Aurora Tiffany-Davis
attiffan
CSC 505
HW 3
Problem 2
10/29/17

2 a For implemented algorithms, please see the following files:

h3p2_recursive_attiffan.java
h3p2_dp_attiffan.java
h3p2_memoized_attiffan.java

In order to test these algorithms for run time, number of recursive calls, and number of scalar multiplications, a suite of test inputs was created. The goal in creating this suite of inputs was to be able to compare algorithms against one another, not only in one narrow case, but across a range of cases. Rather than explicitly designing inputs, and therefore being constrained by my own imagination, I left the inputs to chance. In this way I ensured a range of different inputs that I may not have thought of myself.

I created 54 different test inputs (sequences of dimension).

There were 3 different randomly generated sequences of dimension per set, where a set was characterized by the number of matrices in the sequence.

- Multiple sequences per sequence length were used to mitigate the risk of an unusually “easy” or “hard” sequence skewing test results

The number of matrices ranged from 3 to 20 (for 18 different sequence lengths).

- Multiple sequence lengths were used to mitigate the risk of being mislead by only “easy” (short) or “hard” (long) sequence lengths

Within a given sequence of dimension, each dimension was assigned randomly, between a value of 1 and a value of 20.

- Randomly assigned dimensions were used to get a range of circumstances for the algorithms to try out

2 a The randomly generated test inputs (sequences of dimension) are as follows.

20 Matrices
<10 4 4 15 13 9 3 5 8 9 12 3 4 14 8 15 8 12 19 10 15>
<20 2 11 6 12 13 4 12 14 1 4 11 13 15 6 15 11 15 7 13 10>
<12 11 11 9 1 4 8 12 3 5 7 15 8 3 20 3 10 11 17 6 17>

19 Matrices
<18 13 10 4 19 5 15 1 7 9 9 14 9 12 9 7 18 4 2 6>
<11 19 3 2 8 6 17 10 15 20 7 18 9 19 8 18 9 6 8 8>
<19 6 5 17 1 5 4 1 1 10 12 10 19 3 11 17 6 15 20 14>

18 Matrices
<13 17 17 18 14 13 20 5 6 4 4 17 11 2 12 6 14 3 12>
<15 15 18 1 14 3 15 20 2 11 20 9 10 7 3 12 5 14 6>
<2 18 3 6 12 18 19 13 16 19 9 13 14 7 8 18 18 10 17>

17 Matrices
<13 5 14 8 1 13 15 16 17 1 14 13 3 11 1 7 10 19>
<8 20 8 2 16 8 13 4 16 11 19 14 20 6 13 18 13 1>
<1 8 6 9 18 19 6 3 11 17 2 17 12 14 20 5 1 17>

16 Matrices
<6 11 14 12 18 10 1 2 3 9 16 5 3 15 9 20 15>
<7 16 4 5 13 8 16 6 6 9 7 19 19 19 17 11 6>
<8 10 10 13 5 19 1 12 14 19 8 19 19 7 14 7 10>

15 Matrices
<19 4 9 14 5 3 17 13 5 14 18 7 15 6 7 3>
<20 2 16 16 13 15 14 9 3 19 20 11 19 15 10 2>
<14 17 9 4 9 5 11 9 16 3 7 19 10 7 11 15>

14 Matrices
<9 3 19 7 14 5 19 9 13 10 8 9 8 19 19>
<17 1 19 15 3 7 8 9 14 20 18 8 18 17 3>
<17 6 12 6 20 19 17 11 18 11 14 11 6 18 15>

13 Matrices
<20 18 12 9 19 16 2 9 2 18 7 3 3 12>
<18 1 17 5 12 14 12 2 9 15 5 8 18 16>
<4 3 2 13 15 15 4 5 12 10 16 8 13 19>

