Determining the Activity Series with the Fewest Experiments using Sorting Algorithms

Joshua Schrier, Michael F. Tynes, Lillian Cain*

Department of Chemistry, Fordham University, 441 East Fordham Road, The Bronx, New York, 10458, USA

* jschrier@fordham.edu

Abstract

Determining the activity series of a collection of elements is a classic pedagogical experiment, in which pairs of elements are reacted to determine the relative rank ordering of their reactivity. Determining the optimal sequence of pairwise experiments that minimizes the total number of experiments corresponds to well-known comparison sorting algorithms in computer science. We describe relevant algorithms (insertion sort, binary insertion sort, merge sort, and merge insertion sort), their application to the activity series problem, and discuss ways that this connection can contribute to the introductory chemistry and computer science curricula. In addition to pedagogical interest, this illustrates a simple form of artificial intelligence for chemical experiment planning.

KEYWORDS: First-Year Undergraduate/General, Continuing Education, Interdisciplinary / Multidisciplinary, Problem Solving / Decision Making, Qualitative Analysis

Introduction

The activity series—a list of elements arranged in decreasing order of reactivity— is a standard topic of high school and undergraduate chemistry courses.[1] A classic pedagogical laboratory exercise has students determine the activity series by reacting pairs of species, observing the qualitative outcome of each of experiment to determine which is more reactive, and using the results to construct a ranked list of reactivity. These lab experiences typically include between four species (as a wet lab activity) and nine species (as an online activity where clicking on a hyperlink "performs" a reaction and shows a photograph of the reaction outcome[2]). Activity table charts in textbooks list about 30 species. How many experiments are required to determine the activity series of *n* items?

The problem takes a simpler and more universal form if we consider it as the task of ranking *n* items using pairwise comparisons of a *transitive* characteristic (i.e., a characteristic such that if $A < B$ and $B < C$ then $A < C$). Mathematicians know this as a *tournament problem*, as it can be posed in terms of the design of an athletic tournament, in which *n* players compete against one another in pairwise matches to establish a ranking. An early version of this problem was discussed by the British mathematician Charles Dodgson (better known by his pen name Lewis Carroll), and is discussed in Steinhaus's recreational mathematics book.[3] Computer scientists recognize this as *sorting*—arranging a list of items into an ordered arrangement—and more specifically, as an example of the subclass of *comparison sort* problems in which a single abstract comparison operation determines if a pair of items are in the proper order.[4,5] In an activity series experiment, the qualitative observation of which species is more reactive plays the

same role as determining the winner of an individual athletic match or which of two items in a pair is greater than the other. In all of these examples, the desired outcome is a list in descending order.

Recognizing reactivity series determination as a sorting task allows us to apply known algorithms that solve this problem in an optimal way that minimizes the number of experiments required. [JS1]However, we are not aware of previous pedagogical or research literature discussing the isomorphism between the optimal experiment design and sorting algorithms. Chemical phenomena such as electrophoresis experiments and chromatography can be considered as physically sorting molecules by size or interactions,[6] but no algorithm is applied in these experimental processes. (The sole exception is the demonstration of an ensemble sorting algorithm for purifying mixtures of different droplet types in microfluidic devices.[7]) A growing body of work on artificial intelligence approaches for autonomous ("self-driving") chemical experimentation combines machine-learned surrogates between experimental inputs and outputs and then uses an optimization algorithm to select new experiments with the highest reward.[8] In general, the space of parameters or molecules to be explored is vast, so not all possibilities can be tested, and the goal is to find examples that maximize a given property. In contrast, the reactivity series experimental task requires every species to be tested at least once, using only relative information about pairs of items (i.e., without access to absolute properties of the individual species or other side information), and the goal is to produce a ranking of those items. In both cases, an algorithm is used to plan an optimal experiment sequence, mimicking how a productive scientist performs experiments to yield the most information with the least effort, time, and materials. Therefore, in additional to fundamental interest, teaching algorithmic strategies for experiment planning is of general value for science students and complements

previous work on statistical design of experiments activities for optimization that have appeared previously in this journal.[9–11] The reactivity series experiment has the advantage that efficient planning is achieved by simple algorithms that are amenable to mathematical analysis.

