Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ  
СЕМЕСТРОВАЯ РАБОТА

по дисциплине «Алгоритмы и анализ сложности»  
«Экспериментальный анализ различных методов сортировки»

Обучающийся: Татаринцев Кирилл Владимирович гр. 09–331

(ФИО студента) (Группа)

Руководитель: к.ф.-м.н., доцент КСАИТ, А. В. Васильев

Казань – 2025

[Введение 3](#_Toc200139485)

[1. Постановка задачи 4](#_Toc200139486)

[2. Методика проведения эксперимента 5](#_Toc200139487)

[3. Полученные результаты 7](#_Toc200139488)

[Заключение 41](#_Toc200139489)

[Приложение 1. Программный код 43](#_Toc200139490)

Введение

Сортировки данных являются одними из самых фундаментальных операций в компьютерных науках, от эффективности которых зависит производительность многих алгоритмов и систем. А в условиях постоянно растущих объемов данных, выбор оптимального алгоритма становится критически важной задачей. И поэтому в данной работе главной задачей я ставлю найти самую быстрые сортировки под разные задачи, зависящие от объема данных и их типа. Так же выяснить насколько быстро работает встроенная сортировка c# и есть ли смысл прописывать свои сортировки для более быстрой обработки данных.

1. Постановка задачи
2. Исследование теоретических основ и практической применимости алгоритмов сортировки
3. Реализация и экспериментальная проверка алгоритмов сортировки
4. Разработка и применение методики комплексного тестирования
5. Анализ экспериментальных данных и их визуализация
6. Провести детальный анализ полученных экспериментальных данных
7. Выявить факторы, оказывающие существенное влияние на производительность сортировок (тип данных, их распределение, объем данных, особенности реализации)
8. Определить наиболее подходящие алгоритмы сортировки для различных типов данных и сценариев использования, исходя из их практической эффективности и стабильности
9. Методика проведения эксперимента

Сравнение алгоритмов сортировки проводилось на ноутбуке Honor MagicBook Pro 16 2024, работающем на Windows 11.

Характеристика ноутбука:

1. Процессор: Intel Core Ultra 7 155H (16 ядер, 22 потока, базовая скорость 3,8 ГГц)
2. Оперативная память: 32 гб LPDDR5X c частотой 6400 MГц
3. Видеокарта: NVIDIA GeForce RTX STUDIO 4060 Laptop на 18 гб видеопамяти + 8 Гб выделенной

Проект был разработан на языке программирования C# + Python для построения диаграмм с использованием matplotlib. Проект был создан в Visual Studio.

В рамках данного исследования будет проведено сравнение производительности 9 различных алгоритмов сортировки + две рекурсивные реализации – быстрой сортировки и сортировки слиянием, реализованных на языке C#.

Важно отметить, что все измерения производительности выполняются в условиях, характерных для управляемой среды .NET, включая работу сборщика мусора. Это может приводить к незначительным отклонениям в результатах, особенно при многократных запусках и работе с большими объёмами данных, поскольку сборщик мусора может активироваться в непредсказуемые моменты, влияя на время выполнения.

Сравнение алгоритмов будет производиться на массивах разной длины: [50, 5000, 25000, 50000, 250\_000. 500\_000]. Так же сравнение будет проводиться на разных типах данных с разной генерацией (только для int): int string, CustomDate, DateTime, double.

Разные генерации:

Массив случайных данных**:** Создаётся с помощью встроенной библиотеки System.Random. Для типов double, string и CustomDate используется только этот сценарий.

Массив отсортированных в обратном порядке данных**:** Создаётся путём генерации убывающей последовательности чисел. Этот случай считается худшим для простых сортировок

Массив с большим количеством дубликатов**:** Создаётся с использованием небольшого диапазона значений, что приводит к множеству повторяющихся элементов.

Массив частично упорядоченных данных**:** Создаётся из полностью отсортированного массива, в котором небольшая часть элементов (10%) случайным образом меняется местами. Это имитирует сценарий, где большинство данных уже находится на своих местах.

Массив частично перемешанных данных**:** Создается по принципу: сначала создается отсортированный массив, после чего 10% данных из этого массива достается и перемешиваются. В результате получается почти отсортированный массив, где небольшая часть данных стоит не на своих местах.

1. Полученные результаты

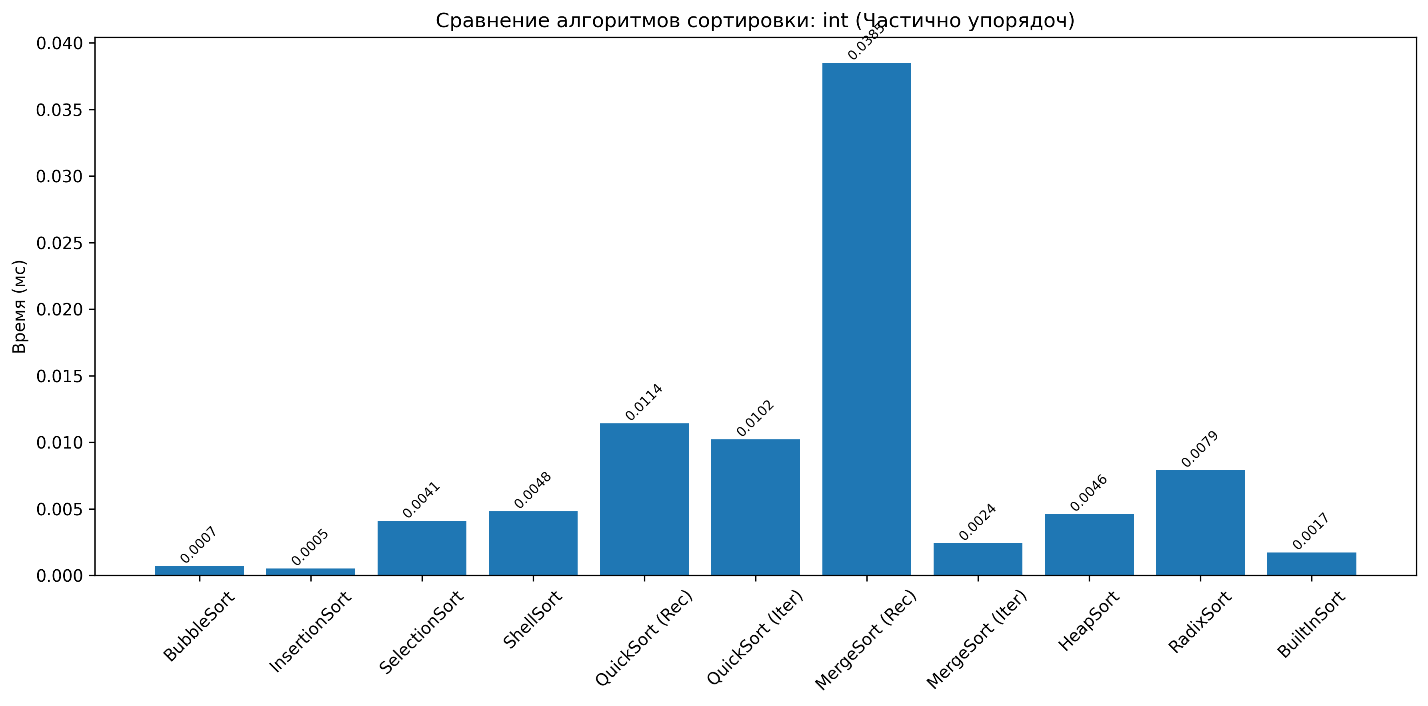
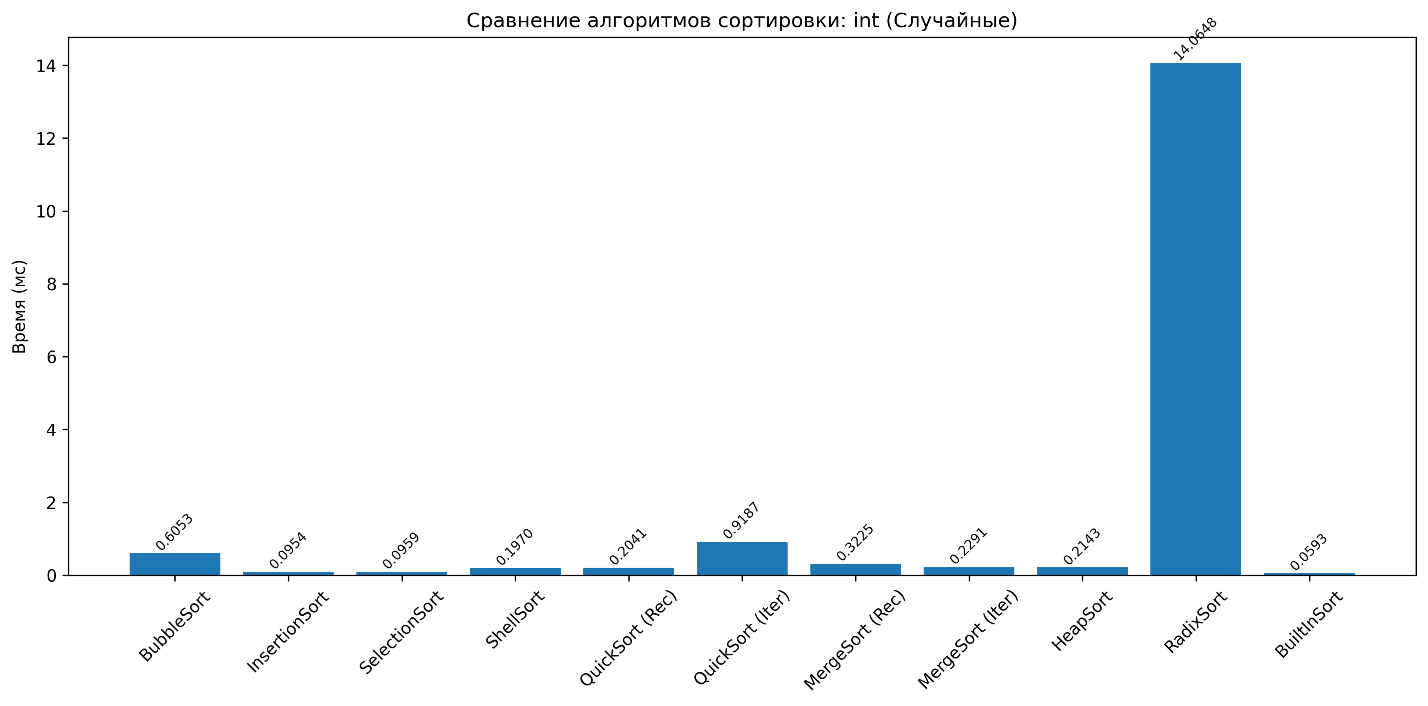
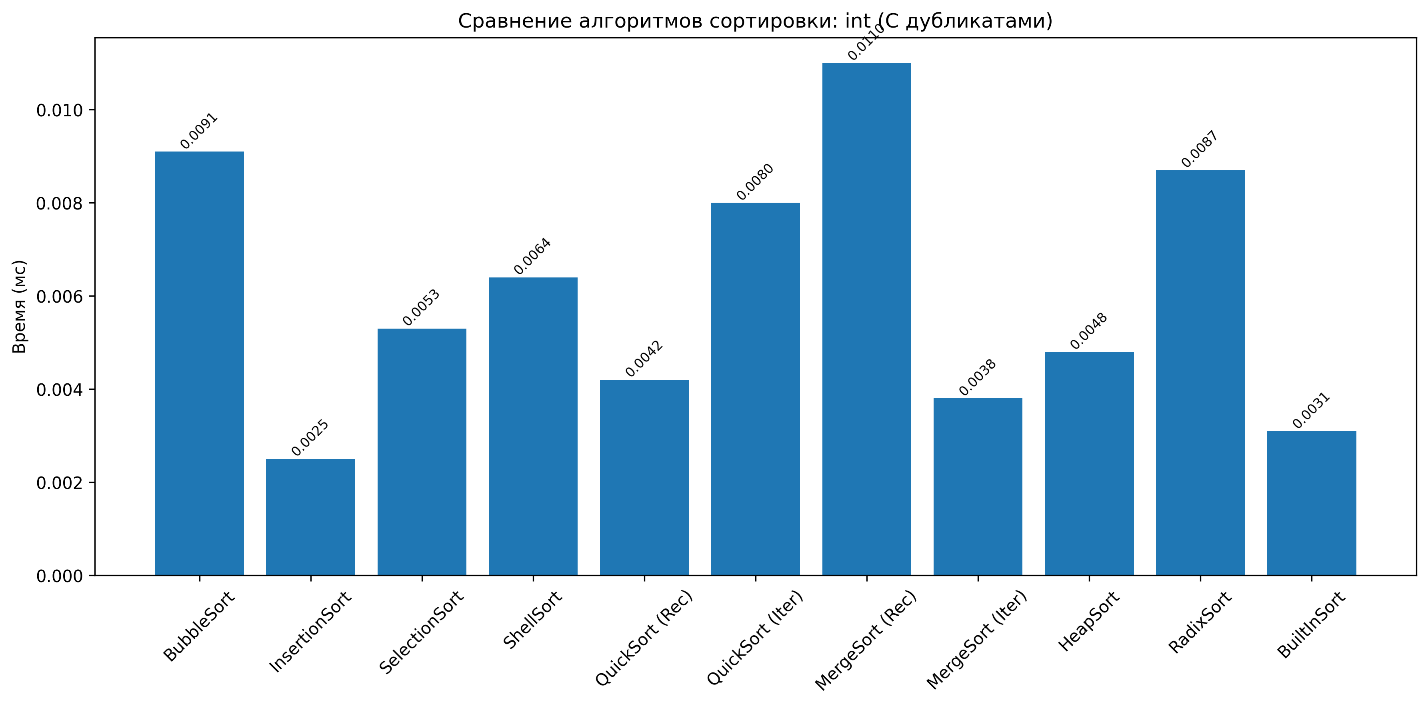
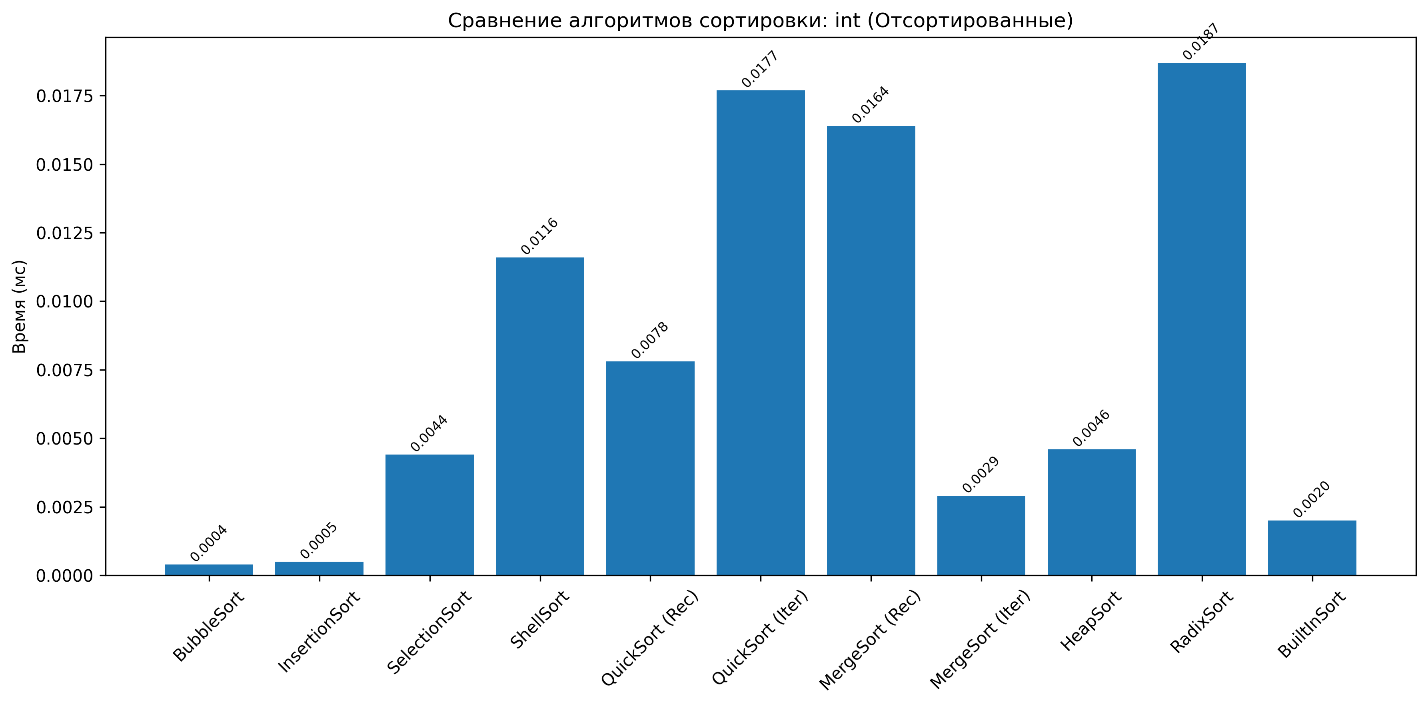
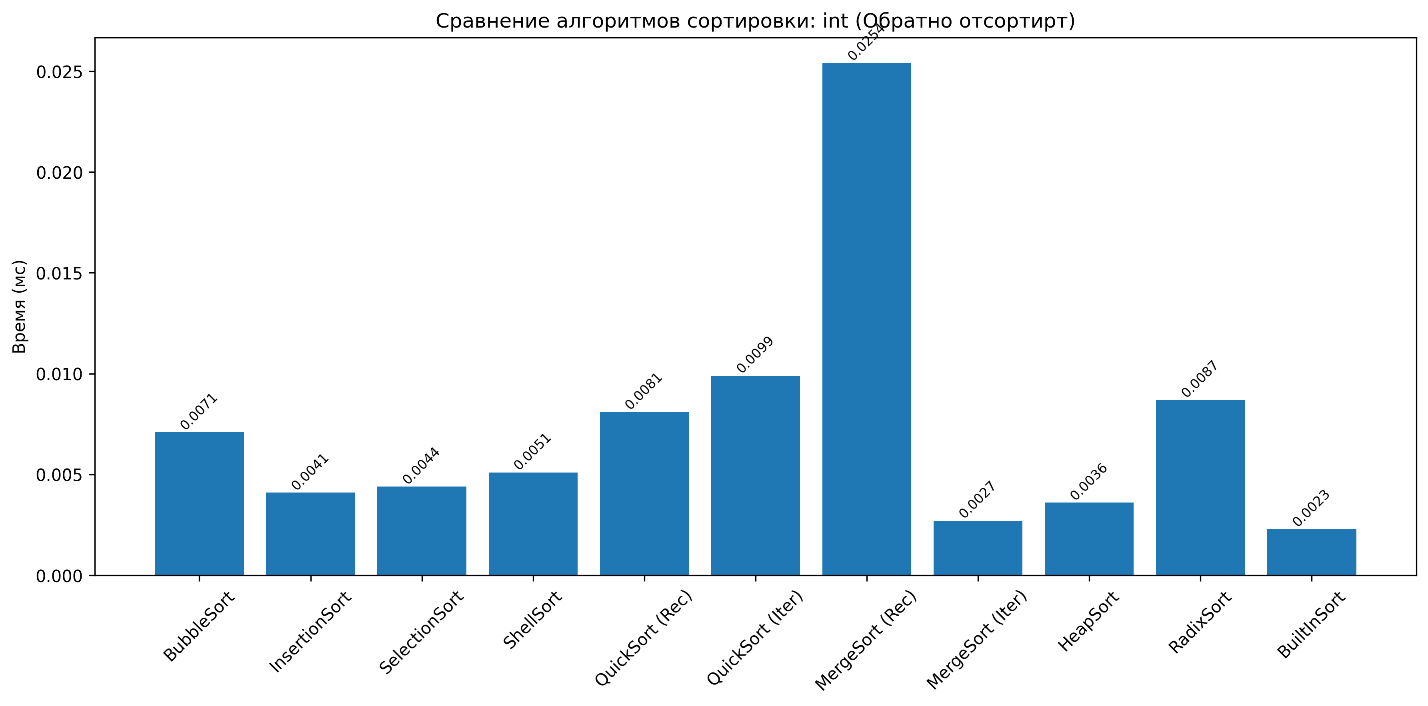
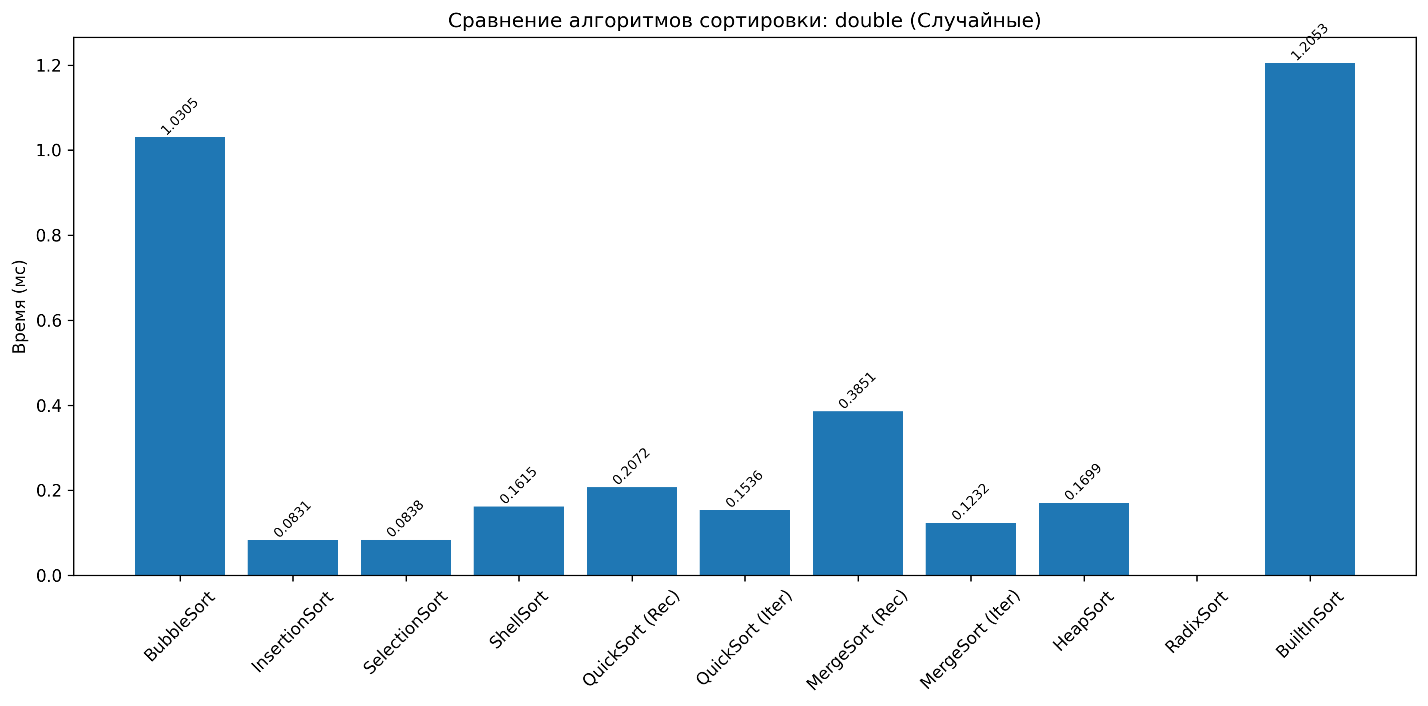
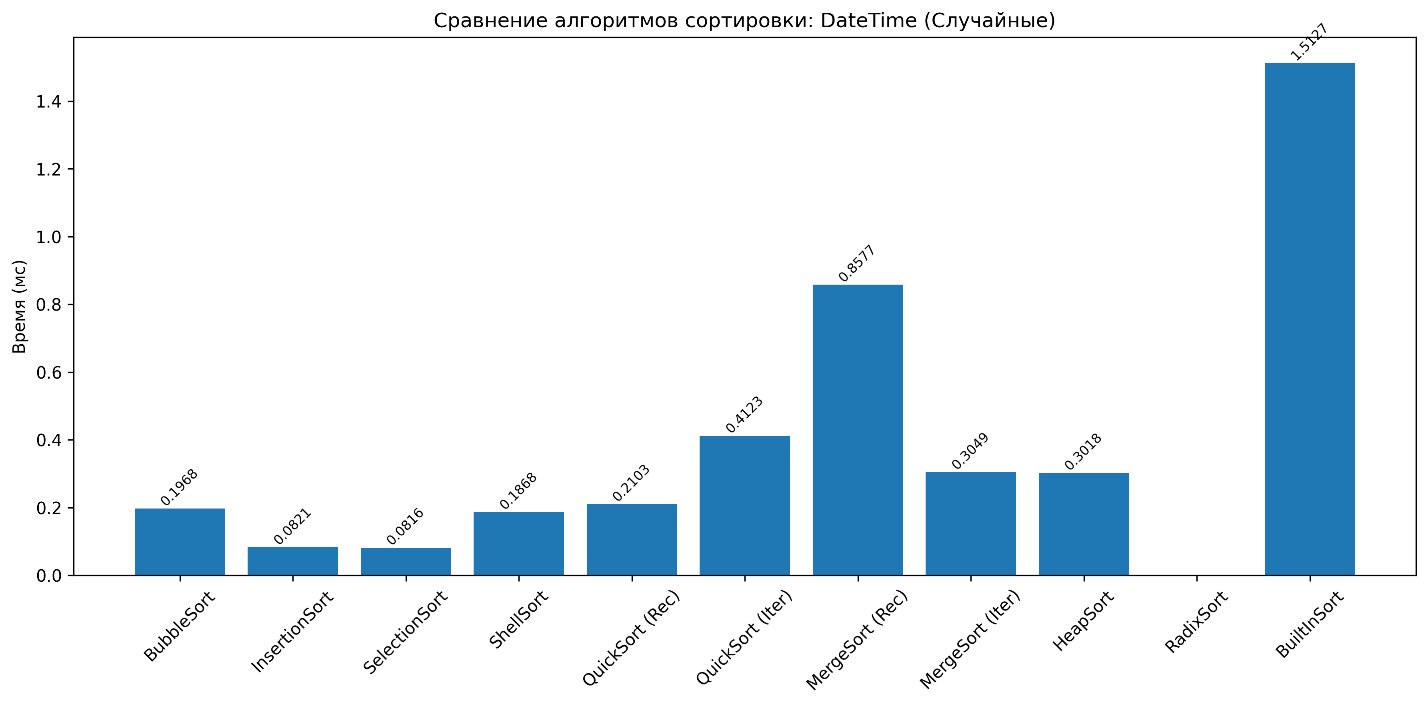
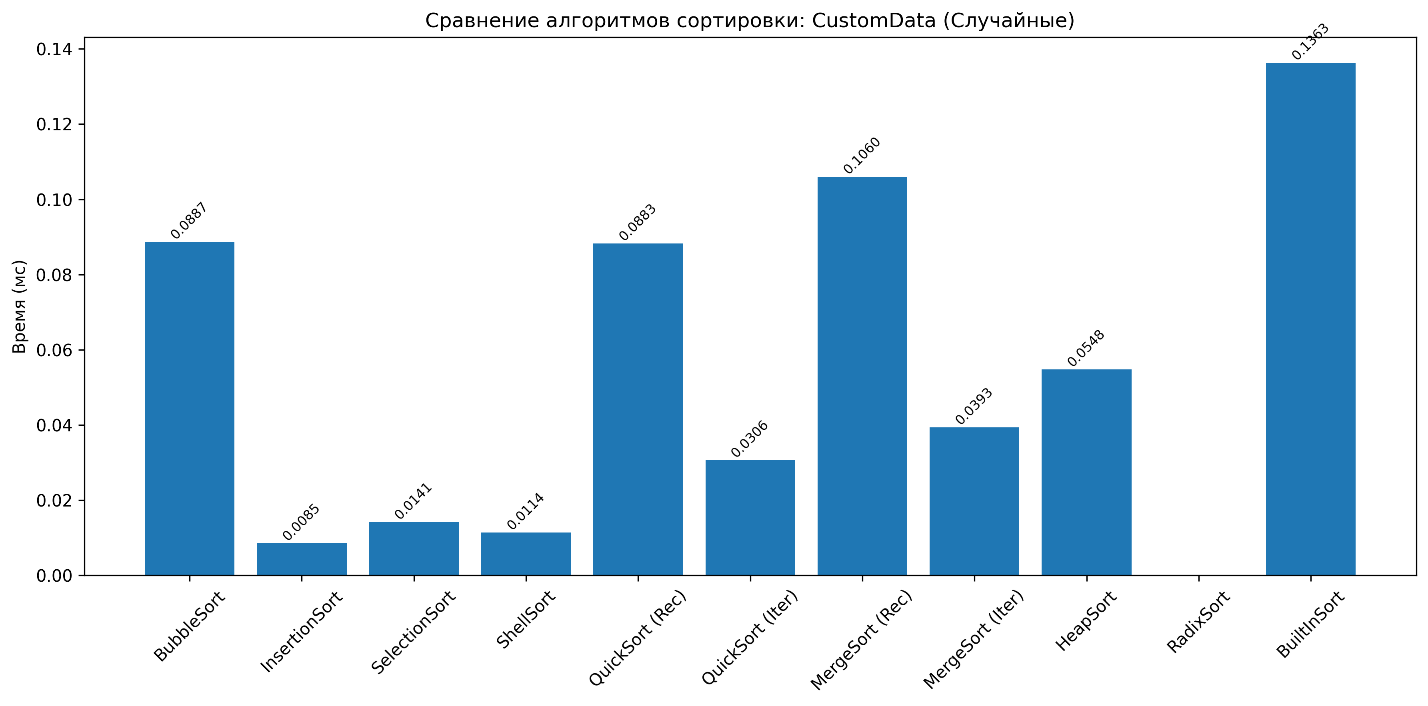
Данные по времени отслеживались в миллисекундах. Все данные записывались в отдельные текстовые документы разбитые по размерам данных для удобства их анализа.

