
COT 6405 Programming Project

Christian Garbin

Contents

1	Introduction	4
2	Longest common subsequence	4
2.1	Definiton	4
2.2	Applications	5
3	Notebook structure	5
4	Algorithm descriptions	6
4.1	Brute-force	6
4.2	Dynamic programming	6
4.3	Hirschberg's linear space	8
4.4	Runtime and space summary	9
5	Planning of experiments	9
5.1	Runs	9
5.2	Input size for tests	10
5.3	Data structures	10
5.4	Input generation	11
5.5	Graphs and tables	11
5.6	Programming language	13
6	Initialization and verification	13
7	Runtime tests and analysis	14
7.1	Constant c calculation	14
7.1.1	Constant c for brute-force	14
7.1.2	Constant c for dynamic programming	15
7.1.3	Constant c for Hirschberg's linear space	16
7.2	Empirical vs. predicted RT graphs	16
7.3	Graphs for larger input sizes	18
8	Space tests and analysis	20
8.1	Empirical vs. predicted space	21
8.2	Dynamic programming vs. Hirschberg's linear space	23
9	Conclusions	24
9.1	Runtime	24
9.2	Space	25
9.3	Methodology and tools	25
10	Code structure and description	26
10.1	Code structure	26

10.2	How reproducibility is ensured	26
10.3	How comparison across algorithms is ensured	26
10.4	How time is measured	27
10.5	How memory is measured	28
10.6	How calculated datapoints are created	29
10.6.1	Aggregated data from the experiments	29
10.6.2	Predicted values	29
10.7	Runtime and memory optimizations	30
10.7.1	Runtime	30
10.7.2	Memory	31
11	References	32

Comparison of longest common subsequence (LCS) algorithms

COT 6405 Analysis of Algorithms, Spring 2020

Christian Garbin

1 Introduction

This notebook compares **longest common subsequence** (defined below) algorithms:

- **Brute-force:** generates combinations of subsequences and check if they are common subsequences.
- **Dynamic programming:** takes advantage of common subproblems to not evaluate the same subsequence more than once.
- **Hirschberg's linear space:** a dynamic programming approach, combined with divide-and-conquer, that uses significantly less space than the dynamic programming algorithm.

The comparison measures:

- Runtime efficiency: how long it takes to find a longest common subsequence.
- Space efficiency: how much space is used to find a longest common subsequence.

The code used in the experiments is written in Python 3.x. The code is available in this GitHub repository.

2 Longest common subsequence

2.1 Definition

Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, another sequence Z is a **subsequence** of X if there is a strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \dots, k$, we have $x_{i_j} = z_j$ [CLRS01].

For example:

- Given the sequence $X = \langle A, B, C, B, D, A, B \rangle$
- The sequence $Z = \langle B, C, D, B \rangle$ is a subsequence of X , with indices $\langle 2, 3, 5, 7 \rangle$

Given two sequences, a **common subsequence** is a sequence that is common to both sequences. A **longest common subsequence** (LCS) is a maximum-length common subsequence.

For example:

- Given the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$

- The sequence $\langle B, C, A \rangle$ is a common subsequence
- The sequence $\langle B, C, B, A \rangle$ is a longest common subsequence
- The sequence $\langle B, D, A, B \rangle$ is another LCS, therefore LCSs do not have to be unique

2.2 Applications

Applications of LCS include determining if two organisms are similar by comparing their DNAs. In this case, "similar" can be determined by the longest common subsequence between the DNAs. The longer the subsequence, the more common the organisms are. For example, given an organism with DNA sequence $SEQ = \text{ACAACGACGGTATTAA}$ and two others with DNA sequences $S_1 = \text{CCCGCGTAGGTTCCAAG}$ and $S_2 = \text{GTAACAATGCTACCGTA}$, the closest organism to SEQ is S_1 because it shares a longer common subsequence with SEQ than S_2 does.

```
1      SEQ = ACAACGACGGTATTAA
2  LCS(SEQ, S1) = .C..C..CGGTATTAA. length = 11
3  LCS(SEQ, S2) = ACAA.G.C..TAT.A.. length = 10
```

Building on the DNA example, LCS can be used as a form of compression. Using a pre-built suffix tree, an LCS-based solution compressed the human genome from about 3 GB to just over 8 MB [BAF16].

LCS is also used in version control systems to produce the "diff", the minimal amount of additions and deletions that transform the older version of a file into the new version, also known as the "edit distance" [NAV01]. For example, this output of Git's `diff` command between two versions of a file shows deletion and addition of two lines to transform the old version of the file into the new one:

```
1 @@ -33,15 +33,11 @@ class ModifiedFile(BaseCommit):
2     '''Number of lines added to the file in this commit.'''
3     deleted = Column(Integer)
4     - '''Number of lines deleted from the file in tihs commit.'''
5     + '''Number of lines deleted from the file in this commit.'''
6     - commit_id = Column(String)
7     + commit_id = Column(String, ForeignKey('commit_info.commit_id'))
```

3 Notebook structure

The remainder of this notebook is structured as follows:

- **Algorithms:** describes the algorithms used in the tests.
- **Planning of experiments:** describes the experiments performed, data collected for each one, the analysis performed on them.
- **Initialization and verification:** initializes the notebook (import Python modules, set important environment values) and verifies that the algorithms are working before using them.
- **Measurement and analysis:** run the tests to collect the metrics and analyzes those metrics.

- **Code:** documents relevant pieces of the code and other technical aspects found during the development and execution of the tests.

4 Algorithm descriptions

Three algorithms will be analyzed:

- Brute-force
- Dynamic programming
- Hirschberg's linear space

In all sections below, m is the length of the longest sequence, and n is the length of the shortest sequence.

4.1 Brute-force

Brute-force is the simplest LCS algorithm: generate subsequences of the smaller sequence and check if they are also a subsequence of the larger sequence.

The pseudocode for the algorithm is:

```
1  LCS_BRUTE_FORCE(X, Y)
2    // Pick the shortest sequence to generate subsequences
3    short_seq = shortest_of(X, Y)
4    long_seq = longest_of(X, Y)
5
6    // Try all subsequences of the shortest sequence
7    for i = length(short_seq) to 1
8      // Try all subsequences of length i, one at a time
9      while "there are subsequences of length i to try":
10         subseq = next_subsequence(short_seq, i)
11         if is_subsequence(subseq, long_seq)
12             return subseq
13
14    // Could not find a subsequence
15    return [] // empty sequence
```

Runtime analysis: the algorithm selects the smaller sequence to generate combinations to test. Since there are 2^n combinations of subsequences that can be generated from the subsequence, and each one has to be tested against the larger sequence, the runtime is $m \times 2^n$. In most cases, the second term is much larger than m , making it a $O(2^n)$ algorithm.

Space analysis: A naive implementation would generate all combinations of the smaller sequence ahead of time, using $O(2^n)$ space. An optimized implementation, as shown above, generates one combination of the smaller sequence at a time, using $O(n)$ space.

4.2 Dynamic programming

Dynamic programming makes use of the optimal substructure of the LCS, solving smaller subproblems only once, combining the solutions.

The pseudocode is shown below. It has two parts: first two matrices are constructed to determine the LCS length and how to construct it (a series of "moves"), then the LCS is extracted by going through the moves matrix.

```

1  LCS_LENGTH(X, Y)
2      m = length(X)
3      n = length(Y)
4
5      // c is an m x n matrix with the top row and
6      // left column initialized to zero
7      c = matrix(m, n)
8      for i = 1 to m
9          c[i, 0] = 0
10     for j = 1 to n
11         c[0, j] = 0
12
13     // b is an m x n empty matrix that will hold the
14     // movements to build the LCS
15     b = matrix(m, n)
16
17     for i = 1 to m
18         for j = 1 to n
19             if X[i] == Y[j]
20                 c[i,j] = c[i-1,j-1] + 1
21                 b[i,j] = "diagonal"
22             else if c[i-1,j] >= c[i, j-1]
23                 c[i,j] = c[i-1,j]
24                 b[i,j] = "up"
25             else
26                 c[i,j] = c[i,j-1]
27                 b[i,j] = "down"
28
29     // c[m,n] has the LCS length and b has the
30     // sequences of moves to extract the LCS
31     return c, b
32
33 EXTRACT_LCS(b, X, i, j)
34     lcs = empty_list()
35
36     while i > 0 and j > 0
37         move = b[i,j]
38         if move == "diagonal"
39             lcs = lcs + X[i]
40             i = i - 1
41             j = j - 1
42         else if move == "up"
43             i = i - 1
44         else // "down"
45             j = j - 1
46
47     // The LCS was built from the bottom up,
48     // need revert it before returning
49     return reverse(lcs)
50
51 LCS(X, Y)
52     b, c = LCS_LENGTH(X, Y)
53     lcs = EXTRACT_LCS(b, X, length(X), length(Y))
54     return lcs

```

Runtime analysis: the `LCS_LENGTH` part of the algorithm is $O(m \times n)$, from its two nested loops. The `EXTRACT_LCS` is $O(m + n)$. For large values of m and n , `LCS_LENGTH` dominates the runtime, making the algorithm overall $O(m \times n)$.