Revisiting this classic laboratory experiment provides an opportunity to cultivate computational problem-solving skills in chemistry students. Algorithmic methods (without computers *per se*) are often invoked when teaching general chemical problem solving related to stoichiometric problems, drawing of Lewis structures, or prediction of molecular shape using VSEPR theory.[12] There is a growing appreciation of the importance of computational thinking across all disciplines,[13] and for teaching chemical principles in particular.[14] The ACS Committee on Professional Training recommends incorporating computational and informatics methodologies into the undergraduate curriculum.[15] However, efforts to explicitly introduce computational thinking into chemistry curricula at the high-school[16] and first-year university level[17,18] have focused more on numerical calculations, data analysis, programming, and simulation aspects of computational thinking. In contrast, our approach to the activity series experiment emphasizes *abstraction* and *algorithmic thinking* competencies for a non-numerical problem, without the need for explicit computer programming. Instructors can incorporate the computational thinking ideas presented here into an existing laboratory without the need for significant modifications or displacing other topics in the curriculum.

Instructors should not expect students to have prior knowledge on this topic. Only about a third of first-year university students can correctly describe a sorting algorithm, and this only increases to two-thirds of students who enroll in a university-level computer science course.[19] Even those students who correctly describe an algorithm typically describe *inefficient* algorithms

such as insertion sort (*vide infra*); the value of avoiding this becomes apparent when placed in an experimental setting. Although our discussion focuses on teaching chemistry students about how sorting relates to their experiments, highlighting this connection may also benefit computer science pedagogy. Students who have completed an introductory computer science course are more likely to default to descriptions of numeric sorts rather than more general descriptions of sorting processes.[19] This may arise because the typical classroom examples involve sorting lists of numbers. Introducing the activity series problem—where the data items are chemical species, rather than numbers and the comparisons are experiments rather than numerical inequalities—may help broaden their perspective appropriately and emphasize the value of efficiency.

Comparison of Sorting Algorithms

Algorithms are characterized by the number of resources used, such as the number of operations needed to achieve a result, the amount of memory required to store intermediate results, communication between distributed workers, complexity of the implementation, or other constraints. An algorithm may be better in one of these metrics while being worse in another. For activity series determination, the most appropriate goal is minimizing the number of comparison operations that must be performed, as each comparison corresponds to an experiment which associated time and reagent costs; any computational effort needed to implement the algorithm or store intermediate results is trivial with respect to that real-world laboratory process. Different numbers of comparisons may be required to sort different possible initial arrangements of the list. For example, if fortuitously provided with a list in the correct order, merely $n - 1$ experiments (testing each adjacent pair) suffice to verify that the order is

correct. However, this is unlikely to occur in practice, as it corresponds to just one of the $n!$ possible arrangements of the list. Common ways to assess the number of comparisons include the average number (over all possible initial guesses) or the maximum ("worst-case scenario") needed to obtain a ranking. The Big-O notation is a shorthand used by computer scientists to denote the asymptotic performance by indicating the leading-order terms as a function of problem size; for example, $O(n^2)$ indicates that the number of operations grows as the square of the size of items, n. Big-O notation does not include numerical prefactors or lower-order terms, and thus emphasizes the asymptotic behavior of the algorithms with problem size. It is sometimes also possible to determine the exact number of operations required, as in the cases discussed below.

Guess and check is one of the worst possible strategies that could be adopted: Propose a possible ordering of the activity series, then performs $n - 1$ experiments to verify that the ranking is correct. If not correct, repeat with the next possible ordering. As there are $n!$ possible arrangements to check, this will require $(n - 1)\ (n!)$ experiments in the worst case. The correct ordering could be any of the permutations, so on average we only need to try half of these experiments. However, for $n=9$ species, this still requires an average of 1.4 million experiments. Clearly a smarter approach is needed.