Было решено проводить анализ по категории: размерность – тип данных, при этом для анализа использовались диаграммы и таблицы.

Рассмотрим диаграммы и таблицы для разных размерностей. (сначала буду приводить общую таблицу, затем отдельных диаграммы)

Размер массива 50 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тп\Алг | BubbleSort | InsertionSort | SelectionSort | ShellSort | QuickSort (Rec) | QuickSort (Iter) | MergeSort (Rec) | MergeSort (Iter) | HeapSort | RadixSort | BuiltInSort |
| int (Сл) | 0,6053 | 0,0954 | 0,0959 | 0,197 | 0,2041 | 0,9187 | 0,3225 | 0,2291 | 0,2143 | 14,0648 | 0,0593 |
| int (сорт) | 0,0004 | 0,0005 | 0,0044 | 0,0116 | 0,0078 | 0,0177 | 0,0164 | 0,0029 | 0,0046 | 0,0187 | 0,002 |
| int (обрсорт) | 0,0071 | 0,0041 | 0,0044 | 0,0051 | 0,0081 | 0,0099 | 0,0254 | 0,0027 | 0,0036 | 0,0087 | 0,0023 |
| int (дуб) | 0,0091 | 0,0025 | 0,0053 | 0,0064 | 0,0042 | 0,008 | 0,011 | 0,0038 | 0,0048 | 0,0087 | 0,0031 |
| int (чассор) | 0,0007 | 0,0005 | 0,0041 | 0,0048 | 0,0114 | 0,0102 | 0,0385 | 0,0024 | 0,0046 | 0,0079 | 0,0017 |
| double | 1,0305 | 0,0831 | 0,0838 | 0,1615 | 0,2072 | 0,1536 | 0,3851 | 0,1232 | 0,1699 | N/A | 1,2053 |
| string | 0,8796 | 0,1359 | 0,1777 | 0,186 | 0,3118 | 0,3765 | 0,5426 | 0,2031 | 0,2862 | N/A | 0,1463 |
| CustomData | 0,0887 | 0,0085 | 0,0141 | 0,0114 | 0,0883 | 0,0306 | 0,106 | 0,0393 | 0,0548 | N/A | 0,1363 |
| DateTime | 0,1968 | 0,0821 | 0,0816 | 0,1868 | 0,2103 | 0,4123 | 0,8577 | 0,3049 | 0,3018 | N/A | 1,5127 |



Результаты экспериментов наглядно демонстрируют, что производительность алгоритмов сортировки критически зависит как от типа входных данных, так и от степени их упорядоченности. На малых массивах (50 элементов) особенно ярко проявляются особенности работы различных алгоритмов.

Для случайных целочисленных данных квадратичные алгоритмы (BubbleSort, SelectionSort) показывают предсказуемо низкую производительность, в то время как более совершенные методы демонстрируют существенно лучшие результаты. Особенно выделяется встроенная сортировка Python, подтверждая свою высокую оптимизацию. Примечательно, что RadixSort показал аномально высокое время выполнения , что объясняется значительными накладными расходами при обработке малых массивов и особенностями реализации.

На уже отсортированных данных InsertionSort и BubbleSort проявляют свои лучшие качества, практически не уступая встроенной сортировке.

При работе с обратно отсортированными массивами наблюдается характерная картина: InsertionSort, для которого это худший случай, существенно замедляется по сравнению с отсортированными данными. В то же время MergeSort и QuickSort сохраняют стабильность, подтверждая свою устойчивость к неблагоприятным входным данным.

Особый интерес представляют результаты для частично упорядоченных массивов. InsertionSort вновь демонстрирует выдающуюся эффективность, превосходя даже встроенную сортировку. ShellSort также показывает отличные результаты, благодаря своей способности эффективно обрабатывать почти упорядоченные последовательности.

Для вещественных чисел наблюдается неожиданный результат: встроенная сортировка уступает по скорости InsertionSort. Это может быть связано с особенностями сравнения чисел с плавающей точкой в Python и дополнительными проверками в реализации Timsort.

При работе со строками встроенная сортировка вновь подтверждает свою эффективность, значительно опережая BubbleSort и другие квадратичные алгоритмы. Это объясняется специальными оптимизациями для строковых операций в Python.

Результаты для пользовательских типов данных и дат показывают, что производительность существенно зависит от сложности операции сравнения. InsertionSort демонстрирует отличные результаты на простых объектах, в то время как встроенная сортировка может проигрывать из-за накладных расходов на вызов методов сравнения.

Сравнение рекурсивных и итеративных реализаций выявило интересные особенности:

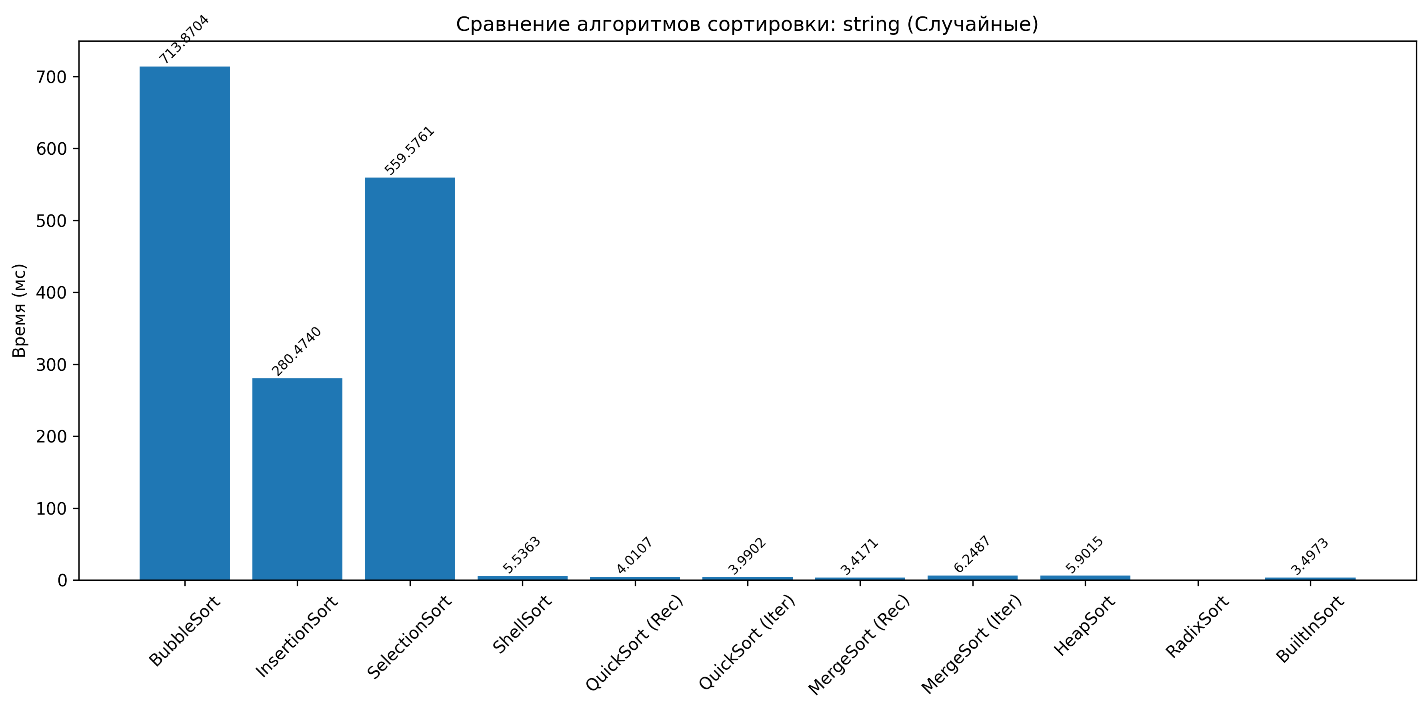
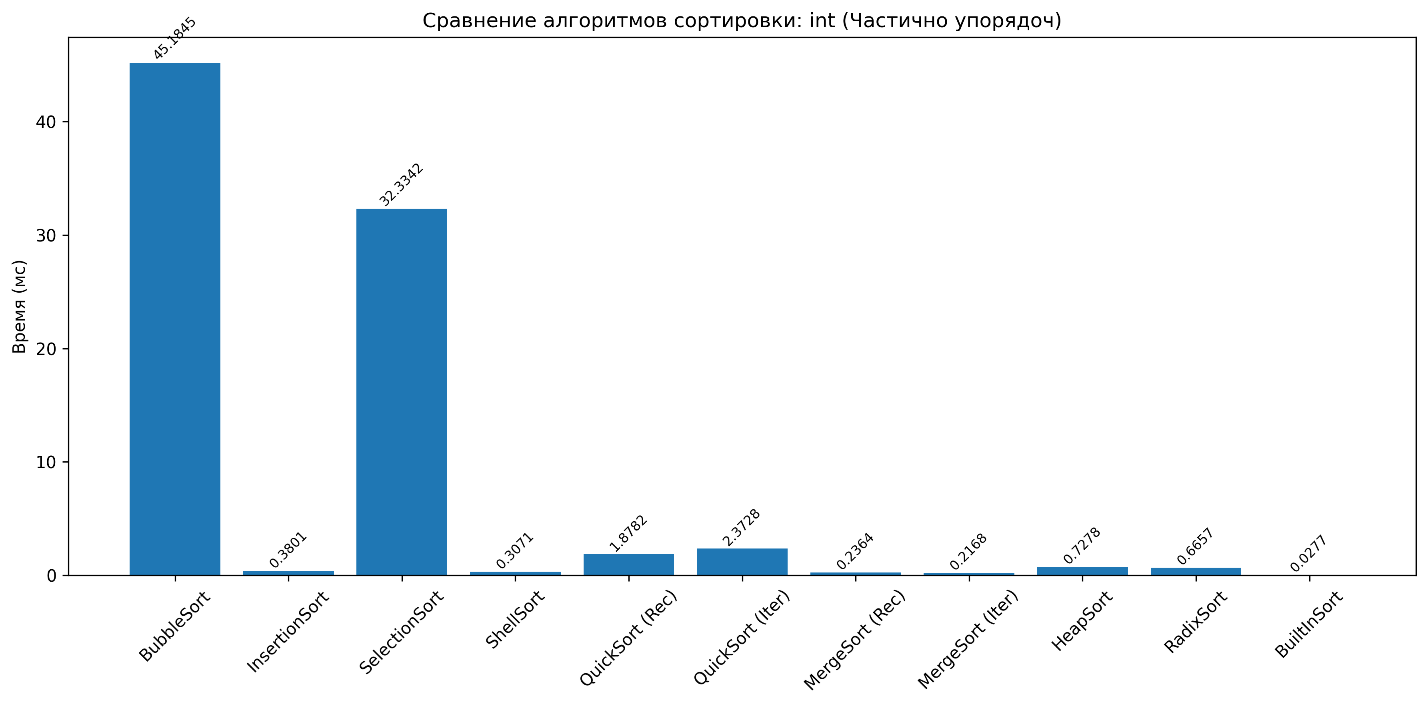
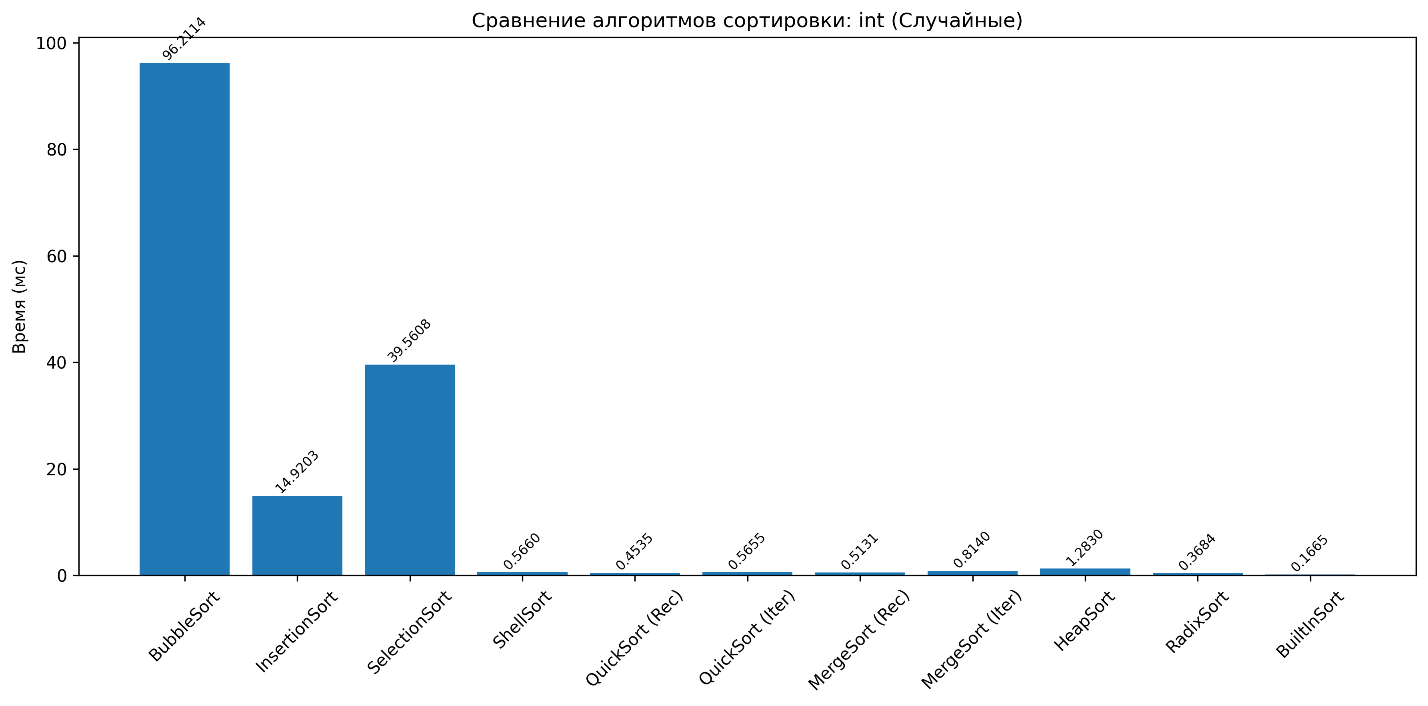
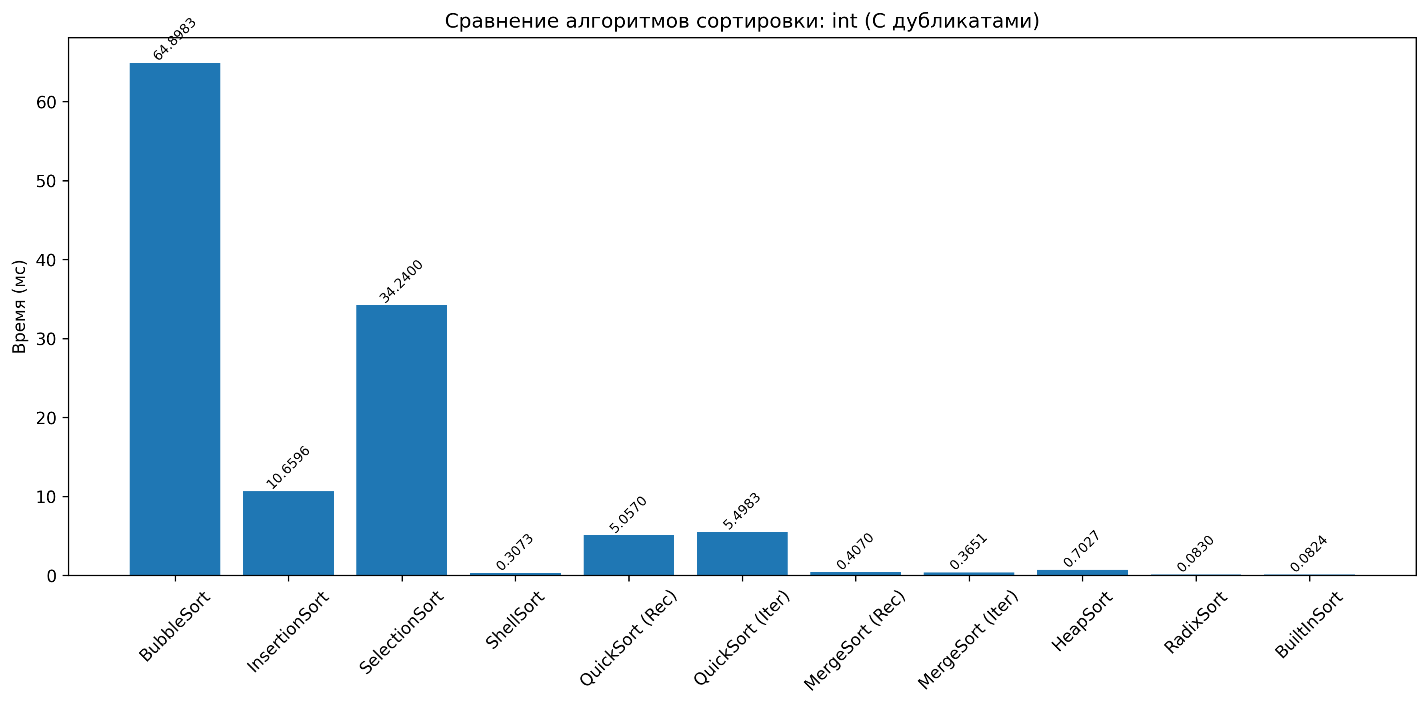
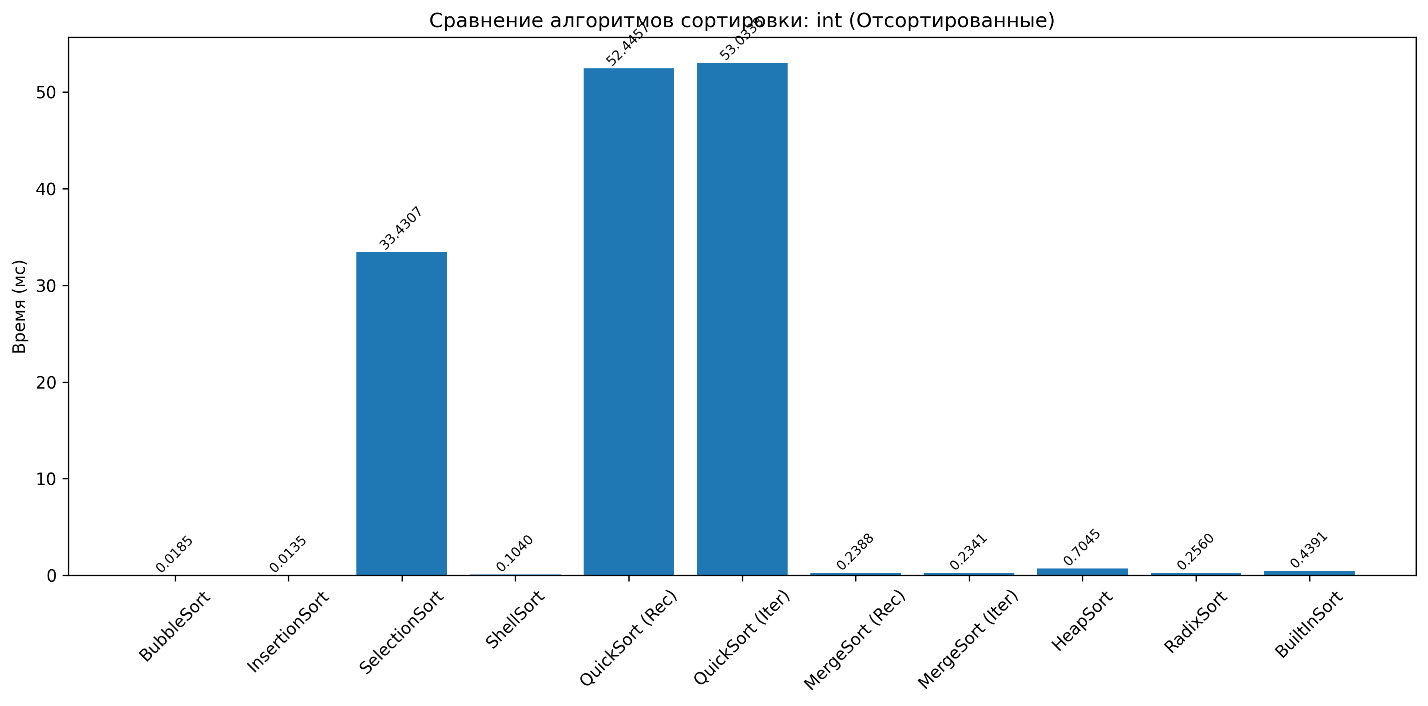
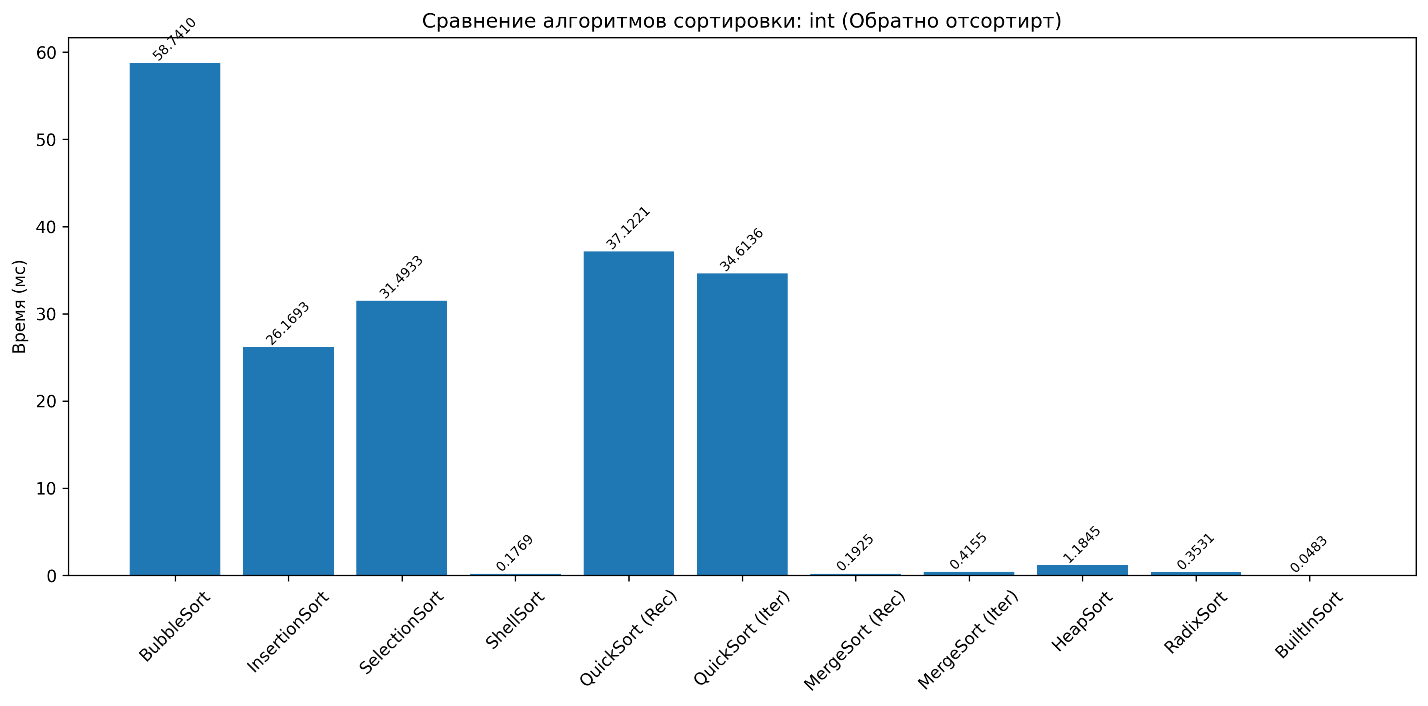
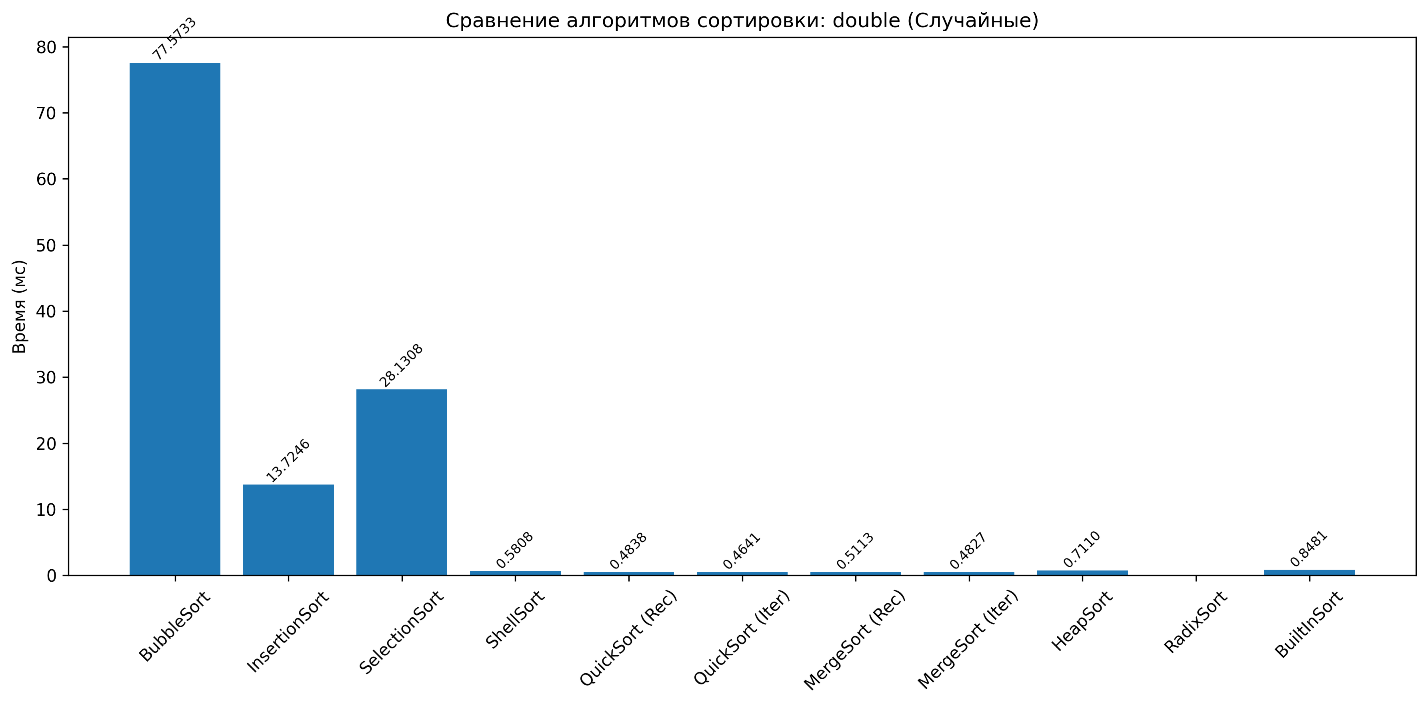
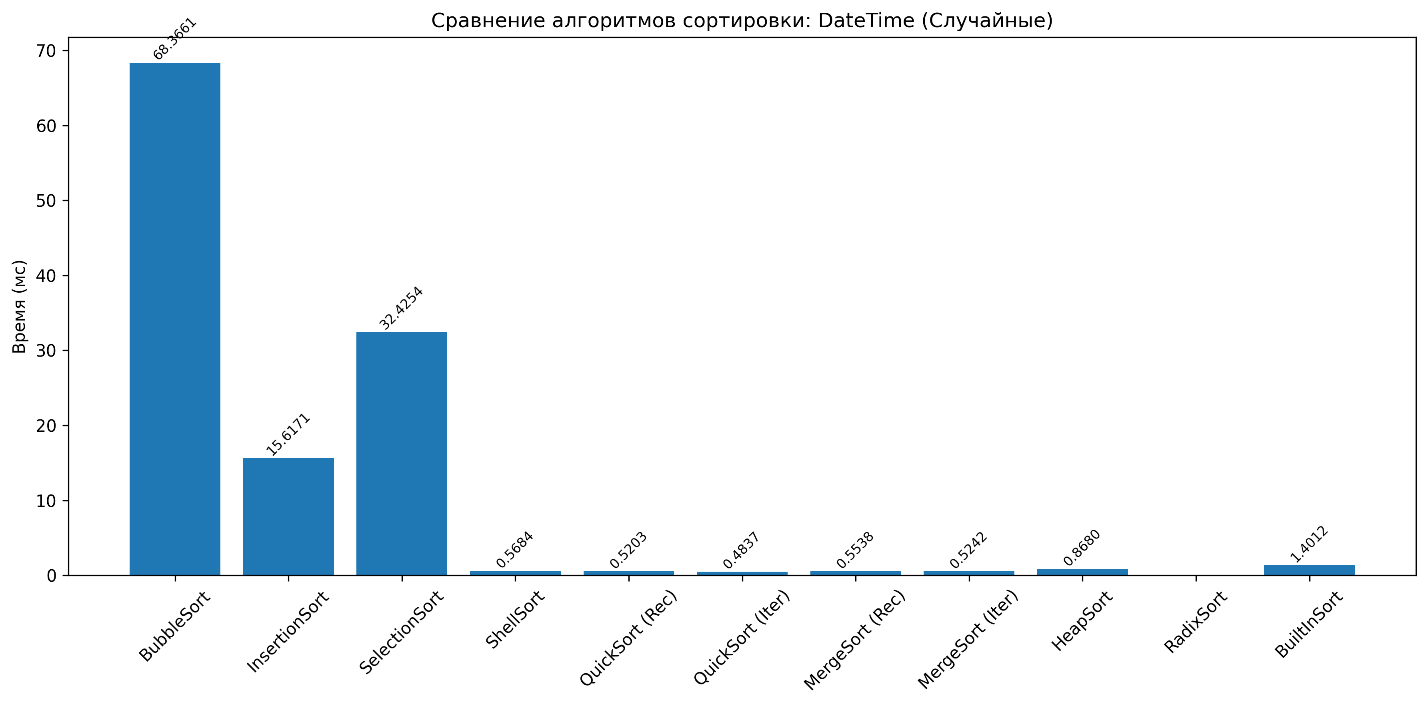
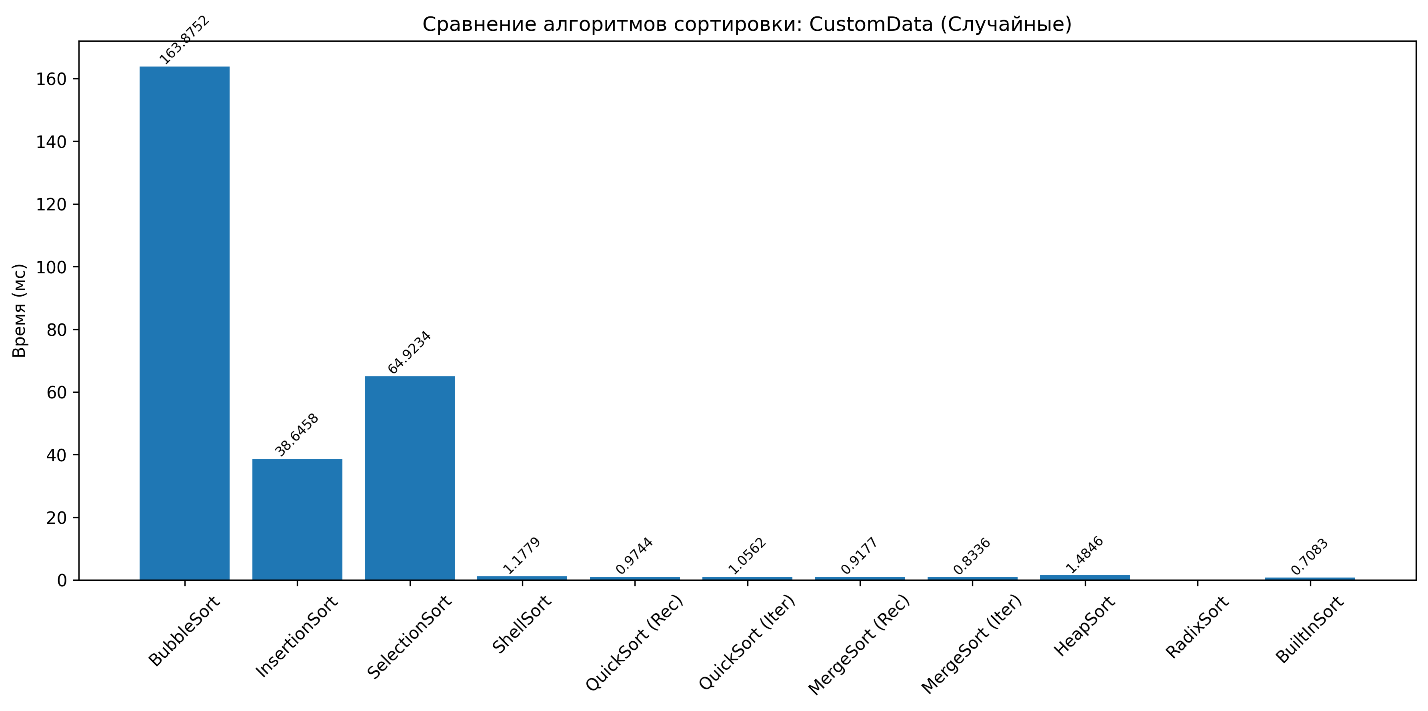
QuickSort (рекурсивный) оказался быстрее итеративной версии что противоречит ожиданиям

