:

x\_index++

continue

if char == y[y\_index % len(y)]

y\_index++

continue

return false

// Step 4

if s\_index == len(s):

return true

x\_index = 0

y\_index = 0

first\_time = true

s\_index = 0

//check if y leads

for char in s:

s\_index++

// Step 1

if first\_time == true:

if y[y\_index] == char:

first\_time = false

// Step 2

y\_index++

continue

// Step 3

if char == y[y\_index % len(y)]

y\_index++

continue

if char == x[x\_index % len(x)]:

x\_index++

continue

return false

// Step 4

if s\_index == len(s):

return true

}

```

#### Analysis:

###### Theoretical

The algorithm will iterate through s once for x leading and once for y leading.

The complexity of this algorithm is therefore O(2n) where n is the size of s.

###### Run Time

I gathered the following data running the untangler on strings of varying length:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Number of comparisons | | | | | |
| Input Size | 10 | 100 | 500 | 1000 | 5000 | 10000 |
| n | 10 | 100 | 500 | 1000 | 5000 | 10000 |
| 2n | 20 | 200 | 1000 | 2000 | 10000 | 20000 |
| Untangler | 13 | 103 | 964 | 1668 | 6660 | 17220 |

As a graph:

The untangler always runs between n and 2n, as described in my algorithm. The theoretical analysis is supported by the run time data.