Space analysis: the two $m \times n$ matrices in `LCS_LENGTH` are responsible for the space the algorithm needs, thus the space is $O(m \times n)$.

4.3 Hirschberg's linear space

Hirschberg's linear space algorithm [HIR75] is a dynamic programming approach that uses divide-and-conquer. As the name indicates, it makes efficient use of space.

The pseudocode is shown below. It has two parts: a *scoring* (also called *cost*) function to help decide where to divide the current subsequence being analyzed, and the function that divides-and-conquers, based on that score.

```

1  SCORE(X, Y)
2    m = length(X)
3    n = length(Y)
4
5    // A list of of scores, initialized with n zeros
6    scores = list(0 * n)
7
8    for i = 1 to m
9      prev_score = scores
10     for j = 1 to n
11       if X[i] == Y[j]
12         scores[j+1] = prev_scores[j] + 1
13       else
14         scores[j+1] = max(scores[j], prev_scores[j + 1])
15
16    return scores
17
18  LCS(X, Y)
19    m = length(X)
20    n = length(Y)
21    if m == 0
22      // Got to the end of the sequence
23      return []
24    else if m == 1
25      // Last character, check if it is in subsequence
26      if X[1] is in Y
27        return X[1]
28      else
29        return []
30    else
31      // Find where to split the current sequences
32
33      // X is split in the middle
34      i = m / 2
35      XB = X[1:i]
36      XE = X[i+1:m]
37
38      // Y is split based on the scores
39      cost_top_left = SCORE(XB, Y)
40      cost_bottom_right = SCORE(reverse(XE), reverse(Y))
41      cost = cost_top_left + reverse(cost_bottom_right)
42
43      k = index of max(cost) // argmax(cost)
44      YB = Y[1:k]
45      YE = Y[k+1:n]
46
47      // Solve for each part of the split sequences
48      return LCS(XB, YB) + LCS(XE, YE)

```

Runtime analysis: in each step, the sequences under examination are split into two subsequences at $m/2$ and a q based on a cost factor (this is the secret cause of the algorithm). It can be shown that the recurrence is $O(mn)$ [KT05] [FAG16].

Space analysis: a naive implementation of the algorithm creates copies of the sequences as it splits them during the recursive calls, using space $O(m + n)$. An implementation that

passes the original sequences around and uses indices to logically split them (without creating copies), uses space $O(\min(m, n))$ (in the `SCORE` function).

4.4 Runtime and space summary

The following table summarizes the runtime and space characteristics of the algorithms.

Table 1: Runtime and space summary

Algorithm	Runtime	Space
Brute-force	$O(2^n)$	$O(n)$
Dynamic programming	$O(m \times n)$	$O(m \times n)$
Hirschberg	$O(m \times n)$	$O(\min(m, n))$

5 Planning of experiments

The experiments compare the runtime and space of the brute-force, dynamic programming recursive, and Hirschberg's linear space algorithms.

To illustrate the algorithms in a typical application, the tests will use two strings that resemble DNA sequences (a combination of the letters `A`, `C`, `G`, and `T`) and will find an LCS for them. Strings of two sizes will be used, a larger string representing a DNA sequence for an organism and a smaller string representing a DNA strand of interest.

5.1 Runs

Two runs will be executed:

- Phase 1: a smaller run, with smaller input size and fewer repetitions of each algorithm, to collect data to determine the constant c , the constant that accounts for the environment (programming language, computer hardware, and so on). This constant will be used in the next to calculate predicted values.
- Phase 2: a larger run, with larger input sizes and more repetitions of each algorithms ($k=10$), to collect data to analyze the algorithms. The average of these runs will be used as the final number for the algorithms.

Two values will be measured in each run, running time (RT) and memory (space) usage:

- Time: measured with Python's `time` package.
- Memory: measured with Python's `memory_profiler` package.

Details of how measurements were conducted are documented in the code section, later in this document.

5.2 Input size for tests

For phase 1, the following sizes will be used.

Table 2: Phase 1 size of strings to test and how they map to the m and n of the RT and memory analysis.

The DNA strain (X) This is the m in RT and space analysis	The possible common sequences (Y) This is the n in RT and memory analysis
1,000	100
2,000	200
3,000	300
4,000	300, 500, and 1,000
5,000	900, 1,000, and 1,200

For phase 2, the following sizes will be used.

Table 3: Phase 2 size of strings to test and how they map to the m and n of the RT and memory analysis.

The DNA strain (X) This is the m in RT and space analysis	The possible common sequences (Y) This is the n in RT and memory analysis
10,000	500, 800, and 1,000
20,000	1,000, 2,000, and 2,500
30,000	2,000, 3,000, and 4,000

5.3 Data structures

Experimental data, the strings, will be stored in the standard data structures for string representation, usually mapped to a constant-time access continuous array in programming languages. Auxiliary data structures to keep track of intermediate results will be kept either in Python arrays or NumPy arrays, whichever is more performant for a specific piece of code.

5.4 Input generation

Strings for the tests will be generated using a pseudorandom number generator initialized with a seed, to ensure the repeatability of the experiments (the same sequence is generated every time). The strings will be generated only once, before each algorithm is executed, to ensure that the results can be compared with each other.

5.5 Graphs and tables

For each algorithm, two tables will be filled in:

- RT analysis: theoretical vs. empirical RT.
- Memory usage: theoretical vs. empirical memory usage.

The following table illustrates the RT analysis for an algorithm.

Table 4: RT analysis table example, using the brute-force algorithm as illustration

Sequence size	Subsequence size	Empirical		Ratio (empirical RT / theoretical complexity)	Predicted RT	% error
		RT (ms)	Theoretical complexity			
10,000	100		2^{100}			
...	...					
30,000	4,000		$2^{4,000}$			

The table columns are computed as follows:

- *Ratio* measures the ratio between the empirical and the theoretical complexity. Its value is always > 0 . It is used to calculate the constant c .
 - The constant c measures the overhead of the steps (computer instructions) that are outside the main loops (or recursive calls) of the algorithms. It is determined as the maximum of the ratio values (with outliers discarded), i.e. the maximum value of the *Ratio* column.
- *Predicted RT* is computed as the constant $c \times \text{Theoretical complexity}$.
- *% error* measures the discrepancy between the predicted and the empirical time: $(\text{Empirical RT} - \text{Predicted RT}) / \text{Empirical RT} \times 100$.

The following table illustrates memory usage for an algorithm.

Table 5: Memory usage analysis table example, using the brute-force algorithm as illustration

Sequence size	Subsequence size	Empirical memory usage (MiB)	Predicted memory usage (MiB)	% error
10,000	100		$100/(1024*1024)$	
...	...			
30,000	4,000		$4,000/(1024*1024)$	

The table columns are computed as follows:

- *Predicted memory usage* is the number of characters needed, times 1 byte per character, divided by $1,024*1,024$ to transform to MiB.
 - "1 byte per character" comes from the variable-length encoding of strings in Python using the CPython environment. Because we are representing DNA strands, we are using only ASCII characters, which are represented as 1 byte [CPY20] [GOL20].
- *% error* measures the discrepancy between the theoretical and empirical memory usage: $(Empirical\ memory\ usage / Predicted\ memory\ usage) / Empirical\ memory\ usage \times 100$.

For the other algorithms, the theoretical values will be adjusted as follows:

- Dynamic programming:
 - Theoretical complexity = $m \times n$
 - Predicted memory usage = $m \times n \times number\ of\ bytes\ per\ matrix\ cell$, divided by $1,024*1,024$ to transform to MiB.
- Hirschberg's linear space algorithm
 - Theoretical complexity = $m \times n$
 - Predicted memory usage = $min(m, n) \times number\ of\ bytes\ per\ array\ entry$, divided by $1,024*1,024$ to transform to MiB.

Once the tables are filled in, two sets of graphs will be created:

- **Algorithm comparison:** this set of graphs compares the empirical runtime and memory usage of the algorithms. There will be one runtime and one memory usage plot for all algorithms (two graphs).
- **Theoretical vs. empirical results:** this set of graphs compares the empirical runtime and memory usage of each algorithm. There will be one set of plots for each algorithm and each size, for a total of 9 graphs (3 algorithms, 3 sizes).

The following graphs will be generated for algorithm comparison:

- RT comparison: a horizontal bar graph with the runtime in ms (horizontal axis) for each algorithm, grouped by the input size. A horizontal bar graph will be used because of the expected large values for large input sizes. This representation makes better use of space.
- Memory comparison: similar to the graph above, using memory usage as the horizontal axis.

The following graphs will be generated for each algorithm for the theoretical vs. empirical results:

- Theoretical vs. empirical runtime in ms for each input size
- Theoretical vs. empirical memory usage in KiB for each input size

5.6 Programming language

The experiments use Python 3.x in a Jupyter Notebook environment.

