Morgan Adams
Couse: Foundations of Algorithms
EN.605.621.85.FA21
Date: Dec 6, 2021

Project 3 Analysis

Overview
This project has us explore Dynamic Programming with some constraints on the input that more closely models a real world application.

Dynamic Programming Signal Processing:
In processing the signal we can use Dynamic programming to build an $O(n^2)$ solution that models determines whether there is an interweaving of signal X and signal Y. Since we're reading this character by character, the approach taken here leverages dynamic data structures to handle an unknown stream length. Consider the pseudocode:

```
SignalProcessor(sigx, sigy, stream)
 SolutionTable = List()
 j = 0
 for i=0 to stream.length + 1

  // dynamically add rows
  SolutionTable.append(list())
  while j <= stream.length and i+j-1 < len(stream)

   // dynamically grow each row
   SolutionTable.append(False)

   // pad sigx with a repeat of itself
   if i == sigx.length
    sigx = sigx.concat(sigx)
   if j == sigy.length
    sigy = sigy.concat(sigy)

   // base case
   if i == 0 and j == 0
    SolutionTable[i][j] = True
   else if i == 0
    SolutionTable[i][j] = (SolutionTable[i][j-1] and sigy[j-1] == stream[i+j-1]
   else if j == 0
```

```
      SolutionTable[i][j] = SolutionTable[i-1][j] and sigx[i-1] == stream[i+j-1]
    else
      SolutionTable[i][j] = (SolutionTable[i-1][j] and sigx[i-1] == stream[i+j]-1) or
                            (SolutionTable[i][j-1] and sigy[j-1] == stream[i+j-1])
    j++
  j = 0
return SolutionTable
```

Hand Run of SignalProcessor:
We'll use the example in the assignment prompt to do a by hand test.

sigx = 101
sigy = 00
s = 100010101
SignalProcessor(sigx, sigy, s)

1. SolutionTable is initialized to an empty list()
2. j = 0
3. i = 0
4. SolutionTable has a list appended to it so we have an empty list in a list [[]].
5. SolutionTable appends False [[False]]
6. End of the inner loop gives [[True]] for the base case. j = 1 now.
7. SolutionTable appends False [[True, False]]
8. Since i == 0 at this point, we go into the second condition and find that False is assigned to SolutionTable[0][1] i.e. [[True, False]] as before
9. Since the algorithm depends on the previous cell being True, the rest of the row will be False giving us [[True, False, False, False, False, False, False, False, False, False]].
10. The outer loop cycles again and appends a new row:
    [[True, False, False, False, False, False, False, False, False, False],
    []]. (I'll skip this step in my commentary now).
11. This time we hit the j == 0 condition in the inner loop which will be the first hit of every first iteration of the inner loop after the outer loop completes a cycle. This time we find that with i=1 that we have a match and that the 0th index of sigx matches the 0th index of stream so we update the position [1][0] in the table.
    [[True, False, False, False, False, False, False, False, False, False],
    [True]]
12. Now j == 1 so now we enter the else clause. The 2nd half of the or expression evaluates to True by matching the 0th character of sigy to the character at the 1st index of signal.
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True]]
13. At j== 2 we find another match with the 1st index of sigy matching the 2nd index of stream.
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True, True]]

