Assignment 1: Algorithm Implementation and Empirical Analysis

Data Generation and Experiment Setup

Method

For the experiment, we decided to generate six total data sets for testing the different implementations of the spreadsheets. All with varying sizes and densities (number of cells filled). We used one data set that was small in size but had medium density, three medium sized data sets that had low, medium and high density respectively and two large data sets that had medium and high density respectively. This was chosen such that all values of size could be tested against one value of density (medium), and all values of density could be tested against one value of size (also medium). This way we can see the effect each variable has. We also tested for a large data size with high density to stress test the respective data structures. The following table demonstrates visually the combinations we decided upon.

Size-Density combinations		size		
		small	medium	large
density	low		X	
	medium	X	X	X
	high		X	X

Small size entailed a 20 x 20 data set, medium was 60 x 60 and large was 250 x 250. Density sizes were 25%, 65% and 95%. Meaning that 25, 65, or 95% of the spreadsheet respectively were filled by data, while the remaining cells were empty. The sizes were determined from testing to be plausible to run within the time constraints of our hardware setup. Both sizes and density values were also decided upon to most meaningfully demonstrate differences between the characteristics of the respective data structures. Fully filled spreadsheets, or 100% density, were not included so that we could still test spreadsheets that had “None” values.

All datasets were generated using the random module and either randInt for columns and rows and random.uniform for float values. All functions that were tested were averaged out over 1000 trials. Excluded from this was the time to build the spreadsheet, rowNum and colNum.

Both rowNum and colNum were excluded from testing as they were implemented the same across all implementations and had a time complexity of O(1). They would therefore not produce any meaningful information. Time to build was timed once and only once, this is because testing the time to build would require generating a new dataset for each trial, which was deemed too computationally expensive.

In order to conduct testing for the linked lists at larger sizes, the system recursion limit needed to be increased. For our timing mechanism we opted for time.perf_counter_ns rather than something like time.time(). This is because perf_counter gives the real time it takes to run the code whereas time.time() is the time spent by the computer to run the code. The computer does not always concentrate 100% of its resources on one process so other processes running could affect results. Therefore perf_counter gives a more accurate result.

Running times

Theoretical running times

Assume a spreadsheet with an equal number of columns and rows, where n = the number of columns/rows

Array based spreadsheet

buildSpreadsheet: Iterates through rows * columns times, creating a 2D array. Rows * columns = n * n = n^2. O(n^2).
appendRow: Makes 1 call to append(). Appending to a list requires creating a new list and copying everything over, this means an operation for each item in the list, i.e. n operations. Therefore O(n).
appendCol: makes a call to append() for each iteration of the loop, which iterates row times. Append requires O(n) operations and is called n times, therefore O(n^2).
insertRow: makes 1 call to insert(). Inserting to a python list requires creating a new list and copying everything over with the additional element added to the new array, meaning O(n) operations.
insertCol: makes a call to insert() for each iteration of the loop, which iterates row times. Therefore O(n) * O(n) operations = O(n^2).
Update: value at row index and col index can be accessed directly and modified at O(1).
Find: Must enumerate all values in spreadsheet to determine which values match. Therefore O(n^2).

Linked List based spreadsheet

buildSpreadsheet: outer loop operates over columns, inner loop operates over rows and calls the append function. Append() function is O(1) and is called columns * rows times, i.e. n * n = O(n^2)
appendRow: iterates through all columns and adds an item to the end (building the row). O(n).
appendCol: constructs a new column and appends it to the end of outer (column) linked list. Construction of the new column involves iterating rows times. O(n).
insertRow: Iterates through each column and inserts an item at the row location. So O(n) calls to insert(). Insert is called on a linkedlist object, in this case the column. It iterates over each node until it finds the node of correct index, then performs O(1) operations to insert it. In worst case the index to be inserted is at the end, so requires iterating through the entire
linkedlist to find it, i.e. O(n). Therefore, there are O(n) calls to insert, which runs at O(n), so the big-O time complexity of insertRow is O(n^2).