6 Initialization and verification

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4 pd.set_option('precision', 3)
```

```
1 # Force reload because this piece of code frequently changes
2 import importlib
3 import metrics as m
4 importlib.reload(m);
```

Check that the algorithms work by testing them against controlled input.

There are three part to the tests:

1. Automated tests that check against well-defined inputs. They are meant to be easy to debug, in case an algorithm fails.
2. Tests with longer inputs that simular DNA strands. They test more realistic scenarios, but still short enough to run fast.
3. A visual check, by printing the aligned subsequence. They guard against the test code itself having a failure that generates false positives.

```
1 m.test()
```

```
1 All basic tests passed
2 All DNA tests passed
3 Visual inspection:
4               lcs_brute_force: CCGTTAATCACAGCCCCTTTGAGCGCTGTG
5               lcs_brute_force: ..GT..CA.A.....TT..A.C.C.GT. size=13
6       lcs_dynamic_programming_matrix_numpy: ..GT..CA.A.....TT..A.C.C.GT. size=13
7               lcs_hirschberg_numpy: ..GT..CA.A.....TT..A.C.C.GT. size=13
```

7 Runtime tests and analysis

To illustrate a real-life scenario, the code checks if a DNA strand is part of a larger DNA sequence [WIK20a].

7.1 Constant c calculation

This section calculates the constant c for each algorithm. This is the constant that accounts for instructions that are not in the loops. For example, given a runtime $O(n^2)$, the constant c allows us to write the runtime more precisely as $c \times n^2$.

The constant is also affected by the language and compiler or interpreter used. In this notebook it is calculated with dynamic analysis: run the code and calculate the overhead.

Each algorithm was executed once with different input sizes, with the pairs representing the length of X and Y , respectively: (1,000, 100), (2,000, 200), (3,000, 300), (4,000, 300), (4,000, 500), (4,000, 1,000).

Once the algorithms are run, we calculate c as $\max(rt_1, rt_2, \dots, rt_n)$, excluding outliers where applicable.

```
1 rt_results_raw, rt_results_summary = m.runtime(m.seq_phase1, verbose=1,
2       file='runtime-phase1')
```

```
1 Loading from file
```

We now have two Pandas DataFrames:

- `rt_results_raw`: results from all 10 executions of each algorithms and each input size.
- `rt_results_summary`: average of the executions for each each algorithm and input size.

Using these DataFrames, the following sections calculate three constants, one for each algorithm. They are stored in the following variables:

- `c_bf`: the constant for the brute-force algorithm.
- `c_dp`: the constant for the dynamic programming algorithm.
- `c_h`: the constant for the Hirschberg linear space algorithm.

The dataframes contain metrics for all algorithms. The sections below filter the dataframe for the algorithm analyzed as needed.

7.1.1 Constant c for brute-force

```
1 rt_bf, c_bf = m.add_runtime_analysis(rt_results_summary, m.ALG_BRUTE_FORCE)
2 print('c_bf={}'.format(c_bf))
3 display(rt_bf.iloc[:,1:])
```

```
1 c_bf=5.206481974415714e-32
```

	Sequence size	Subsequence size	Empirical RT (ms)	Theoretical complexity \
1	0	1000	0.066	1.268e+30
2	1	2000	0.173	1.607e+60
3	2	3000	0.206	2.037e+90
4	3	4000	0.256	2.037e+90
5	4	4000	0.495	3.273e+150
6	5	4000	156.909	1.072e+301
7	6	5000	0.601	8.453e+270
8	7	5000	0.658	1.072e+301
9	8	5000	0.760	inf
10				
11				
	Ratio	Predicted RT	% error	
12	0	5.206e-32	6.600e-02	0.000e+00
13	1	1.077e-61	8.366e+28	-4.836e+31
14	2	1.011e-91	1.061e+59	-5.148e+61
15	3	1.257e-91	1.061e+59	-4.143e+61
16	4	1.512e-151	1.704e+119	-3.443e+121
17	5	1.464e-299	5.579e+269	-3.555e+269
18	6	7.110e-272	4.401e+239	-7.323e+241
19	7	6.141e-302	5.579e+269	-8.478e+271
20	8	0.000e+00	inf	-inf
21				

We can see in the table a very large error for most tests of the brute-force algorithm. This is caused by the "luck factor" of this algorithm: if we are lucky and generate a combination early on that happens to be a common subsequence, the algorithm terminates quickly. The probability of generating a common subsequence is high in this case because of the reduced amount of possible combinations we have when using only the four letter of a DNA sequence.

Given the "luck factor" of this case, analyzing the predicted vs. empirical runtime will not be insightful.

7.1.2 Constant c for dynamic programming

```

1 rt_dp, c_dp = m.add_runtime_analysis(rt_results_summary, m.ALG_DYNAMIC_PROGRAMMING
2 )
3 print('c_dp={}'.format(c_dp))
4 display(rt_dp.iloc[:,1:])

```

```

1 c_dp=0.0001627695555555558

```

1		Sequence size	Subsequence size	Empirical RT (ms)	\
2	9	1000	100	15.257	
3	10	2000	200	62.163	
4	11	3000	300	142.757	
5	12	4000	300	182.718	
6	13	4000	500	308.720	
7	14	4000	1000	612.753	
8	15	5000	900	732.463	
9	16	5000	1000	777.084	
10	17	5000	1200	938.708	
11					
12		Theoretical complexity	Ratio	Predicted RT	% error
13	9	100000	1.526e-04	16.277	-6.685
14	10	400000	1.554e-04	65.108	-4.737
15	11	900000	1.586e-04	146.493	-2.617
16	12	1200000	1.523e-04	195.323	-6.899
17	13	2000000	1.544e-04	325.539	-5.448
18	14	4000000	1.532e-04	651.078	-6.255
19	15	4500000	1.628e-04	732.463	0.000
20	16	5000000	1.554e-04	813.848	-4.731
21	17	6000000	1.565e-04	976.617	-4.038

Since all results are within a small margin of error, none of them will be discarded for the calculation.

7.1.3 Constant c for Hirschberg's linear space

```
1 rt_h, c_h = m.add_runtime_analysis(rt_results_summary, m.ALG_HIRSCHBERG)
2 print('c_h={}'.format(c_h))
3 display(rt_h.iloc[:,1:])
```

```
1 c_h=0.00034574999999999363
```

1		Sequence size	Subsequence size	Empirical RT (ms)	\
2	18	1000	100	34.575	
3	19	2000	200	135.058	
4	20	3000	300	290.896	
5	21	4000	300	387.301	
6	22	4000	500	621.257	
7	23	4000	1000	1226.753	
8	24	5000	900	1399.517	
9	25	5000	1000	1542.752	
10	26	5000	1200	1846.447	
11					
12		Theoretical complexity	Ratio	Predicted RT	% error
13	18	100000	3.457e-04	34.575	0.000
14	19	400000	3.376e-04	138.300	-2.400
15	20	900000	3.232e-04	311.175	-6.971
16	21	1200000	3.228e-04	414.900	-7.126
17	22	2000000	3.106e-04	691.500	-11.307
18	23	4000000	3.067e-04	1383.000	-12.737
19	24	4500000	3.110e-04	1555.875	-11.172
20	25	5000000	3.086e-04	1728.750	-12.056
21	26	6000000	3.077e-04	2074.500	-12.351

Some of the results are off by more than 10%, but not by much, so we will keep all of them for the calculation.

7.2 Empirical vs. predicted RT graphs

The graphs below compare the empirical runtime with the predicted runtime using the smaller input sizes. The predicted runtime is calculated as $c \times \text{theoretical complexity}$.

Each of the graphs below shows the empirical vs. predicted runtime for an algorithm. The empirical runtime is the actual running time of the algorithms, while the empirical runtime is calculated as $c \times$ the theoretical complexity. Note that the vertical axis is in different scales, to accommodate the large differences between the brute-force algorithm and the others. Later sections will compare the dynamic programming algorithm with Hirschberg's.