MergeSort (итеративный) показал лучшие результаты по сравнению с рекурсивной реализацией

Эти различия объясняются особенностями управления стеком вызовов и накладными расходами на организацию циклов в итеративных версиях алгоритмов.

Размер массива 5000 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тп\Алг | BubbleSort | InsertionSort | SelectionSort | ShellSort | QuickSort(Rec) | QuickSort(Iter) | MergeSort(Rec) | MergeSort(Iter) | HeapSort | RadixSort | BuiltInSort |
| int(Сл) | 96,2114 | 14,9203 | 39,5608 | 0,566 | 0,4535 | 0,5655 | 0,5131 | 0,814 | 1,283 | 0,3684 | 0,1665 |
| int(сорт) | 0,0185 | 0,0135 | 33,4307 | 0,104 | 52,4457 | 53,0338 | 0,2388 | 0,2341 | 0,7045 | 0,256 | 0,4391 |
| int(обрсорт) | 58,741 | 26,1693 | 31,4933 | 0,1769 | 37,1221 | 34,6136 | 0,1925 | 0,4155 | 1,1845 | 0,3531 | 0,0483 |
| int(дубл) | 64,8983 | 10,6596 | 34,24 | 0,3073 | 5,057 | 5,4983 | 0,407 | 0,3651 | 0,7027 | 0,083 | 0,0824 |
| int(чупор) | 45,1845 | 0,3801 | 32,3342 | 0,3071 | 1,8782 | 2,3728 | 0,2364 | 0,2168 | 0,7278 | 0,6657 | 0,0277 |
| double | 77,5733 | 13,7246 | 28,1308 | 0,5808 | 0,4838 | 0,4641 | 0,5113 | 0,4827 | 0,711 | N/A | 0,8481 |
| string | 713,8704 | 280,474 | 559,5761 | 5,5363 | 4,0107 | 3,9902 | 3,4171 | 6,2487 | 5,9015 | N/A | 3,4973 |
| CustomData | 163,8752 | 38,6458 | 64,9234 | 1,1779 | 0,9744 | 1,0562 | 0,9177 | 0,8336 | 1,4846 | N/A | 0,7083 |
| DateTime | 68,3661 | 15,6171 | 32,4254 | 0,5684 | 0,5203 | 0,4837 | 0,5538 | 0,5242 | 0,868 | N/A | 1,4012 |



Результаты экспериментов подтверждают, что эффективность алгоритмов сортировки значительно варьируется в зависимости от типа и структуры входных данных. Это особенно заметно на массивах небольшого размера, где критическое значение приобретают накладные расходы и поведение алгоритмов в граничных случаях.

**Для случайных целочисленных данных** квадратичные алгоритмы (BubbleSort, SelectionSort) демонстрируют низкую производительность, ожидаемо уступая более продвинутым методам. Интерес вызывает высокоэффективная встроенная сортировка Python, превосходящая большинство других подходов, включая RadixSort, для которого наблюдаются значительные издержки из-за неэффективности на малых объемах данных.

**На уже отсортированных массивах** особенно ярко проявляется преимущество простых адаптивных алгоритмов. InsertionSort и BubbleSort практически не выполняют лишних операций, обеспечивая отличную производительность. В то же время алгоритмы с фиксированной структурой (например, QuickSort и SelectionSort) теряют в эффективности, так как не способны использовать информацию о предварительном порядке элементов.

**В случае обратно отсортированных данных** ситуация меняется. Для InsertionSort это один из худших сценариев, при котором производительность резко снижается. Однако устойчивые к порядку алгоритмы, такие как MergeSort и встроенная сортировка, демонстрируют стабильность. QuickSort также сохраняет приемлемую эффективность, хотя и чувствителен к подобным входным данным.

**При наличии дубликатов** эффективность большинства алгоритмов остается на высоком уровне. Особенно хорошо справляются встроенная сортировка и QuickSort, эффективно оптимизируя повторяющиеся значения. Квадратичные методы по-прежнему уступают, но не столь катастрофично, как в случае полностью случайных данных.

**На частично упорядоченных массивах** лучшим образом проявляют себя адаптивные алгоритмы — прежде всего, InsertionSort и встроенная сортировка. ShellSort также демонстрирует высокую производительность благодаря своей способности эффективно обрабатывать «почти отсортированные» последовательности. Это соответствует теоретическим ожиданиям и подчеркивает ценность адаптивности.

**Для вещественных чисел** встроенная сортировка по-прежнему эффективна, хотя не всегда лидирует. InsertionSort показывает неожиданно высокую производительность, что может быть связано с особенностями работы с числами с плавающей точкой и спецификой Python-сравнений.