12 Matrices
<17 9 4 18 12 15 6 2 12 4 12 13 15>
<15 4 4 4 12 20 14 7 5 15 13 19 13>
<5 19 5 9 11 17 5 16 12 11 2 9 7>

11 Matrices
<2 18 3 6 13 10 10 7 12 17 5 8>
<8 9 10 16 6 12 5 19 19 13 15 7>
<13 2 16 18 10 17 13 17 13 4 14 19>

10 Matrices
<18 4 6 6 8 7 4 17 8 7 1>
<11 16 7 9 7 9 10 20 13 12 14>
<11 16 19 2 17 3 9 5 1 7 12>

9 Matrices
<2 20 4 2 11 15 2 15 8 15>
<13 3 10 13 18 5 2 8 6 5>
<3 11 4 9 14 9 8 11 1 17>

8 Matrices
<2 8 1 19 18 1 9 5 12>
<2 18 19 18 7 9 3 10 2>
<10 16 15 18 12 2 6 12 7>

7 Matrices
<18 19 2 7 7 7 10 12>
<7 19 13 20 9 3 16 17>
<19 19 12 7 10 7 19 6>

6 Matrices
<3 8 16 15 10 1 12>
<18 5 4 2 3 2 15>
<20 15 15 6 17 15 18>

5 Matrices
<2 17 18 7 16 18>
<16 8 18 11 12 20>
<12 15 4 18 12 16>

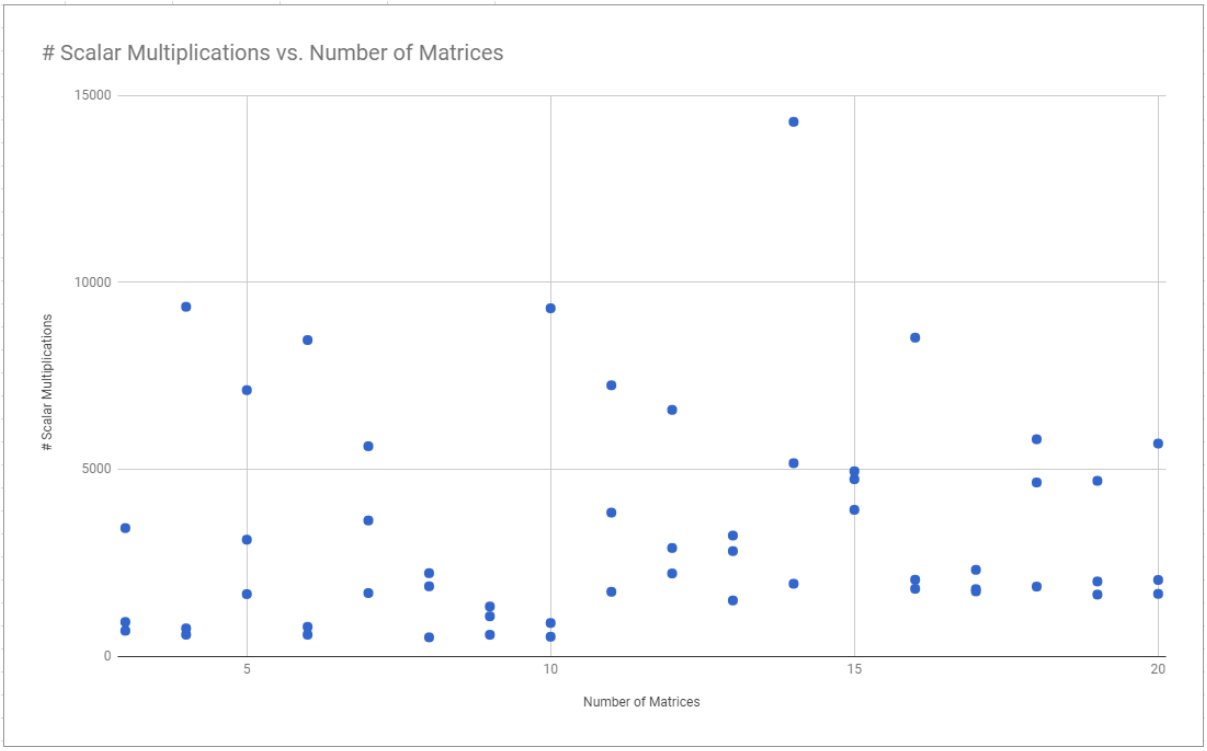
4 Matrices
<17 18 17 14 11>
<2 13 13 4 17>
<6 9 17 3 7>

