ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сравнение алгоритмов сортировки»

Выполнила работу:

Таволжанская Полина

Академическая группа №C3100

Санкт-Петербург

2024

1. ВВЕДЕНИЕ

Цель работы: анализ трех алгоритмов сортировки разной сложности:

- Gnome Sort (сортировка сравнением)

- Heapsort (пирамидальная сортировка)

- Radix Sort (разрядная сортировка)

Задачи:

- Проанализировать логику работы каждого алгоритма, произвести и проверить предположения о худших, средних и лучших случаях входных данных для массива

- Проверить работу алгоритмов на различных данных, в том числе больших (<1000000 элементов)

- Найти асимптотическую и пространственную сложность алгоритмов сортировки

- Сделать выводы о каждом алгоритме исходя из полученных данных

2. ТЕОРЕТИЧЕСКАЯ ПОДГОТОВКА

В данной работе рассматриваются три алгоритма сортировки: **гномья сортировка**, **пирамидальная сортировка** и **поразрядная сортировка**. Далее: описание и принцип работы функций.

**Гномья сортировка** — это простой алгоритм сортировки, который работает по принципу последовательного сравнения соседних элементов. Алгоритм начинает с первого элемента и сравнивает его со следующим. Если текущий элемент меньше предыдущего, они меняются местами, и алгоритм возвращается на один шаг назад. Если текущий элемент больше или равен предыдущему, алгоритм переходит к следующему элементу. Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован. Алгоритм напоминает поведение гнома, который идет вперед, пока не встретит препятствие (элемент, который меньше предыдущего), после чего он возвращается назад, чтобы исправить порядок.

**Пирамидальная сортировка** основана на структуре данных "куча" (heap). Алгоритм сначала строит максимальную кучу из элементов массива, а затем последовательно извлекает максимальный элемент (корень кучи) и помещает его в конец отсортированного массива. После извлечения максимального элемента куча восстанавливается для оставшихся элементов. Структура кучи позволяет эффективно находить максимальный элемент и поддерживать порядок элементов при их извлечении.

**Поразрядная сортировка** — это алгоритм, который сортирует числа по отдельным разрядам, начиная с младшего разряда. Он использует метод сортировки подсчетом (Counting Sort) для упорядочивания элементов по каждому разряду. Алгоритм эффективен для целых чисел с фиксированной длиной и может достигать линейной временной сложности.

3. РЕАЛИЗАЦИЯ

1. Изучение алгоритмов сортировки

Был проведён анализ всех алгоритмов сортировки, исходя из логики их работы были сделаны предположения о худших, средних и лучших случаях входных наборов для каждого алгоритма.

2. Экспериментальная часть (подробно в отдельном блоке)

Для реализации алгоритмов использовался язык программирования C++.

3. Для каждого алгоритма сортировки был произведён подсчёт пространственной и временной сложности для худшего и лучшего случаев.

4. Полученные в ходе экспериментальной части результаты были использованы для составления графиков работы алгоритмов при различных входных данных.

4. ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

Для оценки производительности алгоритмов сортировки были проанализированы асимптотические характеристики каждого алгоритма, а также исследованы затраты памяти и время выполнения на различных наборах данных, размер которых варьировался от 1 тысячи до 1 миллиона элементов с шагом в 1 тысячу.

**Гномья сортировка**

**Временная сложность**: cредняя и худшая сложность: O(n^2), так как алгоритм требует множества сравнений и перестановок для достижения отсортированного состояния. Это связано с тем, что в худшем случае алгоритм может пройти по массиву несколько раз, возвращаясь назад при каждом обмене.  
Лучшая сложность: O(n), если массив уже отсортирован или почти отсортирован, так как в этом случае потребуется всего один проход по массиву без лишних перестановок.

**Пирамидальная сортировка**

**Временная сложность**: cредняя и худшая сложность: **O(n log n)**, поскольку алгоритм строит максимальную кучу и затем последовательно извлекает элементы. Строительство кучи занимает **O(n)**, а извлечение каждого элемента требует **O(log n)**. Лучшая сложность также составляет **O(n log n)**, так как независимо от начального порядка элементов алгоритм всегда проходит через этапы построения кучи и извлечения.