```
1 graph_number = 0
2 def plot_empirical_predicted_rt(data, title):
3     '''Auxiliary function to plot the graphs consistently for all algorithms.'''
4     global graph_number
5     graph_number += 1
6     fig = plt.figure(figsize=(10,4))
7     sns.lineplot(x=m.DF_THEORETICAL_COMPLEXITY, y=m.DF_EMPIRICAL_RT, data=data)
8     sns.lineplot(x=m.DF_THEORETICAL_COMPLEXITY, y=m.DF_PREDICTED_RT, data=data)
9     plt.title('Graph {} - {} empirical vs. predicted runtime'.format(
10         graph_number, title))
11     plt.ylabel("Runtime (ms)")
```



```

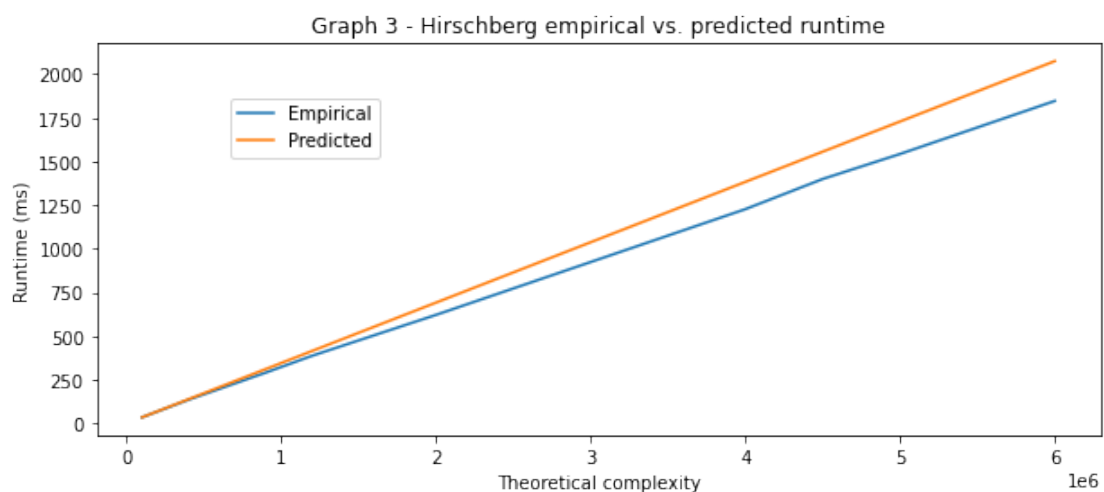
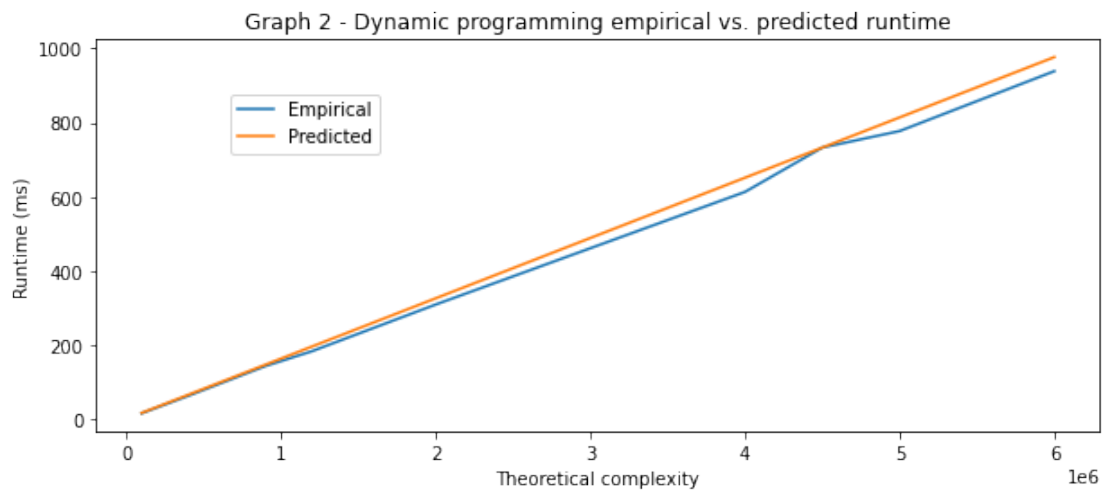
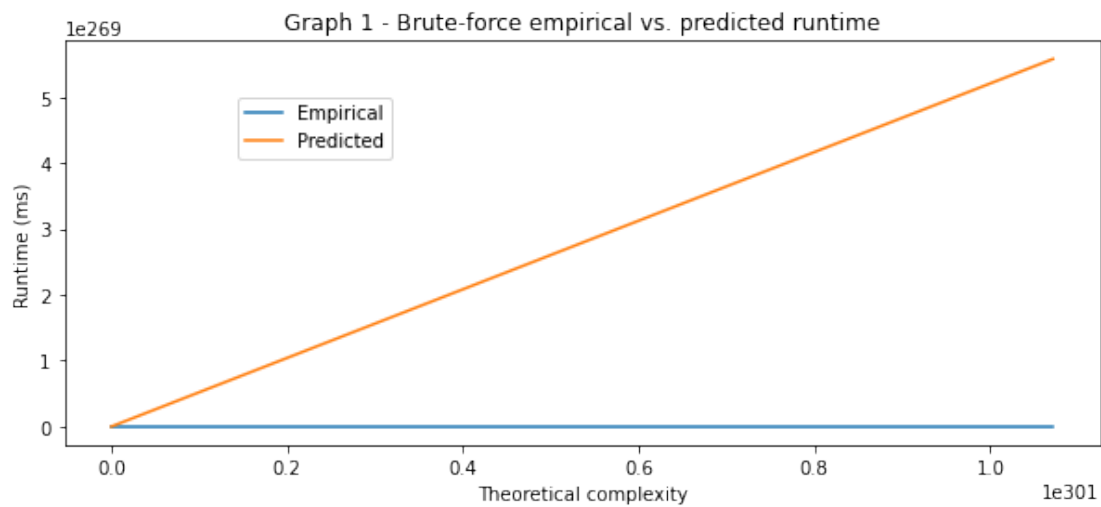
12 fig.legend(['Empirical', 'Predicted'], bbox_to_anchor=(0.3, 0.8))
13 plt.show()

```

```

1 for alg in (rt_bf, rt_dp, rt_h):
2     plot_empirical_predicted_rt(alg, alg.iloc[0][m.DF_ALGORITHM])

```



From these graphs we can see that:

1. The brute-force graph shows again the "luck" effect, where the runtime is low because a subsequence just happens to be found early on. It could have gone the other way, as it in fact did in one of the experiments, where after running for eight hours, it still did not find a common subsequence for $m = 5,000$ and $n = 2,000$.
2. Both the dynamic programming and the Hirschberg algorithms track closely to their predicted runtime.
3. The Hirschberg algorithm's error is larger for larger input sizes, likely because the nature of the implementation and the programming language. It is a recursive algorithm and Python does not support tail recursion optimization [WIK20b] [ROS09]. Thus the loops have the extra cost of function calls that the dynamic programming algorithm does not have. The cost of the function calls is relatively more expensive for small input size, resulting in overestimating c . An improvement for this case could be to do a more rigorous outlier elimination when calculating c .

7.3 Graphs for larger input sizes

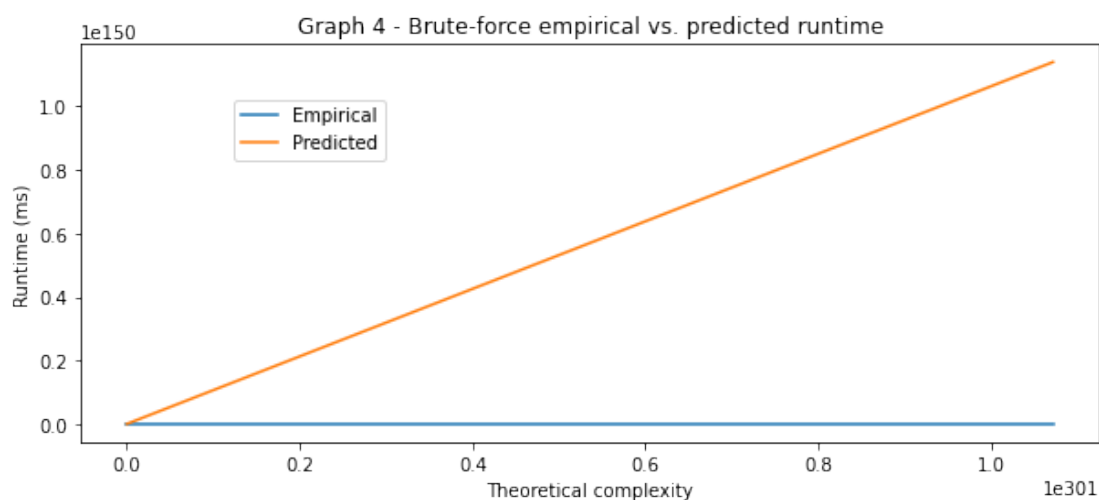
The graphs below compare the empirical runtime with the predicted runtime using the larger input sizes. As before, the predicted runtime is calculated as $c \times \text{theoretical complexity}$.

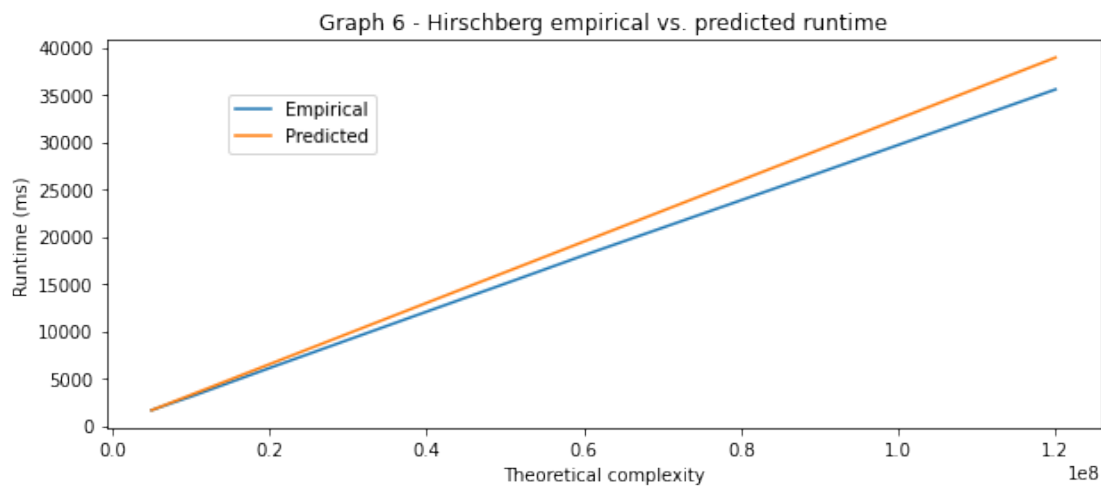
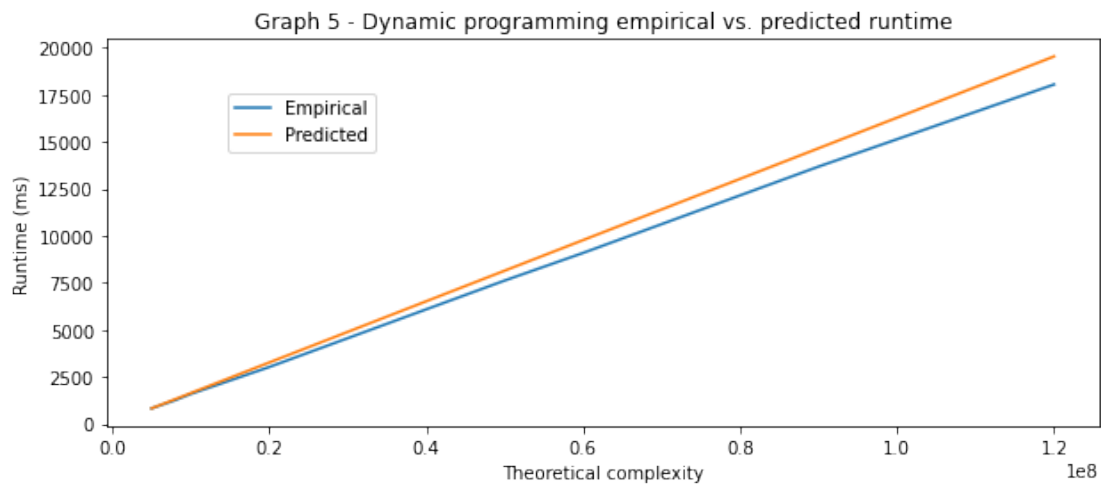
The graphs show the same trend lines as the graphs for the smaller input sizes.