3 Matrices
<2 16 18 3>
<13 17 5 3>
<20 7 15 14>

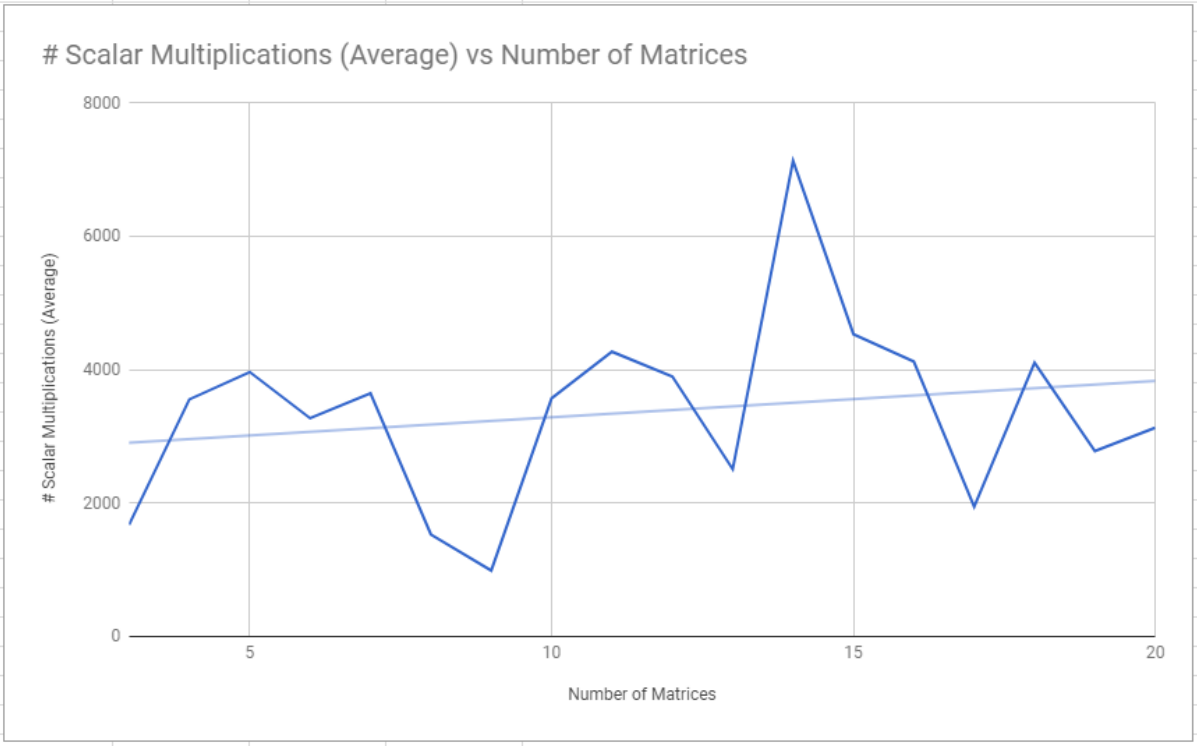
I tested the 54 sequences of dimensions listed above on each of 3 algorithms, and then repeated this test 100 times in order to get average run times per scenario. With 54*3*100 lines of data (16200), the tabulation of measurements is too large to include in this report.

First, to ensure that the algorithms were working properly, I ran 1 trial (54*3 = 162 rows, still too long to include in this report). I checked each sequence of dimensions and ensured that for each one, all 3 algorithms reported the same parenthesization and the same # scalar multiplications.

2 a Here, I show plots drawn from the measurements taken over 1 trial.