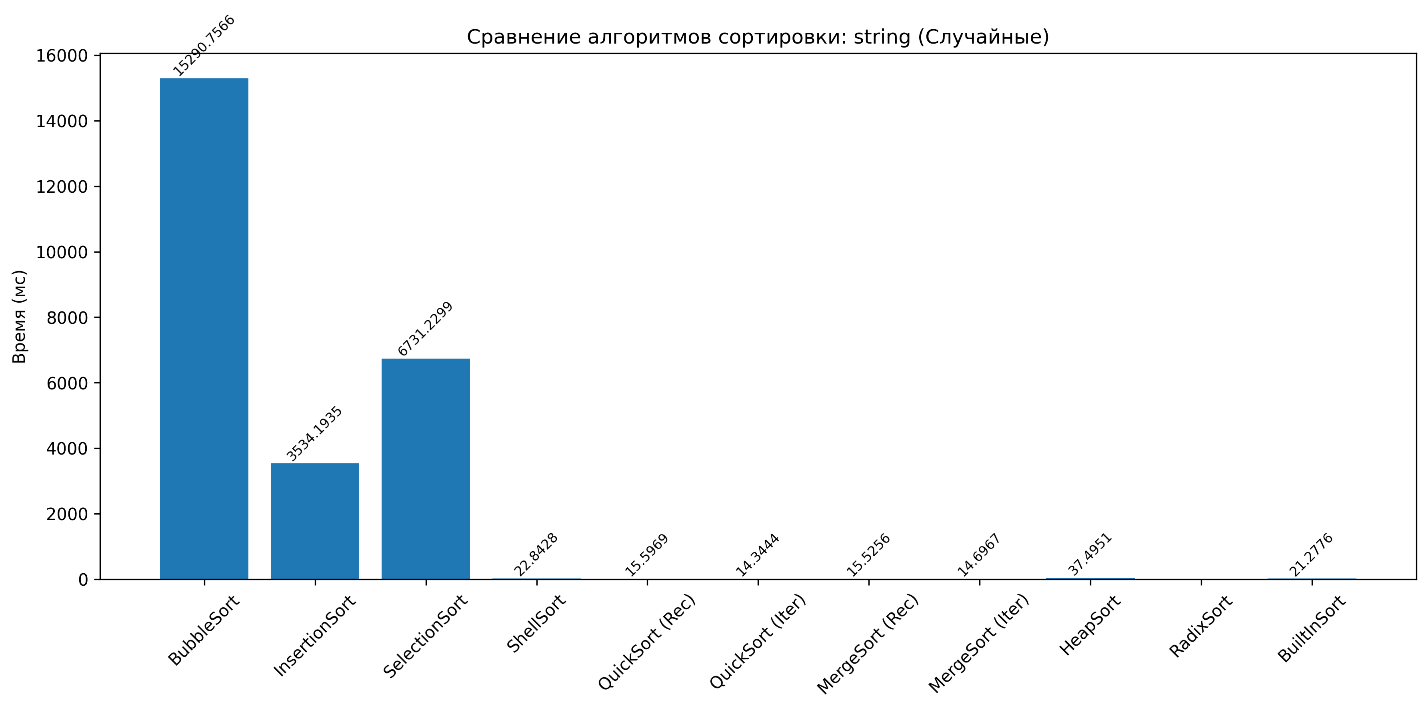
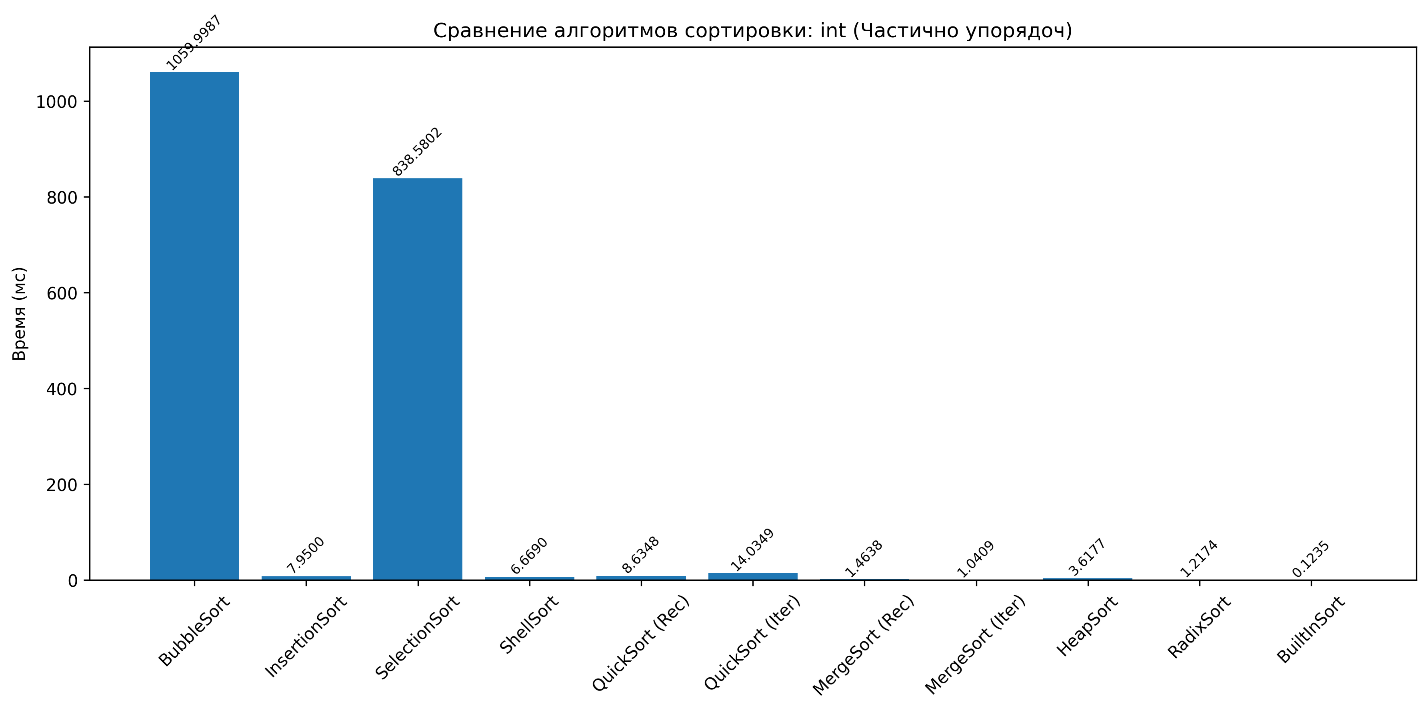
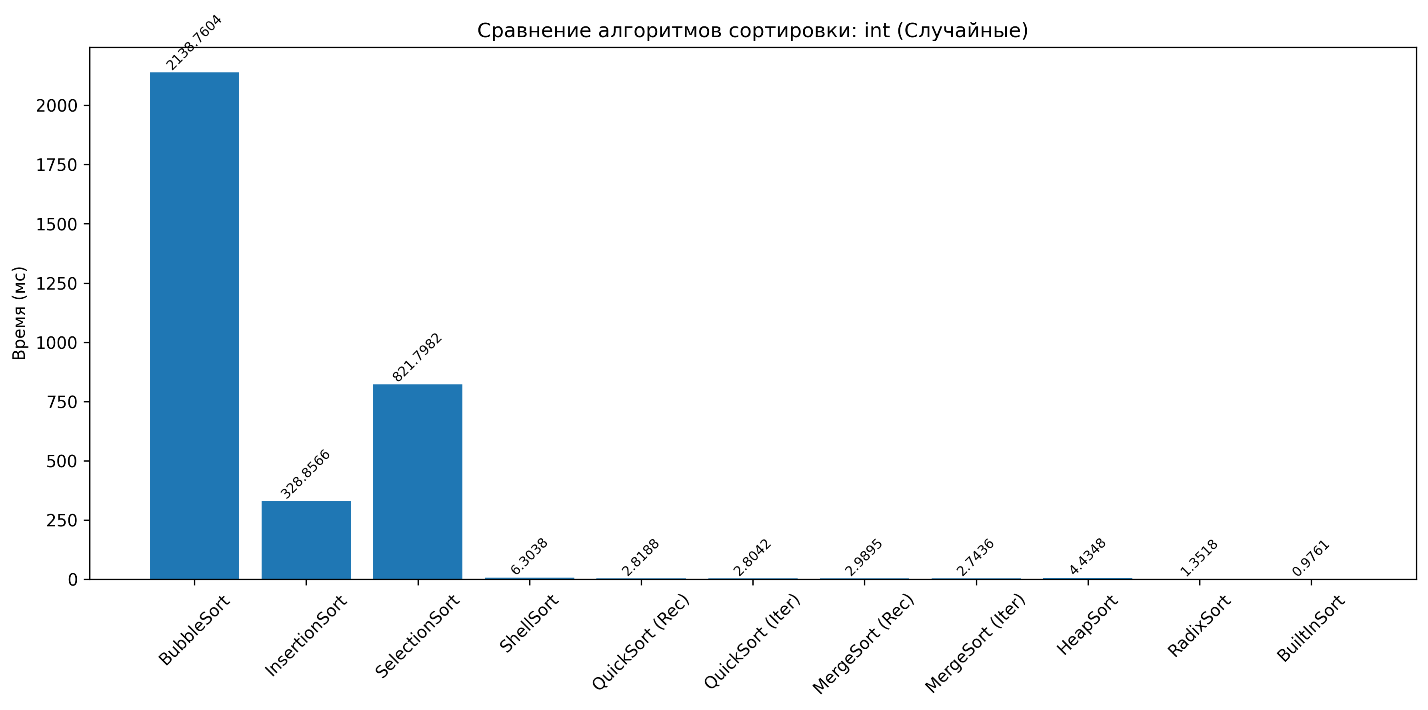
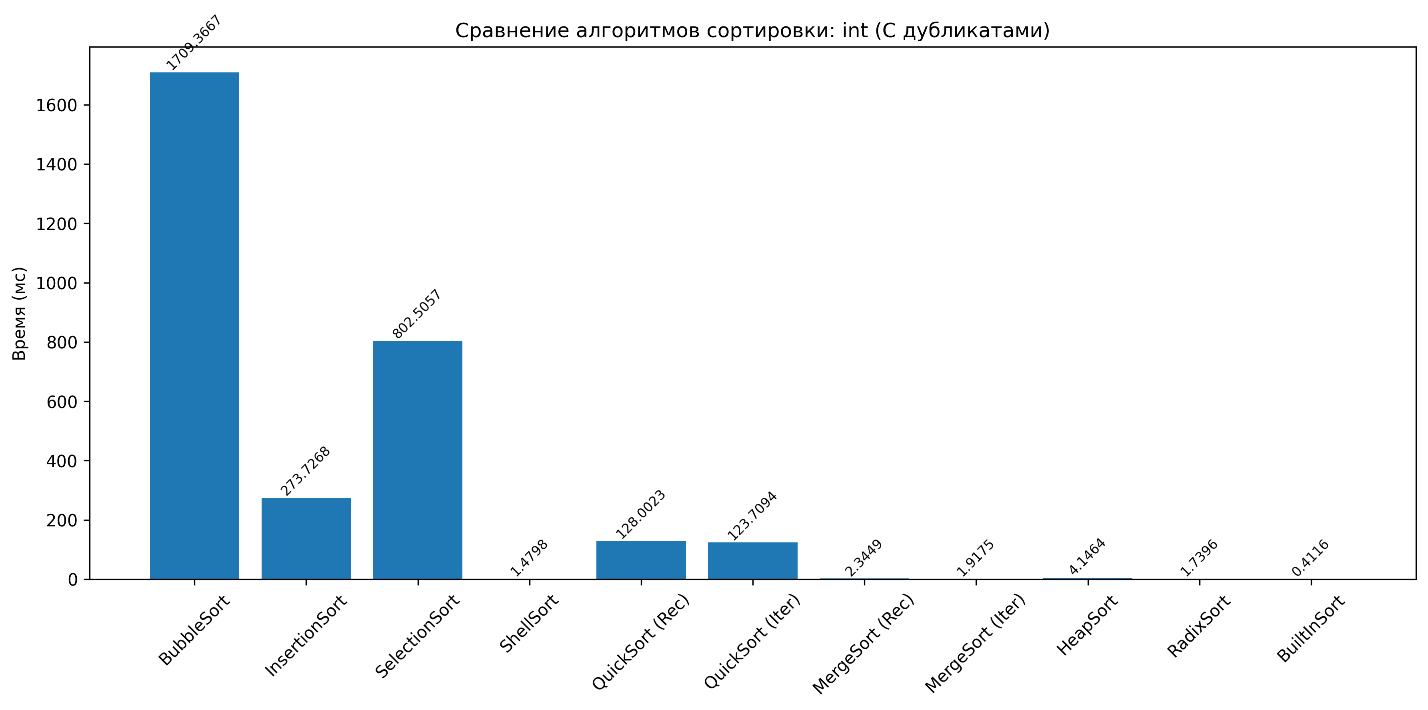
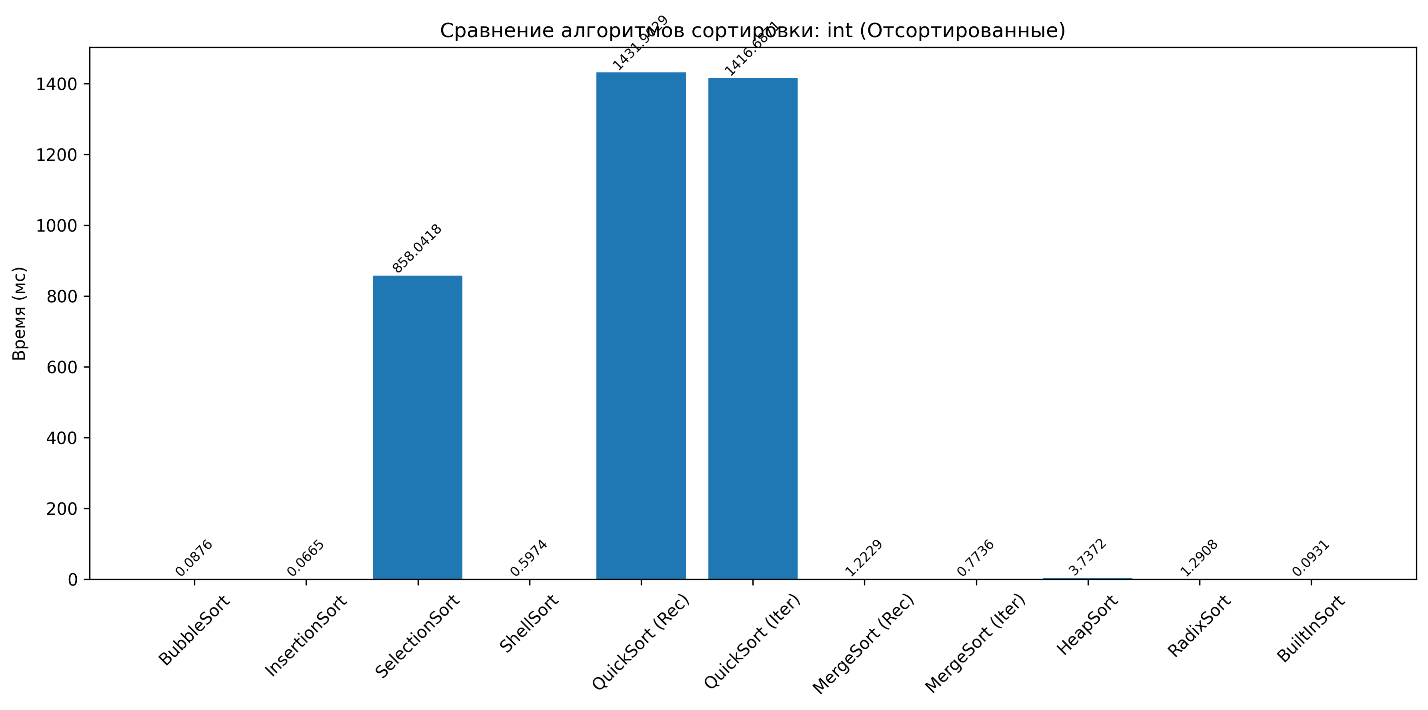
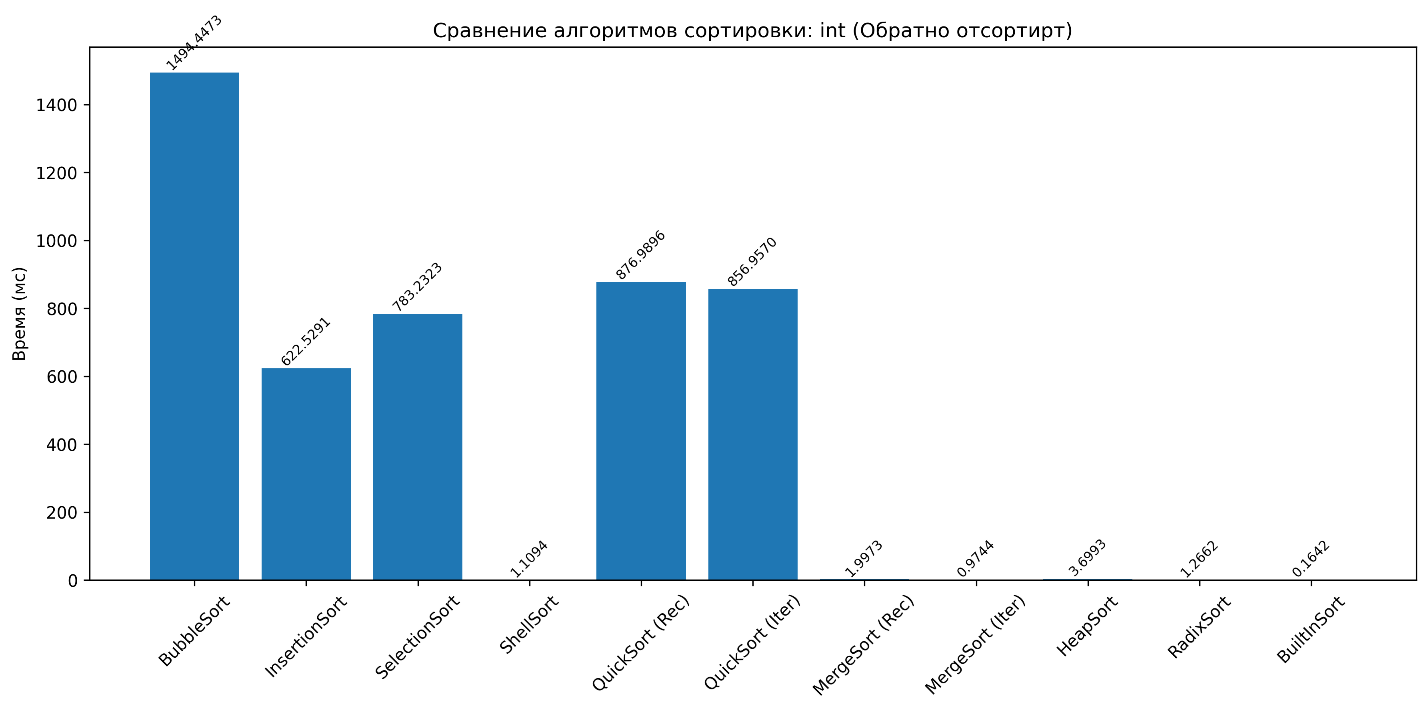
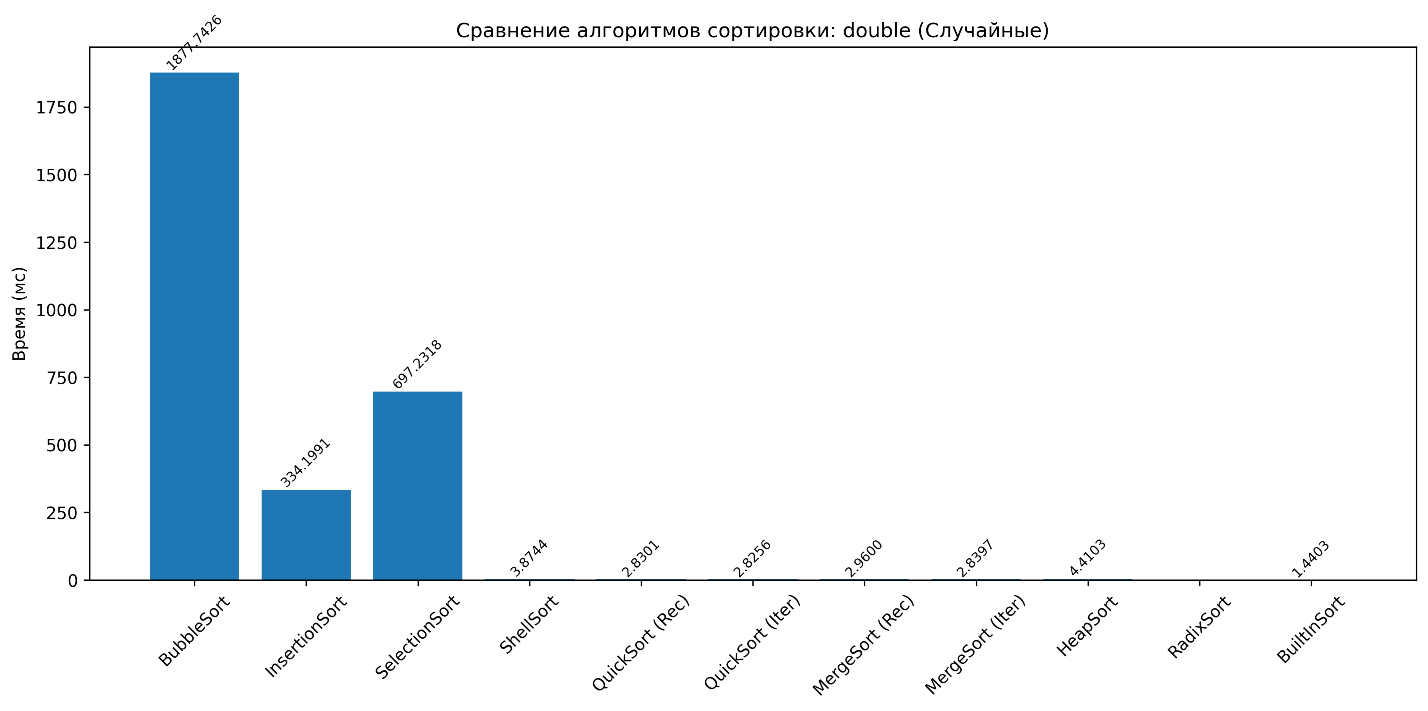
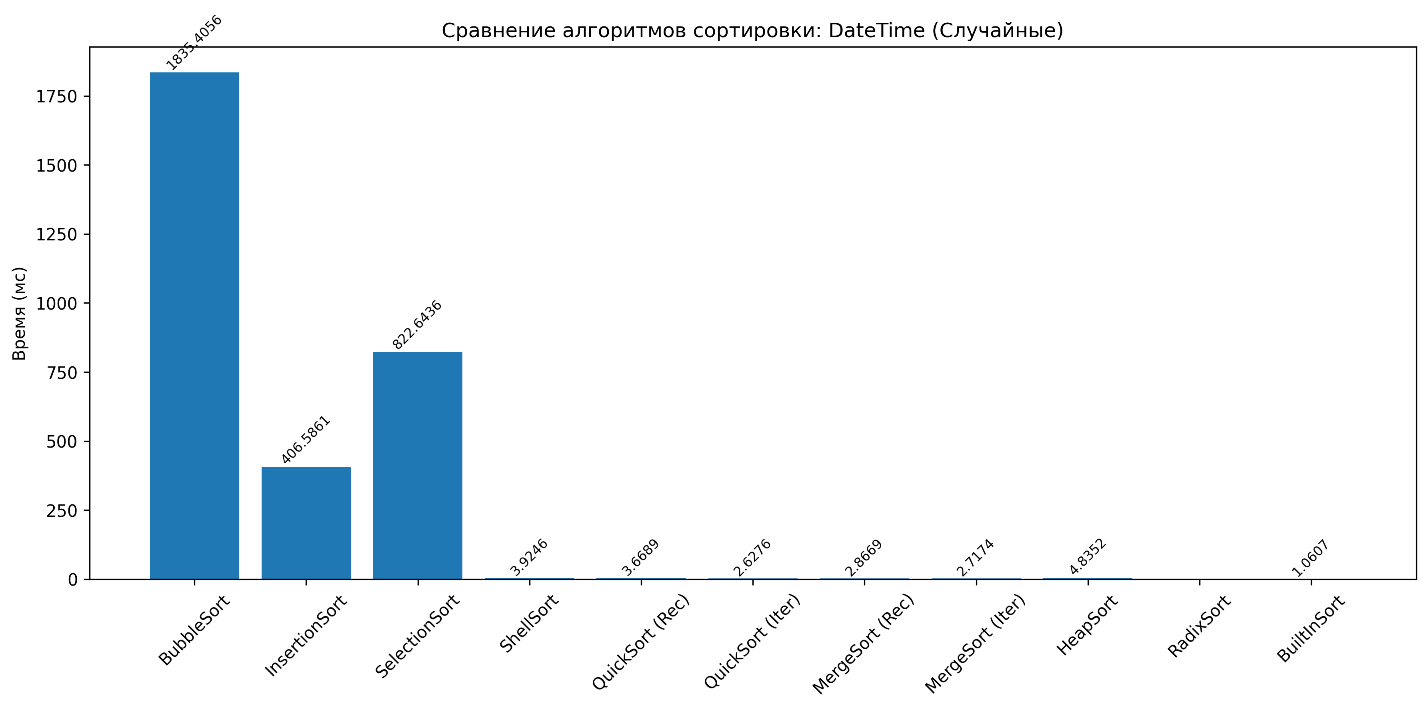
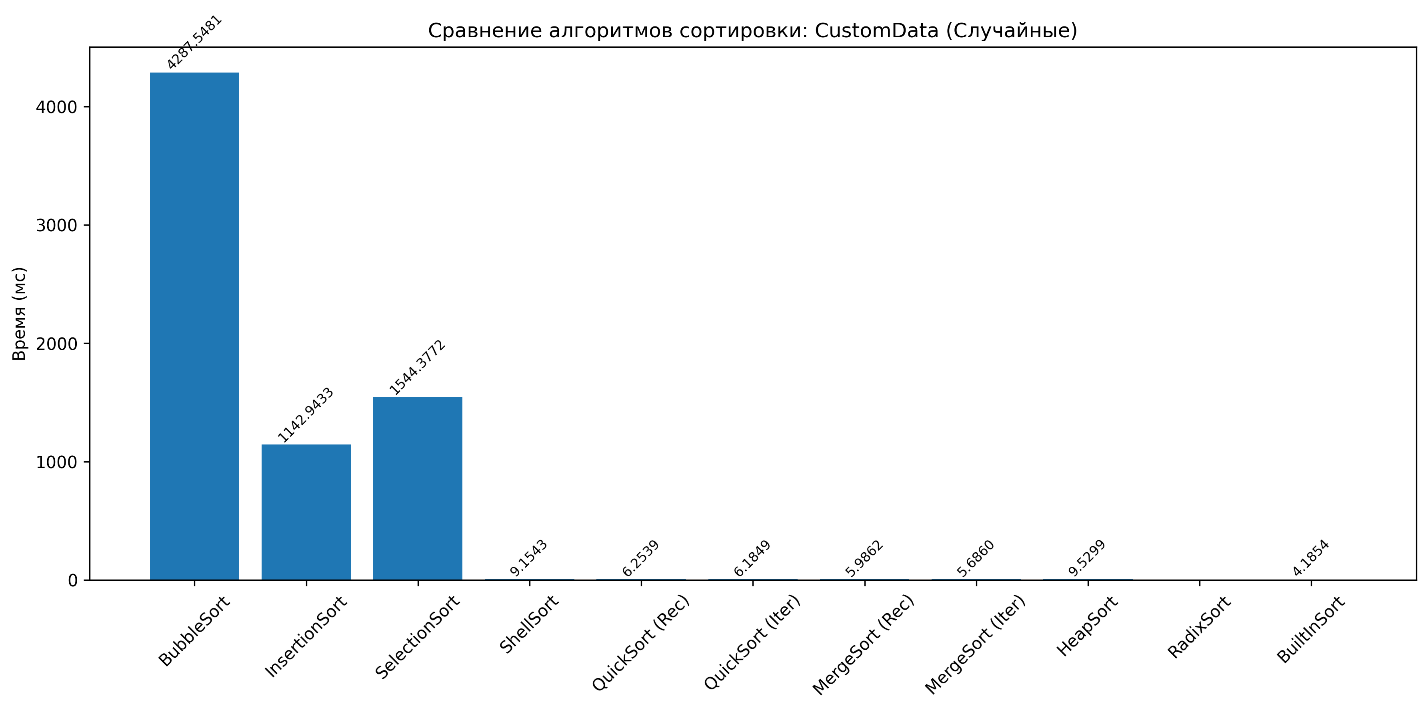
**При сортировке строк** встроенная сортировка уверенно опережает другие алгоритмы, в том числе квадратичные, которые испытывают значительные трудности. Это объясняется специализированной оптимизацией строковых операций в реализации Timsort. QuickSort и MergeSort также показывают хорошую устойчивость.

**В случае пользовательских типов данных** эффективность сортировки определяется сложностью операций сравнения. Простые алгоритмы, такие как InsertionSort, оказываются весьма эффективными благодаря прямолинейной логике и низким накладным расходам. Встроенная сортировка при этом может уступать из-за внутренней сложности механизма и необходимости многократных вызовов пользовательских методов сравнения.

**Сортировка по дате и времени** показывает общую тенденцию: MergeSort, QuickSort и встроенный метод сохраняют высокую производительность. Квадратичные алгоритмы вновь демонстрируют ощутимо худшие результаты, подтверждая свою низкую пригодность для более сложных типов данных.

Размер массива 25000 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тип данных | BubbleSort | InsertionSort | SelectionSort | ShellSort | QuickSort (Rec) | QuickSort (Iter) | MergeSort (Rec) | MergeSort (Iter) | HeapSort | RadixSort | BuiltInSort |
| int(Сл) | 96,2114 | 14,9203 | 39,5608 | 0,566 | 0,4535 | 0,5655 | 0,5131 | 0,814 | 1,283 | 0,3684 | 0,1665 |
| int(сорт) | 0,0185 | 0,0135 | 33,4307 | 0,104 | 52,4457 | 53,0338 | 0,2388 | 0,2341 | 0,7045 | 0,256 | 0,4391 |
| int(обрсорт) | 58,741 | 26,1693 | 31,4933 | 0,1769 | 37,1221 | 34,6136 | 0,1925 | 0,4155 | 1,1845 | 0,3531 | 0,0483 |
| int(дубл) | 64,8983 | 10,6596 | 34,24 | 0,3073 | 5,057 | 5,4983 | 0,407 | 0,3651 | 0,7027 | 0,083 | 0,0824 |
| int(чупор) | 45,1845 | 0,3801 | 32,3342 | 0,3071 | 1,8782 | 2,3728 | 0,2364 | 0,2168 | 0,7278 | 0,6657 | 0,0277 |
| double | 77,5733 | 13,7246 | 28,1308 | 0,5808 | 0,4838 | 0,4641 | 0,5113 | 0,4827 | 0,711 | N/A | 0,8481 |
| string | 713,8704 | 280,474 | 559,5761 | 5,5363 | 4,0107 | 3,9902 | 3,4171 | 6,2487 | 5,9015 | N/A | 3,4973 |
| CustomData | 163,8752 | 38,6458 | 64,9234 | 1,1779 | 0,9744 | 1,0562 | 0,9177 | 0,8336 | 1,4846 | N/A | 0,7083 |
| DateTime | 68,3661 | 15,6171 | 32,4254 | 0,5684 | 0,5203 | 0,4837 | 0,5538 | 0,5242 | 0,868 | N/A | 1,4012 |



При увеличении размера массивов влияние алгоритмической сложности на производительность становится особенно заметным. Эксперименты с большими объемами данных ясно демонстрируют сильные и слабые стороны различных методов сортировки в зависимости от структуры данных.

Для случайных целочисленных массивов наблюдается характерная картина: квадратичные алгоритмы (BubbleSort, InsertionSort, SelectionSort) значительно уступают по производительности более совершенным подходам. Встроенная сортировка Python, QuickSort, MergeSort и HeapSort демонстрируют высокую эффективность, устойчиво обрабатывая большие объемы без резких скачков производительности. RadixSort также показывает хорошие результаты, оправдывая себя при обработке больших массивов.

На уже отсортированных данных простейшие адаптивные алгоритмы вновь демонстрируют выдающуюся эффективность, особенно InsertionSort. В то же время QuickSort оказывается крайне неэффективным — как рекурсивная, так и итеративная версии демонстрируют резкое снижение производительности, что подтверждает чувствительность алгоритма к лучшим случаям при неудачном выборе опорного элемента. Встроенная сортировка остается быстрой и стабильной.

Обратный порядок элементов резко снижает эффективность InsertionSort и BubbleSort, что полностью согласуется с теоретическими предсказаниями. В то же время MergeSort и встроенная сортировка сохраняют стабильную и высокую производительность. QuickSort вновь демонстрирует падение эффективности, хотя и не столь значительное, как в случае отсортированных данных.

При наличии дубликатов значительно возрастает нагрузка на алгоритмы с большим количеством сравнений, особенно SelectionSort и BubbleSort. Более совершенные методы (QuickSort, MergeSort и встроенная сортировка) уверенно справляются с этой задачей, демонстрируя стабильную эффективность. Радикальные алгоритмы остаются конкурентоспособными, что свидетельствует об их устойчивости к повторяющимся значениям.

Для частично упорядоченных данных InsertionSort вновь показывает отличные результаты, подтверждая свою адаптивность. ShellSort также демонстрирует высокую эффективность в таких сценариях. Среди универсальных методов особенно хорошо работают MergeSort и встроенная сортировка, благодаря сбалансированному подходу к оценке упорядоченности.

При сортировке вещественных чисел картина сохраняется аналогичной: встроенная сортировка, MergeSort и QuickSort демонстрируют устойчиво высокую эффективность. Квадратичные алгоритмы вновь показывают слабые результаты, особенно на больших объемах данных. Радикальные подходы оказываются полезными при больших объемах, хотя могут уступать по универсальности.