```
1 rt_results_raw, rt_results_summary = m.runtime(m.seq_phase2, verbose=1,
2       file='runtime-phase2')
```

```
1 Loading from file
```

```
1 for alg in (m.ALG_BRUTE_FORCE, m.ALG_DYNAMIC_PROGRAMMING, m.ALG_HIRSCHBERG):
2     rt, _ = m.add_runtime_analysis(rt_results_summary, alg)
3     plot_empirical_predicted_rt(rt, rt.iloc[0][m.DF_ALGORITHM])
```



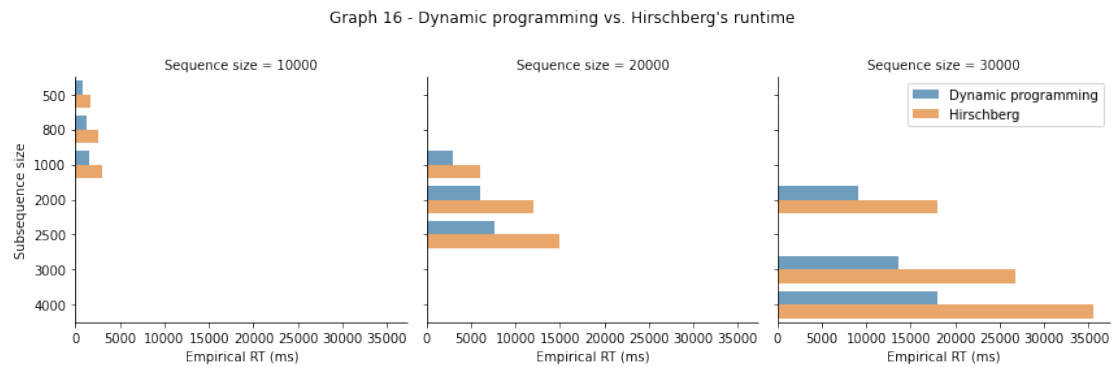


Graphing the algorithms separately does not clearly show how their runtime compare with each other. The next set of graphs compares the runtime of the dynamic programming algorithm with Hirschberg's algorithm. The brute-force algorithm is not shown because its complexity is on another scale entirely, therefore not comparable with the other two algorithms.

The first graph compares the runtime for each sequence size, and within each of those, the subsequence size. It shows that both algorithms approximately linearly with $m \times n$, as expected.

```
1 dp_summary, _ = m.add_runtime_analysis(rt_results_summary, m.
    ALG_DYNAMIC_PROGRAMMING)
2 h_summary, _ = m.add_runtime_analysis(rt_results_summary, m.ALG_HIRSCHBERG)
3 summary = pd.concat([dp_summary, h_summary])
```

```
1 graph_number += 1
2 g = sns.catplot(col=m.DF_SEQ_SIZE, y=m.DF_SUBSEQ_SIZE, x=m.DF_EMPIRICAL_RT, data=
    summary,
3             hue=m.DF_ALGORITHM, height=4, aspect=1, legend=False, alpha=0.7,
4             kind='bar', orient='h')
5 g.fig.suptitle('Graph {} - Dynamic programming vs. Hirschberg's {}'.format(
6     graph_number, 'runtime'))
7 g.fig.subplots_adjust(top=0.8)
8 plt.legend(loc='upper right')
9 plt.show()
```



The next graph show the predicted vs. empirical runtime, with the same scale for the vertical axis.

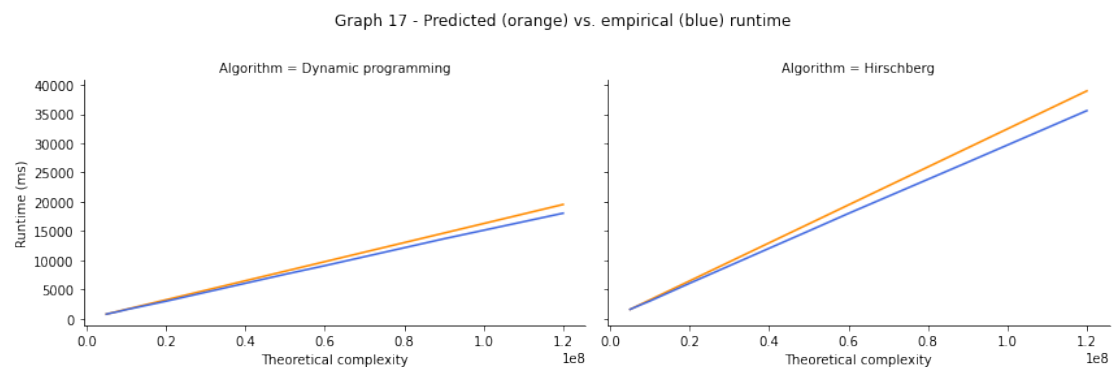
The graphs show that:

- Both algorithms track closely to their predicted runtime.
- Although they have the same growth rate (both have $O(mn)$ complexity), Hirschberg's algorithm has a higher slope, resulting from its higher c constant.

```

1 graph_number += 1
2 g = sns.FacetGrid(summary, col=m.DF_ALGORITHM, height=4, aspect=1.5)
3 g.map(plt.plot, m.DF_THEORETICAL_COMPLEXITY, m.DF_PREDICTED_RT,
4       color='darkorange')
5 g.map(plt.plot, m.DF_THEORETICAL_COMPLEXITY, m.DF_EMPIRICAL_RT,
6       color='royalblue')
7 g.set_ylabels('Runtime (ms)').fig.subplots_adjust(top=0.8)
8 g.fig.suptitle('Graph {} - Predicted (orange) vs. empirical (blue) runtime'.format
9               (
10                graph_number))
10 plt.show()

```



8 Space tests and analysis

```

1 mem_results_raw, mem_results_summary = m.memory(m.seq_phase2, verbose=1,
2         file='memory-phase2')
3 mem_results_summary = m.add_memory_analysis(mem_results_summary)

```

```

1 Loading from file

```

We now have two Pandas DataFrames:

- `mem_results_raw`: results from all 10 executions of each algorithms and each input size.
- `mem_results_summary`: average of the executions for each each algorithm and input size.

The DataFrames contain metrics for all algorithms. Sections below filters the row for the algorithm analyzed in that section.

8.1 Empirical vs. predicted space

The tables below show the empirical and predicted space in MiB. Using these values, an error column is also shown.

Observations from these tables:

1. Measuring small amounts of memory used in short period of times, as used by the brute-force algorithm in all cases and by the other algorithms with smaller input sizes, is unreliable. There is a discussion about the method used later in the notebook, in the code section. Based on these observations, the results from the brute-force algorithm will not be used in further analysis.
2. As the runtime increases and the amount of memory used grows larger, the measurements become more reliable.