14. We again find a match on j==3 (3 0's in a row).
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True, True, True]]
15. We now find there is no match on sigy and no on six since the previous row has False values at i-1 for the rest of the row. Iterating through, we find the rest of the row is False. Of note is that it ends 1 shorter than the previous row. The idea here is that if we found a match every time on the first pass of the outer loop, the top row would be True and the rest of the table False. Finding a match on a column means at most, we would be able to file the rest of the row with 1 less True than the previous row if we found matches on the rest of the row. Therefore, each subsequent row can decrease in length by 1 to have a small savings on space.
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True, True, True, False, False, False, False, False]]
16. The previous row found matches on sigy. This row finds a shared match with sigx. For example, the second character in stream matches the 1st character in sigx. It then finds two matches on sigy similar to the previous row, but representing a different possible solution. After the 4th character is matched to sigy, the rest of the row is False.
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True, True, True, False, False, False, False, False],
    [True, True, True, False, False, False, False, False]]
17. The fourth row finds False on the first couple values, but finds a True where j = 2 and i = 3 matching the 3rd character of sigx to the 5th character in stream. We find a True where j = 3 since the 6th character of stream is 0. The rest of the row is False since there were no sigx matches at this point on the previous row and there are no 1's in sigy.
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True, True, True, False, False, False, False, False],
    [True, True, True, False, False, False, False, False],
    [False, False, True, True, False, False, False]]
18. Moving on to the 5th row (i = 4), we again find the first 3 values to be False, but find the match for the 1 on sigx. My implementation padded sigx to match the length of stream so it's technically the 4th character in sigx, but it's the start of a repetition of sigx. The following character is a match on sigy since it's a 0 so the subsequent value is also True. It then finds another 1 though so the last value in the row is False since there's not a match in the same column at i-1 so it fails to properly match for sigx.
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True, True, True, False, False, False, False, False],
    [True, True, True, False, False, False, False, False],
    [False, False, True, True, False, False, False],
    [False, False, False, True, True, False]]
19. Where i = 5, we only find 1 True value to match up to sigx on the 0 at the 7th index of stream.
    [[True, False, False, False, False, False, False, False, False, False],
    [True, True, True, True, False, False, False, False, False],
    [True, True, True, False, False, False, False, False],
    [False, False, True, True, False, False, False],
    [False, False, False, True, True, False],
    [False, False, False, True, False]]
20. The 6th row matches on the 4th value since it's a 1 and we see a True at i-1 in the same column. This corresponds to a match of the last value in stream which means we have a solution.
    [[True, False, False, False, False, False, False, False, False, False],

[True, True, True, True, False, False, False, False, False],
[True, True, True, False, False, False, False, False],
[False, False, True, True, False, False, False],
[False, False, False, True, True, False],
[False, False, False, True, False]]
[False, False, False, True]]

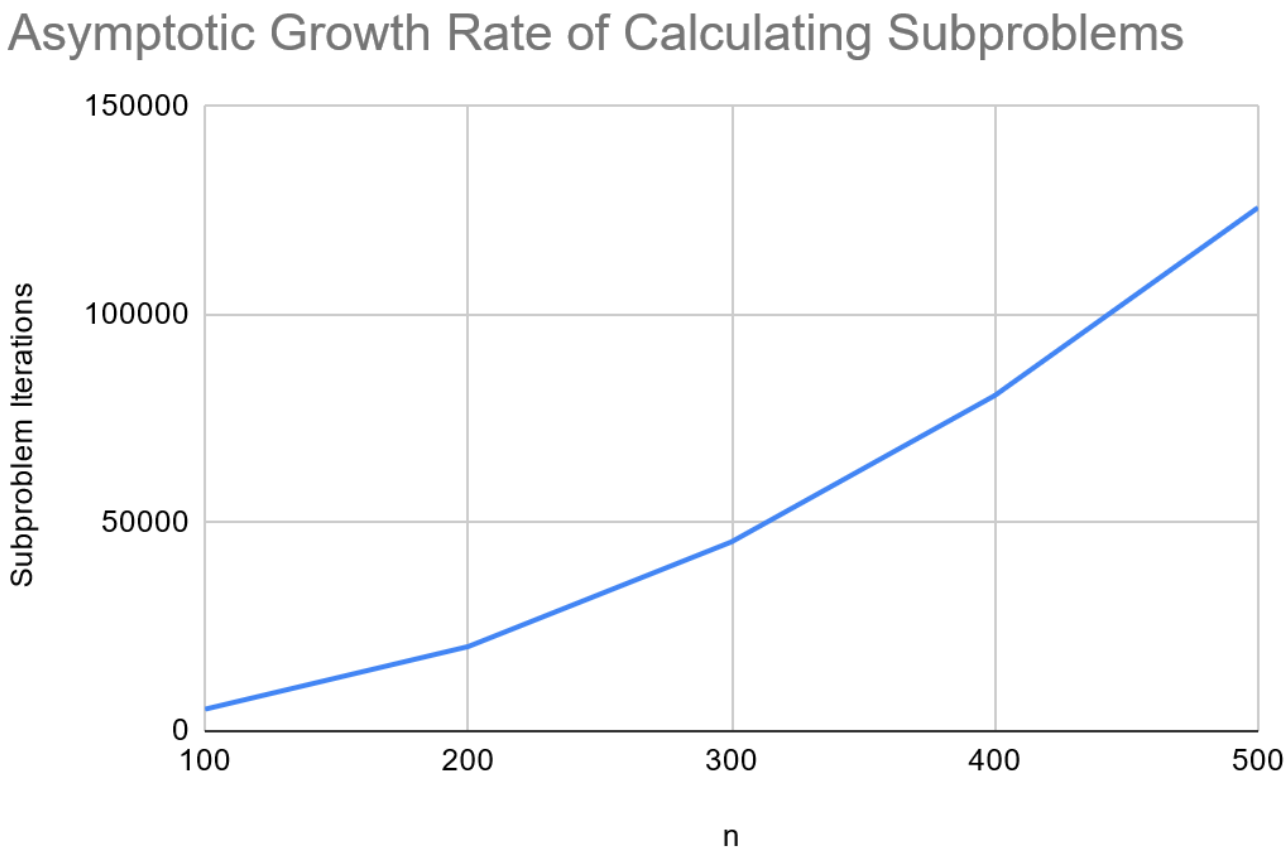
21. The remaining values turn out to be False which makes sense since there are no previous values on i or j that are True.
[[True, False, False, False, False, False, False, False, False, False],
[True, True, True, True, False, False, False, False, False],
[True, True, True, False, False, False, False, False],
[False, False, True, True, False, False, False],
[False, False, False, True, True, False],
[False, False, False, True, False]]
[False, False, False, True],
[False, False, False],
[False, False],
[False]]