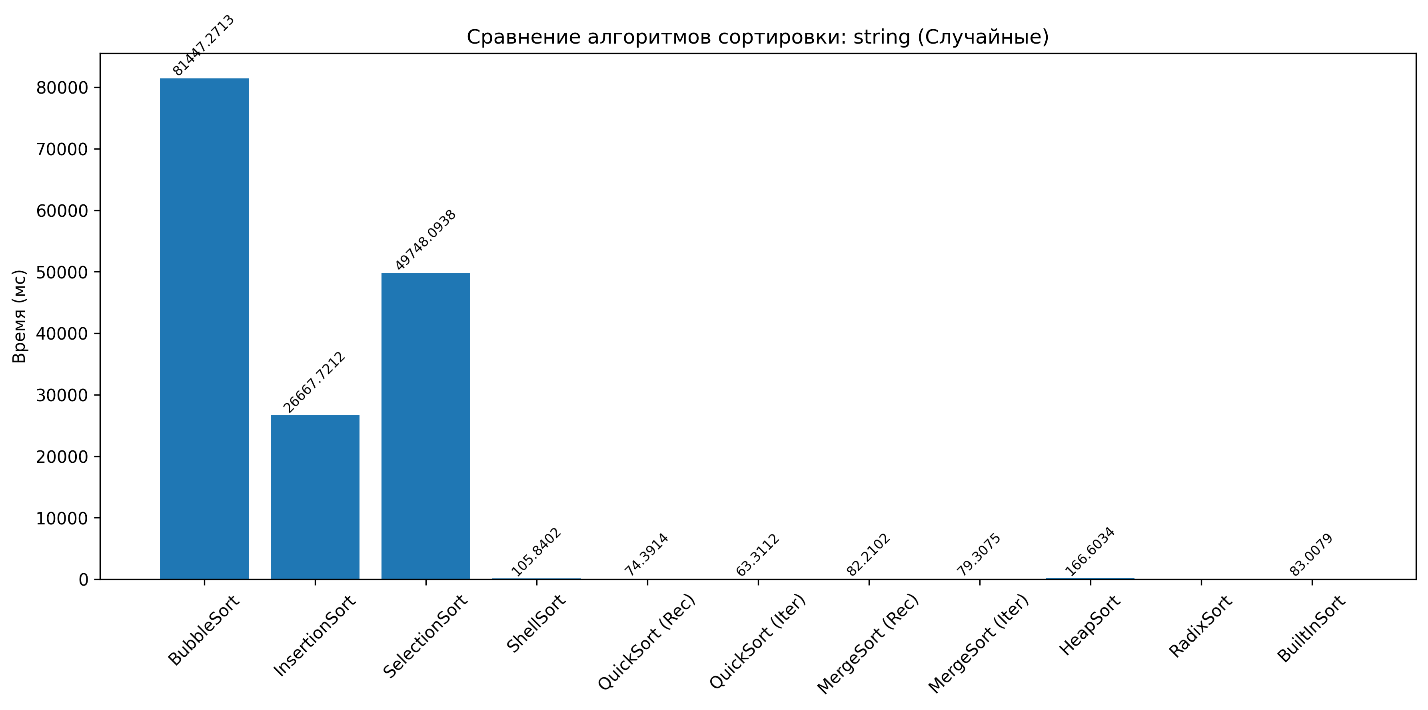
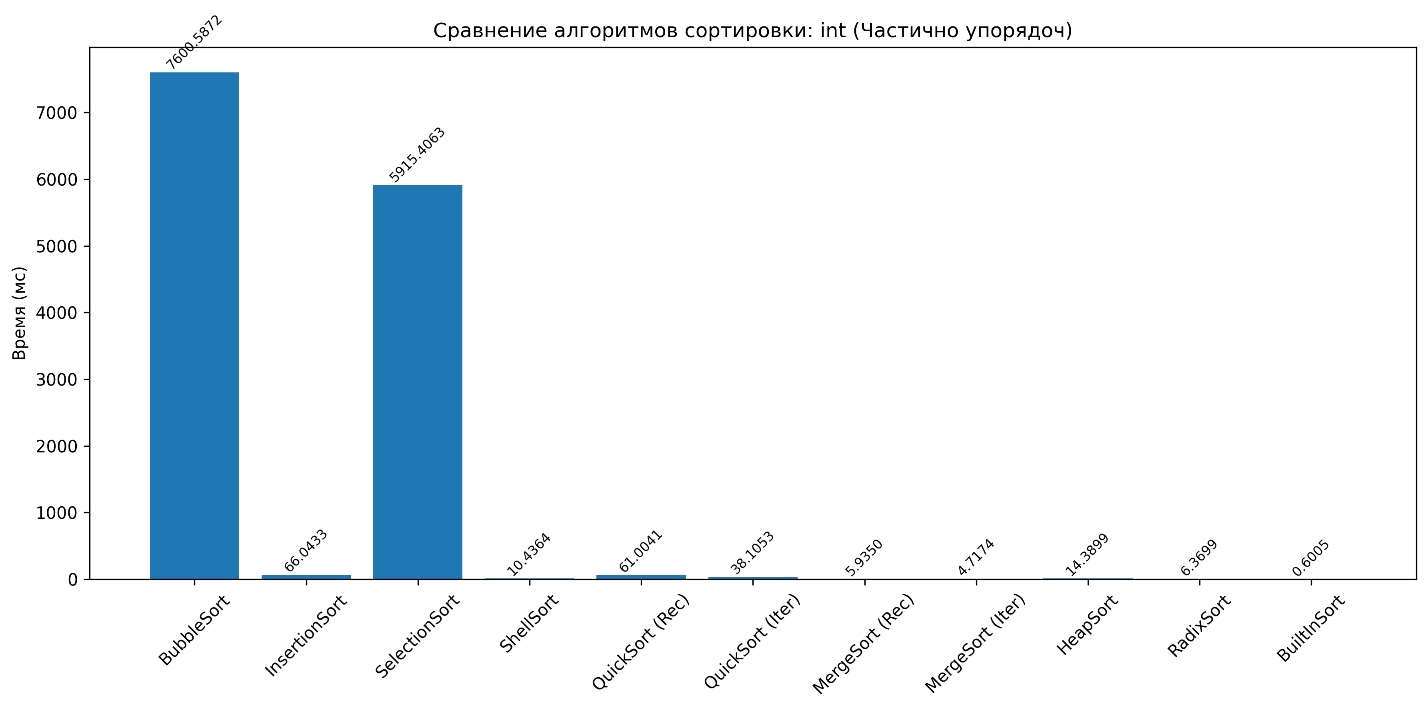
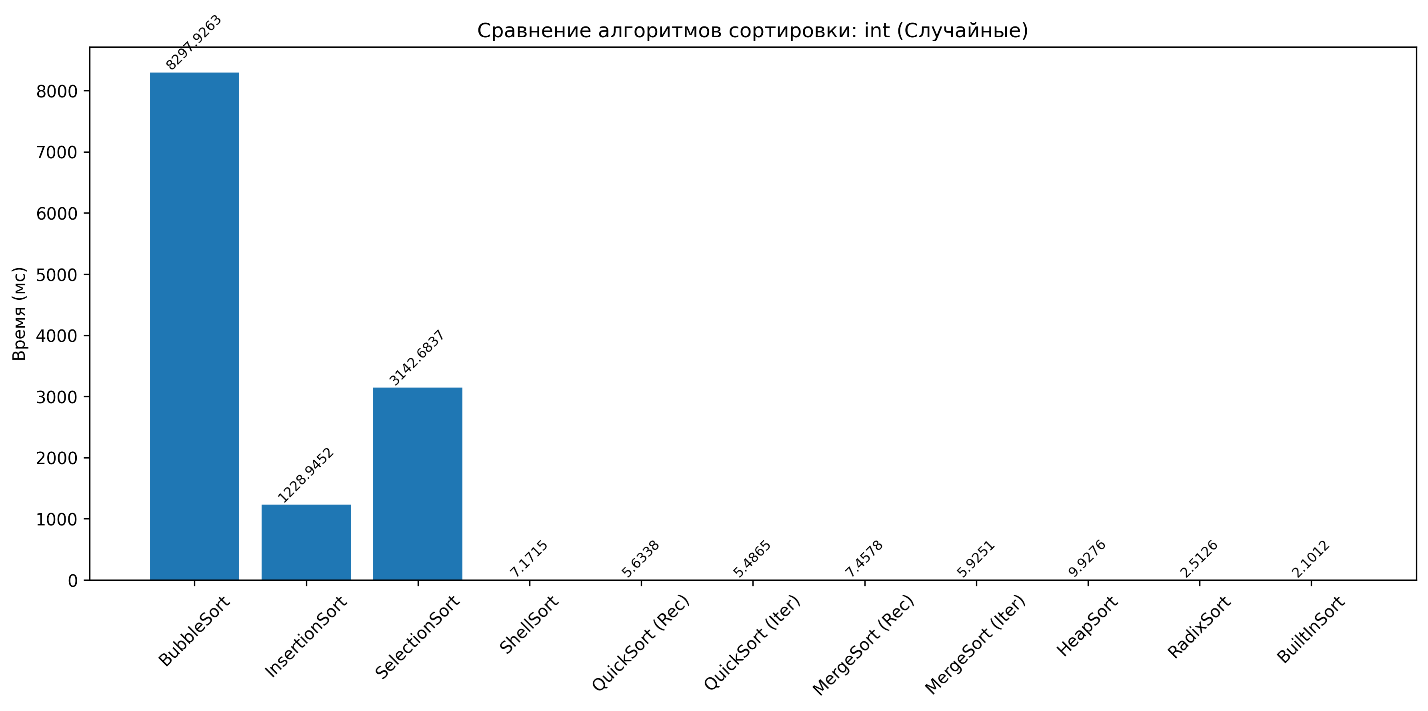
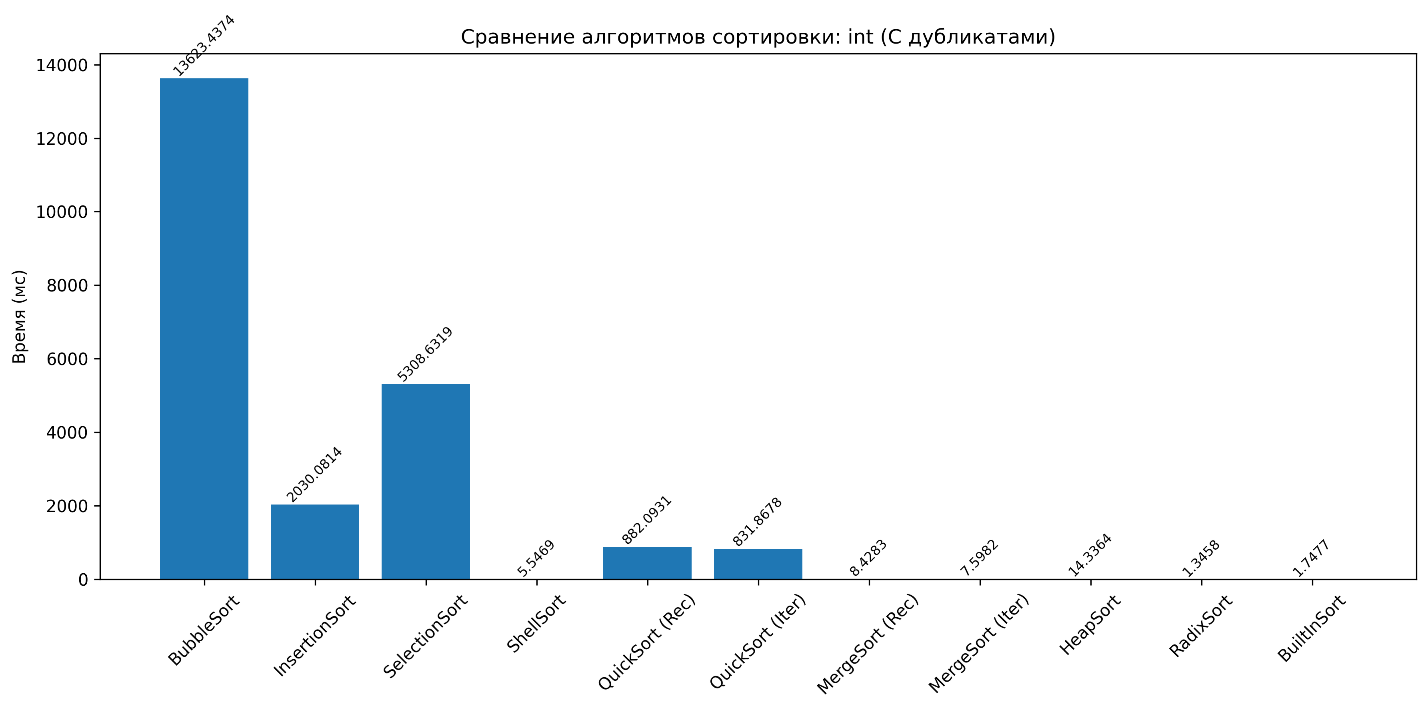
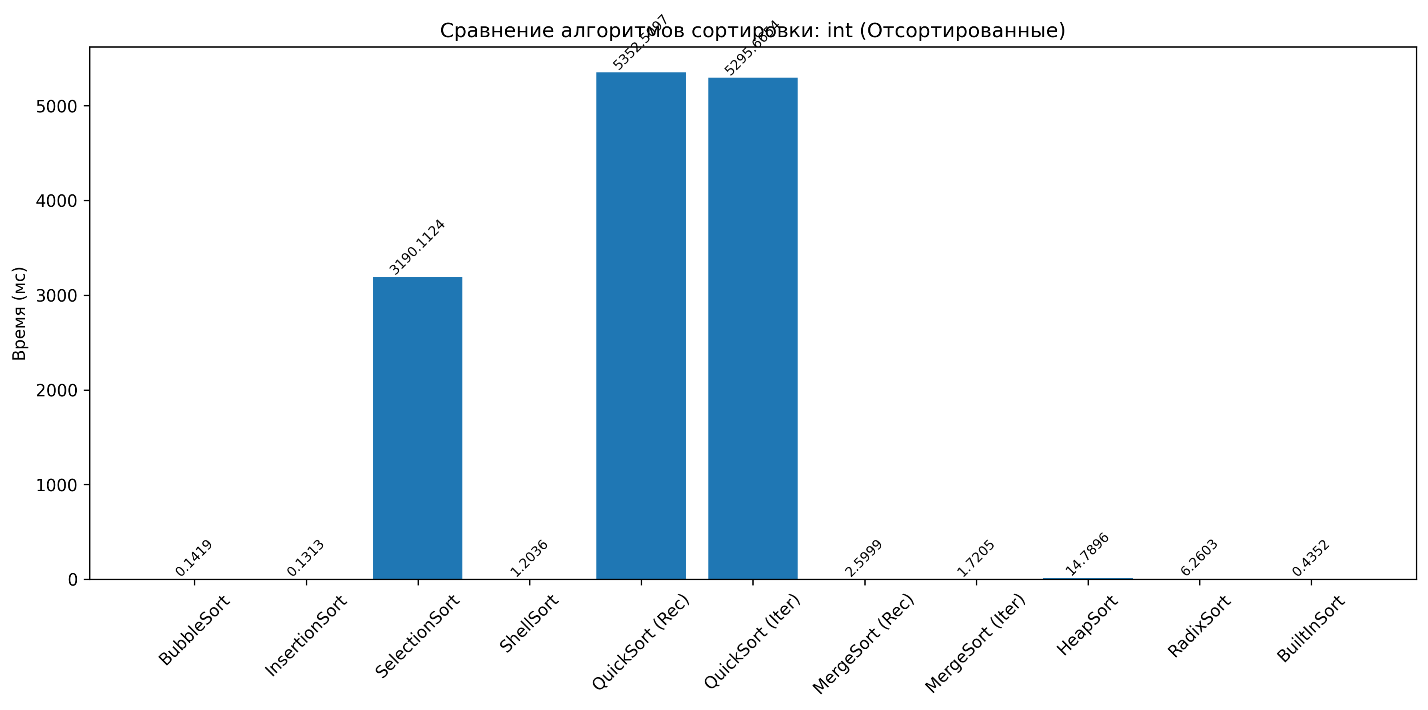
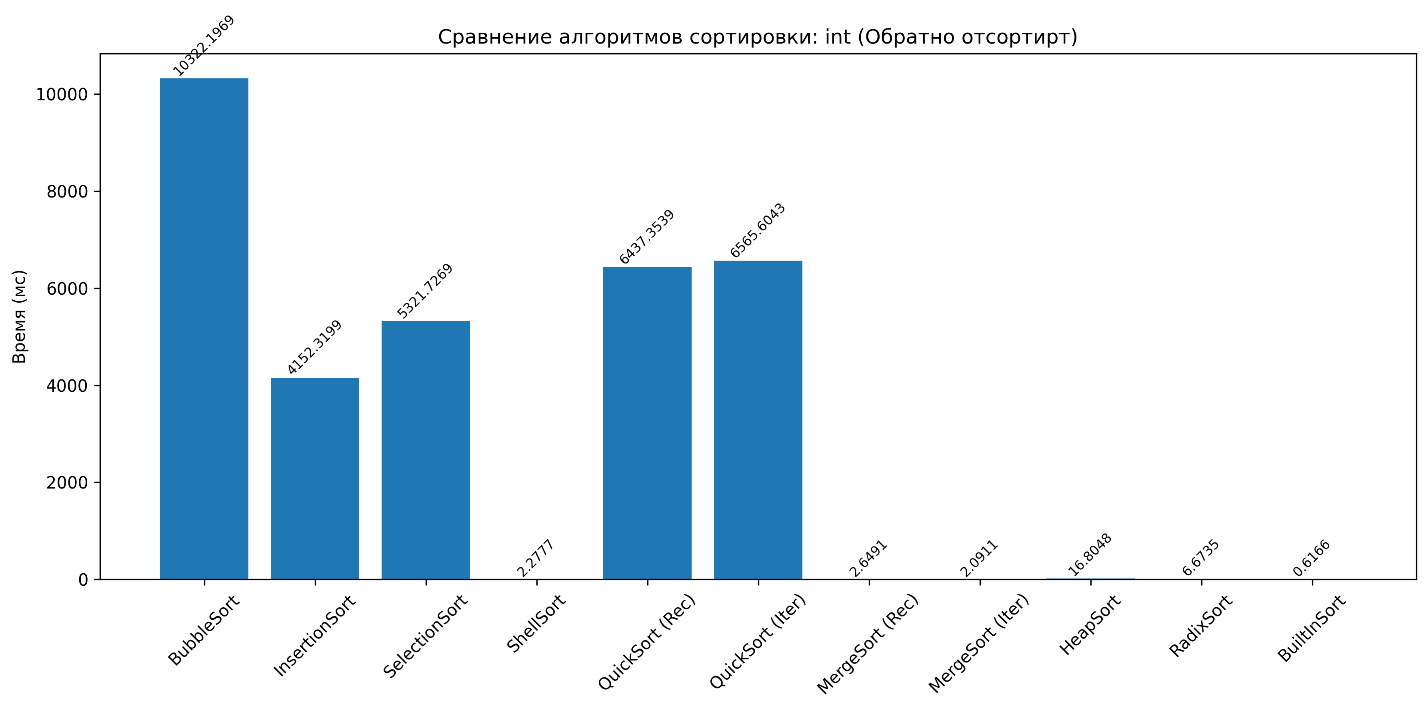
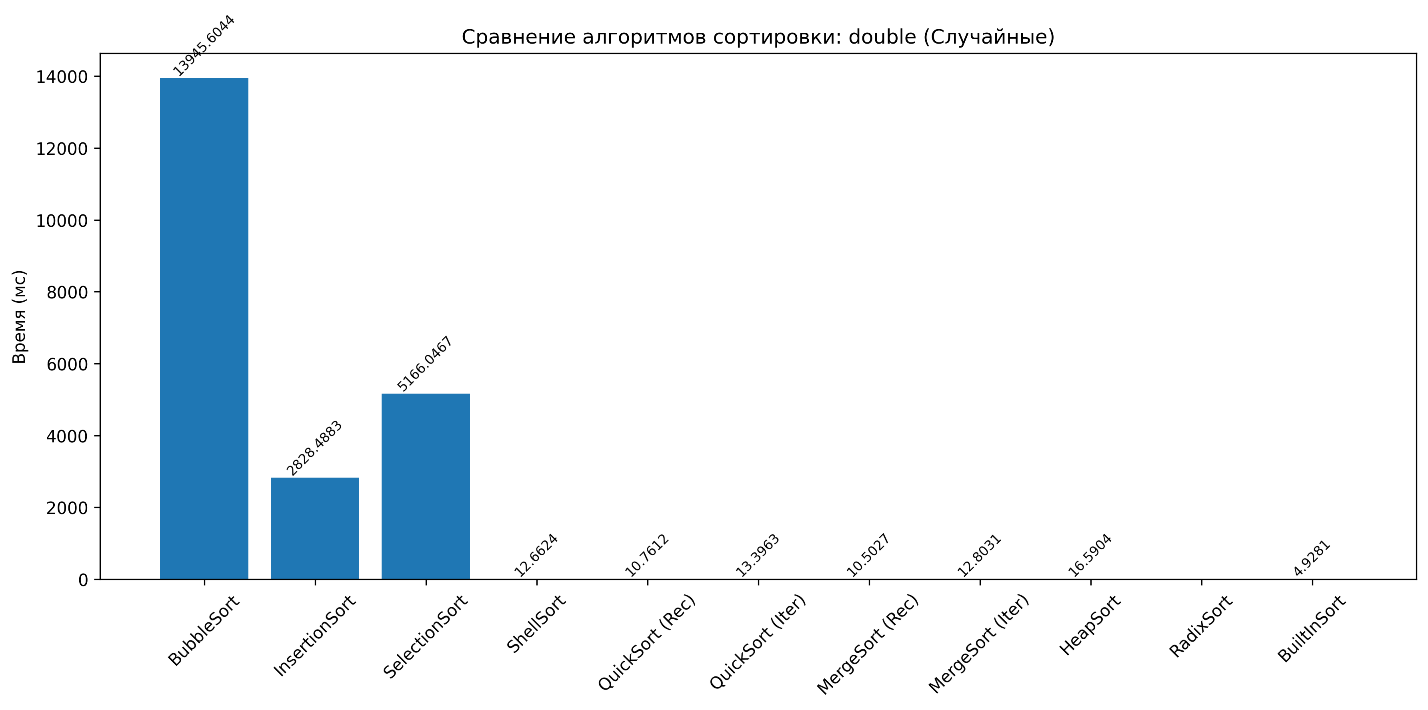
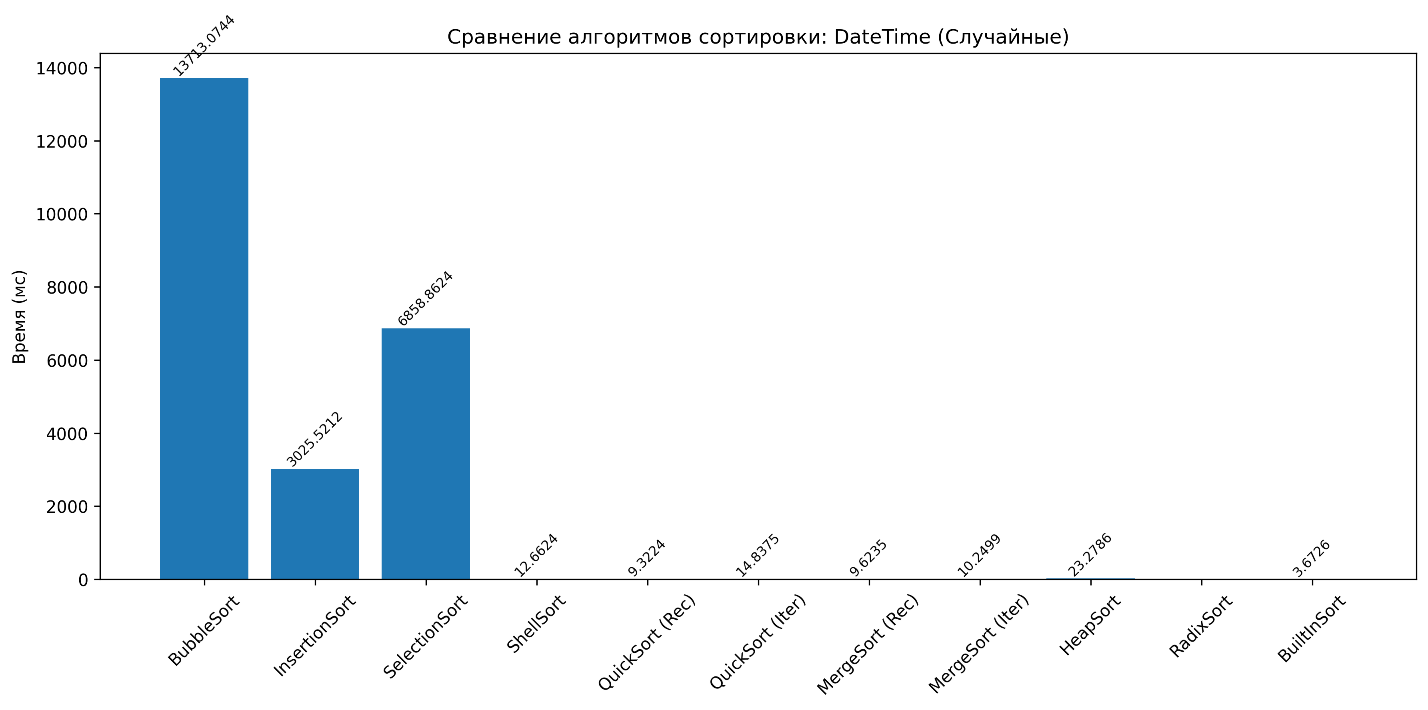
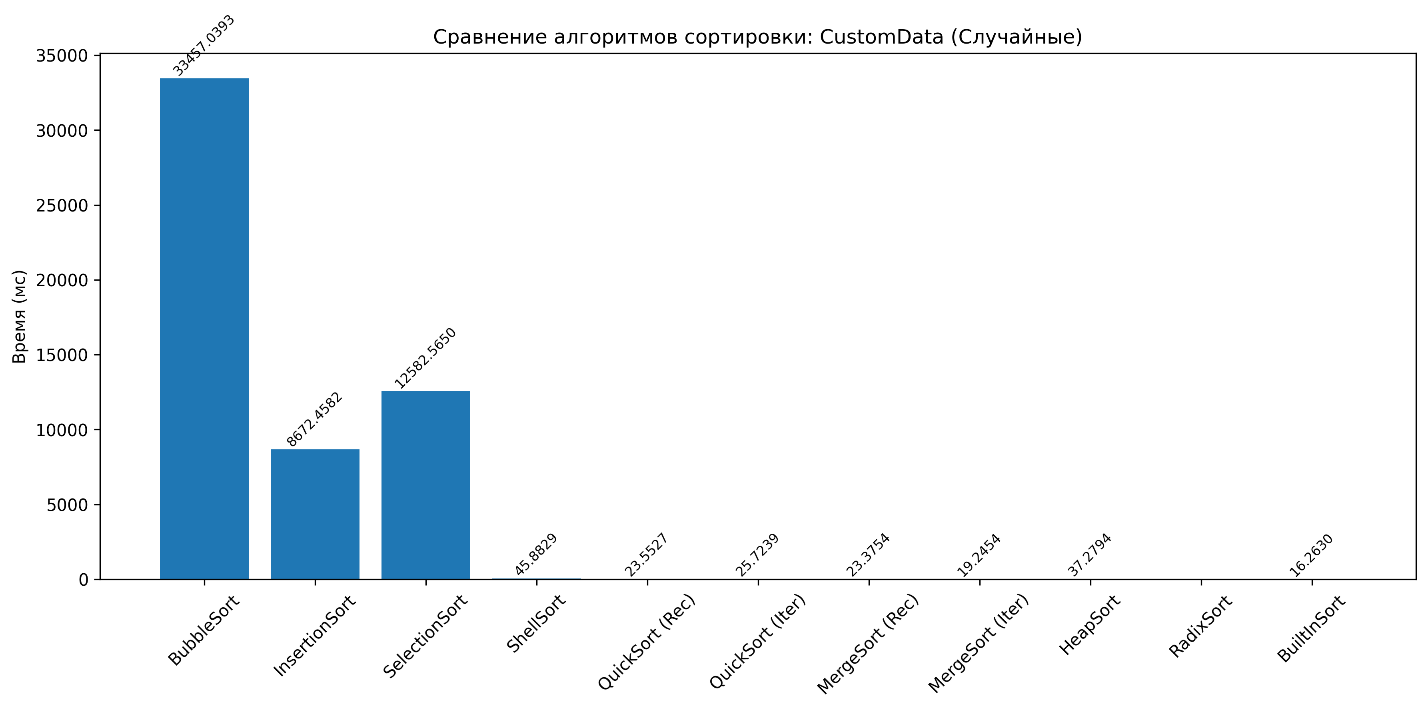
Сортировка строк приводит к резкому увеличению времени выполнения для простых алгоритмов. BubbleSort, InsertionSort и SelectionSort оказываются непригодными из-за высокой стоимости операций сравнения. Встроенная сортировка, напротив, показывает лучшую эффективность, благодаря специализированной оптимизации строковых операций и адаптивности Timsort.