**Поразрядная сортировка**

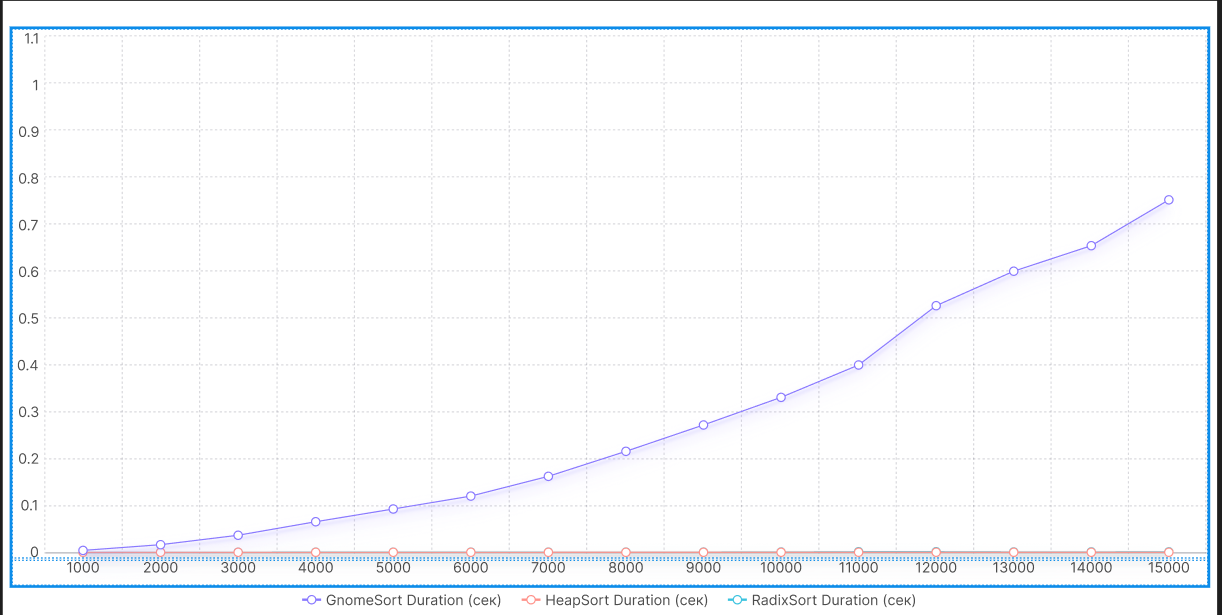
**Временная сложность**: Средняя и худшая сложность: O(nk), где n — количество элементов массива, а k — количество разрядов в наибольшем числе. Алгоритм использует сортировку подсчетом для каждого разряда, что делает его эффективным при работе с целыми числами с фиксированной длиной.  
Лучшая сложность также составляет O(nk), так как даже в оптимальных условиях алгоритм все равно проходит через все разряды для сортировки элементов.

Таким образом, каждый из рассмотренных алгоритмов имеет свои особенности в отношении временной сложности, что влияет на их производительность в зависимости от структуры входных данных.

