**LAB 6 REPORT**

Problem 1: PartitionApp.java

In this problem, we partition an array using different pivot selection strategies and measure the number of comparisons and swaps during the process, calculates the average performance metrics over multiple runs.

These are variables used to count comparisons and swaps during partitioning.

A computer screen shot of a program code

Description automatically generated

Also, The program tests **four pivot strategies**:

* First Element: Pivot is the first element of the array.
* Last Element: Pivot is the last element.
* Middle Element: Pivot is the middle element.
* Random Element: Pivot is chosen randomly.

After that, we run the partitioning process 100 times for a specific pivot strategy,

And calculates and displays the average comparisons and swaps.

A screen shot of a computer program

Description automatically generated

The Pivot selection: function selects the pivot index based on the chosen strategy:

* "first": First element.
* "last": Last element.
* "middle": Middle element.
* "random": A randomly selected element.

A computer screen shot of a program code

Description automatically generated

Next step, we using Partitioning Logic to:

Select Pivot: Chooses the pivot based on the strategy.

Reordering: Places smaller elements to the left of the pivot and larger elements to the right.

Count Metrics:

* Comparisons: Incremented during each comparison in the loop.
* Swaps: Incremented when elements are exchanged.

A screen shot of a computer program

Description automatically generated

Beside that, we swap two elements in the array and increments the swaps counter.

A screen shot of a computer code

Description automatically generated

Finally, we create a random array of integers to test the partitioning.

A screen shot of a computer code

Description automatically generated

*Output:*

A black screen with white text

Description automatically generated

Problem 2: Merge sort, Shell sort, Quick sort

In this problem, we add counters for the number of comparisons, copies, and swaps in Merge sort, Shell sort, Quick sort and display them after sorting.

output:

A screenshot of a computer

Description automatically generated

Table to visualize:

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Merge Sort COPIES | Shell Sort COPIES | Quick Sort COPIES |
| 10000 | 133616 | - | - |
| 15000 | 208616 | - | - |
| 20000 | 287232 | - | - |
| 25000 | 367232 | - | - |
| 30000 | 447232 | - | - |
| 35000 | 529464 | - | - |
| 40000 | 614464 | - | - |
| 45000 | 699464 | - | - |
| 50000 | 784464 | - | - |
|  |  |  |  |

Merge Sort COPIES:

* Merge Sort requires the most copies, as it needs to merge subarrays during each divide-and-conquer step.
* **Trend**: As the input size increases, the number of copies increases linearly with it.

Shell Sort COPIES and Quick Sort COPIES:

* Shell Sort COPIES: Shell Sort requires fewer copies than Merge Sort because it's an in-place algorithm, but the number can still vary based on the gap sequence and sorting steps.
* Quick Sort COPIES: Quick Sort also requires fewer copies compared to Merge Sort, as it performs partitioning in place (without needing extra space for arrays).

Merge Sort > Quick Sort > Shell Sort (Merge Sort needs the most copies, Quick Sort performs more than Shell Sort).

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Merge Sort COMPS | Shell Sort COMPS | Quick Sort COMPS |
| 10000 | 120402 | 149876 | 159004 |
| 15000 | 189226 | 261478 | 249995 |
| 20000 | 260773 | 360419 | 346201 |
| 25000 | 334130 | 499921 | 463541 |
| 30000 | 408555 | 628943 | 528532 |
| 35000 | 484609 | 690538 | 602417 |
| 40000 | 561869 | 899984 | 790217 |
| 45000 | 639691 | 954968 | 929279 |
| 50000 | 718231 | 1251972 | 949021 |

For COMPS:

* Merge Sort is generally the most efficient in terms of the number of comparisons across all input sizes.
* Shell Sort has a higher number of comparisons as the input size increases due to the inefficient gap sequence.
* Quick Sort generally performs fewer comparisons than Shell Sort but more than Merge Sort, particularly as the input size increases.

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Merge Sort SWAPS | Shell Sort SWAPS | Quick Sort SWAPS |
| 10000 | - | 149876 | 84498 |
| 15000 | - | 261478 | 124755 |
| 20000 | - | 360419 | 186059 |
| 25000 | - | 499921 | 259670 |
| 30000 | - | 628943 | 299959 |
| 35000 | - | 690538 | 304097 |
| 40000 | - | 899984 | 411197 |
| 45000 | - | 954968 | 494889 |
| 50000 | - | 1251972 | 459659 |

For SWAPS:

* Merge Sort does not perform swaps, relying on copying during its merge phase.
* Shell Sort involves significantly more swaps than both Merge Sort and Quick Sort, especially as the input size grows.
* Quick Sort performs fewer swaps than Shell Sort, making it more efficient in terms of swapping, particularly for larger datasets.