We'll pass this table to the ReconstructMatches function shortly. One thing worth noting is that there's some minor optimizations that could be made here when we know the rest of a row will be False. I elected to not do this for this implementation in order to keep things simple, but I felt it was worth noting.

Runtime of SignalProcessor:
We note that the first loop runs n times where n equals the length of stream + 1. We add plus one because SolutionTable[0][0] doesn't correspond to a match and is a base case with a default value of True used as a starting place for the rest of the algorithm. The second loop runs at most stream.length + 1 as well, but as matches are found, the length of each row gets shorter which results in a total of $n*(n-1)/2$ ($O(n^2)$) loop iterations. Every other line runs in O(1) time which means the asymptotic runtime for SignalProcessor is $O(n^2)$.

Examining the trace files confirms this runtime. Based on line 100 of the trace files, where the numer inner loop iterations is recorded, we get the following iterations for the associated signal under analysis of length, n.

n	Subproblem Iterations
100	5,251
200	20,301
300	45,451
400	80,601
500	125,751



By doing some rough calculations we can confirm these values are accurate:

100 * 100 / 2 = 5,000
200 * 200 / 2 = 20,000
300 * 300 / 2 = 45,000
400 * 400 / 2 = 80,000
500 * 500 / 2 = 125,000

which corresponds with the data produced by the trace runs.

Construction of Interwoven Signals:
ReconstructMatches first scans the end of each row to look for a Truth value which indicates that some series of interwoven x and y signals was found. From that coordinate we can begin tracing back to construct what those interwoven signals actually are. Consider the pseudocode:

```
ReconstructMatches(SolutionTable, sigx, sigy):
    // find end match marker
    endi = 0
    while endi < SolutionTable.length
        endj = SolutionTable[endi].length - 1
        if SolutionTable[endi][endj]
```