Таблица 1: Результаты времени работы алгоритмов на сложных для них массивах

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | GnomeSort (сек) | HeapSort (сек) | RadixSort (сек) |
| 1000 | 0.00403659 | 0.000010404 | 0.000089029 |
| 2000 | 0.0162458 | 0.000020173 | 0.000166761 |
| Array Size | GnomeSort (сек) | HeapSort (сек) | RadixSort (сек) |
| 3000 | 0.0364122 | 0.000030931 | 0.000273277 |
| 4000 | 0.065323 | 0.000035363 | 0.000338758 |
| 5000 | 0.092534 | 0.000024516 | 0.000303390 |
| 6000 | 0.120107 | 0.000028674 | 0.000357807 |
| 7000 | 0.162340 | 0.000034340 | 0.000411978 |
| 8000 | 0.215792 | 0.000038388 | 0.000487639 |
| 9000 | 0.272095 | 0.000043413 | 0.000557184 |
| 10000 | 0.331143 | 0.000048723 | 0.000608787 |
| 11000 | 0.400307 | 0.000140038 | 0.00114461 |
| 12000 | 0.527084 | 0.000140038 | 0.00114461 |
| 13000 | 0.600747 | 0.000062954 | 0.000772399 |
| 14000 | 0.655299 | 0.000067446 | 0.000815512 |
| 15000 | 0.753239 | 0.000087897 | 0.000881271 |
| 16000 | 5.18799 | 0.0080384 | 0.0068614 |
| 17000 | 5.41843 | 0.0128971 | 0.0095876 |
| 18000 | 6.19189 | 0.0168601 | 0.0074744 |
| 19000 | 6.74565 | 0.0149649 | 0.0124898 |
| Array Size | GnomeSort (сек) | HeapSort (сек) | RadixSort (сек) |
| 20000 | 7.97881 | 0.0181203 | 0.0097861 |
| 21000 | 10.4015 | 0.0131492 | 0.0093488 |
| 22000 | 10.0973 | 0.0190007 | 0.0171048 |
| 23000 | 11.9039 | 0.0188221 | 0.0151277 |
| 24000 | 12.2875 | 0.0128936 | 0.010931 |
| 25000 | 14.6112 | .0194518 | .015507 |
| 26000 | 13..5288 | - .0169856 | - .0148418 |
| 27000 | -16..7306 | - .0207039 | - .0146494 |
| 28000 | -17..623 | - .0250031 | - .0173873 |
| 29000 | -18..5506 | - .0476258 | - .0250475 |
| 30000 | -19..7859 | - .0251645 | - .0217522 |
| 31000 | -18..0714 | - .0198479 | - .0162681 |
| 32000 | -18..4731 | - .0278916 | - .0156234 |
| 33000 | -20..6878 | - .0266554 | - .0267339 |
| 34000 | -21..8176 | - .0383685 | - .0249828 |
| 35000 | -23..5657 | - .0235335 | - .0170228 |
| 36000 | -27..2926 | - .0241677 | - .0155966 |
| Array Size | GnomeSort (сек) | HeapSort (сек) | RadixSort (сек) |
| 37000 | -26..3929 | - .0294598 | - .019737 |
| 38000 | -26..9611 | - .0228633 | - .0193306 |
| 39000 | -28..3528 | - .0251332 | - .0156218 |
| 40000 | -29..9961 | - .0266567 | - .0222287 |

Рисунок 1. График, демонстрирующий зависимость времени обработки массива от его размера.



**Стабильность**. Для проверки стабильности работы алгоритмов сортировки было проведено 100 запусков каждого алгоритма на одном и том же наборе данных размеров в 10000 и 100000 элементов:

GnomeSort не был использован в этом анализе так как он является самым худшим в тестировании на больших наборах данных.

| Алгоритм сортировки | Размер | Q1 | Median | Q3 | Mean | IQR | Min | Max |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Heap Sort | 10000 | 44.066 | 45.1837 | 46.2316 | 44.9333 | 2.1656 | 41.709 | 49.1157 |
| Radix Sort | 10000 | 39.2964 | 39.5098 | 41.129 | 40.1744 | 1.8326 | 37.0793 | 47.3649 |

Таблица 2. Данные полученные из 100 запусков двух алгоритмов сортировок на отсортированных в обратном порядке массивах (время указано в 10-5 сек)

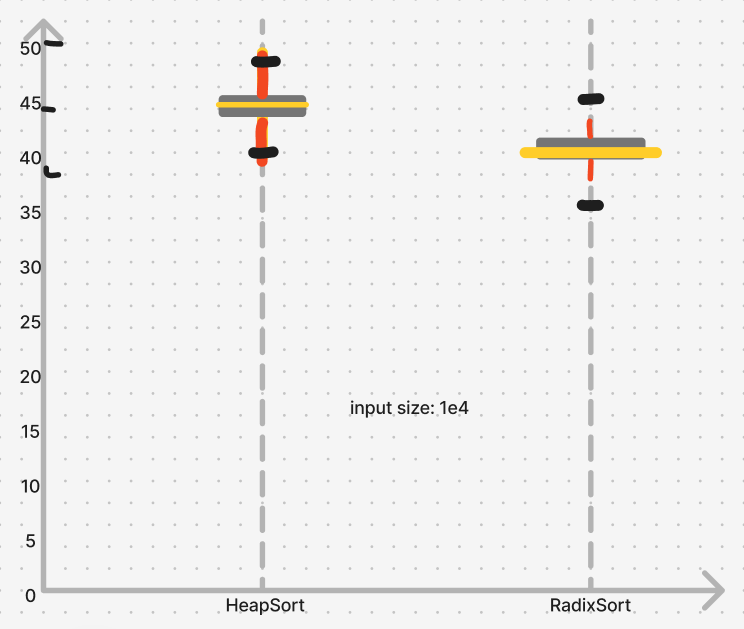


Рис. 2 График «ящик с усами» времени выполнения двух алгоритмов сортировки на наборах данных размера 10000 элементов