В случае пользовательских типов данных эффективность сортировки напрямую зависит от сложности операций сравнения. Встроенная сортировка демонстрирует уверенное преимущество благодаря оптимизациям под произвольные типы. Простые алгоритмы не справляются с объемами, показывая значительные задержки даже на относительно простых структурах данных.

С датами и временем наблюдается схожая картина: эффективны встроенная сортировка, MergeSort и QuickSort. Алгоритмы с квадратной сложностью демонстрируют значительное отставание. RadixSort применим только к числовым типам и здесь не участвует, что подчёркивает его ограниченность.

Размер массива 50000 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тип данных | BubbleSort | InsertionSort | SelectionSort | ShellSort | QuickSort (Rec) | QuickSort (Iter) | MergeSort (Rec) | MergeSort (Iter) | HeapSort | RadixSort | BuiltInSort |
| int(Сл) | 8297,926 | 1228,945 | 3142,684 | 7,1715 | 5,6338 | 5,4865 | 7,4578 | 5,9251 | 9,9276 | 2,5126 | 2,1012 |
| int(сорт) | 0,1419 | 0,1313 | 3190,112 | 1,2036 | 5352,51 | 5295,665 | 2,5999 | 1,7205 | 14,7896 | 6,2603 | 0,4352 |
| int(обрсорт) | 10322,2 | 4152,32 | 5321,727 | 2,2777 | 6437,354 | 6565,604 | 2,6491 | 2,0911 | 16,8048 | 6,6735 | 0,6166 |
| int(дубл) | 13623,44 | 2030,081 | 5308,632 | 5,5469 | 882,0931 | 831,8678 | 8,4283 | 7,5982 | 14,3364 | 1,3458 | 1,7477 |
| int(чупор) | 7600,587 | 66,0433 | 5915,406 | 10,4364 | 61,0041 | 38,1053 | 5,935 | 4,7174 | 14,3899 | 6,3699 | 0,6005 |
| double | 13945,6 | 2828,488 | 5166,047 | 12,6624 | 10,7612 | 13,3963 | 10,5027 | 12,8031 | 16,5904 | N/A | 4,9281 |
| string | 81447,27 | 26667,72 | 49748,09 | 105,8402 | 74,3914 | 63,3112 | 82,2102 | 79,3075 | 166,6034 | N/A | 83,0079 |
| CustomData | 33457,04 | 8672,458 | 12582,57 | 45,8829 | 23,5527 | 25,7239 | 23,3754 | 19,2454 | 37,2794 | N/A | 16,263 |
| DateTime | 13713,07 | 3025,521 | 6858,862 | 12,6624 | 9,3224 | 14,8375 | 9,6235 | 10,2499 | 23,2786 | N/A | 3,6726 |



При работе с массивами большого размера особенности алгоритмов сортировки становятся особенно очевидны, что подтверждается результатами экспериментов.

Случайные целочисленные данные показывают, что квадратичные алгоритмы (BubbleSort, InsertionSort, SelectionSort) крайне неэффективны и демонстрируют колоссальное замедление, что делает их непригодными для практического использования при больших объемах. Встроенная сортировка Python и более продвинутые алгоритмы (QuickSort, MergeSort, HeapSort, RadixSort) проявляют устойчивую и высокую производительность. При этом ShellSort выделяется как метод с умеренной сложностью, демонстрируя значительно лучшее поведение по сравнению с квадратичными алгоритмами, но уступая более эффективным подходам.

Для отсортированных массивов адаптивные алгоритмы (InsertionSort, BubbleSort) сохраняют высокую производительность, однако QuickSort (в обеих версиях) испытывает значительные трудности, демонстрируя крайне низкую эффективность из-за специфики выбора опорного элемента в худшем случае. MergeSort и встроенная сортировка показывают себя стабильными и эффективными, что делает их предпочтительными для подобных сценариев.

Обратно отсортированные данные усиливают негативные эффекты для квадратичных алгоритмов, дополнительно замедляя их работу. QuickSort вновь показывает значительные ухудшения производительности, что указывает на его уязвимость к неблагоприятным входным данным. Алгоритмы с гарантированным временем выполнения, такие как MergeSort и встроенная сортировка, сохраняют конкурентоспособность и стабильность.

Массивы с дубликатами увеличивают нагрузку на алгоритмы с большим количеством сравнений, особенно SelectionSort и BubbleSort, делая их ещё менее практичными. Быстрые алгоритмы и встроенная сортировка с оптимизациями для таких случаев демонстрируют хорошую устойчивость и сохраняют приемлемое время работы. RadixSort, ориентированный на числовые данные, продолжает показывать хорошие результаты там, где применим.

Частично упорядоченные массивы вновь подчеркивают преимущества адаптивных алгоритмов, таких как InsertionSort и ShellSort, которые значительно выигрывают за счет оптимизации под частично отсортленные данные. Несмотря на это, при экстремально больших объемах встроенная сортировка остаётся самым сбалансированным и эффективным решением.

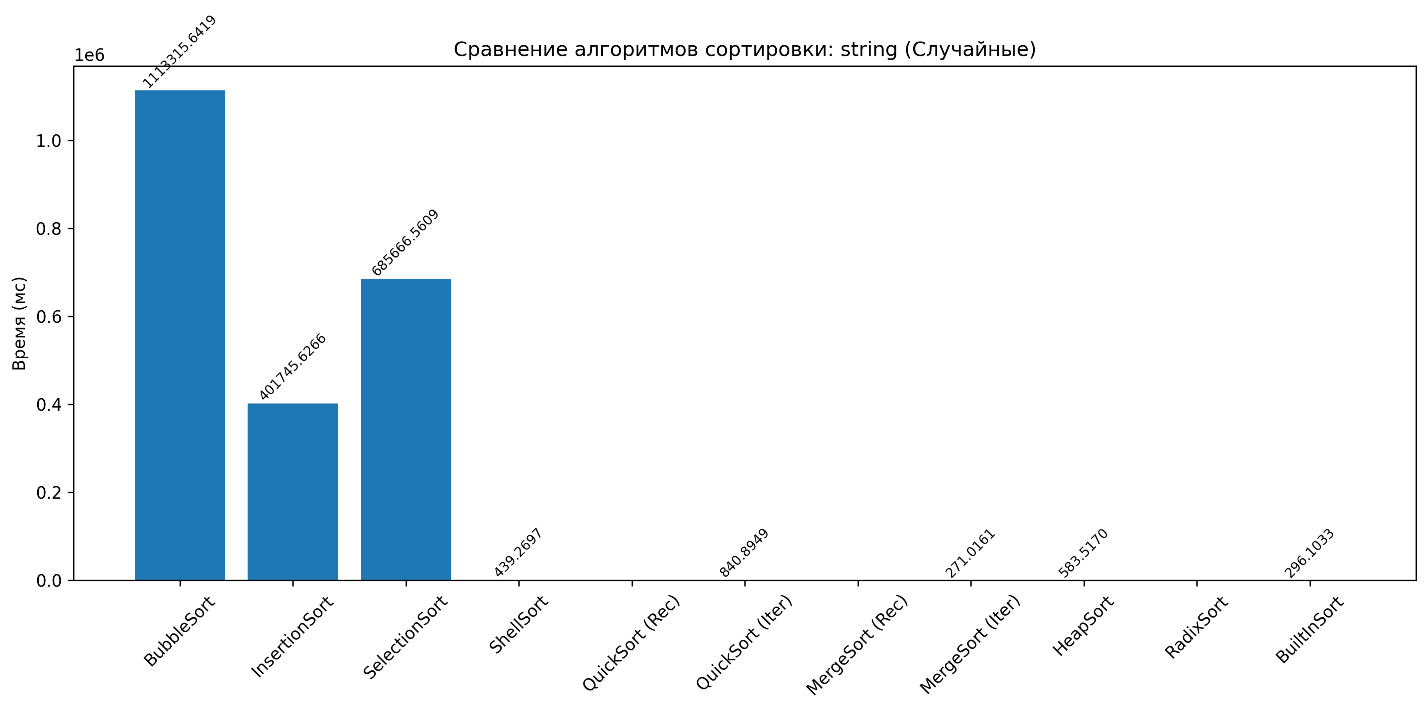
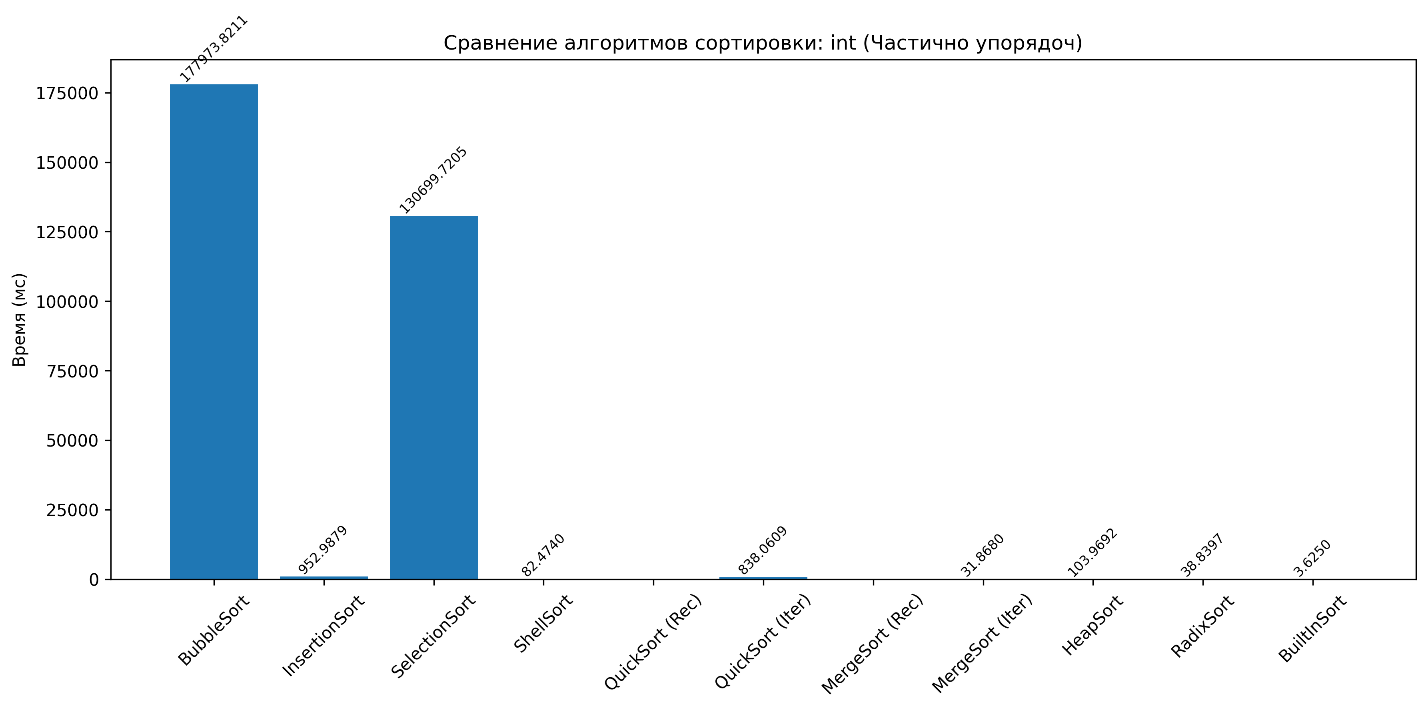
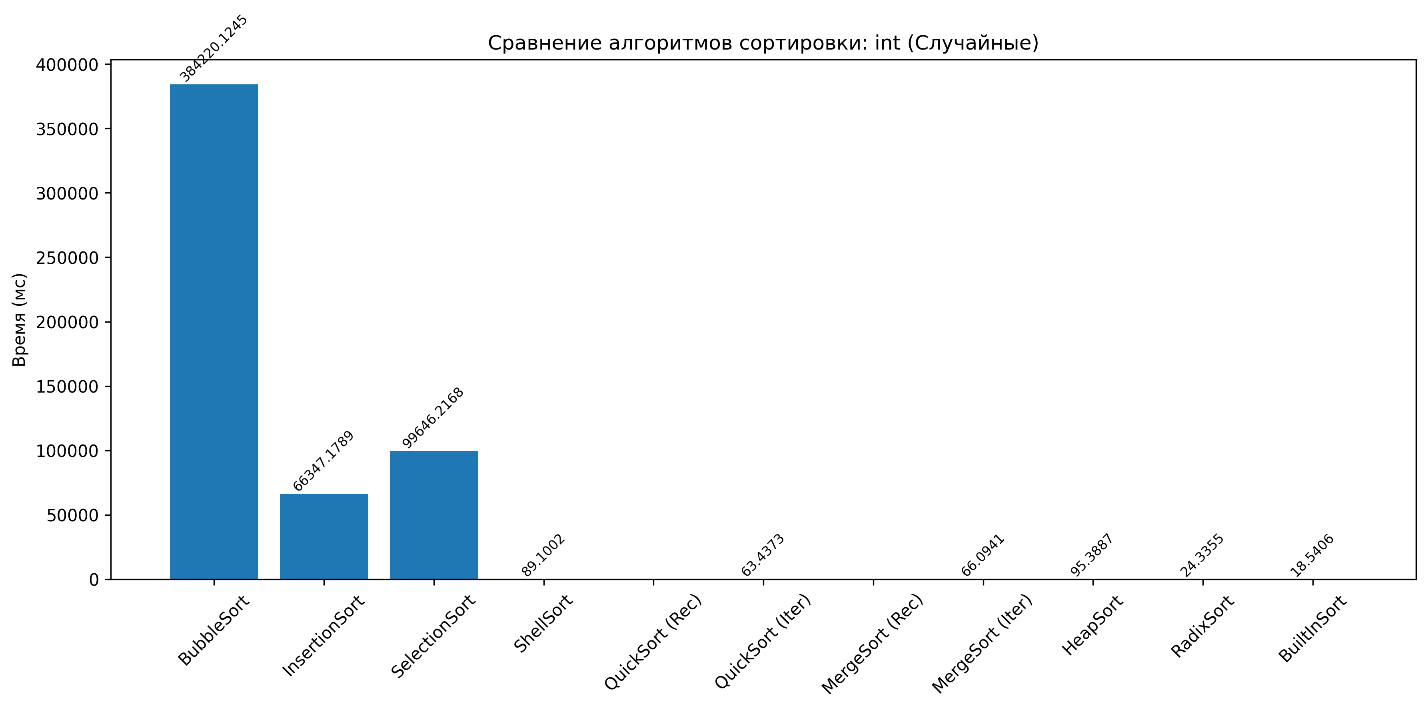
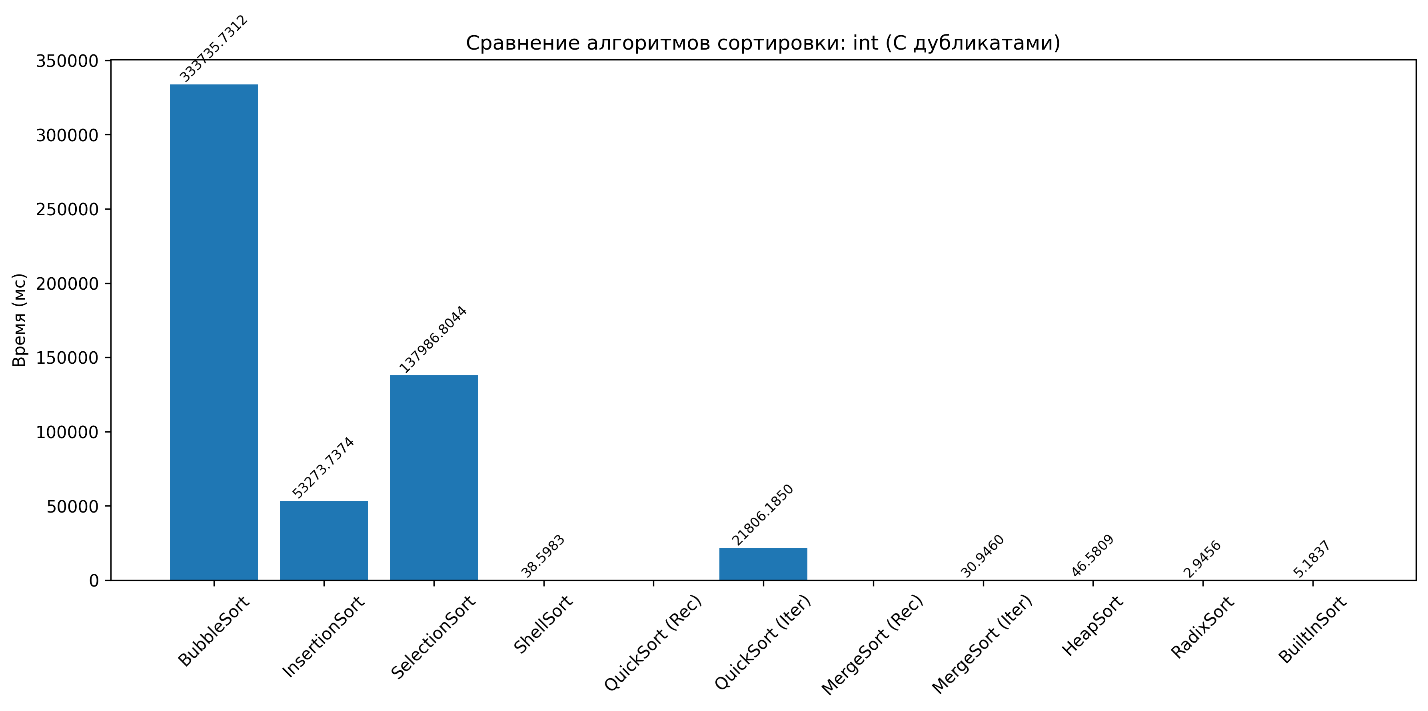
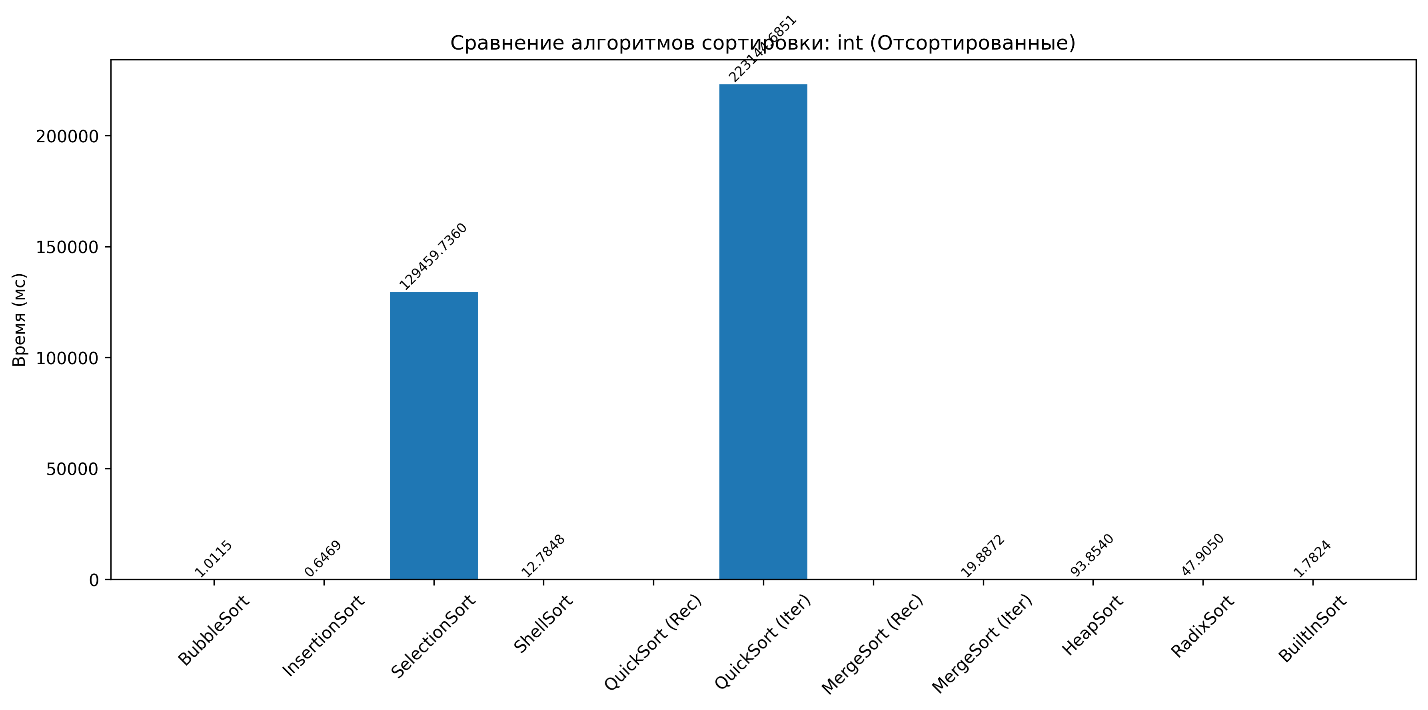
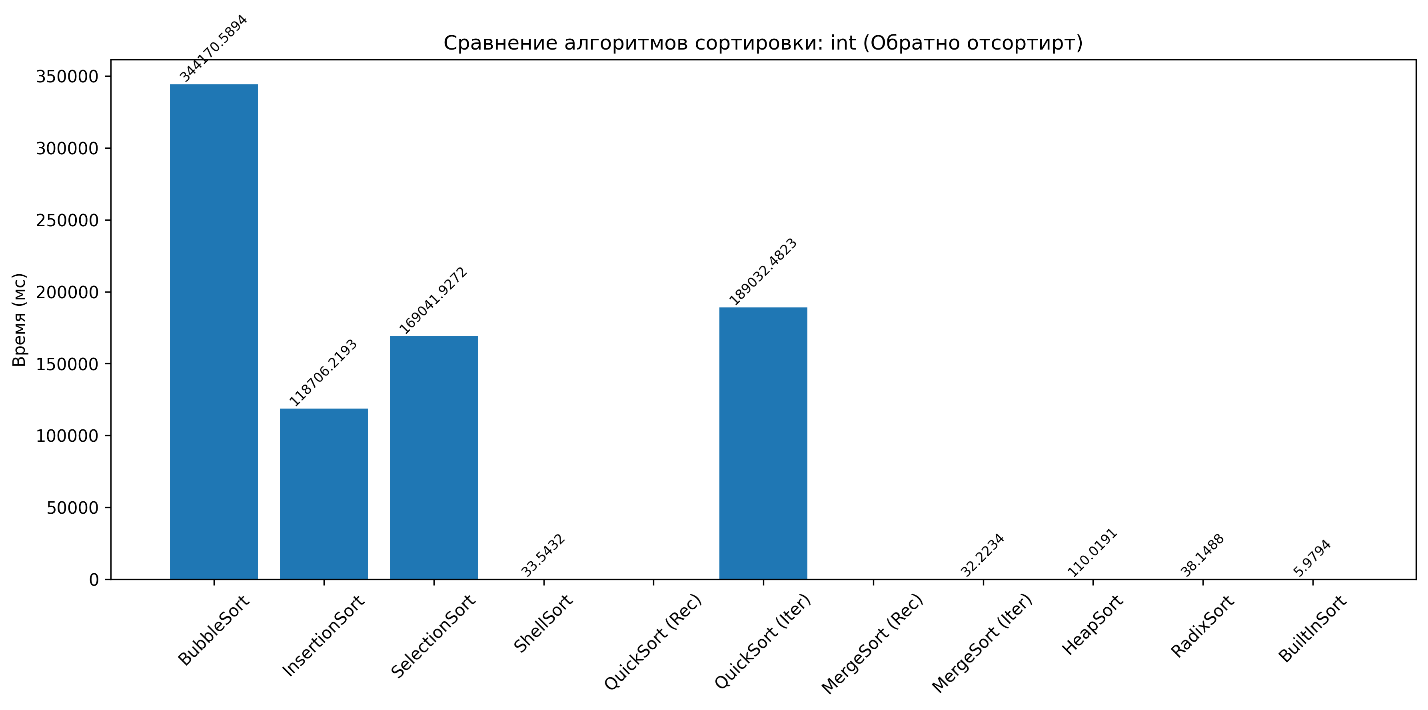
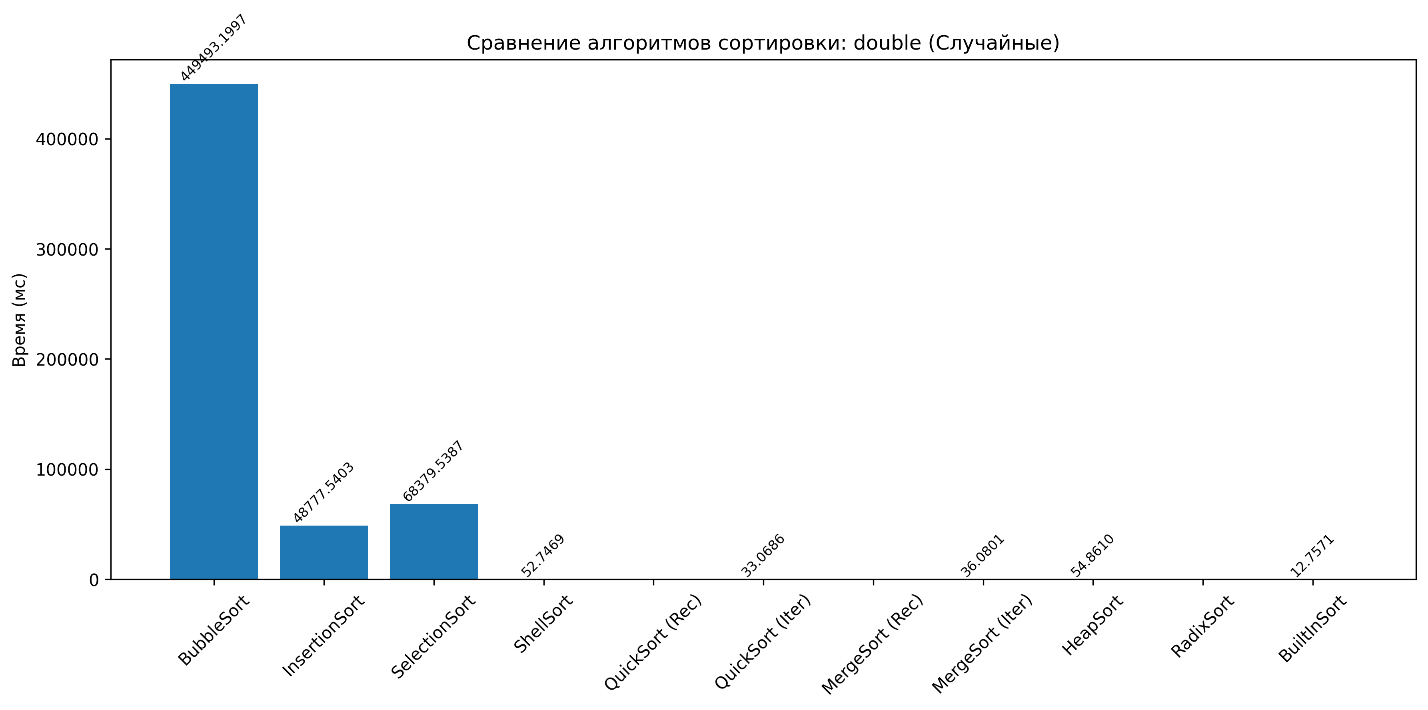
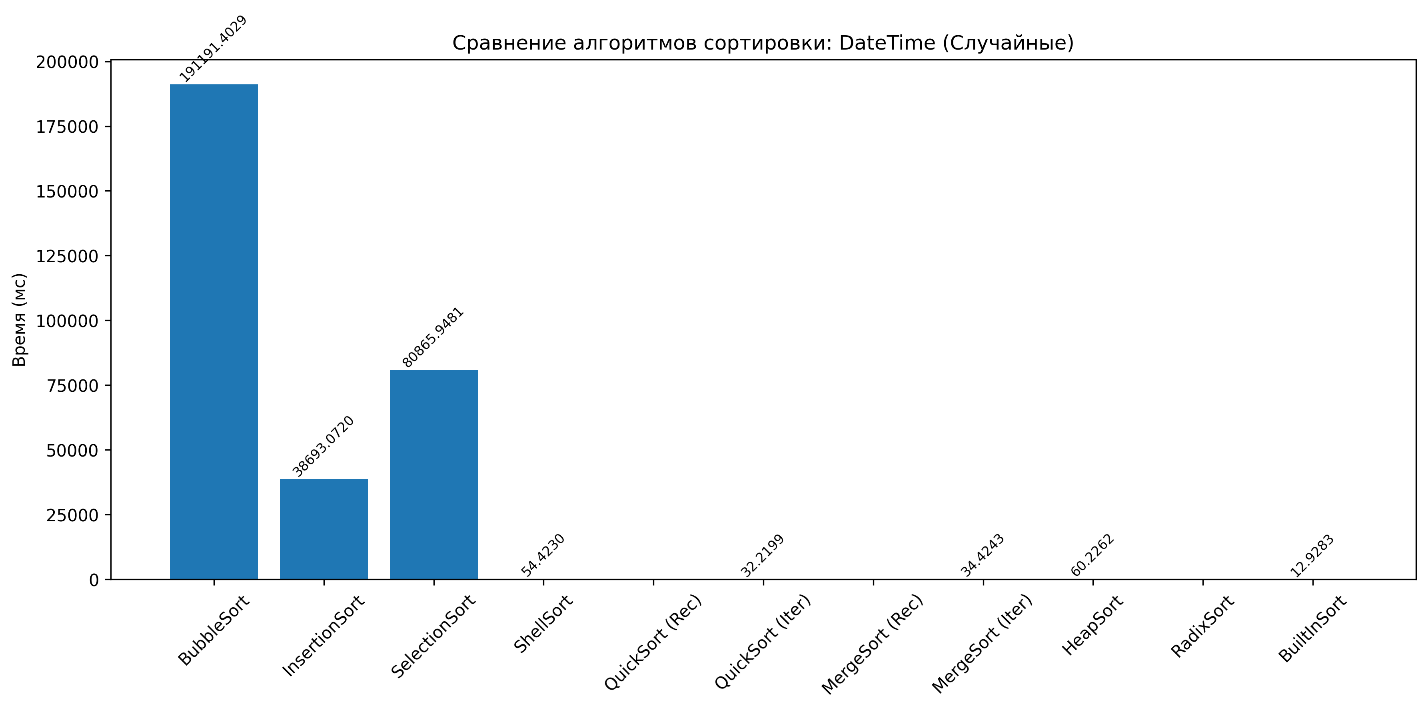
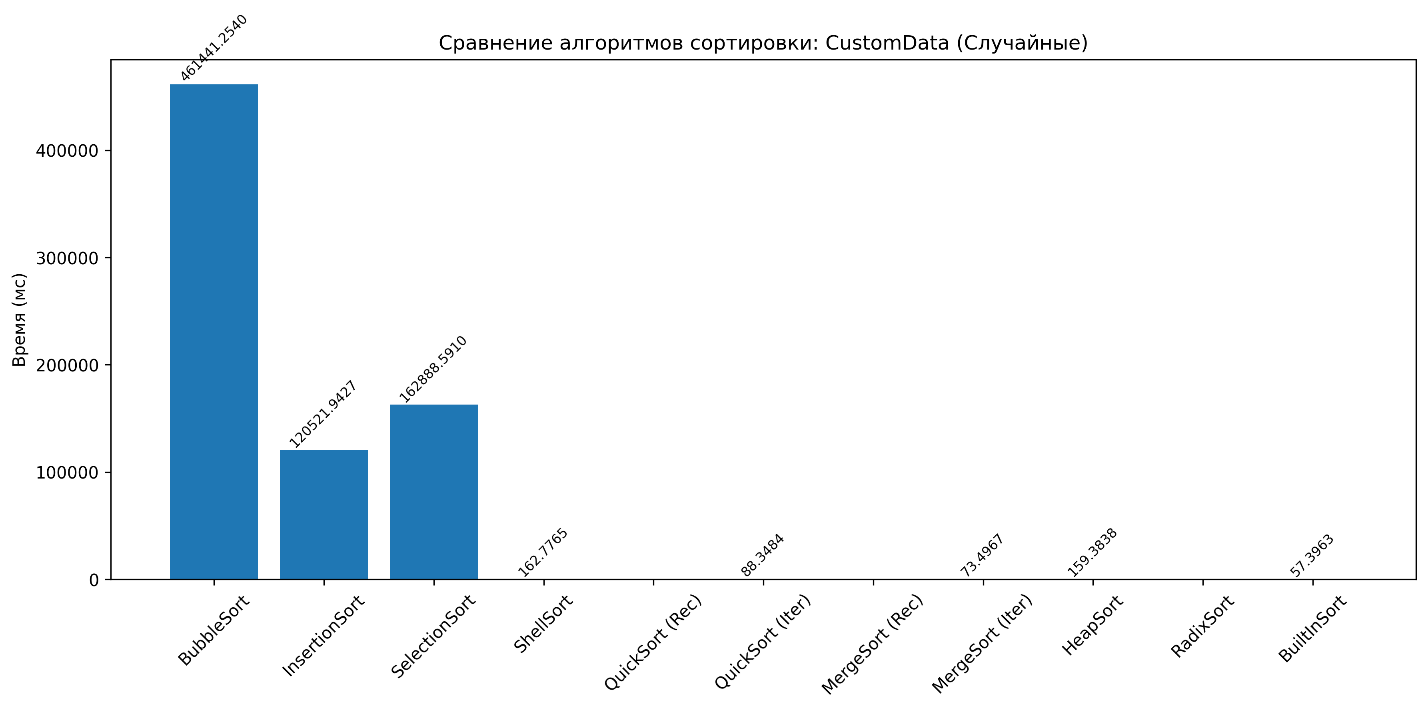
Для вещественных чисел квадратичные алгоритмы оказываются практически неприменимыми из-за значительного роста времени выполнения. Современные алгоритмы с хорошей асимптотикой, в том числе встроенная сортировка, сохраняют приемлемую производительность, что свидетельствует о правильном выборе при работе с такими типами данных.

