1. W moim programie zostały zaimplementowane następujące algorytmy:
2. Quick Sort – Jego średnia złożoność powinna wynosić O(n\*log(n)) , a pesymistycznie O). Z tablicy przekazanej do algorytmu wybiera się element, który dzieli ją na dwie podtablice. Do pierwszej przenoszone są wszystkie elementy nie większe od wybranego, a do tej drugiej wszystkie większe. Następnie wywołuje się te samą funkcję na pierwszej połowie jak i na drugiej sortując je aż do momentu, gdy kolejny fragment podziału zawiera jeden element.
3. Merge Sort – Jest to rekurencyjny algorytm sortowania, którego złożoność wynosi O(n\*log(n)). Algorytm działa tak, że dzielimy tablice na dwie równe części, na których stosuje się merge sort, chyba że został pojedynczy element w tablicy. Następnie łączy się podciągi w ostateczny posortowany ciąg.
4. Comb Sort – Kolejny algorytm o złożoności O(n\*log(n)) =. Działa podobnie do Bubble Sort, z tym wyjątkiem, że porównuje ze sobą elementy różniące się od siebie nie o co jeden, ale o tzw. gap, który jest początkowo długością tablicy a potem zmniejszany o 1.3 odrzucając część ułamkową. Gap jest dzielony przez 1.3 aby uzyskać najszybszą złożoność obliczeniową. Jego pesymistyczna złożoność wynosi O() .
5. Sposób generowania ciągów:

W metodzie main tworzymy tablice intów, następnie użytkownik wybiera jaką metodą chce posortować ciąg, jaki rodzaj posortowania będzie przy jego generowaniu oraz długość ciągu używanego do badania.

Ciągi są generowane za pomocą funkcji createRandom, która tworzy nam tablice intów o długości wybranej przez użytkownika, a następnie do każdego miejsca w tablicy przypisuje wartość losową z zakresu <0, 1000000> przy pomocy obiektu klasy Random.

Jeśli użytkownik wybrał aby ciąg był posortowany od połowy to najpierw pętla przechodzi przez połowę tablicy i każdemu elementowi przypisuje poprzedni powiększony o losową wartość z zakresu <0, 1000000>,a następnie generuje losowe liczby dla pozostałej liczby miejsc w ciągu. Jeśli została wybrana opcja w pełni posortowanego ciągu to wykonuje się tylko pierwsza część funkcji createHalfSorted, tyle że na całej długości tablicy. Funkcja createReverselySorted działa tak jak createFullySorted, z tym wyjątkiem, że idzie od końca tablicy i każdy poprzedni element jest różnicą następnego oraz losowej wartości z zakresu

<0, 1000000>.

Następnie w pętli for, która generuje nam 101 różnych ciągów o tej samej specyfikacji (w analizie odrzucamy pierwszy wynik ze względu na JVM, której zajmuje chwilkę na skompilowanie kodu co powoduje wydłużony czas dla pierwszego sortowania) sortujemy te tablice i za pomocą funkcji System.currentTimeMillis() otrzymujemy czas przed sortowaniem oraz zaraz po jego zakończeniu co pozwala nam odczytać czas w milisekundach potrzebny na posortowanie danego ciągu przez algorytm, dla każdego ze 100 przypadków.

System jest obciążony przez różne procesy, które wymagają, aby w ogóle mógł działać, jednak każdy inny proces mogący spowolniać działanie procesora został wyłączony, z wyjątkiem środowiska programowania, bez którego oczywiście kod się nie skompiluje, które natomiast zajmuje znaczną część RAM-u co może powodować opóźnienia w sortowaniu.

1. Następnie dla każdego ciągu-sposobu posortowania wyniki zostały wrzucone do Excela, gdzie znajdują się tabele z danymi oraz wykresy z wynikami. Plik załączam razem z programem i README.txt.

