**Collections Demo**Max Crawford

* **Experiment Design**
* *Using Python*
* collection\_utils.py: handles creation/population of collections, tracking of benchmark results
  + make\_collections() initiates the generation and population of collections; collections are filled with *length* random integers, up to max\_length and at interval determined by scale
  + collectionTest super class with arrayTest, listTest, and dequeTest sub-classes contain each generated collection as well as its type, length, and benchmark results (starting with population runtime)
* benchmarks.py: contains various benchmarks for evaluating collection performance
  + sort\_items() tests the time to sort a collection; uses sorted() for sake of flexibility (deques have no in-place sorting method)
  + filter\_items() tests the time to filter a collection by a simple condition; uses filter() with lambda expression, in this case searching for multiples of 10
  + insert\_item() tests the time to insert a random integer into a collection at various indices: start, middle, and end
* collections\_demo.py: main file, calls all other functions
  + Repeated for *n* trials: Calls make\_collections() to create/populate collections, calls benchmark functions on each collection, compiles results into a Pandas dataframe
  + Averages results from all compiled dataframes, returning a final avg\_results df; also creates a ‘log’ version of this df for sake of graphing
  + Writes both df’s to separate sheets of an Excel spreadsheet
  + Calls graph\_results on both df’s
* graph\_results.py: handles graphing of results
  + Creates a Seaborn FacetGrid with a lineplot for each benchmark
  + Each lineplot has log(length) on x-axis, runtime in ms on y-axis, and different-color lines for each collection type
  + Saves the final graph as an image
* **Results**
* For the final evaluation, I ran the program with n\_trials=5, max\_length=10^8, and scale=2. This took approximately 51 minutes to complete. See graphs at bottom of doc (note that the graphs do *not* share a y-axis).
* NOTE: In a few of the benchmarks, there is a notable peak for 10^2.107 (128) integers. After further testing, it appears this is an anomaly. It is likely the program briefly slowed down during one of the trials, resulting in outliers that were then included in the average. An improved version of this test might try to ‘punish’ such outliers before averaging. And in general, it may be useful to examine individual trials more closely, such as to analyze the typical spread of runtimes.
* Population: All collections took a similar amount of time to populate, although arrays were slightly slower; at 10^8 integers, it was a difference of about 2 and a half seconds. Lists and deques were nearly identical in performance.
* Sort: Again, the results appear similar across the board. Arrays were again slightly slower, by about 1.5-2 seconds at 10^8 integers. Lists appear to be faster than deques by a very slightly margin, about 0.8 seconds at 10^8.
* Filter: It is hard to draw any meaningful results from this benchmark, as comparative performance was fairly inconsistent. At 10^8, lists appear to be slowest, but this is not the case just one interval before (10^7.827), and is likely the result of an outlier data point. Notably, the runtimes here were all less than a tenth of a millisecond, so it’s safe to assume any collection could perform adequately for simple filtering.
* Insert:
* Start: Deques performed extremely fast (nearly instantaneous); arrays were slightly slower, followed by lists (time also grew significantly faster than deques).
* Middle: Arrays performed best, followed by lists and deques (which performed almost identically until the last couple of values, where lists overtook the latter).
* End: All runtimes were less than a hundredth of a second, so differences are neglible; but, arrays ran fastest, then lists, then deques.
* NOTE: For simplicity, I used the insert() method for all of these benchmarks; it is possible that performance varies with other approaches, especially for deques which have dedicated appendleft() and appendright() methods.
* For all insert benchmarks, there appears to be significant variation in the graphs. This is likely because a different number is generated each time insert\_items() is called (so, for every collection). It would probably be better to use the same number for all inserts, or at least for all collections of the same length.
* *Average Runtime (ms)*
* A graph of a function

  Description automatically generated with medium confidence
* *Log(Runtime)*
* A graph of different colored lines

  Description automatically generated