При работе со строками наблюдается резкий рост времени выполнения для квадратичных алгоритмов, вызванный дороговизной операций сравнения строк. Встроенная сортировка и алгоритмы с логарифмической сложностью (QuickSort, MergeSort) значительно лучше справляются с нагрузкой, хотя общая стоимость операций остаётся высокой из-за особенностей строковых сравнений.

Пользовательские типы данных и объекты DateTime также подчеркивают важность эффективной реализации операций сравнения. Встроенная сортировка благодаря глубоким оптимизациям и специализированной обработке сложных объектов показывает лучшие результаты, в то время как простые алгоритмы страдают из-за высокой стоимости сравнений.

Размер массива 250000 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тип данных | BubbleSort | InsertionSort | SelectionSort | ShellSort | QuickSort(Rec) | QuickSort(Iter) | MergeSort(Rec) | MergeSort(Iter) | HeapSort | RadixSort | BuiltInSort |
| int(Сл) | 384220,1 | 66347,18 | 99646,22 | 89,1002 | INF (SOF) | 63,4373 | INF (SOF) | 66,0941 | 95,3887 | 24,3355 | 18,5406 |
| int(сорт) | 1,0115 | 0,6469 | 129459,7 | 12,7848 | INF (SOF) | 223144,7 | INF (SOF) | 19,8872 | 93,854 | 47,905 | 1,7824 |
| int(обрсорт) | 344170,6 | 118706,2 | 169041,9 | 33,5432 | INF (SOF) | 189032,5 | INF (SOF) | 32,2234 | 110,0191 | 38,1488 | 5,9794 |
| int(дубл) | 333735,7 | 53273,74 | 137986,8 | 38,5983 | INF (SOF) | 21806,19 | INF (SOF) | 30,946 | 46,5809 | 2,9456 | 5,1837 |
| int(чупор) | 177973,8 | 952,9879 | 130699,7 | 82,474 | INF (SOF) | 838,0609 | INF (SOF) | 31,868 | 103,9692 | 38,8397 | 3,625 |
| double | 449493,2 | 48777,54 | 68379,54 | 52,7469 | INF (SOF) | 33,0686 | INF (SOF) | 36,0801 | 54,861 | N/A | 12,7571 |
| string | 1113316 | 401745,6 | 685666,6 | 439,2697 | INF (SOF) | 840,8949 | INF (SOF) | 271,0161 | 583,517 | N/A | 296,1033 |
| CustomData | 461441,3 | 120521,9 | 162888,6 | 162,7765 | INF (SOF) | 88,3484 | INF (SOF) | 73,4967 | 159,3838 | N/A | 57,3963 |
| DateTime | 191191,4 | 38693,07 | 80865,95 | 54,423 | INF (SOF) | 32,2199 | INF (SOF) | 34,4243 | 60,2262 | N/A | 12,9283 |



Результаты экспериментов с большими массивами демонстрируют резкое ухудшение производительности квадратичных алгоритмов (BubbleSort, InsertionSort, SelectionSort), которые показывают астрономические времена работы, достигающие сотен тысяч миллисекунд. Это подтверждает их полную непригодность для обработки больших объемов данных.

Рекурсивные реализации QuickSort и MergeSort сталкиваются с критическими проблемами переполнения стека (INF (SOF)), что делает их использование невозможным на таких масштабах. В то же время итеративные версии этих алгоритмов демонстрируют устойчивую и приемлемую производительность, что указывает на необходимость выбора именно таких реализаций при работе с большими массивами.

Для целочисленных данных итеративный QuickSort (63,4373 мс) и MergeSort (19,8872 мс) показывают стабильные и быстрые результаты. RadixSort сохраняет конкурентоспособность (24,3355 мс), а встроенная сортировка Python остается одной из самых быстрых (18,5406 мс), демонстрируя высокую оптимизацию.

В случае с отсортированными массивами квадратичные алгоритмы работают быстро лишь в ограниченных случаях (InsertionSort и BubbleSort), но SelectionSort и рекурсивные алгоритмы продолжают страдать от высокой стоимости. Итеративные реализации продолжают показывать стабильную эффективность.

При работе с обратно отсортированными массивами квадратичные алгоритмы значительно замедляются, а рекурсивные версии QuickSort и MergeSort испытывают сбои из-за переполнения стека. Итеративные варианты сохраняют приемлемое время выполнения.

Для данных с дубликатами и частично упорядоченных массивов наблюдается схожая картина: квадратичные алгоритмы непригодны, а итеративные и встроенные методы остаются наиболее эффективными.