```
1 for algorithm in m.algorithms:
2     alg= algorithm[1]
3     df = mem_results_summary[mem_results_summary[m.DF_ALGORITHM] == alg]
4     print('\n{} memory analysis'.format(alg))
5     display(pd.pivot_table(df,
6         values=[m.DF_EMPIRICAL_SPACE, m.DF_PREDICTED_SPACE, m.DF_ERROR],
7         index=[m.DF_SEQ_SIZE, m.DF_SUBSEQ_SIZE]))
```

1 Brute-force memory analysis

	Sequence size	Subsequence size	% error	Empirical space (MiB)	\
3	10000	500	-inf	0.000	
4		800	-inf	0.000	
5		1000	-inf	0.000	
6	20000	1000	-inf	0.000	
7		2000	-inf	0.000	
8		2500	38.965	0.004	
9	30000	2000	-inf	0.000	
10		3000	75.586	0.012	
11		4000	87.793	0.031	
13	Predicted space (MiB)				
14	Sequence size	Subsequence size			
15	10000	500		4.768e-04	
16		800		7.629e-04	
17		1000		9.537e-04	
18	20000	1000		9.537e-04	
19		2000		1.907e-03	
20		2500		2.384e-03	
21	30000	2000		1.907e-03	
22		3000		2.861e-03	
23		4000		3.815e-03	

1 Dynamic programming memory analysis

	% error	Empirical space (MiB)	\
--	---------	-----------------------	---

2	Sequence size	Subsequence size		
3	10000	500	-162660.417	0.012
4		800	0.134	30.559
5		1000	0.167	38.211
6	20000	1000	0.101	76.371
7		2000	0.058	152.676
8		2500	0.047	190.824
9	30000	2000	0.057	229.012
10		3000	0.037	343.449
11		4000	0.029	457.895
12				
13			Predicted space (MiB)	
14	Sequence size	Subsequence size		
15	10000	500		19.073
16		800		30.518
17		1000		38.147
18	20000	1000		76.294
19		2000		152.588
20		2500		190.735
21	30000	2000		228.882
22		3000		343.323
23		4000		457.764

1 Hirschberg memory analysis

1			% error	Empirical space (MiB)	\
2	Sequence size	Subsequence size			
3	10000	500	-inf	0.000	
4		800	92.188	0.039	
5		1000	91.862	0.047	
6	20000	1000	91.862	0.047	
7		2000	91.862	0.094	
8		2500	91.862	0.117	
9	30000	2000	86.979	0.059	
10		3000	97.287	0.422	
11		4000	97.543	0.621	
12					
13			Predicted space (MiB)		
14	Sequence size	Subsequence size			
15	10000	500		0.002	
16		800		0.003	
17		1000		0.004	
18	20000	1000		0.004	
19		2000		0.008	
20		2500		0.010	
21	30000	2000		0.008	
22		3000		0.011	
23		4000		0.015	

```

1 def plot_empirical_predicted_mem(data, title):
2     '''Auxiliary function to plot the graphs consistently for all algorithms.'''
3     global graph_number
4     graph_number += 1
5     fig = plt.figure(figsize=(10,4))
6     sns.lineplot(x=m.DF_THEORETICAL_COMPLEXITY, y=m.DF_EMPIRICAL_RT, data=data)
7     sns.lineplot(x=m.DF_THEORETICAL_COMPLEXITY, y=m.DF_PREDICTED_RT, data=data)
8     plt.title('Graph {} - {} empirical vs. predicted runtime'.format(
9         graph_number, title))
10    plt.ylabel("Runtime (ms)")
11    fig.legend(['Empirical', 'Predicted'], bbox_to_anchor=(0.3, 0.8))
12    plt.show()

```

```

1 for alg in (rt_bf, rt_dp, rt_h):
2     plot_empirical_predicted_rt(alg, alg.iloc[0][m.DF_ALGORITHM])

```

8.2 Dynamic programming vs. Hirschberg's linear space

Hirschberg's algorithm main advantage over the standard dynamic programming approach is its efficient use of space.

Before going into the analysis, a review of some implementation details that affect the space characteristics of the algorithms:

1. The classic dynamic programming algorithm uses two $m \times n$ matrices, one for the length and another for the direction of the moves. Space-optimized implementations combine these matrices into one, reserving bits in each cell for the length and direction. Such an implementation was used here. It is discussed in the code section of the notebook.
2. Hirschberg's traditional implementation is recursive. In languages that do not support tail recursion, such as Python, each recursion creates a new stack frame. Thus, some of the memory used by the algorithm is in the form of the stack frames, in addition to the arrays it needs for the algorithm itself.
3. Also for Hirschberg's, a simplistic implementation of the algorithm creates copies of the sequences as it finds where to split them. A space-optimized algorithm uses indices into the original sequence to avoid creating copies. Such an implementation was used here. It is discussed in the code section of the notebook.

With that in mind, from the tables we observe that:

1. The dynamic programming algorithm tracks closely to the predicted space. This is due to two factors. First, it uses a large amount of memory, which seems to favor this particular method of measuring memory utilization (discussion in the code section). And second, it uses loops, as opposed to recursion, which does not create the overhead of stack frames (in languages without tail recursion).
2. The Hirschberg's linear space algorithm has a large error. This error is likely caused by it being a recursive algorithm. Some of the memory measured during the execution comes from the stack frames created in each recursion.

Finally, the most important observation is the difference between the dynamic programming and Hirschberg's linear space algorithm. As expected, dynamic programming uses significantly more memory.

On the other hand, in this implementation, dynamic programming is twice as fast as Hirschberg's. In another programming language, with better support for tail recursion, their performance may be comparable.

In most real-life applications, with large input sizes, a fine-tuned implementation of Hirschberg's linear space is the preferable option. In some cases, for very large input sizes, it may be the only feasible option.

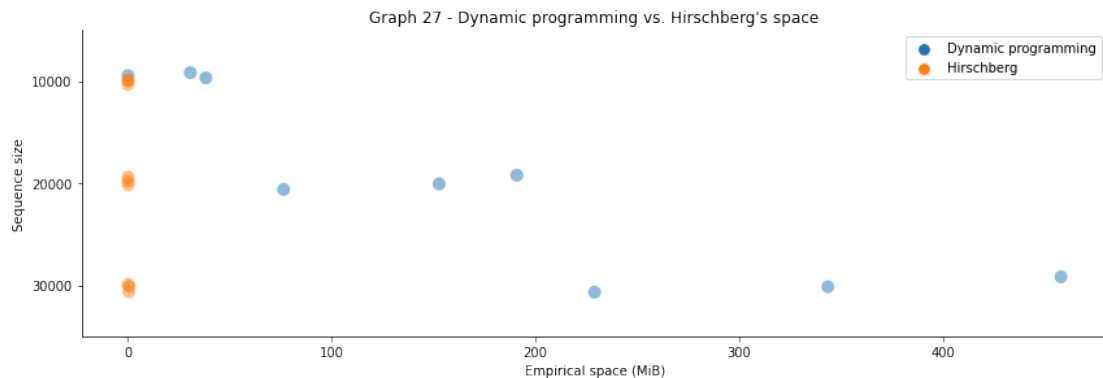
To illustrate the dramatic difference in space, the graph below shows the empirical space used by the two algorithms side-by-side, for each input size. The horizontal axis is the empirical space. Each dot represents a subsequence for a sequence size (vertical axis). We can see

that Hirschberg's linear space algorithm (orange) never goes past one MiB, while the dynamic programming algorithm (blue) escalates quickly, going past hundreds of MiBs, reaching almost half a gigabyte for the largest combination of sequences.

