**Algorithm Representation**

**Selection Sort**

* Time complexity:
  + Best case: O(n^2)
  + Average case: O(n^2)
  + Worst case: O(n^2)
* Space complexity: O(1)
* Idea: Separate the data set into 2 sections: sorted and unsorted. Initially, the sorted section will be empty, and the unsorted will contain the whole data set. Then, for each unsorted element, we choose the smallest, add it to the sorted section and remove it from the unsorted.
* Step-by-step:
  + First, for each element in the list, we will call this “selected element”, select the smallest in range from selected element to the end of the list.
  + Second, swap the smallest element found in the above step with the selected element.
  + Third, repeat the two steps above until the selected element is the penultimate item in the list. A diagram of a swapping elements

    Description automatically generatedA diagram of a swapping process

    Description automatically generated**A diagram of a number

    Description automatically generated**A diagram of a number

    Description automatically generated**A green rectangular box with numbers

    Description automatically generated**
* Improvement:
  + Instead of only choosing the smallest element, we also select the largest element, then swap the smallest element with the selected element and swap the largest element with element at n – 1 – selected element’s position.

**Merge Sort**

* Time complexity:
  + Best case: O(nlogn)
  + Average case: O(nlogn)
  + Worst case: O(nlogn)
* Space complexity: O(n)
* Idea: Split the data set recursively into subsets, until there are all subsets with only one element. Then, every pair of adjacent subsets, merge them following this rule:
  + If the current item in set 1 is larger than the current item in set 2, push the item in set 2 into a common array/list. If not, push the item in set 1.
  + If either set is empty when the other is not, push the rest of the other set into the common list.

There, this also means that if the element number is not even, we are still able to merge them. By merging all available pairs of adjacent subsets, then finally, merge the odd subset when the number of subsets is now even.

* Step-by-step
  + Merge sort is a recursive sorting algorithm.
  + About the merge sort function:
    - The function requires 3 parameters: left limit, right limit, and the array.
    - If the left limit is smaller than the right limit, meaning that the sub array(s) has not been reduced to the size of one, we calculate the middle position by using the formula (left + right) / 2. Then, we continue to call the merge sort function to recursively split the array into halves. This time we call 2 merge sort functions, one with left as it is and right as middle, and one with right as it is and left as middle + 1. This indicates that we split the parameter array into 2 halves.
    - When the sizes of the sub arrays are one, then we can recursively call the merge function to merge the sub arrays.
  + About the merge function:
    - The function requires 4 parameters: left limit, right limit, middle and the array.
    - First, we determine the left and right sizes of the pair of sub array.
    - Second, we push the element from the main array to 2 sub arrays. The left array will start from left limit to middle. The right array will start from the middle + 1 to the right limit.
    - Third, we will replace the elements in range left limit to right limit in the original array. This time, we will compare the front of the left array and right array. For example, if the left array’s front is larger than the right array’s, then the array left limit position will hold the right array’s front’s value and vice versa. This guarantees the array will be sorted and no key being duplicated. Then, we increase the front of either left array or right array, accordingly, and increase the index of left limit. This loop will end until either sub array reaches its end.
    - Finally, we push back the other array whose front did not reach the end, increasing its front and the left limit of the original array until it reaches the right limit. A diagram of a array

      Description automatically generatedA diagram of a diagram

      Description automatically generated

A diagram of a cell number

Description automatically generatedA screenshot of a computer

Description automatically generated

* Improvement:
  + Per this image describing the performance of 3 sorting algorithms, we can see that merge sort performance excels when the input size is large, substantial and it remain faster as the input size grow.**A graph with a line and a line

    Description automatically generated**
  + Therefore, if the size of the subarray is small, we could use insertion sort or binary insertion sort to optimize the speed. And if it is larger than the threshold, then now we could utilize the merge function for speed optimizing.

**Shell sort**

* Time complexity (Original):
  + Best case: O(nlogn).
  + Average case: O(nlogn), depend on the chosen sequence.
  + Worst case: O(n^2).
* Space complexity: O(1).
* Idea: Shell sort is an improvement of insertion sort. It first sort elements that have specific gaps, determined by the used increment rule used. It then reduces the gap recursively until the gap is one. The inner sorting method is insertion sort, customized by the interval or gap. If the gap becomes one, the inner sorting becomes the original insertion sort. (1)
* Step-by-step:
  + The interval first equals the size of the array divided by 2, then the interval will reduce itself by 2 at every iteration, until it becomes 1.
  + We then sort the array/list using the insertion sort with the interval.
  + Consider array {12, 34, 54, 2, 3}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 34 | 54 | 2 | 3 |

* + - First, we have the interval is 5/2 = 2. So, we sort {12, 54, 3} and {34,2} concurrently using insertion sort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 12 | 34 | 54 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 12 | 34 | 54 |

* + - Second, we will have the interval as 5/4 = 1, so we sort {3, 2, 12, 34, 54} using insertion sort.
* Because we have reached interval= 1, we terminate the loop and end the sorting. (2)
* Variation:
  + Because the interval/gap used in shell sort is not fixed, the algorithm can improve based on the increment/decrement sequence we use. Some common sequences:
    - **Knuth’s increments**: 1, 4, 13, …, (3^k – 1)/ 2

**Sedgewick's increments:** 1, 8, 23, 77, 281, 1073, 4193, 16577...4j+1+ 3·2j+ 1

* + - **Hibbard's increments:** 1, 3, 7, 15, 31, 63, 127, 255, 511…
    - **Papernov & Stasevich increment:** 1, 3, 5, 9, 17, 33, 65, ...
    - **Pratt**: 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 27, 16, 24, 36, 54, 81....

**Reference**

* Selection sort’s images and complexity: Selection sort – Data structure and algorithms tutorials, GeeksforGeeks, [Selection Sort – Data Structure and Algorithm Tutorials - GeeksforGeeks](https://www.geeksforgeeks.org/selection-sort/).
* Merge sort’s images and complexity: Merge sort – Data structure and algorithms tutorials, GeeksforGeeks, [Merge Sort - Data Structure and Algorithms Tutorials - GeeksforGeeks](https://www.geeksforgeeks.org/merge-sort/).
* Comparison line chart between insertion, binary insertion and merge sort: CS240, PA4: Sorting improvements, Mike Lam – James Maddison University, [CS240 - PA4 (jmu.edu)](https://w3.cs.jmu.edu/lam2mo/cs240_2014_08/pa04-sorting.html).
* (1): Modified and paraphrasing from Shell Sort Algorithm, Programiz, [Shell Sort (With Code in Python, C++, Java and C) (programiz.com)](https://www.programiz.com/dsa/shell-sort).
* (2): Example taken from article ShellSort, GeeksforGeeks, [ShellSort - GeeksforGeeks](https://www.geeksforgeeks.org/shellsort/).