| **Алгоритм сортировки** | **Размер** | **Q1** | **Median** | **Q3** | **Mean** | **IQR** | **Min** | **Max** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Heap Sort | 100000 | 58.1521 | 58.5451 | 59.3353 | 63.6375 | 1.18314 | 57.2906 | 102.428 |
| Radix Sort | 100000 | 49.4551 | 49.8420 | 50.2527 | 53.9528 | 0.79959 | 47.6201 | 92.3338 |

Таблица 3. Данные полученные из 100 запусков двух алгоритмов сортировок на отсортированных в обратном порядке массивах (время указано в 10-5 сек)

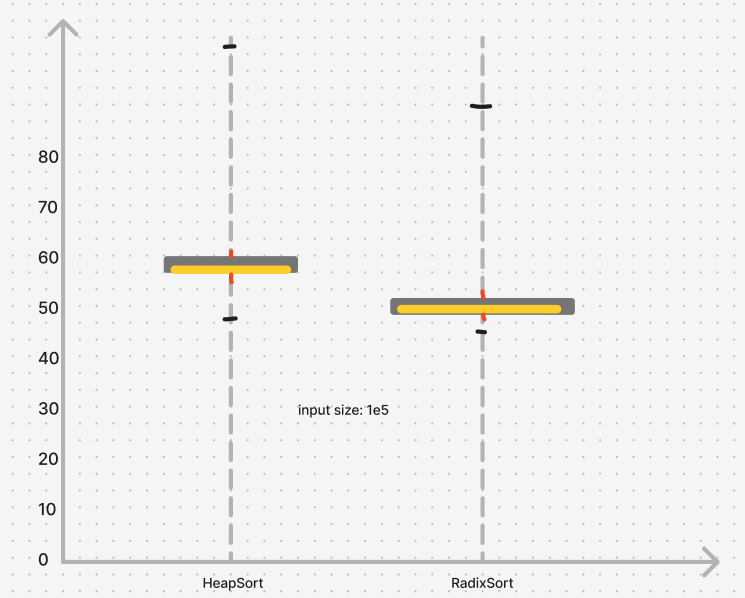


Рис. 3 График «ящик с усами» времени выполнения двух алгоритмов сортировки на наборах данных размера 100000 элементов

Исходя из результатов запусков программ можно сделать вывод, что алгоритм сортировки HeapSort менее устойчив нежели алгоритм RadixSort в силу меньшего разброса (длины усов ящика), хотя оба алгоритма имеют выбросы по времени, далёкие от стабильных показателей. Выбросы для RadixSort: массивы с числами очень больших и наоборот малых размеров (малого кол-ва разрядов). Выбросы для HeapSort могут быть связаны с излишним использованием памяти, а также с неэффективностью алгоритма сортировки на почти отсортированных массивах. Можно предположить, что выбросы для GnomeSort возникнут при обработке алгоритмом как полностью (почти) отсортированных, так и отсортированных в обратном порядке массивах в силу количества необходимых перестановок.

5. ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были изучены и реализованы три алгоритма сортировки: гномья сортировка, пирамидальная сортировка и поразрядная сортировка.

Анализируя результаты, можно сделать следующие выводы. Гномья сортировка демонстрирует низкую производительность из-за своей временной сложности O(N^2). Это приводит к значительному увеличению времени выполнения на больших массивах. Алгоритм показал себя неэффективным для массивов крупных размеров, что ставит перспективу его применения в реальных задачах под сомнение. Использовать Gnome Sort можно на массивах почти полностью отсортированных, когда заранее известно что количество перестановок будет небольшим.