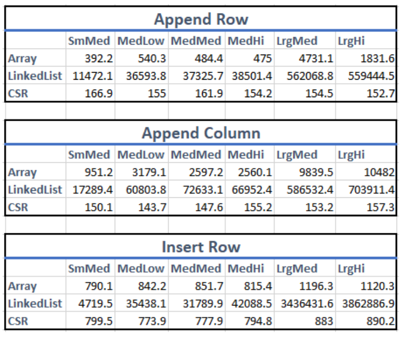
insertCol: First constructs the new column, calling append() rows times (O(n)). Then attempts to insert this column at the end of the spreadsheet, insert() requires O(n) operations, O(n) + O(n) = O(n).
Update: in order to find the position to be updated, must iterate through columns and then rows, worst case row and column are at the end. O(n) + O(n) = O(n).
find: must enumerate all values of the spreadsheet in order to determine which values match. This requires iterating through all rows for each column. O(n^2).

CSR based spreadsheet

appendRow: occurs a constant amount of time as it requires no loops and just adds to the end of sumA. O(1)
appendCol: occurs a constant amount of times as it is incrementing total columns by one. O(1)
insertRow: occurs a constant amount of times as it will directly insert the same sum value at the index into sumA. O(1)
insertCol: iterates through all columns in colA and if the column has an index that is greater than or equal to the target column index, then it will increment it. O(n)
Update: to find the desired update position, it must iterate through all columns within the indices range and find the matching existing cell. If the cell does not exist then it inserts at the end of the row and then increments the sum of all rows that are within the range of the rowIndex + 1 till the length of sumA. O(n).
find: to find all cells with the target value, it must enumerate through all the values in the spreadsheet. This means enumerating through all column indices ranges and then the subsequent values in valA. O(n^2)

Empirical running times

Raw Data



Append Row

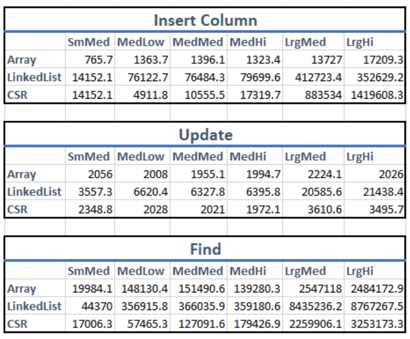
	SmMed	MedLow	MedMed	MedHi	LrgMed	LrgHi
Array	392.2	540.3	484.4	475	4731.1	1831.6
LinkedList	11472.1	36593.8	37325.7	38501.4	562068.8	559444.5
CSR	166.9	155	161.9	154.2	154.5	152.7

Append Column

	SmMed	MedLow	MedMed	MedHi	LrgMed	LrgHi
Array	951.2	3179.1	2597.2	2560.1	9839.5	10482
LinkedList	17289.4	60803.8	72633.1	66952.4	586532.4	703911.4
CSR	150.1	143.7	147.6	155.2	153.2	157.3

Insert Row

	SmMed	MedLow	MedMed	MedHi	LrgMed	LrgHi
Array	790.1	842.2	851.7	815.4	1196.3	1120.3
LinkedList	4719.5	35438.1	31789.9	42088.5	3436431.6	3862886.9
CSR	799.5	773.9	777.9	794.8	883	890.2



Insert Column

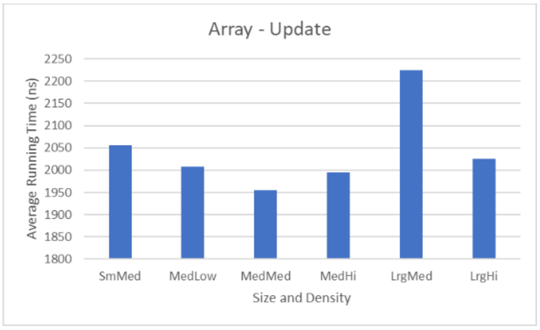
	SmMed	MedLow	MedMed	MedHi	LrgMed	LrgHi
Array	765.7	1363.7	1396.1	1323.4	13727	17209.3
LinkedList	14152.1	76122.7	76484.3	79699.6	412723.4	352629.2
CSR	14152.1	4911.8	10555.5	17319.7	883534	1419608.3