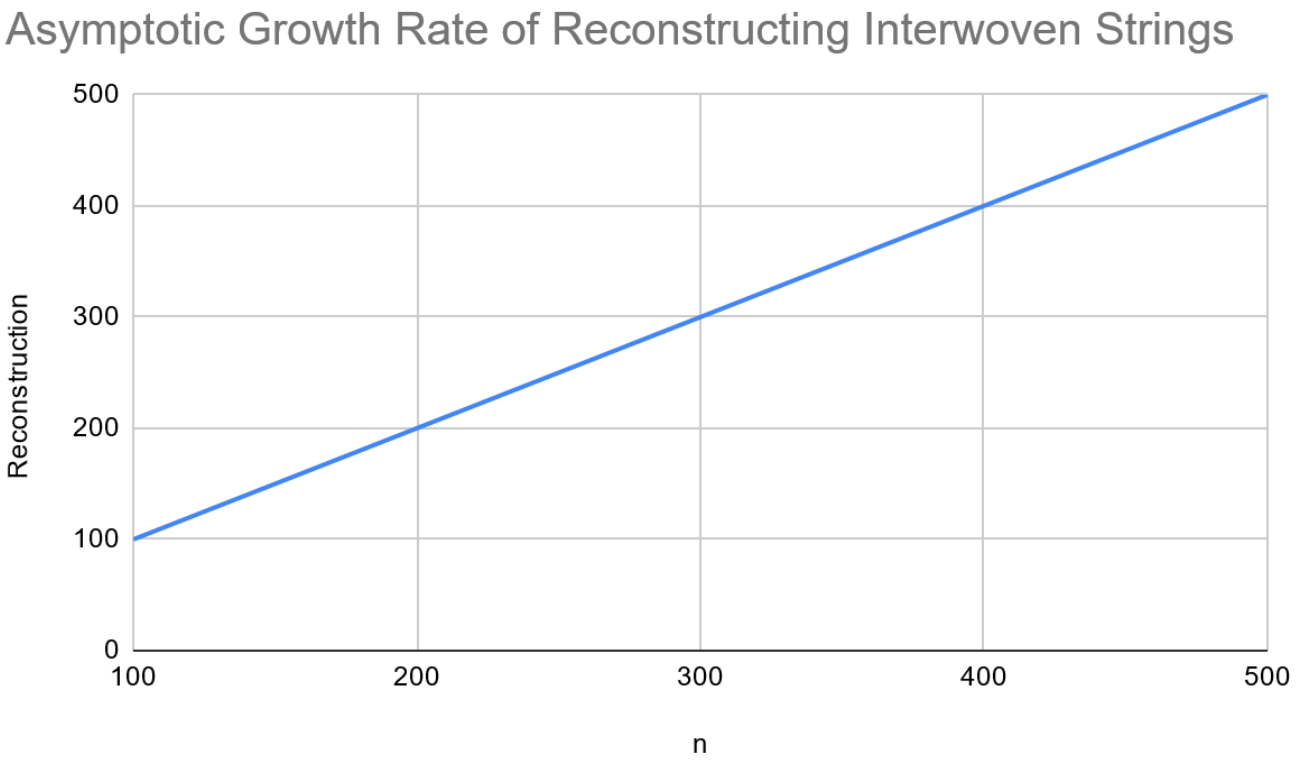
```
        break
    endi++

// return empty strings if no match found
if endi >= SolutionTable.length
    return '', ''

// count matched signals
sigx_c = 0
sigy_c = 0

// pad signals to greater than the table size
sigx = sigx * ceiling(SolutionTable.length/sigx.length)
sigy = sigy * ceiling(SolutionTable.length/sigy.length)

// find interwoven signals
while endi > 0 or endj > 0
    if endj > 0 and SolutionTable[endi][endj - 1]
        sigy_c += 1
        endj -= 1
    elif SolutionTable[endi - 1][endj]
        sigx_c += 1
        endi -= 1

return sigx.substring(0, sigx_c), sigy.substring(0, sigy_c)
```

Runtime of ReconstructMatches:
Since there are stream.length + 1 rows this first loop has a runtime of O(n). The final loop traces back the interwoven signals which will be at most O(n) since the interwoven signals make up a substring of stream. Thus ReconstructMatches runs in O(n) time.

Based on the data in the trace files, we can see the largest cost is on line 167 where the actual reconstruction happens which exactly matches n.

n	Reconstruction
100	100
200	200
300	300

400	400
500	500



Hand Run of ReconstructMatches:
We'll reuse the table generated above and pass it to the function along with sigx, and sigy.

1. The first part of ReconstructMatches looks over the end of each row to find a True value which corresponds to a solution.
   [[True, False, False, False, False, False, False, False, False, False],
   [True, True, True, True, False, False, False, False, False],
   [True, True, True, False, False, False, False, False],
   [False, False, True, True, False, False, False],
   [False, False, False, True, True, False],
   [False, False, False, True, False]]
   [False, False, False, True],
   [False, False, False],
   [False, False],
   [False]]
2. A True is found at (6, 3). These coordinates get used to traceback and construct the matched interwoven signals.
3. The values sigx and sigy are again padded with repeats of themselves.
4. We also establish counters for sigx and sigy that will be used as indices against the padded values of sigx and sigy which will gives us our interwoven signals.