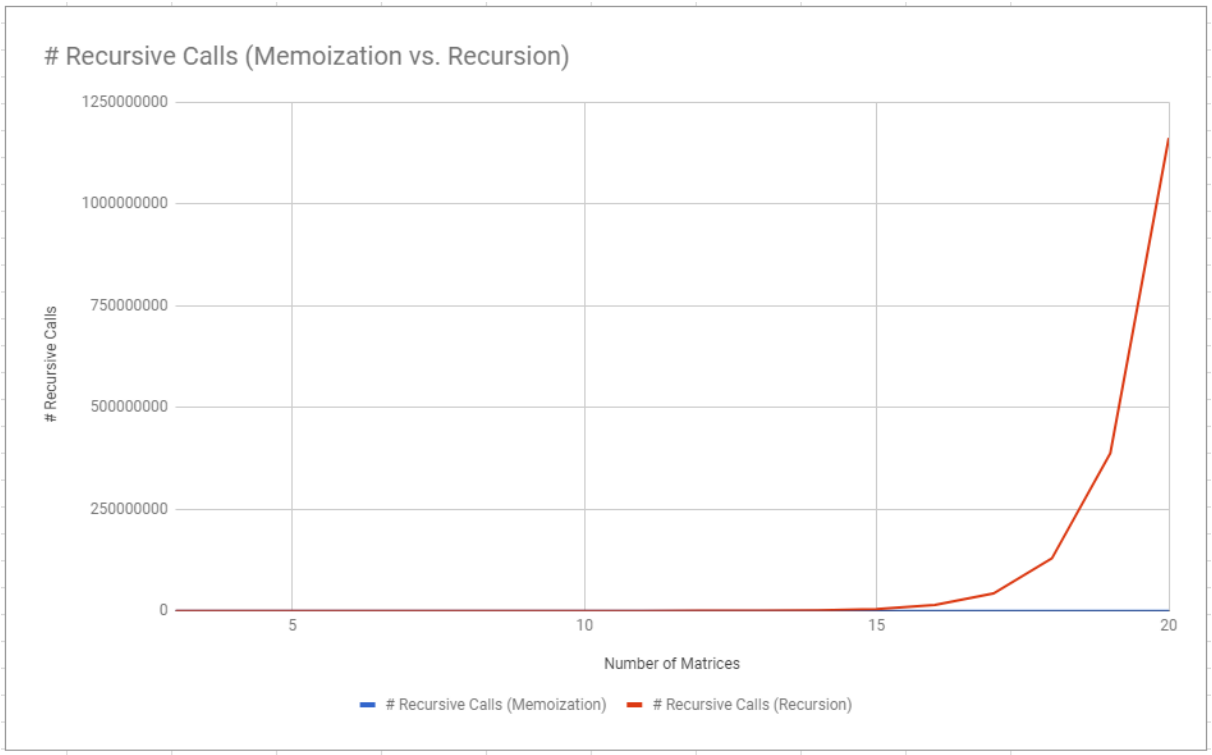


For each value on the X-axis, there are 3 points on the Y axis, showing the 3 test sequences per sequence length. The randomization of test inputs makes this data a bit messy.

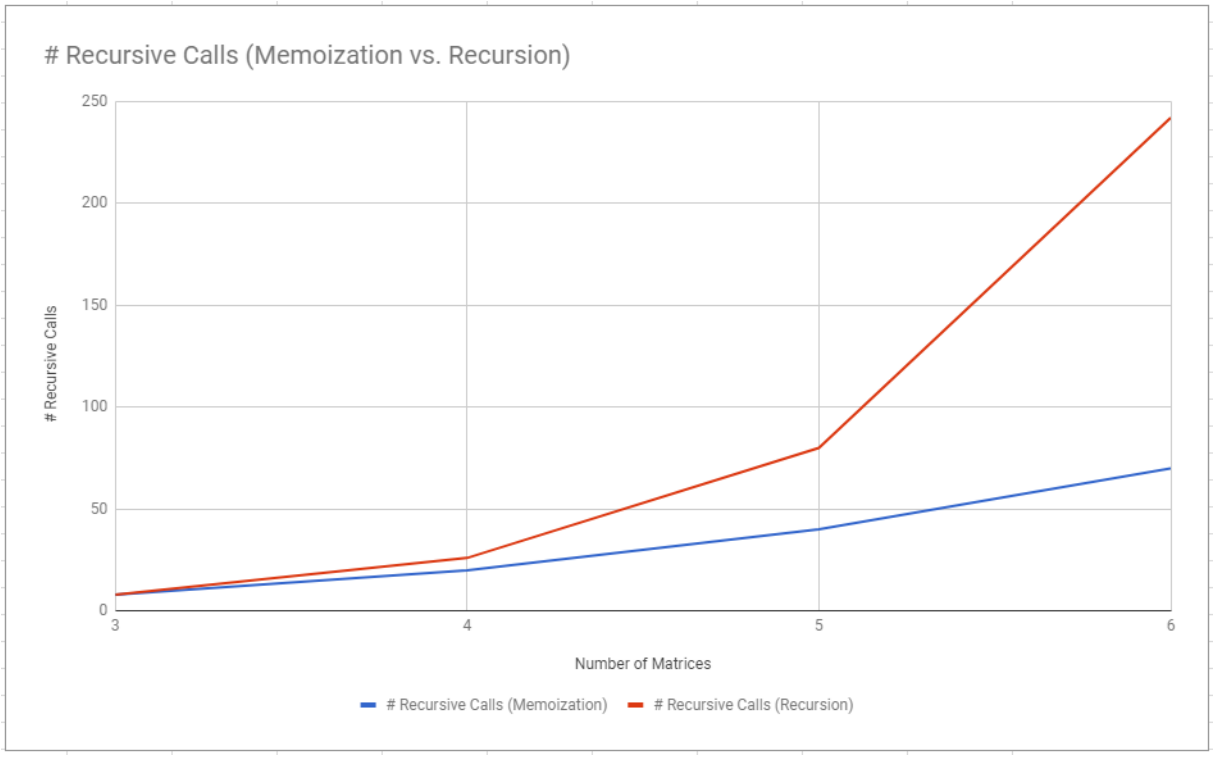


Here, the 3 sequences' results are averaged for each sequence length, and a trendline is superimposed in a lighter color. The overall trend is as expected (more matrices, in general, lead to a higher # scalar multiplications).