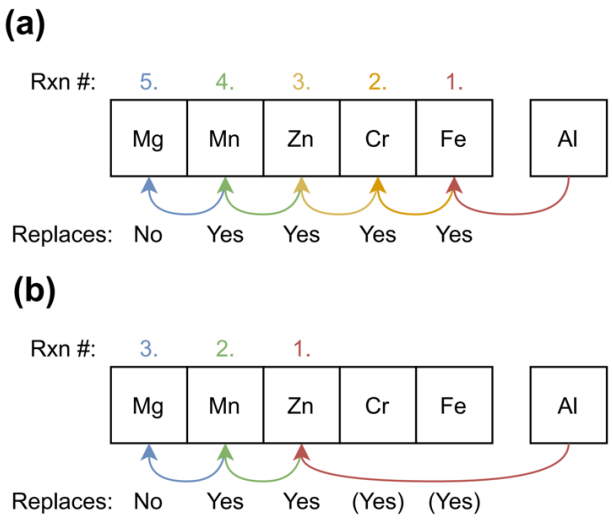

Figure 1: Insertion Sort and Binary Insertion Sort. We depict the final step in the insertion sort algorithm applied to sorting 6 elements by reactivity; the previous 4 steps proceeded identically to this one and put Mg-Fe in order of decreasing activity. The goal in the current step is to find the correct place in the list in which to insert Al (between Mg and Mn). The reactions are numbered and proceed from right to left. **(a)** Insertion sort performs a linear search for the insertion point, traversing from the least reactive to most reactive elements (or vice versa), performing an experiment until a reaction does not occur. **(b)** Binary insertion sort performs a binary search for the insertion point, starting at the median element and taking advantage of the fact that the Mg-Fe are already sorted. Given the pre-sorting of Mg-Fe, an observation that Al reacts with Zn implies it will react with Cr and Fe.

Insertion Sort (depicted in Figure 1a) is one of the simplest algorithms to consider; incidentally, it is one of the most common algorithms described by computer science undergraduate students when asked to define a sorting algorithm.[19] (An "out-of-place" implementation is described here, although computer science courses will often describe an "in-place" variant which requires the same number of comparisons but reduces computer memory by

eliminating the need for a separate sorted list.) Given a list of unsorted items, begin by creating an empty list into which we will add sorted entries in the correct rank order. At each iteration, take the next item from the unsorted list and determine where to insert it into the sorted list by comparing it against each of the items in the sorted list. For the activity series, this corresponds to testing each new species against the species already in the sorted list until a reaction does not occur to determine its placement within the descending list of reactivities. Insertion sort requires $O(n^2)$ comparison operations for n items to be compared, because each of the n items must be compared against each of the other $n-1$ items. More precisely, only half of these comparisons are needed as the sorted list is constructed, so the worst-case scenario requires $n(n-1)/2$ comparisons; for an activity series of n=9 species, at most 36 comparison experiments are required. The average number of required comparisons (averaged over possible arrangements of the input) is somewhat lower than the worst-case, and can be determined by numerical simulations (see the Supporting Information). For insertion sort, an average of 24.2 experiments are required to sort n=9 species.

Binary Insertion Sort (depicted in Figure 1b) greatly reduces the number of comparisons needed by performing a binary search to determine where to insert a new entry into the sorted list, rather than testing the entries sequentially. At each step, test the new item against the median entry in the sorted list. If the new item is more (reactive) than this median item, its correct placement is in the first half of the sorted list, and if it is less (reactive) than the median then its correct placement is in the last half. Repeat the process against the median entry in the relevant half-list, until a single location is identified. In general, this requires $\log_2 n$ subdivisions (expressed in terms of the binary, i.e., base-2, logarithm). The binary search is repeated for each

of the n items to be sorted, resulting in an asymptotic worst-case requirement of $O(n \log_2 n)$ comparisons. The exact expression can be described analytically,[20] and at most 21 comparisons are required to rank 9 items. Adopting the binary insertion algorithm reduces the maximum number of experiments needed by more than 40% compared to the insertion algorithm. Numerical simulations indicate that an average of 19.9 experiments are needed to perform binary insertion sorting of n=9 species.