Пирамидальная сортировка (Heapsort) обладает хорошей производительностью с временной сложностью O(N log N). Она подходит для работы с большими объемами данных и обеспечивает стабильное время выполнения. Однако использование дополнительной памяти для создания кучи может быть ограничением в некоторых случаях. Также Heapsort непредпочтительна к использованию на массивах, где количество перестановок до отсортированного массива мало, так как в любом случае перераспределяет все элементы.

Поразрядная сортировка (Radix Sort) демонстрирует неплохие результаты при работе с массивами целых чисел с небольшим диапазоном значений. С её линейной временной сложностью O(Nk) алгоритм эффективно обрабатывает большие объемы данных, но требует значительных ресурсов памяти для хранения промежуточных результатов.

Таким образом, гномья сортировка не рекомендуется для практического использования из-за своей низкой эффективности. Пирамидальная сортировка подходит для универсального применения, особенно на больших массивах, однако не является стабильной. Поразрядная сортировка является предпочтительным выбором для массивов с ограниченным диапазоном значений и большими объемами данных.

**Приложение**

**Полный набор результатов работы трёх алгоритмов при количестве входных от 1000 до 1e6:**

|  |  |  |  |
| --- | --- | --- | --- |
| 1000 | 0.0207883 | 0.0004693 | 0.0005527 |
| 2000 | 0.12696 | 0.0011443 | 0.0015105 |
| 3000 | 0.28743 | 0.0016906 | 0.0365139 |
| 4000 | 0.493237 | 0.0241769 | 0.0021141 |
| 5000 | 0.757329 | 0.0056294 | 0.0177956 |
| 6000 | 1.63777 | 0.0081154 | 0.003966 |
| 7000 | 2.06429 | 0.0072413 | 0.0084148 |
| 8000 | 1.94887 | 0.0167912 | 0.013443 |
| 9000 | 2.08532 | 0.0081829 | 0.0048919 |
| 10000 | 2.53861 | 0.0078416 | 0.0061861 |
| 11000 | 3.42503 | 0.0103492 | 0.0065572 |
| 12000 | 2.7497 | 0.0072385 | 0.0056789 |
| 13000 | 3.99052 | 0.011081 | 0.0106531 |
| 14000 | 4.68424 | 0.0095489 | 0.0078273 |
| 15000 | 3.98794 | 0.012107 | 0.008663 |
| 16000 | 5.18799 | 0.0080384 | 0.0068614 |
| 17000 | 5.41843 | 0.0128971 | 0.0095876 |
| 18000 | 6.19189 | 0.0168601 | 0.0074744 |
| 19000 | 6.74565 | 0.0149649 | 0.0124898 |
| 20000 | 7.97881 | 0.0181203 | 0.0097861 |
| 21000 | 10.4015 | 0.0131492 | 0.0093488 |
| 22000 | 10.0973 | 0.0190007 | 0.0171048 |
| 23000 | 11.9039 | 0.0188221 | 0.0151277 |
| 24000 | 12.2875 | 0.0128936 | 0.010931 |
| 25000 | 14.6112 | 0.0194518 | 0.015507 |
| 26000 | 13.5288 | 0.0169856 | 0.0148418 |
| 27000 | 16.7306 | 0.0207039 | 0.0146494 |
| 28000 | 17.623 | 0.0250031 | 0.0173873 |
| 29000 | 18.5506 | 0.0476258 | 0.0250475 |
| 30000 | 19.7859 | 0.0251645 | 0.0217522 |
| 31000 | 18.0714 | 0.0198479 | 0.0162681 |
| 32000 | 18.4731 | 0.0278916 | 0.0156234 |
| 33000 | 20.6878 | 0.0266554 | 0.0267339 |
| 34000 | 21.8176 | 0.0383685 | 0.0249828 |
| 35000 | 23.5657 | 0.0235335 | 0.0170228 |