2 a



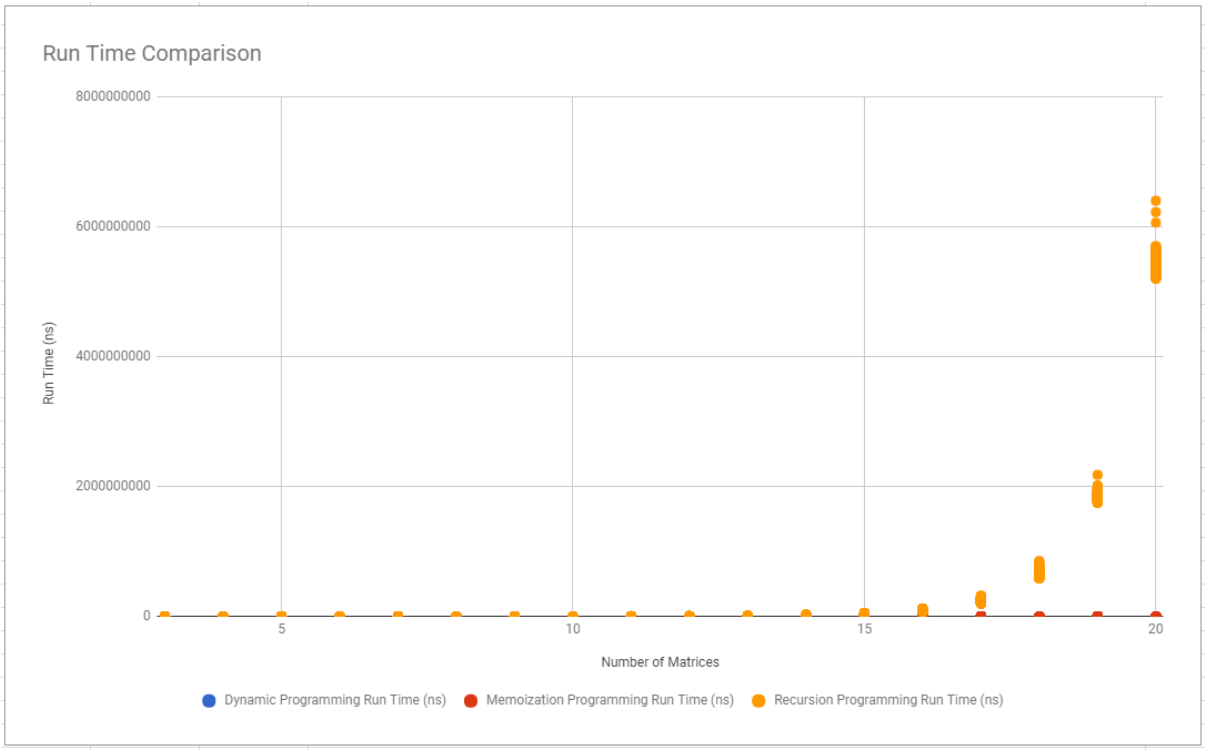
In this chart, the dynamic programming algorithm is excluded, as it never has any recursive calls. The actual sequence of dimensions does not impact the # recursive calls, which is based solely on the sequence length - this is why only one measurement is shown for each sequence length. We can see here that the # recursive calls made by the memoization algorithm vanishes to insignificance when compared to the calls made by the recursion algorithm, for the longer sequence lengths.