Update

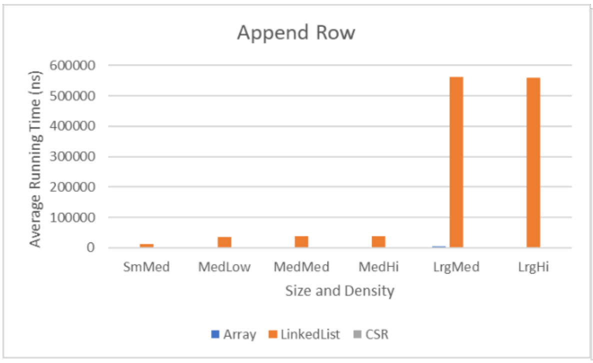
	SmMed	MedLow	MedMed	MedHi	LrgMed	LrgHi
Array	2056	2008	1955.1	1994.7	2224.1	2026
LinkedList	3557.3	6620.4	6327.8	6395.8	20585.6	21438.4
CSR	2348.8	2028	2021	1972.1	3610.6	3495.7

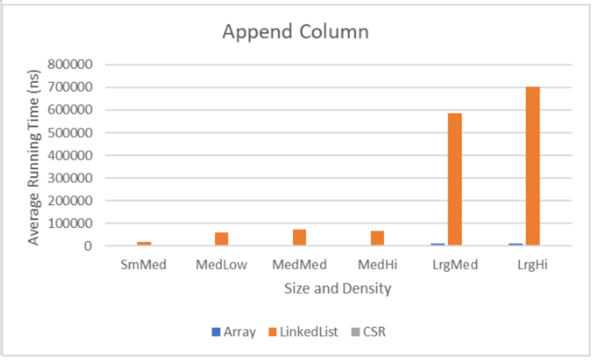
Find

	SmMed	MedLow	MedMed	MedHi	LrgMed	LrgHi
Array	19984.1	148130.4	151490.6	139280.3	2547118	2484172.9
LinkedList	44370	356915.8	366035.9	359180.6	8435236.2	8767267.5
CSR	17006.3	57465.3	127091.6	179426.9	2259906.1	3253173.3



Array-based performs at a very similar level on Update regardless of size or density, which corroborates the theoretical estimate that the running time is O(1), so should vary little with input size.