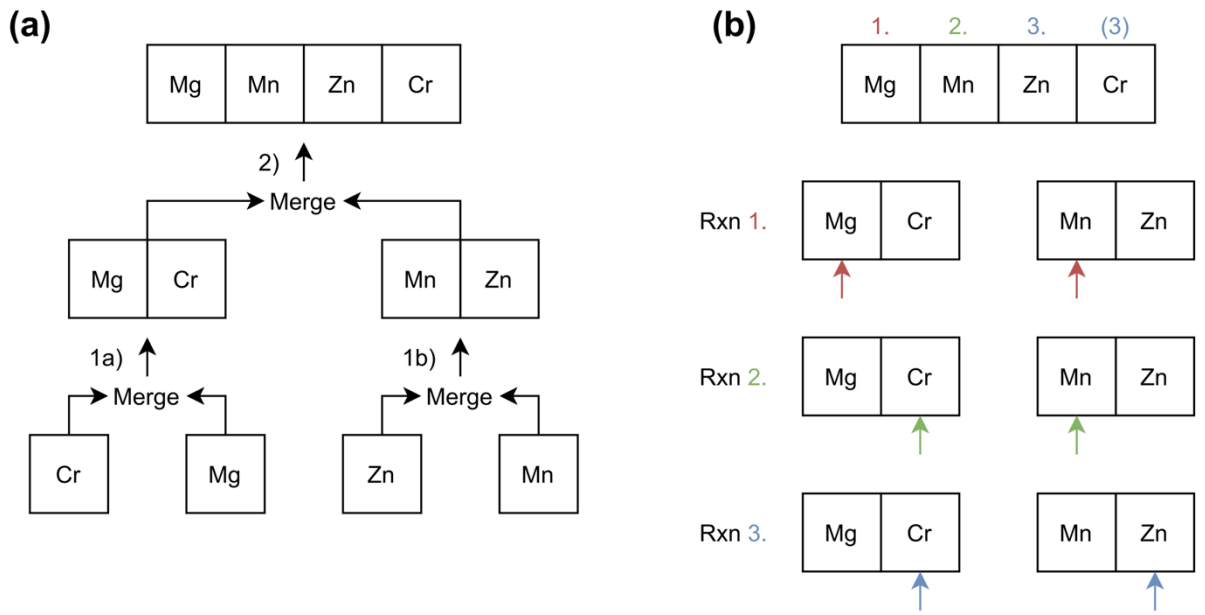

Figure 2: Merge Sort. We depict the merge sort algorithm applied to rank the elements Cr, Mg, Zn, Mn in order of decreasing reactivity. (a) Overall structure of the algorithm: Starting from individual elements in the original unsorted order, themselves each sorted lists of length 1, pairs of sorted lists are merged together until only one sorted list remains. Steps 1a and 1b can be performed in parallel. (b) Detailed view of the 'merge' routine in step 2 of panel (a): At each step of 'merge', pairs of elements across the two lists are compared, beginning with the most reactive element from each list. The more reactive element from the experiment is chosen and inserted into an empty list (top) which will store the final sorted output of 'merge'. The next element from the list from which the 'winner' of the previous comparison is compared with the 'loser' thereof, and so on until the empty list is filled.

Merge Sort (depicted in Figure 2)**,** developed by John von Neuman in 1945, was one of the first programs written for a stored memory computer.[21] The unsorted list is divided into *n*

sublists, each containing one item; a list of only one item is intrinsically "sorted". These sorted lists are merged and resorted to produce new sorted sublists. The merging process is repeated until only a single (sorted) list remains. An advantage of merge sort is that each sublist merge step can be performed in parallel. This can be useful in a laboratory setting, as the students can "divide and conquer" the experiments in an efficient way. Like binary insertion sort, this algorithm is also $O(n \log_2 n)$ and requires the same number of (exact) worst-case comparisons.[20] However, the average number of comparisons for merge sort is slightly lower; for $n=9$ only 19.2 comparisons are required.