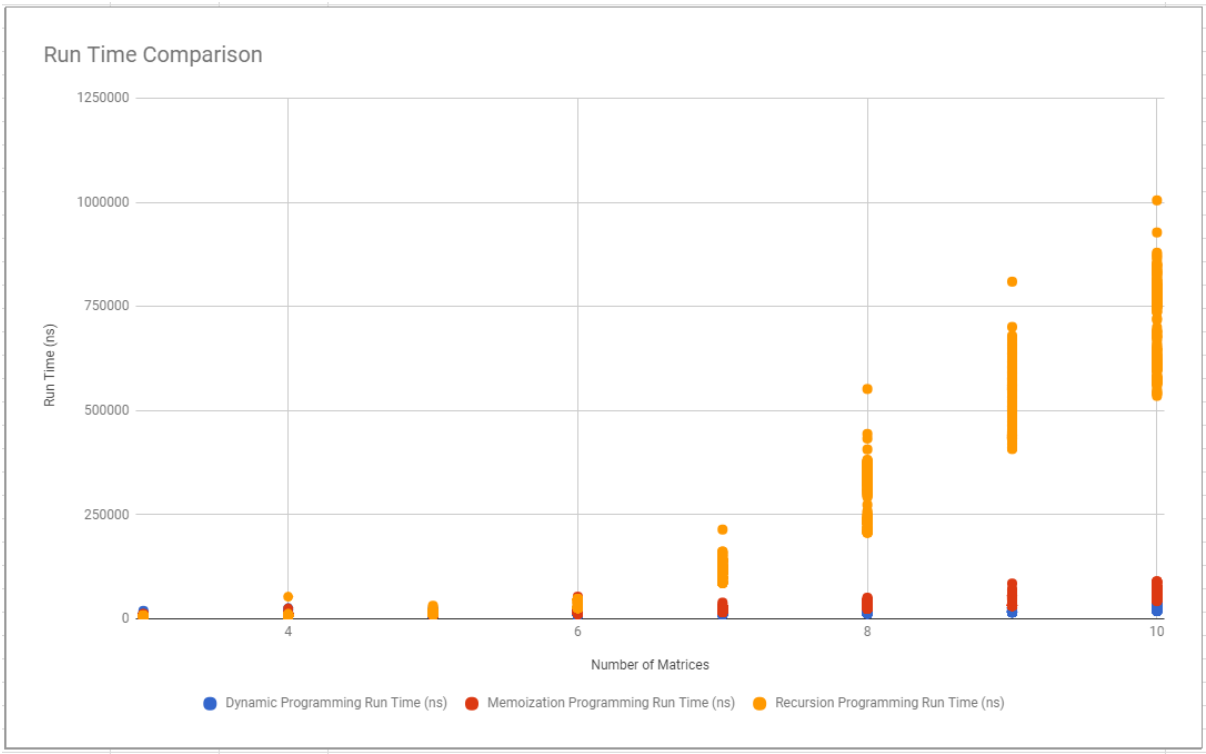
Here, we “zoom in” to show these algorithms starting to widely diverge at smaller sequence lengths.

2 a Next, after examining the single trial, I ran 100 trials to get run times. This test was performed on one PC, with no other applications running during the test.

System Model: HP Notebook
System Type: x64-based PC
Processor: Intel64 Family 6 Model 78 Stepping 3 GenuineIntel ~2600 Mhz