Patrząc na wyniki porównamy średni czas sortowania dla każdego z algorytmów

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Randomly Sorted | Randomly Sorted | Randomly Sorted | Randomly Sorted |
| Algorytm | Quick Sort | Quick Sort | Quick Sort | Quick Sort |
| Czas średni | 5.9 | 32.86 | 68.01 | 144.49 |
| Odchylenie | 0.3 | 0.49 | 0.61 | 4.87 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Half Sorted | Half Sorted | Half Sorted | Half Sorted |
| Algorytm | Quick Sort | Quick Sort | Quick Sort | Quick Sort |
| Czas średni | 4.98 | 28.6 | 59.27 | 126.55 |
| Odchylenie | 0.35 | 2.65 | 1.46 | 4.67 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Fully Sorted | Fully Sorted | Fully Sorted | Fully Sorted |
| Algorytm | Quick Sort | Quick Sort | Quick Sort | Quick Sort |
| Czas średni | 3.99 | 23.64 | 50.08 | 106.79 |
| Odchylenie | 0.26 | 1.76 | 3.41 | 4.56 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Revesely Sorted | Revesely Sorted | Revesely Sorted | Revesely Sorted |
| Algorytm | Quick Sort | Quick Sort | Quick Sort | Quick Sort |
| Czas średni | 4 | 24.26 | 50.25 | 106.13 |
| Odchylenie | 0.2 | 5.04 | 0.95 | 4.42 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Randomly Sorted | Randomly Sorted | Randomly Sorted | Randomly Sorted |
| Algorytm | Merge Sort | Merge Sort | Merge Sort | Merge Sort |
| Czas średni | 10.34 | 59.45 | 124.36 | 256.86 |
| Odchylenie | 0.57 | 0.82 | 1.14 | 4.96 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Half Sorted | Half Sorted | Half Sorted | Half Sorted |
| Algorytm | Merge Sort | Merge Sort | Merge Sort | Merge Sort |
| Czas średni | 7.89 | 47.03 | 98.45 | 207.72 |
| Odchylenie | 0.55 | 0.59 | 0.88 | 14.58 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Fully Sorted | Fully Sorted | Fully Sorted | Fully Sorted |
| Algorytm | Merge Sort | Merge Sort | Merge Sort | Merge Sort |
| Czas średni | 6.09 | 37.26 | 79.52 | 166.66 |
| Odchylenie | 0.38 | 0.61 | 0.96 | 4.17 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Revesely Sorted | Revesely Sorted | Revesely Sorted | Revesely Sorted |
| Algorytm | Merge Sort | Merge Sort | Merge Sort | Merge Sort |
| Czas średni | 5.94 | 37.65 | 80.71 | 166.26 |
| Odchylenie | 0.44 | 0.92 | 2.97 | 4.48 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Randomly Sorted | Randomly Sorted | Randomly Sorted | Randomly Sorted |
| Algorytm | Comb Sort | Comb Sort | Comb Sort | Comb Sort |
| Czas średni | 8.39 | 49.81 | 103.12 | 249.36 |
| Odchylenie | 0.53 | 1.32 | 13.2 | 147.25 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Half Sorted | Half Sorted | Half Sorted | Half Sorted |
| Algorytm | Comb Sort | Comb Sort | Comb Sort | Comb Sort |
| Czas średni | 6.7 | 41.85 | 87.12 | 205.23 |
| Odchylenie | 0.48 | 0.62 | 3.4 | 6.55 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Fully Sorted | Fully Sorted | Fully Sorted | Fully Sorted |
| Algorytm | Comb Sort | Comb Sort | Comb Sort | Comb Sort |
| Czas średni | 5.48 | 36.35 | 76.82 | 184.17 |
| Odchylenie | 0.56 | 0.93 | 6.95 | 13.42 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Długość | 100 000 | 500 000 | 1 000 000 | 2 000 000 |
| Typ sortowania | Revesely Sorted | Revesely Sorted | Revesely Sorted | Revesely Sorted |
| Algorytm | Comb Sort | Comb Sort | Comb Sort | Comb Sort |
| Czas średni | 5.46 | 36.22 | 75.41 | 184.54 |
| Odchylenie | 0.56 | 0.58 | 1.36 | 14.56 |

Z wykresów możemy zauważyć, że niektóre wyniki znacznie odstają od średnich. Wyniki niższe mogą być spowodowane na przykład presortem ciągu w taki sposób, że algorytm wykona mniej kroków niż jego złożoność. Natomiast te wyższe są spowodowane niekorzystnym sortem ciągu w momencie jego utworzenia albo/oraz chwilowym skokiem przeciążenia systemu.

1. Podsumowanie

Quick sort – Patrząc na wszystkie algorytmy quicksort działa najszybciej. Dzięki ustawieniu pivotu w pozycji losowej program nie ma problemu z błędem Stack Overflow, który jest spowodowany tworzeniem zbyt dużej ilości ramek w rekurencyjnym sposobie zaimplementowania kodu. Można było to obejść implementując sortowanie szybkie algorytmem iteracyjnym, który ma taką samą złożoność czasową jak rekurencyjny, ale mniejszą złożoność pamięciową. Patrząc na wyniki można stwierdzić, że quick sort działa szybciej jeśli ciąg jest albo w pełni albo odwrotnie posortowany. Jeśli jest ustawiony randomowo albo w połowie czas ten zmienia się praktycznie na połowę większy w przypadku długich ciągów liczbowych. Duże znaczenie ma tu ustawienie pivota, który ustawiony na końcu zaburza pracę algorytmu dla w pełni posortowanego a na srodku ciągu- dla posortowanego w połowie. Dzieje się tak dlatego, że program wykona bardzo dużo wywołań samej siebie co spowoduje przeciąża JVM powodując wywołanie błędu nadmiaru stacku.

Merge Sort- analizując wyniki stwierdzamy, że czas wykonania algortymu drastycznie się zmniejsza wraz z większym zakresem posortowania ciągu startowego. Przy w pełni posortowanym oraz odwrotnie posortowanym a nawet w połowie posortowanym (tylko dla wartości 2mln +) merge sort jest drugim najszybszym algorytmem z pośród zaimplementowanych. Merge sort sortuje tablice w pełni posortowaną oraz odwrotnie szybciej niż comb sort dlatego że nie wymaga tylu sprawdzeń dla już posortowanego ciągu a w przypadku posortowanego odwrotnie tylko zamienia miejscami każdy z elementów.

Comb Sort- ulepszony algorytm bubble sort, który dla ranodmowo oraz w połowie posrotowanej tablicy jest drugim najszybszym z wybranych. Porównuje on tylko te, które są oddalone od siebie o gap, który zmniejsza się o 1.3 co cykl w ten sposób następuje optymalizacja algorytmu.

Najszybszym algorytmem okazał się Quick Sort. Wszystkie algorytmy przypominają wykres funkcji n\*log(n), która jest jednocześnie złożonością każdej z powyższych metod sortowania. Jedyny problem pojawia się w poszczególnych wynikach które znacznie odbiegają od standardowych. Dzieje się tak z różnych powodów, ale głównymi są przeciążenia systemu, jego chwilowe zawieszenia oraz ciąg posortowany w sposób taki, który wydłuża działanie algorytmu poprzez wymuszenie większej ilości porównań razem z zamianami poszczególnych wartości ciągów.