Merge Insertion Sort (also known as the Ford-Johnson algorithm) was invented in 1959,[22] and combines merge and binary insertion ideas. The general premise is that $k + 1$ comparisons are needed to perform binary insertion into sorted sublists of $2^{k+1} - 1$ and 2^k, so sublists should be organized such that insertions are performed into the largest sublists possible. In theory it is one of the most optimal sorting algorithms in terms of number of comparisons required, but it is rarely used in practice as the required data structures are complicated and require additional processor operations that erode this advantage for typical sorting tasks. (Other algorithms requiring fewer comparisons for large n are known to exist,[23] but Ford-Johnson requires the fewest comparisons for $n \leq 47$ items.[24]) The exact number of comparisons needed in the worst case can be calculated exactly,[25] and for $n=9$ items at most 19 comparisons are required; this is two fewer experiments than the worst cases for binary insertion and merge sort algorithms, and even slightly lower than the *average* number of comparisons needed for either of those other algorithms. The average number of required comparisons, 18.6, is only slightly lower than the worst-case requirement.[26]

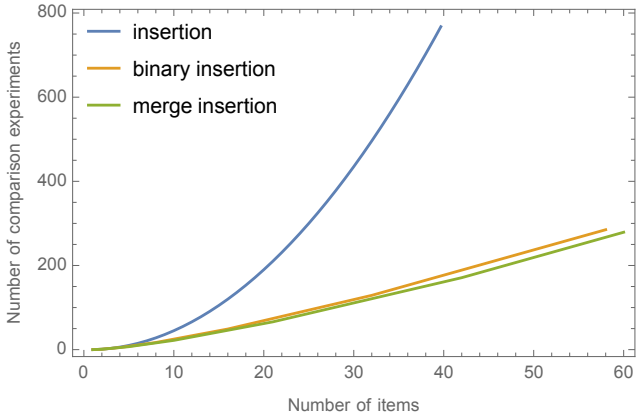

Figure 3: Comparison of exact number of comparison experiments needed as a function of number of items to be compared. The curve for merge sort is identical to binary insertion, so it is not shown.

Larger problems. By changing the algorithm, the maximum number of experiments needed for the $n=9$ problem can be reduced from 36 (using insertion sort) to 19 (using merge insertion sort). This is an appreciable savings, although an intellectually lazy (but hard working) student might argue that the less efficient approach is still tractable. This notion can be dispelled by plotting the number of experiments needed for each algorithm as the number of items grows in Figure 3; insertion sort grows quadratically with the number of items, whereas the other algorithms we have discussed require only $O(n \log_2 n)$ experiments. How many experiments are needed to determine an n=30 activity series? Insertion sort requires at most 435 experiments (and on average 243.5), whereas binary insertion and merge sort require at most 119 experiments (and on average 114.4 and 111.5, respectively), while merge insertion requires at most 111 experiments (and on average 108.5 experiments). The Mathematica notebook in the Supporting Information contains interactive functions for computing the exact worst-case values for

arbitrary numbers of items, implementations of the insertion, binary insertion, and merge sort algorithms, and numerical simulations for estimating the average number of comparisons.

Generalizations. The algorithms discussed above sort an arbitrary (and non-predetermined) number of items using only information obtained from pairwise comparison operations. Further improvements are possible by removing these assumptions. If the number of items is known ahead of time, and the goal is to find a predetermined work plan that uses the shortest amount of *elapsed* time by distributing comparison experiments across students working in parallel, *sorting networks* may be a more appropriate model. Background information known about the species (e.g., "alkali metals are more reactive than coinage metals") can be incorporated into *non-comparison sorting algorithms* (e.g., bucket sort); essentially this information is a form of pre-sorting of the items into smaller groups which reduces the total number of comparison operations needed. If instead of relying solely upon pairwise comparisons of relative ordering one has access to a value for each item which establishes its absolute ranking (e.g., the electrochemical half-cell potential of each species relative to a common reference electrode), then the n experiments needed to determine those values suffices, after which a (solely numerical) sort can be performed to order the items without further pairwise experiments. The analysis above assumed that each comparison has the same unit cost and that all comparisons can be performed; this is not necessarily true in a laboratory setting, where some reagents might be more expensive than others (e.g., gold versus copper) or some reaction pairs may be forbidden (e.g., because of safety concerns). The corresponding problems of sorting items by minimizing the total of the costs of comparison operation[27] and sorting under forbidden pairs[28] have been treated; it is common to analyze these in terms of a graph where the vertices

