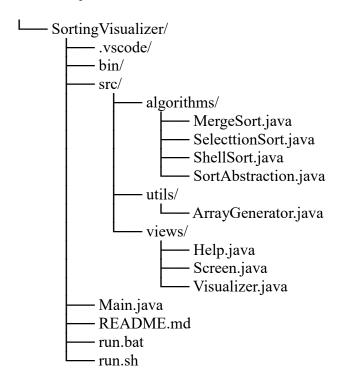
GROUP 3: SORTING VISUALIZATION OF SELECTION SORT, MERGE SORT, SHELL SORT AND QUICK SORT ALGORITHMS.

Participants: Bùi Xuân Son – 20226065: GUI and SelectionSort

Phạm Thành Nam – 20225989: MergeSort Luân Quang Minh – 20225985: QuickSort Vũ Đức Manh – 20226054: ShellSort

1. File system



Source code:

- 'Main' class: entry point initializing program.
 - + extending 'javax.swing.JFrame' class: create application window.
 - + add confirmation dialog when closing window.
- 'views' package: including components for user interface
- + '**Help**' class: extending javax.swing.JLabel, <u>display guideline</u> for users in each sorting animation.
- + 'Screen' class: extending javax.swing.JPanel, as a <u>wrapper</u> including elements(controls) as JButton, JPanel, JTextField, JSlider, ...etc.
 - + 'Visualizer' class: extending java.awt.Canvas, display sorting animation.
- 'utils' package: including class(es) used as utilities:
- + 'ArrayGenerator' class: helper class which can generate random array or transform input sequence to array of numbers.
- 'algorithms' package: includes classes <u>performing sorting process</u>, using methods of Visualizer class for visualizing tasks.
 - + 'SortAbstraction' class: a abstract class, generalization for 3 sorting classes.

+ 'SelectionSort', 'MergeSort', 'ShellSort' class: extending SortAbstraction to *implement* 'SortAbstraction::sort()' method which is setting up logic for animation in 'Visualizer' class.

2. Algorithm

2.1. Selection Sort:

- Step-by-step:

- 1. **Step 1**: i=1.
- 2. **Step 2**: Find the minimum a[min] in array from a[i] to a[n].
- 3. **Step 3**: Swap a[min] and a[i]
- 4. **Step 4**: If i<=n-1, i=i+1; repeating step 2. Or else stop (n-1 element(s) sorted)

- Time complexity:

© Best Case: O(n^2)

O Average Case: O(n^2)

10 Worst Case: O(n^2)

2.2. Merge Sort:

- Step-by-step:

- **O Divide** by finding the number of the position midway between and . Do this step the same way we found the midpoint in binary search: add and, divide by 2, and round down.
- **©** Conquer by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray array[p..q] and recursively sort the subarray array[q+1..r].
- **©** Combine by merging the two sorted subarrays back into the single sorted subarray array[p. . r].

- Time complexity:

- **©** Best Case: O(n log n), When the array is already sorted or nearly sorted.
- **O** Average Case: O(n log n), When the array is randomly ordered.
- **Worst Case:** O(n log n), When the array is sorted in reverse order.

2.3. Shell Sort:

- Step-by-step:

- 1. **Step 1**: Start
- 2. **Step 2**: Initialize the value of gap size, say h.
- 3. **Step 3**: Divide the list into smaller sub-part. Each must have equal intervals to h.
- 4. **Step 4**: Sort these sub-lists using insertion sort.
- 5. **Step 5**: Repeat this step 2 until the list is sorted.
- 6. **Step 6**: Print a sorted list.

7. **Step 7**: Stop.

- Time complexity:

© Best Case: When the given array list is already sorted the total count of comparisons of each interval is equal to the size of the given array.

So best case complexity is $\Omega(n \log(n))$.

- **O** Average Case: $O(n \log n) \sim O(n^{1.25})$.
- **©** Worst Case: $O(n^2)$.

2.4. Quick Sort:

- Step-by-step:
 - 1. **Step 1**: Choose a pivot element from the array.
 - 2. **Step 2**: Partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
 - 3. **Step 3**: Recursively apply quicksort to the left and right sub-arrays.
 - 4. **Step 4**: The recursion terminates when the sub-array has 0 or 1 elements.

- Time complexity:

- **©** Best Case: Ω (N log (N)) when pivot is located near the middle
- **\Phi** Average Case: θ (N log (N))
- **1** Worst Case: $O(n^2)$