5. We start at (6, 3) and trace backward. It's a solution to the interwoven signals problem so starting here will guarantee we trace a proper solution.
6. For each iteration of the loop we check if the previous column has a True, if so we add 1 to sigy_c. If instead we find a True in the row above at the same column, we add 1 to sigx_c. In other words, matches on the previous row correspond to a match on sigx and matches on the previous column correspond to sigy. Following the Trace we get the following for each pass of the while loop:
    a. (init) sigx_c: 0, sigy_c: 0, endi: 6, endj = 3
    b. sigx_c: 1, sigy_c: 0, endi: 5, endj = 3
    c. sigx_c: 2, sigy_c: 0, endi: 4, endj = 3
    d. sigx_c: 3, sigy_c: 0, endi: 3, endj = 3
    e. sigx_c: 3, sigy_c: 1, endi: 3, endj = 2
    f. sigx_c: 4, sigy_c: 1, endi: 2, endj = 2
    g. sigx_c: 4, sigy_c: 2, endi: 2, endj = 1
    h. sigx_c: 4, sigy_c: 3, endi: 2, endj = 0
    i. sigx_c: 5, sigy_c: 3, endi: 1, endj = 0
    j. sigx_c: 6, sigy_c: 3, endi: 0, endj = 0
7. We then return substrings of the padded sigx and sigy corresponding to these matched values.
   sigx.substring(0, sigx_c), sigy.substring(0, sigy_c)
   This substring function returns the substring of the padded sigx and sigy starting at the 0'th index and then the next sigx_c and sigy_c characters respectively. The advantage of this approach is it makes it easy to include partial matches of repeats of sigx and sigy.
8. Indeed, once we print out the values we get the following from the application:
   Best Match:
   SIGX: 101101
   SIGY: 000

Total Runtime:
Based on these two methods we have a runtime of $n^2 + n$ or simply $O(n^2)$ for this solution. However, there is some complexity in the solution when we consider the parameters of the application.

Since we're reading the characters from some kind of communication stream and are unable to look at the string as a whole, we have to look at it, as it comes in. We simulate this in the application with a SignalReader object with the following functions:

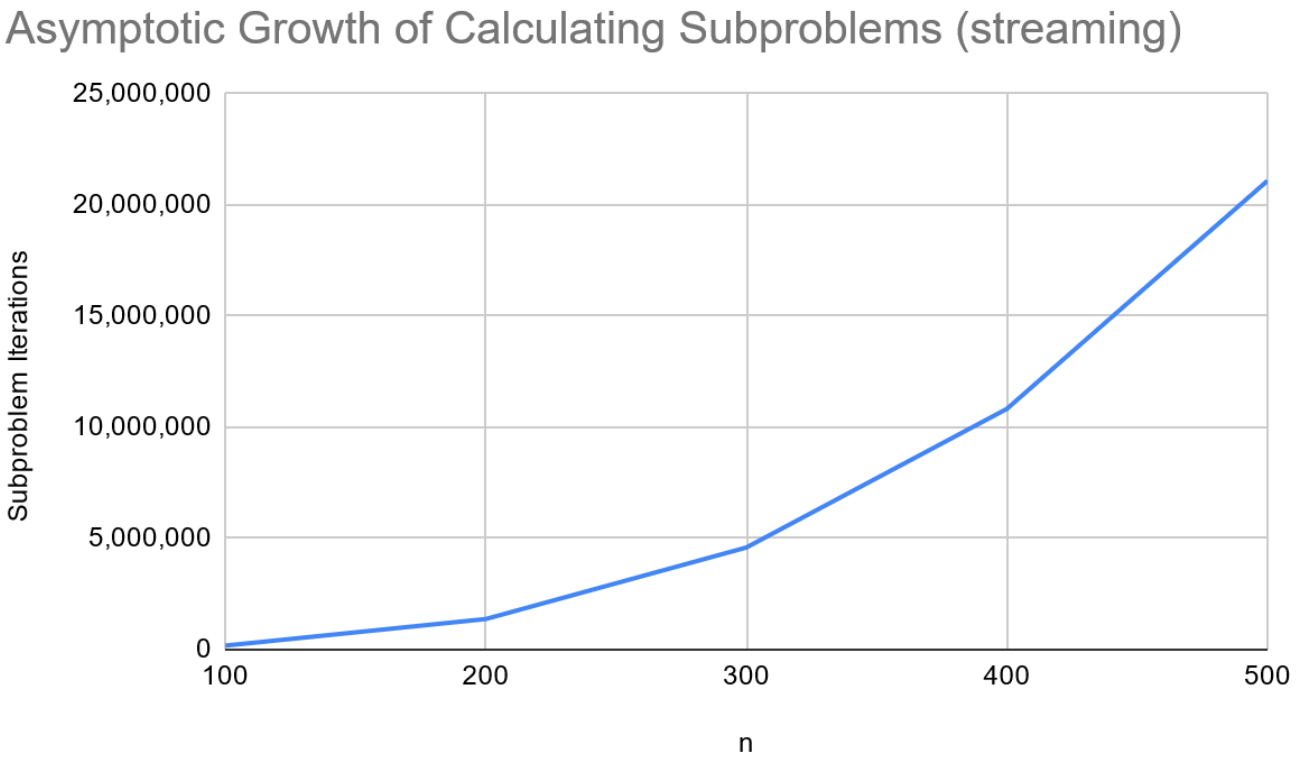
- network_read - simulates reading a single character form the network, but really just accepts a character parameter and appends it to the stream
- next_signal - reads a possible signal on the stream; the catch here is that if network_read adds a new character on the stream and and it is contiguous on the stream with the previous "signal" generated by next_signal, then it next_signal will return the previous signal with the new character(s) amended
- has_next - fast forwards through noise on the stream to see if there is a set of characters in x or y to match against

