Lab Week 3 Reflection

1. Looking at the data, the execution time scales with the number of rows in a roughly linear relationship. This makes sense, because I don't really have any nested loops in my functions, so the highest complexity I would have would be O(n) or O(n log n).

2. Out of all my functions, the one with the highest coefficient of complexity is R, that is, the function that retrieves the latest sale. This is probably because it not only has to look at every single element in the ArrayList allSales, but it has to make anywhere from 1 to 6 comparisons between each Sale and the Sale currently stored as the latest. Also, the way I generated sale IDs meant that it was trivial to both check for duplicate IDs and find a sale by ID, since IDs were basically just the row numbers to the CSV. If I was using a more complex system like UUIDs or some other kind of tracking number, those functions would likely balloon in runtime, resulting in O(n^2) complexity. I'm lucky that I chose the right data structure to avoid this.

3. I tested my functions only on data generated by generateCSV(), so there are a couple blind spots that were outside the scope of my lab. If I were to make this code more rigorous I would change these things:
   - Currently, column headers/positions are hard-coded in. if the structure were to change, the construction of Sale instances would have to be updated as well.
   - It's assumed that all the cells have valid types (whole integers for saleId, valid dates for saleDate, non-Strings for amount, etc). Real data might have mistakes or inaccuracies that would break this.
   - Like I mentioned before, a real company might want to use UUIDs or tracking numbers rather than ints for their sale IDs. For example, if a user's order is visible at www.example.com/my-orders/**12345**, changing the link to www.example.com/my-orders/**1234<u>6</u>** might result in seeing some else's private order.

rows	G(enerate CSV)	L(oad CSV)	R(etrieve Latest	T(otal Revenue)	D(uplicate IDs)	F(ind ID #)
100	5.43683	7.98279	0.19554	0.14617	1.09028	0.23275
100	11.53475	6.11433	0.21850	0.06046	0.85392	0.23183
100	11.84371	7.91004	0.19704	0.08388	1.26508	0.20533
100	12.52238	8.60513	0.22600	0.09854	0.63608	0.25763
1000	18.47292	24.37979	0.98825	0.30575	2.67158	0.22754
1000	17.30454	25.16150	0.88013	0.29621	3.80146	0.14850
1000	19.18425	22.62071	0.74433	0.26475	2.94700	0.21150
10000	41.11333	49.05138	3.79446	2.70396	12.53117	0.16077
10000	42.06275	43.42442	3.67713	3.58125	12.56983	0.14813
10000	41.66454	47.42517	4.33638	2.91221	12.37342	0.16867
100000	106.67600	125.09429	11.45596	8.38929	48.01071	0.11125
100000	94.07142	132.06504	12.79479	9.01358	47.83400	0.12658
1000000	491.73483	602.03200	60.65183	20.13104	79.43188	0.21554
1000000	551.94788	542.84904	30.78475	18.69738	80.45692	0.34896
10000000	4128.67667	4686.09896	312.00121	47.39525	886.45142	0.27404

According to ChatGPT, you can find the power k of a complexity O(n^k) by calculating the slope of a log-log plot. Since I can't just eyeball a slope in a Google Sheets plot, below I took the log_10 of every data point, then in bold i used the =SLOPE() function to calcualte what the slope of the lines would be numerically. Because of how I structured my test data construction, D and F have very good complexities of O(n) and O(1). The worst function that I have is R, which is somewhere between O(n) and O(n log n). **This is the only place I used AI for this lab. AI was not used to write any code. AI was not used to write this textbox. AI was only used to learn how to read/calculate Big O complexities from a log-log plot.**

2	*0.73535*	*0.90215*	*-0.70876*	*-0.83515*	*0.03754*	*-0.63311*
2	*1.06201*	*0.78635*	*-0.66055*	*-1.21854*	*-0.06858*	*-0.63482*
2	*1.07349*	*0.89818*	*-0.70544*	*-1.07637*	*0.10212*	*-0.68754*
2	*1.09769*	*0.93476*	*-0.64589*	*-1.00638*	*-0.19649*	*-0.58901*
3	*1.26654*	*1.38703*	*-0.00513*	*-0.51463*	*0.42677*	*-0.64294*
3	*1.23816*	*1.40074*	*-0.05546*	*-0.52840*	*0.57995*	*-0.82827*
3	*1.28294*	*1.35451*	*-0.12823*	*-0.57716*	*0.46938*	*-0.67469*
4	*1.61398*	*1.69065*	*0.57915*	*0.43200*	*1.09799*	*-0.79378*
4	*1.62390*	*1.63773*	*0.56551*	*0.55403*	*1.09933*	*-0.82937*
4	*1.61977*	*1.67601*	*0.63713*	*0.46422*	*1.09249*	*-0.77297*
5	*2.02807*	*2.09724*	*1.05903*	*0.92373*	*1.68134*	*-0.95370*
5	*1.97346*	*2.12079*	*1.10703*	*0.95490*	*1.67974*	*-0.89762*
6	*2.69173*	*2.77962*	*1.78284*	*1.30387*	*1.89999*	*-0.66647*
6	*2.74190*	*2.73468*	*1.48834*	*1.27178*	*1.90556*	*-0.45723*
7	*3.61581*	*3.67081*	*2.49416*	*1.67573*	*2.94765*	*-0.56218*
log slope:	**0.46149**	**1.03452**	**1.19043**	**0.96863**	**0.89185**	**-0.00783**