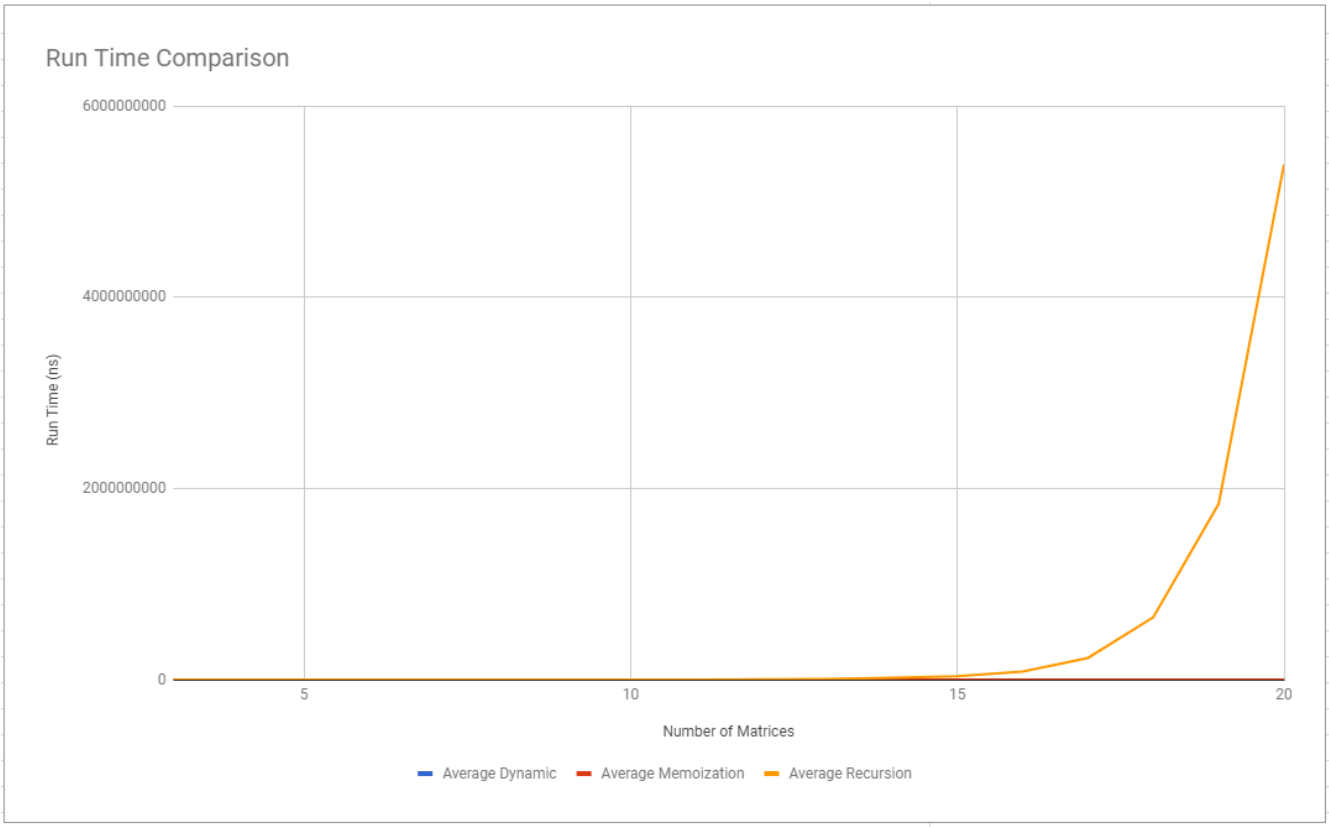


The scatter plot shows that there is variation in run time within the recursion algorithm, but that it runs for longer time than the other two. Memoization appears to run longer than Dynamic Programming.

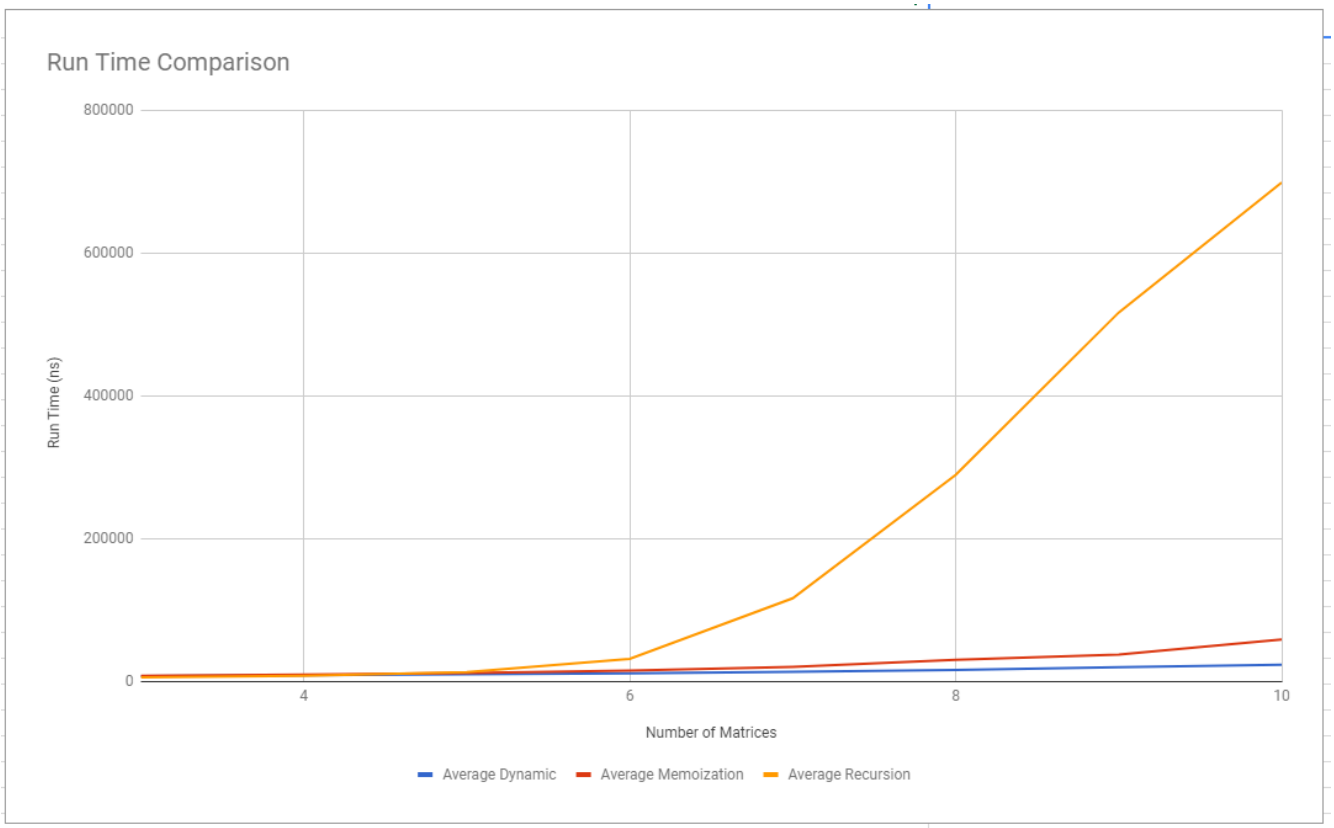


Here, we “zoom in” to show these algorithms starting to widely diverge at smaller sequence lengths.