The program has 2 operating modes: as a stream or the whole signal. If the --stream flag is passed to the application it will read 1 character at a time and attempt to run it through SignalProcessor. For example if a stream m has some set of characters @#!101110#%^Y then the stream operating mode will read through the noise to the first 1 and pass "1" as the signal to the SignalProcessor function and determine whether it is a valid interweaving of x and y. It will then read the next character, 0, and then pass 10 to SignalProcessor until the signal n=101110 has been read and then passed to SignalProcessor.

Here a significant assumption is made. It's possible that the 1's and 0's on either end of n are noise. If we tried to generate every substring of n and pass that to SignalProcessor, it would cost n(n+1)/2 or $O(n^2)$. Combined with the actual cost of SignalProcessor, we obtain a running time of $O(n^4)$. This implementation builds contiguous substrings based on the previous substring giving the total runtime $O(n^3)$.

Indeed, if use the summation formula for squares with a ½ multiplier since, as discussed earlier, the SignalProcessor function also is based on a summation, we find that the trace file results show a $O(n^3)$ runtime:

n	Subproblem Iterations	(n(n+1)(2n+1)/6) * 1/2
100	176,850	169,175
200	1,373,700	1,343,350
300	4,590,550	4,522,525
400	10,827,400	10,706,700
500	21,084,250	20,895,875