| 36000 | 27.2926 | 0.0241677 | 0.0155966 |
| 37000 | 26.3929 | 0.0294598 | 0.019737 |
| 38000 | 26.9611 | 0.0228633 | 0.0193306 |
| 39000 | 28.3528 | 0.0251332 | 0.0156218 |
| 40000 | 29.9961 | 0.0266567 | 0.0222287 |
| 41000 | 41.8355 | 0.031234 | 0.0223103 |
| 42000 | 40.4271 | 0.0322616 | 0.0169928 |
| 43000 | 42.1248 | 0.0343688 | 0.0275948 |
| 44000 | 37.0428 | 0.0329464 | 0.019117 |
| 45000 | 38.6336 | 0.0346867 | 0.020633 |
| 46000 | 39.7639 | 0.0346476 | 0.0179052 |
| 47000 | 41.4139 | 0.0257693 | 0.0225558 |
| 48000 | 43.398 | 0.0481883 | 0.0335197 |
| 49000 | 45.5132 | 0.027184 | 0.0287628 |
| 50000 | 46.1125 | 0.0386551 | 0.0194384 |
| 51000 | 49.3338 | 0.0500448 | 0.0337493 |
| 52000 | 52.9849 | 0.0426862 | 0.0290015 |
| 53000 | 52.6711 | 0.0404981 | 0.029629 |
| 54000 | 55.0175 | 0.044435 | 0.0299191 |
| 55000 | 56.3532 | 0.0335421 | 0.0266699 |
| 56000 | 57.5732 | 0.0357207 | 0.0287305 |
| 57000 | 61.032 | 0.0582873 | 0.0422581 |
| 58000 | 62.7115 | 0.0437642 | 0.0339077 |
| 59000 | 65.1063 | 0.0552995 | 0.0362092 |
| 60000 | 66.2152 | 0.0413335 | 0.0252544 |
| 61000 | 68.4994 | 0.0486581 | 0.0306745 |
| 62000 | 70.9988 | 0.0489655 | 0.0381537 |
| 63000 | 74.6809 | 0.038004 | 0.0257895 |
| 64000 | 76.2603 | 0.040475 | 0.0341256 |
| 65000 | 77.3744 | 0.0508422 | 0.0323345 |
| 66000 | 81.025 | 0.045706 | 0.0280086 |
| 67000 | 83.2899 | 0.0534551 | 0.0351026 |
| 68000 | 85.9318 | 0.0493262 | 0.0389213 |
| 69000 | 98.4631 | 0.0613015 | 0.0398667 |
| 70000 | 113.175 | 0.0494963 | 0.0334762 |
| 71000 | 93.4147 | 0.061678 | 0.0462653 |
| 72000 | 96.9554 | 0.0579896 | 0.0358933 |
| 73000 | 120.062 | 0.0582977 | 0.0347074 |
| 74000 | 110.743 | 0.0584254 | 0.0380995 |
| 75000 | 103.542 | 0.0665579 | 0.0404884 |
| 76000 | 107.131 | 0.0939371 | 0.0501148 |
| 77000 | 123.884 | 0.0956656 | 0.0621456 |
| 78000 | 126.669 | 0.0712042 | 0.0420007 |
| 79000 | 158.357 | 0.0772212 | 0.0781886 |
| 80000 | 151.478 | 0.0757444 | 0.0790206 |
| 81000 | 178.102 | 0.0656753 | 0.0434423 |
| 82000 | 155.738 | 0.060385 | 0.0419661 |
| 83000 | 152.972 | 0.140232 | 0.148742 |
| 84000 | 171.046 | 0.0921842 | 0.0486491 |
| 85000 | 160.877 | 0.0909159 | 0.0624278 |
| 86000 | 193.169 | 0.0786728 | 0.0519715 |
| 87000 | 180.565 | 0.113116 | 0.0785917 |
| 88000 | 198.215 | 0.0808862 | 0.0548563 |
| 89000 | 151.965 | 0.0759744 | 0.050002 |
| 90000 | 183.251 | 0.0745864 | 0.047966 |
| 91000 | 206.963 | 0.0738863 | 0.0529326 |
| 92000 | 233.787 | 0.116868 | 0.0658374 |
| 93000 | 241.382 | 0.0972015 | 0.0641958 |
| 94000 | 164.537 | 0.0841749 | 0.0499901 |
| 95000 | 162.363 | 0.0757655 | 0.0509191 |
| 96000 | 168.829 | 0.0740042 | 0.0505448 |
| 97000 | 172.189 | 0.103552 | 0.056977 |
| 98000 | 176.804 | 0.0777646 | 0.0633541 |
| 99000 | 178.927 | 0.0796041 | 0.0510771 |
| 100000 | 180.431 | 0.087449 | 0.0617400 |