2 a



The overall average run times for each sequence length bear out the trends discussed above in the scatter plots. For a given sequence length and algorithm, the average was over 3 sequences tested 100 times each.



Again, we “zoom in” to show these algorithms starting to widely diverge at smaller sequence lengths.

2 a The overall run time averages for each algorithm are shown below. These are averaged over the 54 input sequences of dimensions, and the 100 trials run on each. Based on these test results, Dynamic Programming appears to be the most time-efficient approach.

Algorithm	Average Run Time (ns)	Average Run Time (ms)
h3p2_dp_attiffan.java	45344	0.04
h3p2_memoized_attiffan.java	127348	0.12
h3p2_recursive_attiffan.java	458516277	458.51

Running the programs: Each underlined section below lists the compile command, then the run command (to run the sequence of dimensions from Q 2.b), and finally the output (for that sequence of dimensions).

h3p2_dp_attiffan.java

```
javac h3p2_dp_attiffan.java
java h3p2_dp_attiffan 5 2 4 7 7 3 9 7 8 8 6 3 7 5 5
Dynamic Programming     14      5, 2, 4, 7, 7, 3, 9, 7, 8, 8, 6, 3, 7, 5, 5
(A1(((((((((((A2A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14))     59920   0       960
```

h3p2_memoized_attiffan.java

```
javac h3p2_memoized_attiffan.java
java h3p2_memoized_attiffan 5 2 4 7 7 3 9 7 8 8 6 3 7 5 5
Memoization     14      5, 2, 4, 7, 7, 3, 9, 7, 8, 8, 6, 3, 7, 5, 5
(A1(((((((((((A2A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14))     564730  910     960
```

h3p2_recursive_attiffan.java

```
javac h3p2_recursive_attiffan.java
java h3p2_recursive_attiffan 5 2 4 7 7 3 9 7 8 8 6 3 7 5 5
Recursion       14      5, 2, 4, 7, 7, 3, 9, 7, 8, 8, 6, 3, 7, 5, 5
(A1(((((((((((A2A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14))     32741223        1594322 960
```

The output for each program has 7 parts, each separated by a tab character.

- Algorithm type
- Number of matrices
- Comma-separated sequence of dimension
- Parenthesization
- Run time expressed in nanoseconds
- Number of recursive calls
- Number of scalar multiplications

These outputs were produced, in this standardized way, to facilitate testing and analysis (a batch file was used to append output of all tests to one text file, which could then be manipulated for analysis).

2 b Parenthesization of a matrix-chain product whose sequence of dimension is <5, 2, 4, 7, 7, 3, 9, 7, 8, 8, 6, 3, 7, 5, 5>

Algorithm	Parenthesization	# Scalar Mult's
h3p2_recursive_attiffan.java	(A1((((((((((((A2A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14))	960
h3p2_dp_attiffan.java	(A1((((((((((((A2A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14))	960
h3p2_memoized_attiffan.java	(A1((((((((((((A2A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14))	960

Multiplications required for a parenthesization that multiplies the input matrices in their input order (significantly more multiplications needed than in our algorithms above):

Matrix Name	Rows	Columns	Matrix to Multiply (1)	Matrix to Multiply (2)	# Scalar Mults	Resultant Rows	Resultant Columns
A	5	2					
B	2	4	A	B	40	5	4
C	4	7	AB	C	140	5	7
D	7	7	ABC	D	245	5	7
E	7	3	ABCD	E	105	5	3
F	3	9	ABCDE	F	135	5	9
G	9	7	ACBDEF	G	315	5	7
H	7	8	ABCDEFG	H	280	5	8
I	8	8	ABCDEFGH	I	320	5	8
J	8	6	ABCDEFGHI	J	240	5	6
K	6	3	ABCDEFGHIJ	K	90	5	3
L	3	7	ABCDEFGHIJK	L	105	5	7
M	7	5	ABCDEFGHIJKL	M	175	5	5
N	5	5	ABCDEFGHIJKLM	N	125	5	5
				Total ----->	**2315**		