denote each item and the edges are weighted (or absent) depending on the cost. Finally, we have assumed that each comparison returns a deterministic outcome; while this is the case for the activity series, other types of experiments may yield a statistical distribution of outcomes, necessitating additional experiments beyond the minimum requirements above discussed above.

Further Resources

Chemical educators can draw upon an extensive computer science pedagogy literature around teaching sorting algorithms using song,[29] folk dancing,[30] and student-led exercises analyzing written algorithms without explicit computer programming.[31] Because sorting is a core part of the computer science curriculum, a plethora of learning resources exist. Here we highlight a few resources we have found most helpful: Christian and Griffiths' *Algorithms to Live By: The Computer Science of Human Decisions* is an entertaining, no-code introduction of algorithms and their applications to real-life problems intended for a popular audience; Chapter 3 discusses the sorting problem using anecdotes related to laundry, athletic tournaments, libraries, and animal behavior.[4] Joe James has created a series of YouTube videos with animated sorting algorithms and companion code in the Python[32] and Java[33] programming languages. The Wolfram Demonstration project has two interactive demonstrations of sorting algorithms.[34,35] Khan Academy's "Computer Science: Algorithms" unit provides a self-paced series of videos and activities introducing algorithms, binary search, asymptotic (Big-O) notation, and several sorting algorithms (including insertion and merge sort).[36] The authoritative reference on sorting algorithms is Knuth's *The Art of Computer Programming, Volume 3: Sorting and Searching*, a highly technical book devoted to implementation and mathematical analysis of the algorithms discussed here.[5] Equally authoritative, but more approachable, is the CLRS *Introduction to*

Algorithms textbook;[37] insertion sort is used as an introductory example of algorithm analysis in Chapter 2, and a more extensive discussion of sorting algorithms occupies Chapters 6-9, however it does not describe the Ford-Johnson merge insertion algorithm.

Conclusion

We have elucidated the relationship between the classic activity series determination chemistry experiment and sorting algorithms from computer science. Selecting an appropriate algorithm drastically reduces the number of experiments required to determine the relative reactivities. Even for a modest pedagogical-scale determination of an activity series of 9 species, the number of pairwise experiments can be nearly halved by adopting more efficient algorithms. Furthermore, adopting an appropriate algorithm such as the merge sort allows experimental tasks to be distributed across independently acting students in an efficient way. For laboratory tasks where the time and materials costs of performing experimental comparisons outweighs any computational or mental resources, the use of theoretically optimal algorithms (such as the Ford-Johnson merge insertion sort) is warranted. In general, sorting algorithms are an optimal way of planning *any* experiment whose goal is producing a rank ordering of items and where the only information comes from pairwise comparisons of those items.

Associated Information

Supporting Information

Supporting information is available at [ACS Information].

- Mathematica 12.1 notebook containing exact scaling results used to generate Figure 3, implementations of the algorithms, and numerical simulations of an activity series experiment. (ZIP)

Acknowledgements

We thank Isaac Tamblyn for suggesting the problem of "chemical games" based upon introductory chemistry laboratory experiments, which motivated this work. We acknowledge support by the National Science Foundation (DMR-1928882) and the Henry Dreyfus Teacher-Scholar Award (TH-14-010).

References

(1) Kieffer, W. F. The Activity Series of the Metals. *J. Chem. Educ.* **1950**, *27*, 659. https://doi.org/10.1021/ed027p659.