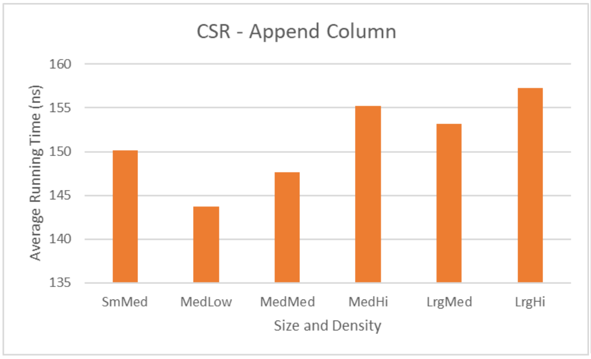


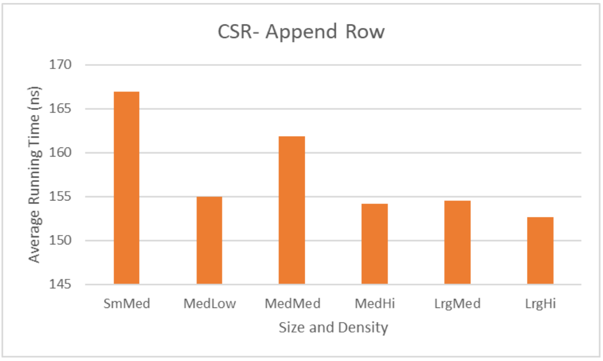


LinkedList is excessively slow at appending new rows and columns, it is similarly slow at inserting rows and columns, but CSR is also poor (even worse performing in fact) when it comes to inserting columns. The theoretical running time for appending a column for linkedList is O(n), whereas it is O(n^2) for the array. Despite this, the array-based implementation is still significantly faster. This is likely because the append() function of a python list may be O(n), but has a much faster constant factor.

CSR-based spreadsheet		size		
		small	medium	large
density	low		9568	
	medium	3327	20,365	450,020
	high		28,775	668,425

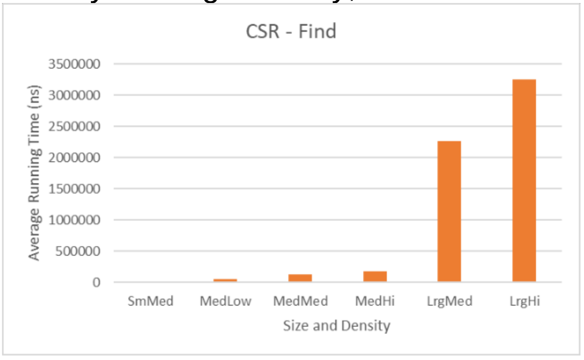
The above table contains the average running time data across all tasks for the CSR implementation. It is clear from this data that the running time increases not just with size, but also with density, which is to be expected from the CSR data structure as it is intended for circumstances with low density where it can maximally save space and time.



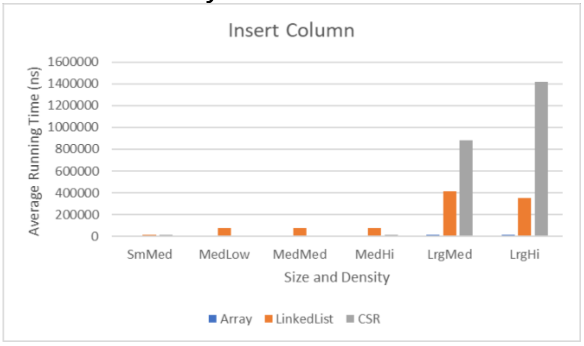


Another example of an O(1) theoretical prediction being supported by empirical evidence, the variation by size when CSR performs the Append Column and Append Row tasks is negligible.

This graph demonstrates the factor that was not included in the asymptotic time analysis for the CSR based spreadsheet, which is that CSR is built to save space and time for certain operations by not storing empty cells. While in this case the size of the dataset clearly impacts the running time, there is also a significant difference noticeable between medium density and high density, where medium density is faster as it must search less.



CSR appears to take longer to insert a column compared to the other two implementations. Compared to the other CSR functions that have a O(1) runtime complexity, the insert column has a complexity of O(n). This would explain the increase of running time here compared to other methods. One possibility that may explain the very large difference between CSR and the LinkedList implementation here could be that the constant factors not taken into account by asymptotic analysis are more costly for CSR here than the LinkedList or Array implementation.



Conclusion

On a general basis an array-based spreadsheet implementation was found to be a more efficient solution in relation to most tasks. For a use case in which a spreadsheet is going to often be updated the array-based solution is most suitable as it is the highest performing at the Update task due to its ability to directly access any element in the spreadsheet at O(1). With the exception of InsertColumn CSR is much faster at adding new information to the spreadsheet (AppendColumn, AppendRow, InsertRow) making it a better choice for a spreadsheet that will often be modified. However, CSR still performs worse on the Update task, meaning that there is a tradeoff between a use case that prioritises modification vs. accessing the data. The LinkedList implementation is outperformed on almost every metric, and the data collected does not indicate it is a good choice for essentially any circumstance.