```

1 graph_number += 1
2 df = mem_results_summary[
3     (mem_results_summary[m.DF_ALGORITHM] == m.ALG_DYNAMIC_PROGRAMMING) |
4     (mem_results_summary[m.DF_ALGORITHM] == m.ALG_HIRSCHBERG)]
5 sns.catplot(y=m.DF_SEQ_SIZE, x=m.DF_EMPIRICAL_SPACE, hue=m.DF_ALGORITHM, data=df,
6             orient='h', height=4, aspect=3, legend=False, s=10, alpha=0.5)
7 plt.title('Graph {} - Dynamic programming vs. Hirschberg's space'.format(
8             graph_number))
9 plt.legend(loc='upper right')
10 plt.show()

```



9 Conclusions

9.1 Runtime

From the runtime experiments we can conclude that:

1. The brute-force algorithm is basically a coin flip, where the odds for "it will run quickly" are much lower than the odds for "it will run very, very slowly". It may find an LCS in extremely fast times (under 100 ms), or it may take hours and still not find an LCS (as it happened once during the research for this project). However, in some applications, where the probability of finding a common subsequence is high (a combination of small input sizes and small set of characters to pick from), running the brute-force algorithm in parallel with a dynamic programming algorithm may be worthwhile. When it pays off, it pays off big.
2. The dynamic programming and Hirschberg's algorithm have stable runtime characteristics, that is given an input size, it is easy to calculate how long it will take to find an LCS. Predictability is a desirable characteristic in real-life applications, making these algorithms useful for large input sizes.

9.2 Space

1. The pseudocode for the algorithms are, as a general rule, not a good example of memory efficiency. For example, the pseudocode for the dynamic programming algorithm usually shows two tables, one for the length and another for the moves. While this approach makes the pseudocode easy to understand, it is not an efficient utilization of space. To make the most space-intensive algorithms work, careful use of memory management techniques is needed.
2. The difference in space utilization between the dynamic programming and Hirschberg's linear space algorithms is astounding when they are put side by side. It shows one of them barely using one MiB, while the other reaches half of a gigabyte to perform the same work. Hats off to Dr. Hirschberg.

9.3 Methodology and tools

1. Getting the algorithms to run fast and use a reasonable amount of memory requires knowledge of the particular environment (e.g a Python environment, compared to a C++ environment). The naive implementation, one that follows the pseudocode from textbooks closely, is usually slow, uses too much memory, or both.
2. Python is a good choice for experimentation, but not for performance. For example, for the brute-force algorithm Python's `itertools` is a fast and efficient way to generate combinations. On the other hand, the lack of tail recursion handicaps algorithms that make use of recursion, such as Hirschberg's. This introduces potentially artificial differences between the algorithms, i.e. differences that are caused by the environment, not necessarily by fundamental differences in the algorithms.
3. Jupyter notebooks speeds up the "experiment, evaluate" cycle greatly. It also makes the process repeatable and transparent by exposing the code used for analysis, facilitating peer review of the methods and assumptions used in the work.
4. Measuring memory usage was surprisingly hard, especially when the usage was on the low side (less than one MiB). Part of the reason is the environment (Python, which is a mixture of reference counting and garbage collection) and tools (`memory_profile`). If I had to do this again, I would write these pieces of the code in C or C++, as written in another point, and spawn individual processes for each test case, measuring memory usage with operating system tools, instead of language modules/libraries.

To summarize, if I had to do it again, I would have written the algorithms in C or C++ to take advantage of their performance and write the results into a file, then use Python to read the file and perform the analysis in a Jupyter notebook. This approach would combine the strengths of all tools: the performance of C/C++, the reporting tools from Python (Pandas, Seaborn, among others), and the transparency of analysis in the Jupyter notebook.

10 Code structure and description

This section highlights pieces of the code that are significant for the experiments.

10.1 Code structure

The code is organized in the following folders and files:

- Main folder
 - `metrics.py` drives the collection of measurements and empirical vs. predicted calculations.
 - `report.ipynb` is this notebook - it analyzes the results and is the source for the PDF report.
 - `lcs_brute_force.py`, `lcs_dynamic_programming_matrix_numpy.py`, and `lcs_hirschberg_numpy.py` have the code for the respective algorithms.
- `data` has the cached raw test data - they are saved in CSV files after the runs.
- `utils` has several utilities functions, such as DNA sequence generation, LCS verification, and others.
- `alternatives` has different versions of the algorithms tried out when running the experiments - see the header in each file for the details of each variation.

10.2 How reproducibility is ensured

The code uses a pseudorandom generator to create the sequences used in the test. To ensure that the results are consistent across runs, the pseudorandom generator is initialized with a seed as soon as the program starts.

This snippet is from the main function in `metrics.py`, the module that drives the experiments. It shows the pseudorandom generator being initialized right after the program starts up. From this point on, any function that relies on random numbers will be driven by the same seed.

```
1 if __name__ == "__main__":
2     if len(sys.argv) is not 2:
3         print('Specify phase1 or phase2')
4         exit()
5
6     # For consistency across runs
7     random.seed(42)
```

10.3 How comparison across algorithms is ensured

To make sure we can compare one algorithm against the other, we need to use the same sequence and subsequence for all algorithms.

The code does that by creating the sequences only once, then executing all algorithms on the same sequences, for the same number of repetitions.

This code snippet from `metrics.py` shows these steps (code is cleaned a bit to highlight important parts). First the sequences for a given input size are generated, then all algorithms run on those sequences.

```
1 for dna_size, dna_strand_size in sequences:
2     # Create the test strings only once to correctly compare algorithms
3     dna = lcs_utils.random_dna_sequence(dna_size)
4     dna_strand = lcs_utils.random_dna_sequence(dna_strand_size)
5
6     for alg in algorithms:
7         for i in range(repeat):
8             _run_test(alg, dna, dna_strand)
```

The `for` loops are driven by data structures, further ensuring that the same input is used for all algorithm.

This is the `sequences` data structure used in the outer loop, specifying the input sizes (one for each phase):

```
1 seq_phase1 = [
2     (1_000, 100), (2_000, 200), (3_000, 300),
3     (4_000, 300), (4_000, 500), (4_000, 1_000),
4     (5_000, 900), (5_000, 1_000), (5_000, 1_200),
5 ]
6
7 seq_phase2 = [
8     (10_000, 500), (10_000, 800), (10_000, 1_000),
9     (20_000, 1_000), (20_000, 2_000), (20_000, 2_500),
10    (30_000, 2_000), (30_000, 3_000), (30_000, 4_000),
11 ]
```

This is the `algorithms` data structure used in the internal loop, specifying the algorithms to test:

```
1 Algorithm = namedtuple('Algorithm', ['function', 'description'])
2 algorithms = [
3     Algorithm(lcs_brute_force.lcs, ALG_BRUTE_FORCE),
4     Algorithm(lcs_dynamic_programming_matrix_numpy.lcs,
5               ALG_DYNAMIC_PROGRAMMING),
6     Algorithm(lcs_hirschberg_numpy.lcs, ALG_HIRSCHBERG),
7 ]
```

10.4 How time is measured

Time is measured with Python's time module, using the `process_time` function. It was chosen because it accounts for both user and system time:

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep.

Time measurement is done by getting the time before the LCS algorithm starts and again when it returns, then multiplied by 1,000 to convert to milliseconds:

```
1 start = time.process_time()
2 lcs = alg.function(dna, dna_strand)
3 total_time = (time.process_time() - start) * 1000
```

Note that this method includes the first call to the LCS function in the time. However, the time to perform the initial call is negligible in most cases.

10.5 How memory is measured

Memory is measured with the `memory_profiler` module, using its `memory_usage` function.

The LCS algorithm is invoked through `memory_usage`, which collects the process memory usage at the specified intervals:

```
1 mem_usage = memory_usage((alg.function, (dna, dna_strand)),
2                           interval=0.01)
```

A short interval (0.01) was used because some of the tests run in under 100 ms. This small interval gives more datapoints in those cases.

Because the `memory_usage` function measures the memory used by the entire process, not only the function, and Python is a garbage collected language, we must eject objects that are ready to be garbage-collected, so they don't count against memory usage:

```
1 # Garbage-collect to not measure memory left over from other runs
2 gc.collect()
```

Also because `memory_usage` measures the memory used by the entire process, we need to measure how much memory the process is already using before the LCS algorithm is executed. This is done by calling an empty function a few times.

```
1 # Get the current memory usage of the Python environment
2 mem_baseline = min([min(memory_usage(
3     (lcs_empty.lcs, (dna, dna_strand)), interval=0.01))
4     for _ in range(3)])
```

Measuring memory with this method slows down the code. The effect is more visible in fast algorithms and small input sizes. Because of this, memory usage is measured separately from runtime measurement. First the runtime measurements are collected for all algorithms and input sizes, then the algorithms are executed again for the same input sizes to measure memory usage.

Despite all these steps, measuring memory usage consistently proved to be more difficult than anticipated. The measurements in this report are reasonably accurate, but they should be used as an "order of magnitude" measure, not as an exact number. For example, we can decisively say that Hirschberg's algorithm uses just under one MiB even for the largest input size, while dynamic programming uses over 400 MiB, but we cannot say that Hirschberg's uses (for example) exactly 448.34 MiB less memory than dynamic programming.

If I had the chance to do this again, I would use a different method to measure memory. For example, spawn a separate program just with the algorithm execution and use operating

system tools to measure the memory usage of that process, for example with Linux `time` command.

10.6 How calculated datapoints are created

10.6.1 Aggregated data from the experiments

In phase 2 of the experiments we run each algorithm ten times to get better measurements. This is controlled by the `repeat` parameter:

```

1  if __name__ == "__main__":
2      ...
3      elif test == "phase2":
4          print('Running phase 2 tests')
5          runtime(seq_phase2, repeat=10, verbose=2, file='runtime-phase2')
6          memory(seq_phase2, repeat=10, verbose=2, file='memory-phase2')
7
8  ...later in the test function
9
10 def _runtime_tests(sequences, repeat=2, verbose=1):
11     for dna_size, dna_strand_size in sequences:
12         ...
13         for alg in algorithms:
14             for i in range(repeat): # <---- number of repetitions
15                 _run_test(alg, dna, dna_size)

```

Mean values for runtime in phase 2 is created with Pandas' `groupby`, aggregating by `mean()`:

```

1  summary = all_results.groupby(
2      [DF_ALGORITHM, DF_SEQ_SIZE, DF_SUBSEQ_SIZE]).mean()

```

This method avoids mistakes can could creep in if we attempt to calculate it by hand.

Similarly, for the memory usage in phase 2, `groupby` is also used to aggregate the values, this time using `max()`. It proved to be a better measurement of memory usage than `mean()`, given the method used to measure memory.

10.6.2 Predicted values

The predicted values, and other variables need to calculate it, are calculated in the code and added to the dataframes. All calculations are done with NumPy broadcasting, also to avoid mistakes that could creep in if we attempt to calculate it by hand.

This the code that calculates the predicted runtime. It calculates the theoretical complexity based on the algorithm, then the other values are derived from it.

```

1  if alg == ALG_BRUTE_FORCE:
2      # "/" is a trick to force Pandas/NumPy to calculate with maximum
3      # precision - without it, it overflows and sets the value to zero
4      df[DF_THEORETICAL_COMPLEXITY] = 2 ** (df[DF_SUBSEQ_SIZE] / 1)
5  else:
6      df[DF_THEORETICAL_COMPLEXITY] = df[DF_SEQ_SIZE] * df[DF_SUBSEQ_SIZE]
7
8  df[DF_RATIO] = df[DF_EMPIRICAL_RT] / df[DF_THEORETICAL_COMPLEXITY]
9  c = max(df[DF_RATIO])
10 df[DF_PREDICTED_RT] = c * df[DF_THEORETICAL_COMPLEXITY]

```

```

11 df[DF_ERROR] = (df[DF_EMPIRICAL_RT] - df[DF_PREDICTED_RT]) / \
12    df[DF_EMPIRICAL_RT] * 100

```

Similarly for space, the predicated value varies by algorithm. Once that value is determined, the other values are calculated from it.

```

1  INT32 = 4 # bytes in an int32
2
3  def predicted_space(x):
4      # Memory in bytes
5      alg = x[DF_ALGORITHM]
6      if alg == ALG_BRUTE_FORCE:
7          return x[DF_SUBSEQ_SIZE] # Uses one byte per character
8      elif alg == ALG_DYNAMIC_PROGRAMMING:
9          return x[DF_SEQ_SIZE] * x[DF_SUBSEQ_SIZE] * INT32
10     elif alg == ALG_HIRSCHBERG:
11         return x[DF_SUBSEQ_SIZE] * INT32
12
13     summary[DF_PREDICTED_SPACE] = summary.apply(
14         predicted_space, axis='columns')
15     summary[DF_PREDICTED_SPACE] /= 1024 * 1024 # Convert to MiB
16
17     summary[DF_ERROR] = \
18         (summary[DF_EMPIRICAL_SPACE] - summary[DF_PREDICTED_SPACE]) / \
19         summary[DF_EMPIRICAL_SPACE] * 100

```

10.7 Runtime and memory optimizations

The initial implementation of the algorithms took a significant time to complete or used so much memory that the Python interpreter quit.

Several techniques were used to speed up the code and reduce the amount of memory it uses.

10.7.1 Runtime

Python is an interpreted language, making it slower than compiled languages, such as C or C++. However, Numba offers just-in-time compilation for Python code. It does not support all Python modules and language constructs. Some code changes are needed to make it compilable by Numba.

With these changes in place, the speedup is significant. The table below shows an example of the Hirschberg's algorithm with and without Numba.

Table 6: Comparison of Numba-optimzied code with non-optimized code.

Input size	Numba time (ms)	Without Numba (ms)
m=4,000, n=500	623	1,129
m=5,000, n=1,000	1,529	2,920

The main change done to the code was to help Numba do type-inference by creating some data types ahead of time, with explicit type information. For example, in several branches of the code we need to return an empty array to signal the end of a sequence. A `return []` does not work with Numba because it cannot infer the type of the array elements. To get around that, empty arrays with explicit types are created, as shown below.

```

1 # Empty array of strings, to let numba infer the type
2 empty_string_list = [str('X') for _ in range(0)]
3 nx = x_end - x_start
4 if nx == 0:
5     # Empty input - got to the end of the sequence
6     return empty_string_list

```

In addition to that, regular Python lists and arrays were replaced with NumPy arrays wherever possible. Numba optimizes NumPy code better than regular Python, as shown in the table below.

Table 7: Comparison of Numba with and without NumPy.

Input size	Numba with NumPy (ms)	Numba with Python arrays (ms)
m=4,000, n=500	623	648
m=5,000, n=1,000	1,529	1,593

10.7.2 Memory

Memory optimization is relevant for the dynamic programming algorithm. The traditional representation of the algorithm, found in textbooks and other places, shows two $m \times n$ matrices, one for the lengths and another for the direction of moves. When translated literally to code, it uses significantly more memory than a space-optimized version.

A space-optimized version uses only a few bits for the moves. Since there are only three moves (up, down, diagonal), they can be encoded in as little as two bits. When encoded into bits, they can be stored into the higher bits of an integer, reserving the lower bits for the length. This collapses the two matrices into one.

This is the technique used in the algorithm tested in this report.

```

1 # To save memory and speed-up the code, the move is stored
2 # in the upper bits of a cell and the length in the lower bits
3 MOVE_START = 25 # leaves 24 bits for the length
4 MOVE_DIAGONAL = 1 << (MOVE_START)
5 MOVE_UP = 1 << (MOVE_START+1)
6 MOVE_LEFT = 1 << (MOVE_START+2)
7 EXCLUDE_LENGTH = (MOVE_DIAGONAL + MOVE_UP + MOVE_LEFT)
8 EXCLUDE_MOVE = ~EXCLUDE_LENGTH
9 ...
10
11 def _lcs_grid(xs, ys):
12     ...
13     for i in range(1, len(xs) + 1):
14         x = xs[i-1] # Remember that we use index 0 as a sentinel

```

```

15     for j in range(1, len(ys)+1):
16         y = ys[j-1]
17         if x == y:
18             # A match - move diagonally
19             length = grid[i-1][j-1] & EXCLUDE_MOVE
20             grid[i][j] = (length + 1) | MOVE_DIAGONAL
21         else:
22             left_length = grid[i][j-1] & EXCLUDE_MOVE
23             above_length = grid[i-1][j] & EXCLUDE_MOVE
24             if left_length < above_length:
25                 grid[i][j] = above_length | MOVE_UP
26             else:
27                 grid[i][j] = left_length | MOVE_LEFT

```

11 References

[BAF16] Beal, R., Afrin, T., Farheen, A. et al. 2016. *A new algorithm for “the LCS problem” with application in compressing genome resequencing data*. BMC Genomics 17, 544 (2016). <https://doi.org/10.1186/s12864-016-2793-0>, accessed 2020-03-16.

[CLRS01] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. 2001. *Introduction to Algorithms*, 2nd edition. MIT Press.

[CPY20] *The CPython source code*. <https://github.com/python/cpython/blob/v3.6.3/Include/unicodeobject.h#L202>, accessed 2020-03-18.

[FAG16] Fagerberg, R. 2016. *Dynamic Programming: Hirschberg’s Trick*. <https://imada.sdu.dk/~rolf/Edu/DM823/E16/Hirschberg.pdf>, accessed 2020-04-25.

[GOL20] Golubin, A. *How Python saves memory when storing strings*. <https://rushter.com/blog/python-strings-and-memory/>, accessed 2020-03-18.

[HIR75] Hirschberg, D. *A linear space algorithm for computing maximal common subsequences*. Commun. ACM 18 (1975): 341-343.

[KT05] Kleinberg, J. and Tardos, E. 2005. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA.

[NAV01] Navarro, G. 2001. *A guided tour to approximate string matching*. ACM Computing Surveys. <http://users.csc.calpoly.edu/~dekhtyar/570-Fall2011/papers/navarro-approximate.pdf>, accessed 2020-03-27.

[ROS09] van Rossum, G. 2009. *Tail recursion elimination*. <http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html>, accessed 2020-04-28.

[WIK20a] Wikipedia. *Subsequence* entry, *Applications* section. <https://en.wikipedia.org/wiki/Subsequence#Applications>, accessed 2020-03-18.

[WIK20b] Wikipedia. *Tail call*. https://en.wikipedia.org/wiki/Tail_call, accessed 2020-04-28.

References used in the code are annotated directly in the code.