(2) Metal Activity Series http://dept.harpercollege.edu/chemistry/chm/100/dgodambe/thedisk/series/series.htm (accessed Nov 26, 2020).

(3) Steinhaus, H. *Mathematical Snapshots*, 3rd ed.; Dover Publications: Mineola, N.Y, 1999, pp. 37-40.

(4) Christian, B.; Griffiths, T. *Algorithms to Live By: The Computer Science of Human Decisions*; Picador: New York, 2017, pp. 59-83.

(5) Knuth, D. E. *The Art of Computer Programming: Volume 3: Sorting and Searching*, 3rd ed.; Addison-Wesley: Reading, Mass, 1997; Vol. 3.

(6) Murphy, N.; Woods, D.; Naughton, T. J. *Stable Sorting Using Special-Purpose Physical Devices*; http://www.cs.nuim.ie/~tnaughton/pubs/varasto/BCRI-Preprint-06-2006.pdf; Boole Centre for Research in Informatics, University College Cork, Ireland, 2006; p 10.

(7) Turk-MacLeod, R.; Henson, A.; Rodriguez-Garcia, M.; Gibson, G. M.; Aragon Camarasa, G.; Caramelli, D.; Padgett, M. J.; Cronin, L. Approach to Classify, Separate, and Enrich Objects in Groups Using Ensemble Sorting. *Proc Natl Acad Sci USA* **2018**, *115*, 5681–5685. https://doi.org/10.1073/pnas.1721929115.

(8) Häse, F.; Roch, L. M.; Aspuru-Guzik, A. Next-Generation Experimentation with Self-Driving Laboratories. *TRECHEM* **2019**, *1*, 282–291. https://doi.org/10.1016/j.trechm.2019.02.007.

(9) Lang, P. L.; Miller, B. I.; Nowak, A. T. Introduction to the Design and Optimization of Experiments Using Response Surface Methodology. A Gas Chromatography Experiment for the Instrumentation Laboratory. *J. Chem. Educ.* **2006**, *83*, 280. https://doi.org/10.1021/ed083p280.

(10) Krawczyk, T.; Słupska, R.; Baj, S. Applications of Chemiluminescence in the Teaching of Experimental Design. *J. Chem. Educ.* **2015**, *92*, 317–321. https://doi.org/10.1021/ed5002587.

(11) Snetsinger, P.; Alkhatib, E. Flexible Experiment Introducing Factorial Experimental Design. *J. Chem. Educ.* **2018**, *95*, 636–640. https://doi.org/10.1021/acs.jchemed.7b00431.

(12) Bodner, G. M. The Role of Algorithms in Teaching Problem Solving. *J. Chem. Educ.* **1987**, *64*, 513. https://doi.org/10.1021/ed064p513.

(13) Wing, J. M. Computational Thinking and Thinking about Computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* **2008**, *366*, 3717–3725. https://doi.org/10.1098/rsta.2008.0118.

(14) *Using Computational Methods To Teach Chemical Principles*; Grushow, A., Reeves, M. S., Eds.; American Chemical Society, Series Ed.; ACS Symposium Series; American Chemical Society: Washington, DC, 2019; Vol. 1312. https://doi.org/10.1021/bk-2019-1312.

(15) ACS Guidelines for Bachelor's Degree Programs https://www.acs.org/content/acs/en/education/policies/acs-approval-program/guidelines-supplements.html (accessed Nov 26, 2020).