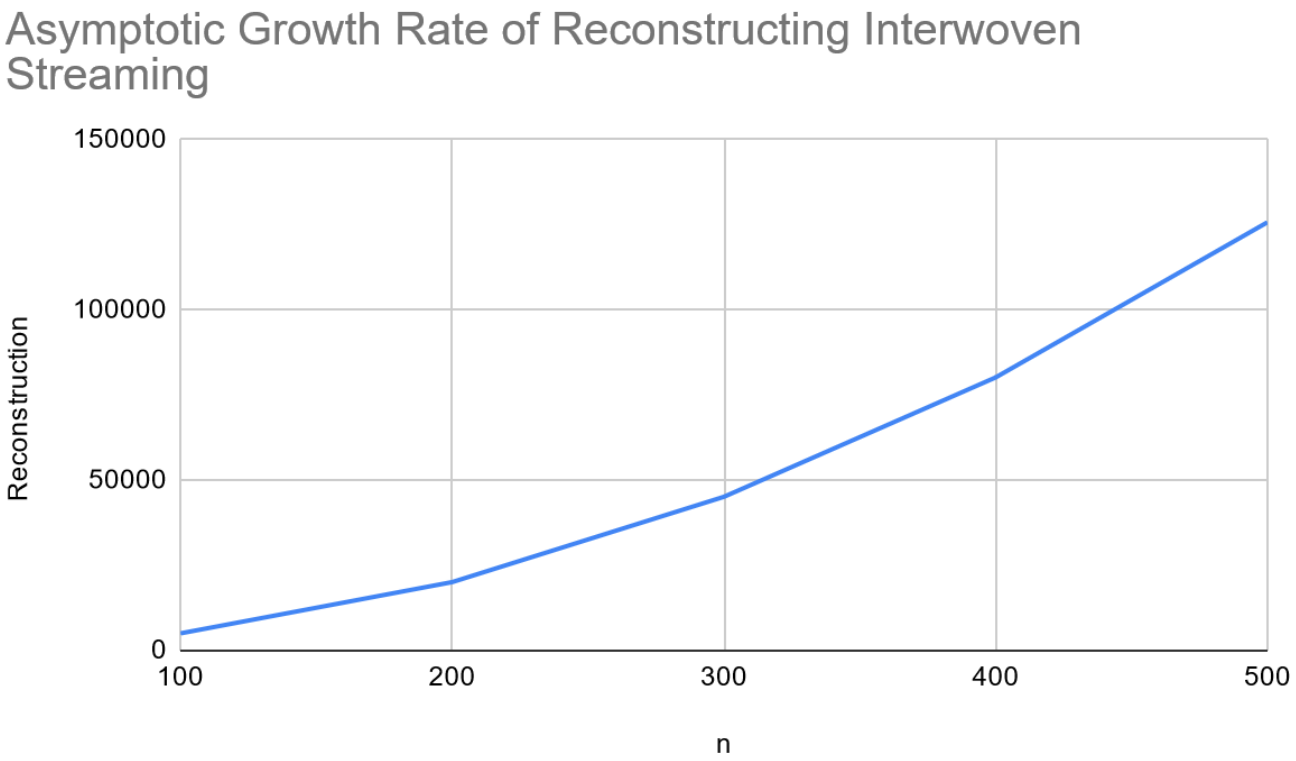


If we simply leave off the --stream flag, the SignalReader will look for a valid string in the stream and pass that to SignalProcessor giving us the $O(n^2)$ runtime as described earlier. This suggests the user of this algorithm should not run the calculations character by character from the stream. Ideally instead, the

signal stream would be batched to achieve the $O(n^2)$ time. In either case, both options are available to the user.

Similarly if we observe the cost of reconstructing the interwoven strings with the --stream option enabled we see the growth rate increases from linear to quadratic:

n	Reconstruction
100	5050
200	20100
300	45150
400	80200
500	125750



This implementation also attempts to find the best string within the signal by recognizing that valid contiguous symbols could appear as noise. The signal "1@@1@@@@100010101@@@@@@@@@@ 10001@1" would find 100010101 to be the best match on a x and y of 101 and 00. 10001 would also be valid, but we assume "longer" to be better.

A major takeaway from this implementation is the importance of protocols in network communications - especially secure protocols like TLS that prevent tampering. Even if there were not an attacker tampering though, implementing a protocol for the communication would make it much easier to identify a valid sequence within the stream since then noise could contain valid characters.

Design Assumptions:

- Noise could have partial valid contiguous sequences of valid characters
- The best result is the longest match detected against a contiguous sequence of valid characters. For example, if 10001 is analyzed, a match over 1000 would not weigh as heavily as a match against 10001. The absence of a protocol for this communication stream makes such assumptions necessary
- I acknowledge the possibility of having a stream containing valid characters such as 11010101010001, but perhaps the first and last characters are part of the noise. That complicates the runtime significantly though having to enumerate every possible substring so I instead took an approach to allow a stream to come in character by character and test the contiguous signal with each additional character: e.g. test1: 1, test2: 11, test3: 110, etc. This allows for testing for the best possible match and possibly rooting out some noise at the end of the string even if those characters are valid, but maybe don't match the desired interwoven strings.
- As a result of testing progressively longer sequences of valid characters, I added a SignalProcessor class to extract the next signal from the stream to be tested. It helped simulate the stream at the cost of O(n) to stream characters 1 by 1. It also helps with sifting through noise too so it is a helpful utility. In the worst case if there is a lot of noise we would have a cost of O(m) where m is the length of the entire stream (noise included), but we would incur that cost anyway since we can't avoid examining the stream for valid characters.