(16) Matsumoto, P. S.; Cao, J. The Development of Computational Thinking in a High School Chemistry Course. *J. Chem. Educ.* **2017**, *94*, 1217–1224. https://doi.org/10.1021/acs.jchemed.6b00973.
(17) Sharma, A. A Model Scientific Computing Course for Freshman Students at Liberal Arts Colleges. *JOCSE* **2017**, *8*, 2–9. https://doi.org/10.22369/issn.2153-4136/8/2/1.
(18) Holme, T. Systems Thinking as a Vehicle To Introduce Additional Computational Thinking Skills in General Chemistry. In *It's Just Math: Research on Students' Understanding of Chemistry and Mathematics*; ACS Symposium Series; American Chemical Society, 2019; Vol. 1316, pp 239–250. https://doi.org/10.1021/bk-2019-1316.ch014.
(19) Simon, B.; Chen, T.-Y.; Lewandowski, G.; McCartney, R.; Sanders, K. Commonsense Computing: What Students Know before We Teach (Episode 1: Sorting). In *Proceedings of the 2006 International Workshop on Computing Education Research - ICER '06*; ACM Press: Canterbury, United Kingdom, 2006; p 29. https://doi.org/10.1145/1151588.1151594.
(20) A001855 - OEIS https://oeis.org/A001855 (accessed Nov 26, 2020).
(21) Knuth, D. E. Von Neumann's First Computer Program. *ACM Comput. Surv.* **1970**, *2*, 247–260. https://doi.org/10.1145/356580.356581.
(22) Ford, L. R.; Johnson, S. M. A Tournament Problem. *The American Mathematical Monthly* **1959**, *66*, 387. https://doi.org/10.2307/2308750.
(23) Manacher, G. K. The Ford-Johnson Sorting Algorithm Is Not Optimal. *J. ACM* **1979**, *26*, 441–456. https://doi.org/10.1145/322139.322145.
(24) Peczarski, M. The Ford–Johnson Algorithm Still Unbeaten for Less than 47 Elements. *Information Processing Letters* **2007**, *101*, 126–128. https://doi.org/10.1016/j.ipl.2006.09.001.
(25) A001768 - OEIS https://oeis.org/A001768 (accessed Nov 26, 2020).
(26) Stober, F.; Weiß, A. On the Average Case of MergeInsertion. *Theory Comput Syst* **2020**, *64*, 1197–1224. https://doi.org/10.1007/s00224-020-09987-4.
(27) Gupta, A.; Kumar, A. Sorting and Selection with Structured Costs. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*; IEEE: Newport Beach, CA, USA, 2001; pp 416–425. https://doi.org/10.1109/SFCS.2001.959916.
(28) Banerjee, I.; Richards, D. Sorting Under Forbidden Comparisons; Leibniz International Proceedings in Informatics (LIPIcs); Schloß Dagstuhl--Leibniz-Zentrum für Informatik, 2016; Vol. 53, p 22:1--22:13. https://doi.org/10.4230/LIPICS.SWAT.2016.22.
(29) Schreiber, B. J.; Dougherty, J. P. Assessment of Introducing Algorithms with Video Lectures and Pseudocode Rhymed to a Melody. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*; ACM: Seattle Washington USA, 2017; pp 519–524. https://doi.org/10.1145/3017680.3017789.
(30) AlgoRythmics - YouTube https://www.youtube.com/user/AlgoRythmics/videos (accessed Nov 26, 2020).
(31) Nasar, A. A Mathematical Analysis of Student-Generated Sorting Algorithms. *The Mathematics Enthusiast* **2019**, *16*, 315–330.
(32) James, Joe. Python Sorting Algorithms - YouTube https://www.youtube.com/playlist?list=PLj8W7XIvO93rJHSYzkk7CgfiLQRUEC2Sq (accessed Feb 16, 2021).

(33) James, Joe. Java Sorting Algorithms - YouTube https://www.youtube.com/playlist?list=PLj8W7XIvO93qVnnXxyeWmCSvMFqRBP4Jw (accessed Feb 16, 2021).
(34) Stevens, P. Sorting Algorithms https://demonstrations.wolfram.com/SortingAlgorithms/ (accessed Feb 17, 2021).
(35) Urban, K. Comparing Sorting Algorithms on Rainbow-Colored Bar Charts https://demonstrations.wolfram.com/ComparingSortingAlgorithmsOnRainbowColoredBarCharts/ (accessed Feb 17, 2021).
(36) Algorithms | Computer science | Computing https://www.khanacademy.org/computing/computer-science/algorithms (accessed Nov 26, 2020).
(37) Cormen, T. H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, Mass, 2009.

Graphical Table of Contents Image