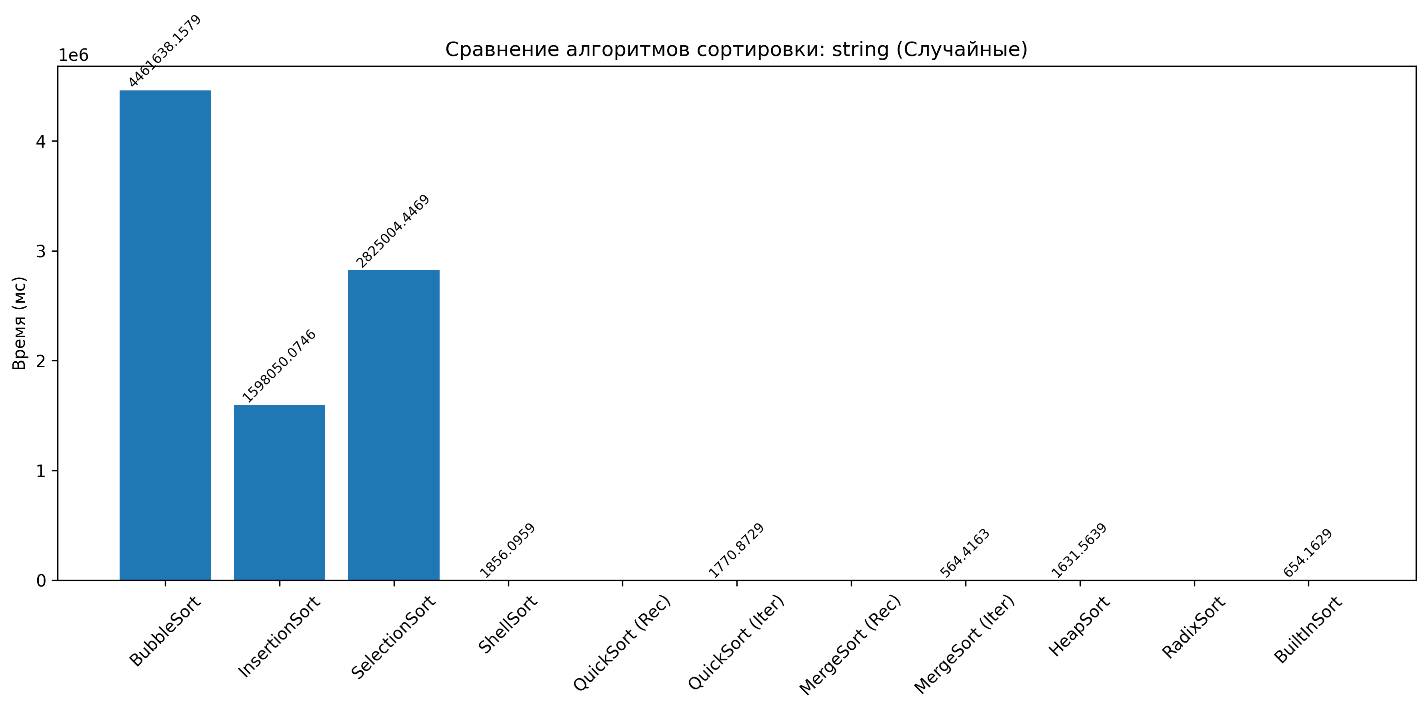
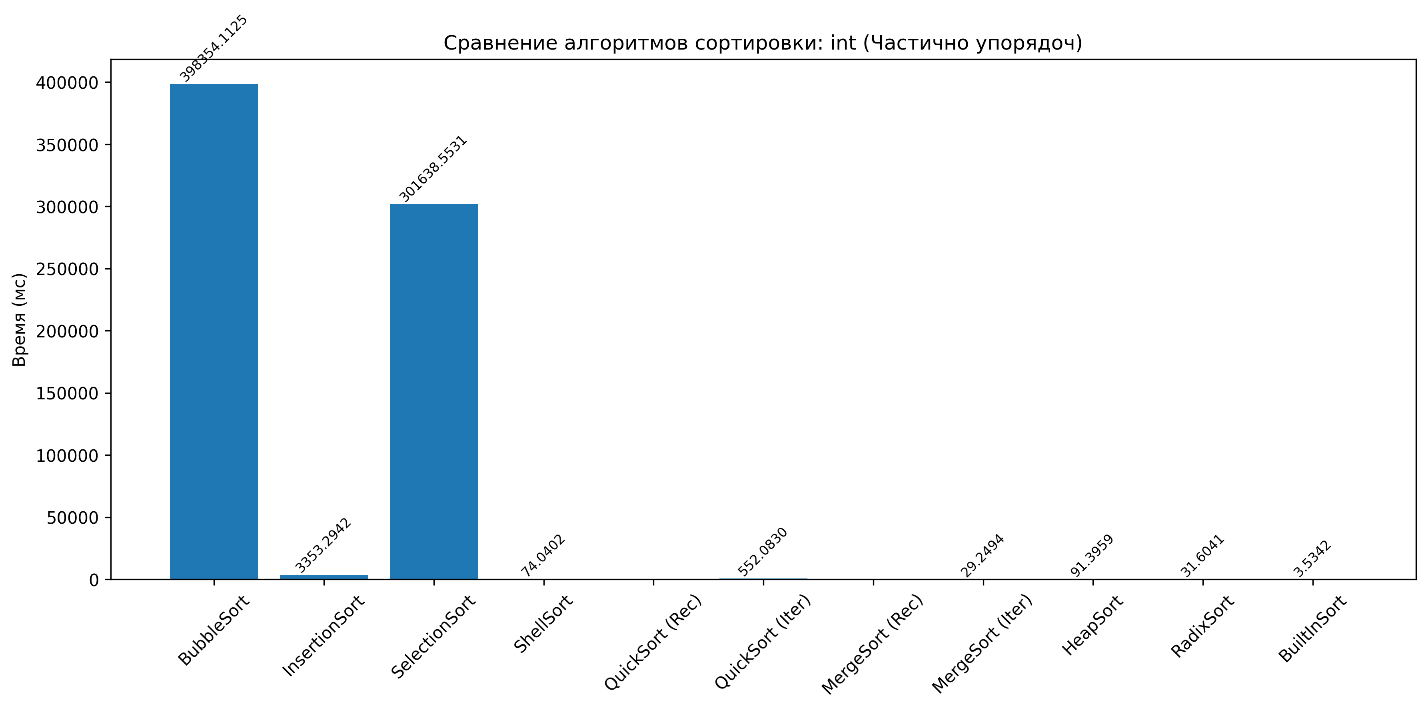
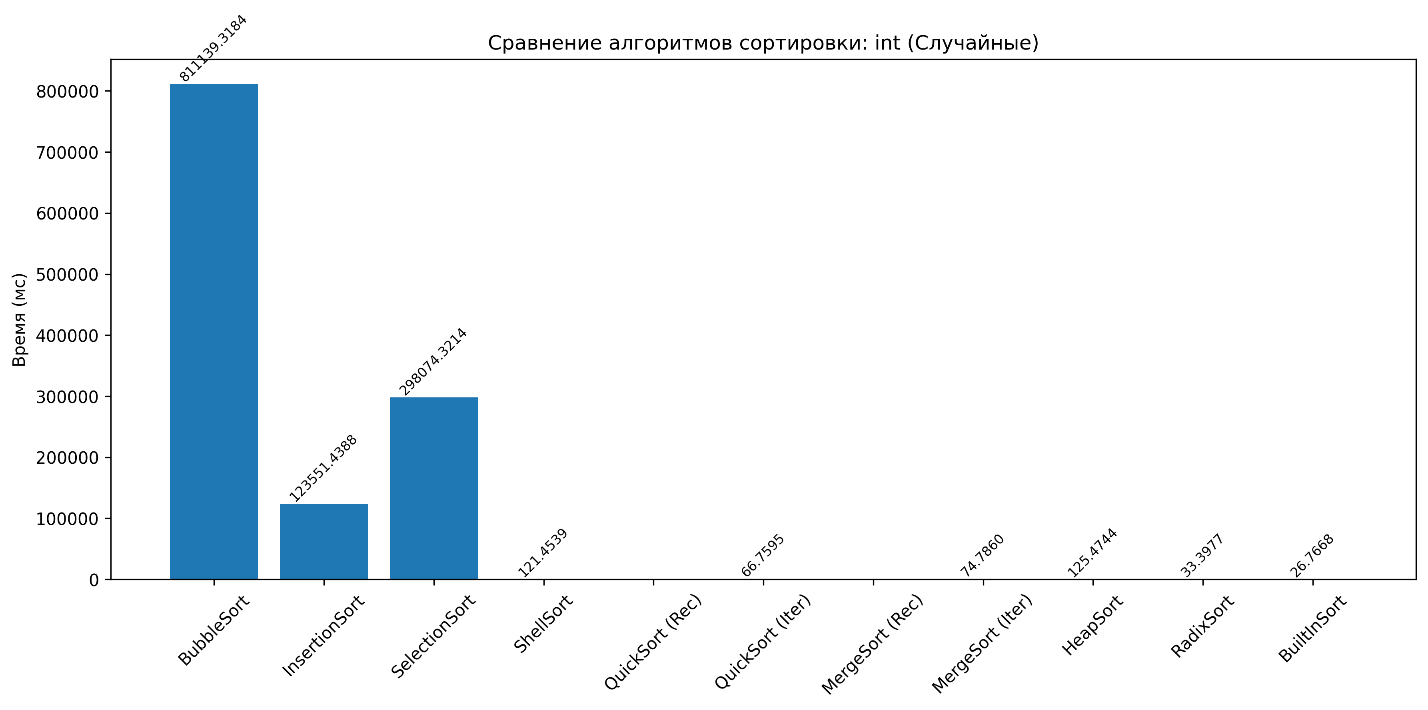
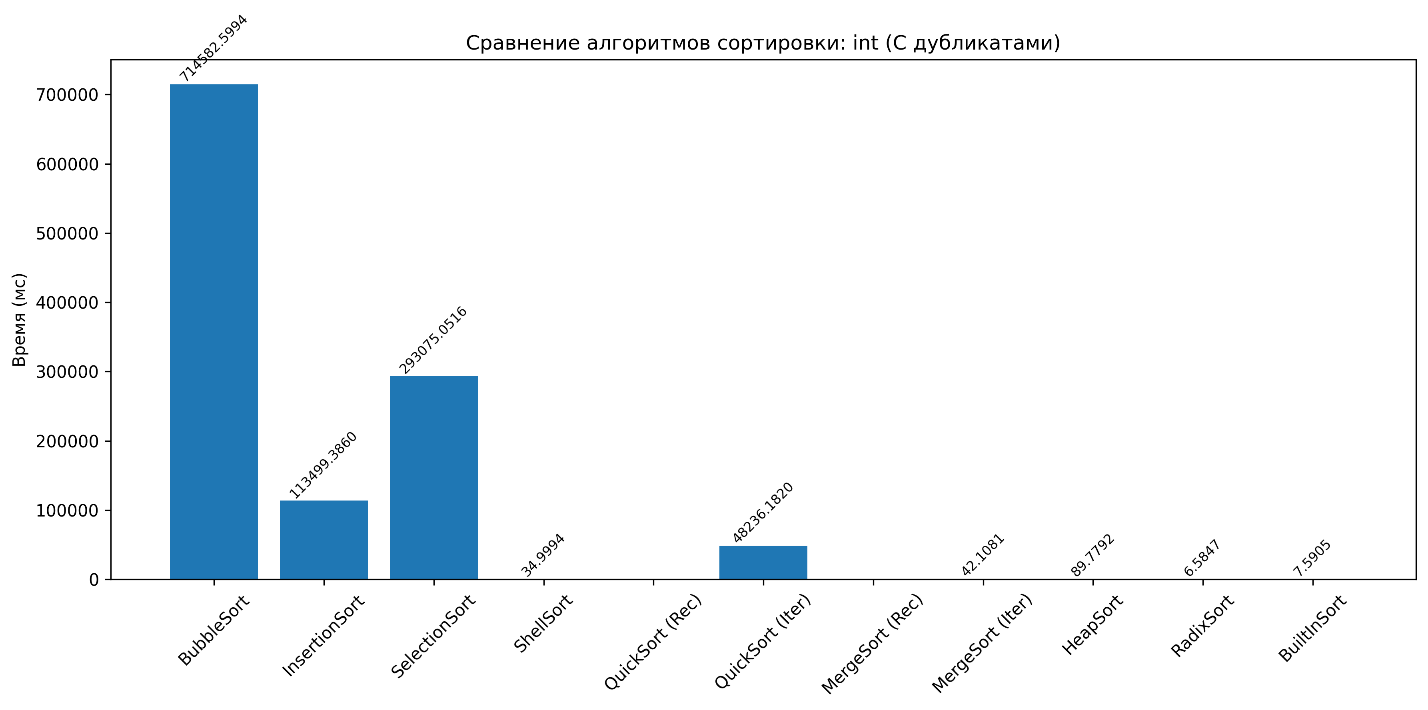
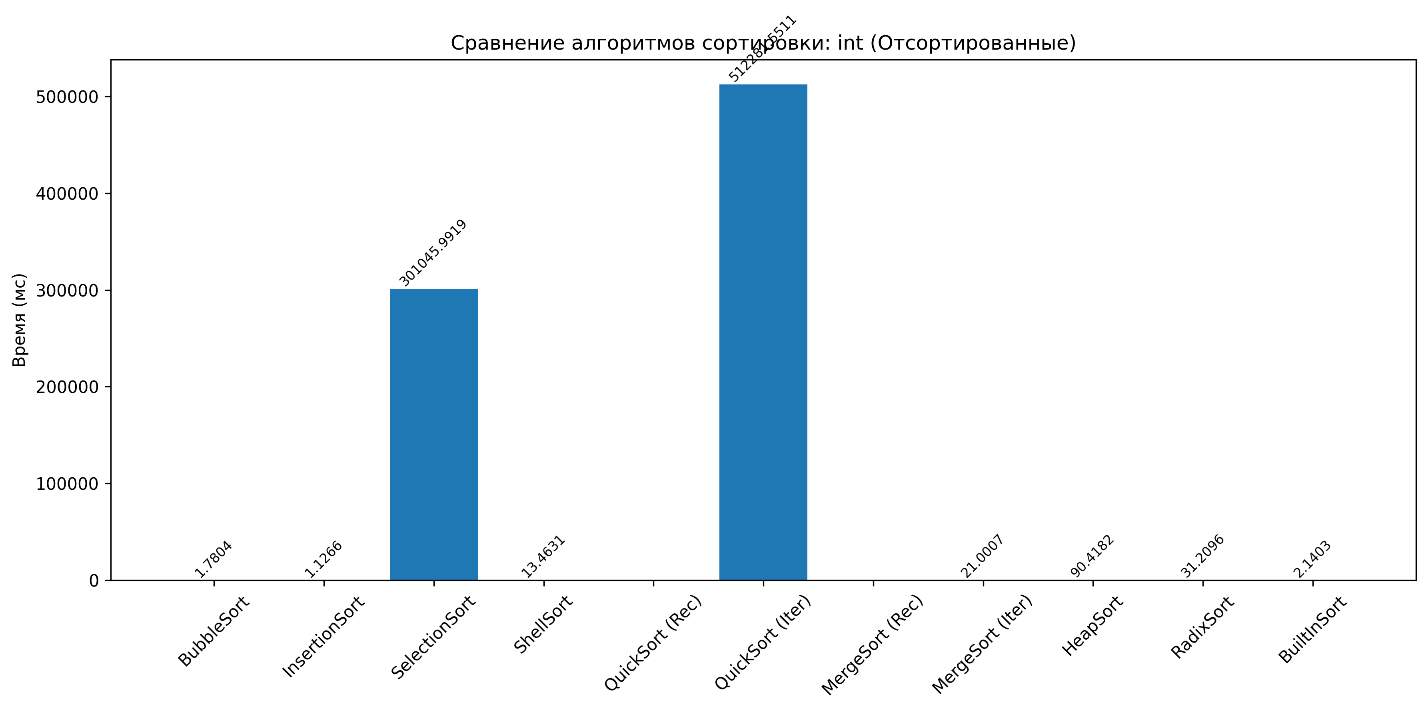
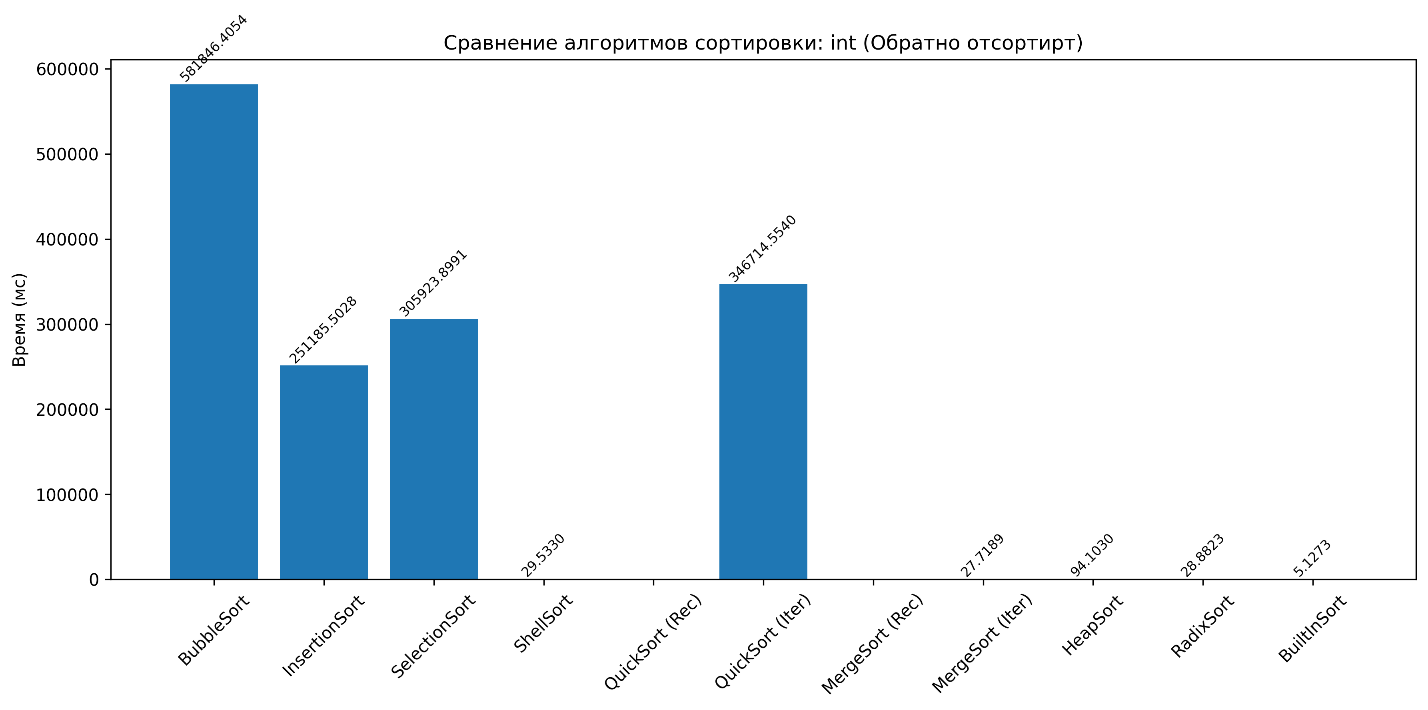
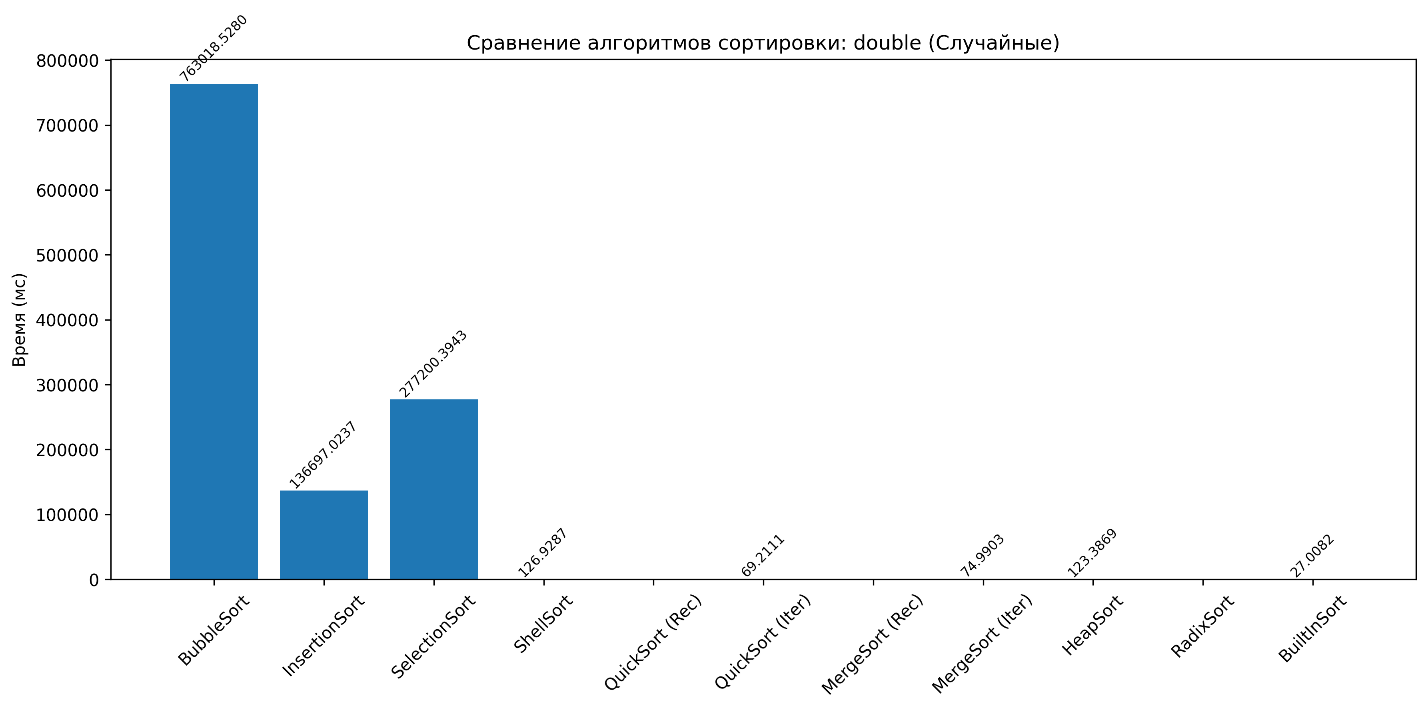
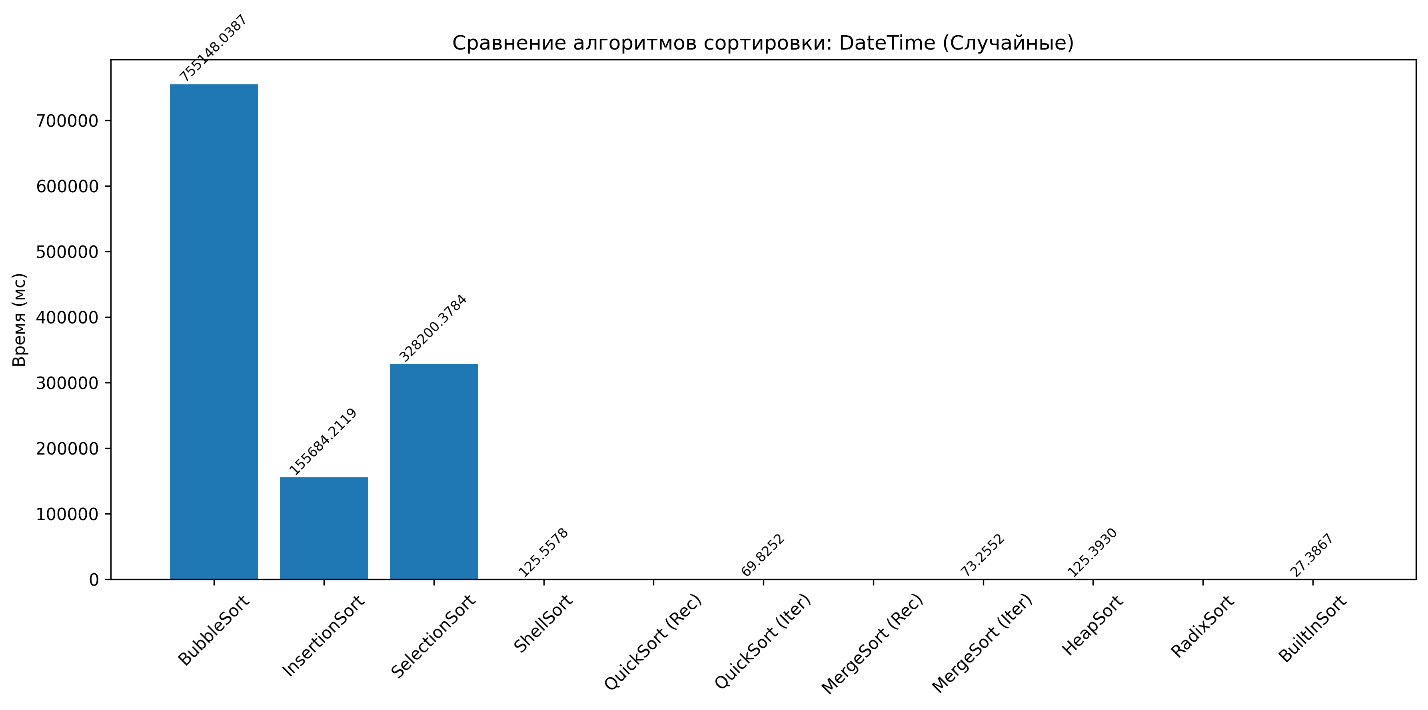
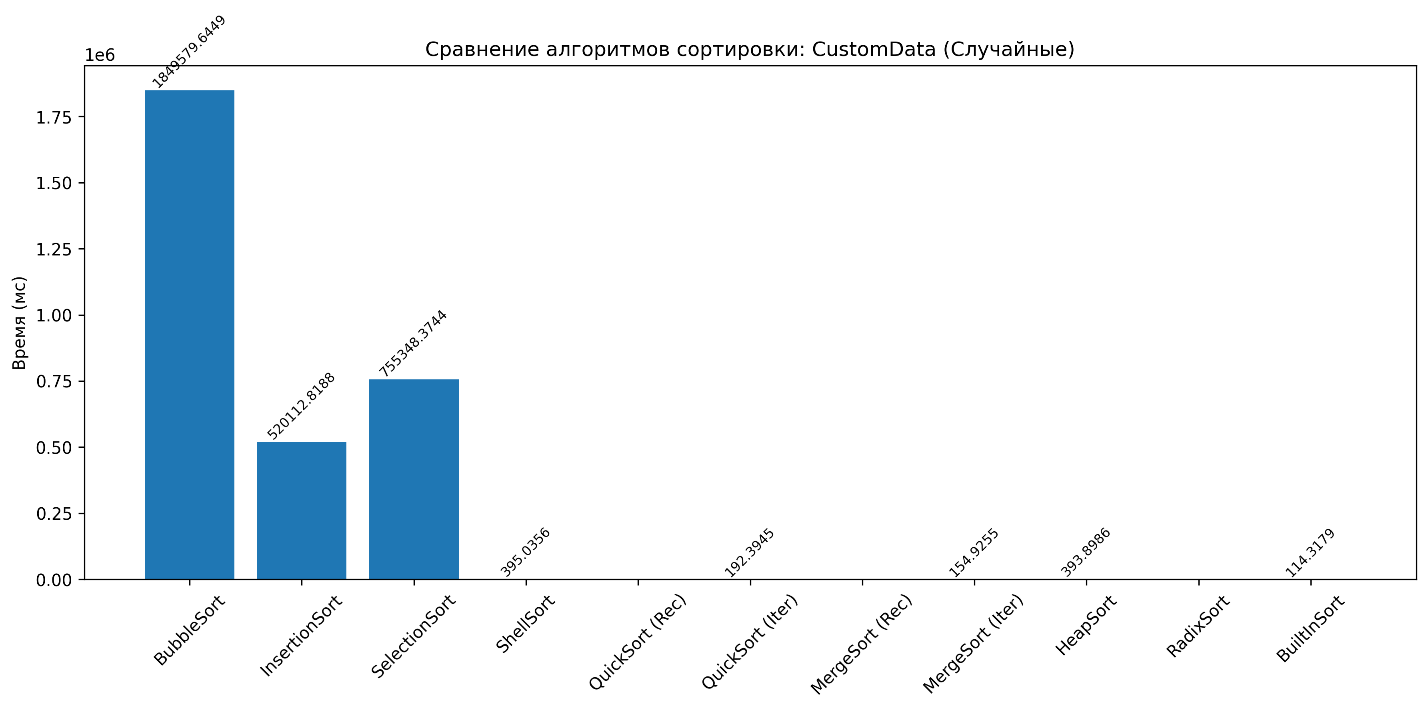
С вещественными числами и строками общая тенденция сохраняется: встроенная сортировка эффективна, но проигрывает по скорости итеративным QuickSort и MergeSort. Строковые данные требуют существенно больших затрат времени, что связано с дороговизной операций сравнения.

Для пользовательских типов данных и дат производительность определяется стоимостью сравнения объектов. Итеративные версии алгоритмов и встроенная сортировка показывают лучшие результаты, тогда как квадратичные методы оказываются крайне неэффективными.

В целом, эксперименты подтверждают необходимость использования итеративных реализаций алгоритмов сортировки и встроенных высоко оптимизированных функций при работе с большими объемами данных. Рекурсивные алгоритмы и квадратичные методы на таких масштабах не подходят.

Размер массива 500000 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тип данных | BubbleSort | InsertionSort | SelectionSort | ShellSort | QuickSort(Rec) | QuickSort(Iter) | MergeSort(Rec) | MergeSort(Iter) | HeapSort | RadixSort | BuiltInSort |
| int(Сл) | 811139,3 | 123551,4 | 298074,3 | 121,4539 | INF (SOF) | 66,7595 | INF (SOF) | 74,786 | 125,4744 | 33,3977 | 26,7668 |
| int(сорт) | 1,7804 | 1,1266 | 301046 | 13,4631 | INF (SOF) | 512281,6 | INF (SOF) | 21,0007 | 90,4182 | 31,2096 | 2,1403 |
| int(обрсорт) | 581846,4 | 251185,5 | 305923,9 | 29,533 | INF (SOF) | 346714,6 | INF (SOF) | 27,7189 | 94,103 | 28,8823 | 5,1273 |
| int(дубл) | 714582,6 | 113499,4 | 293075,1 | 34,9994 | INF (SOF) | 48236,18 | INF (SOF) | 42,1081 | 89,7792 | 6,5847 | 7,5905 |
| int(чупор) | 398354,1 | 3353,294 | 301638,6 | 74,0402 | INF (SOF) | 552,083 | INF (SOF) | 29,2494 | 91,3959 | 31,6041 | 3,5342 |
| double | 763018,5 | 136697 | 277200,4 | 126,9287 | INF (SOF) | 69,2111 | INF (SOF) | 74,9903 | 123,3869 | N/A | 27,0082 |
| string | 4461638 | 1598050 | 2825004 | 1856,096 | INF (SOF) | 1770,873 | INF (SOF) | 564,4163 | 1631,564 | N/A | 654,1629 |
| CustomData | 1849580 | 520112,8 | 755348,4 | 395,0356 | INF (SOF) | 192,3945 | INF (SOF) | 154,9255 | 393,8986 | N/A | 114,3179 |
| DateTime | 755148 | 155684,2 | 328200,4 | 125,5578 | INF (SOF) | 69,8252 | INF (SOF) | 73,2552 | 125,393 | N/A | 27,3867 |



Результаты экспериментов на очень больших массивах данных продолжают подтверждать ключевые закономерности и выявляют новые особенности.

Квадратичные алгоритмы (BubbleSort, InsertionSort, SelectionSort) демонстрируют крайне низкую производительность, с временами сортировки, исчисляемыми сотнями тысяч и даже миллионами миллисекунд. Это ещё раз подчёркивает их непригодность для масштабных задач.

Рекурсивные версии QuickSort и MergeSort сталкиваются с критическим ограничением – переполнением стека (INF (SOF)), что полностью исключает возможность их применения для таких объёмов данных. В отличие от них, итеративные реализации QuickSort и MergeSort сохраняют устойчивость и показывают приемлемое время работы, хотя и значительно выше по сравнению с меньшими массивами.

Итеративный QuickSort демонстрирует время в диапазоне от десятков до сотен миллисекунд в зависимости от типа и структуры данных, что значительно лучше рекурсивных аналогов. Итеративный MergeSort также стабильно работает с массивами большого размера, при этом показывая производительность, сопоставимую с QuickSort.

RadixSort сохраняет высокую эффективность для целочисленных и вещественных данных, однако с увеличением размера массива его время работы заметно возрастает, но остаётся конкурентоспособным.

Встроенная сортировка Python продолжает демонстрировать высокую оптимизацию и относительно низкое время выполнения, что делает её предпочтительным выбором для большинства практических задач.

Анализ по типам данных показывает:

Для целочисленных массивов результаты аналогичны общему тренду: квадратичные алгоритмы крайне медленные, рекурсивные алгоритмы не применимы, итеративные и встроенные решения – наиболее эффективны.

Отсортированные данные выгодно ускоряют работу BubbleSort и InsertionSort, но рекурсивные алгоритмы страдают от переполнения стека.

Обратно отсортированные и частично упорядоченные массивы приводят к значительному замедлению квадратичных методов, при этом итеративные методы продолжают работать стабильно.

Для вещественных чисел ситуация похожа на целочисленные данные, с отсутствием RadixSort (N/A) и сохранением высокой производительности встроенной сортировки.

Строковые данные требуют значительно больше времени на сортировку, что связано с дороговизной операций сравнения и копирования. Итеративные алгоритмы и встроенная сортировка остаются лучшими вариантами.

Пользовательские типы данных и даты демонстрируют, что сложность сравнения объектов напрямую влияет на время работы алгоритмов. Итеративные методы и встроенная сортировка показывают устойчивость к усложнённым операциям.

В целом, экспериментально подтверждается необходимость отказа от рекурсивных реализаций на больших данных в пользу итеративных, а также использование встроенных высоко оптимизированных сортировок Python для достижения максимальной эффективности.

Заключение

В результате проведенной работы были рассмотрены и протестированы различные алгоритмы сортировки с целью выявления наиболее эффективных методов обработки данных разных типов данных и размерностей. Результаты тестирований показали, что выбор алгоритма сортировки сильно зависит от типа, распределения и размерности исходных данных.

Алгоритмы с квадратичной временной сложностью (BubbleSort, InsertionSort, SelectionSort) проявили приемлемую производительность только на небольших объемах данных и в определённых условиях, например, при частично отсортированных массивах. При увеличении размера данных их эффективность существенно снижается, что делает их непрактичными для крупных наборов.

Рекурсивные варианты QuickSort и MergeSort показали хорошие результаты на средних объемах данных, однако столкнулись с ограничениями, связанными с переполнением стека при очень больших наборах. Итеративные реализации этих алгоритмов оказались более надежными и сохраняли высокую производительность даже на больших объемах.

Алгоритм RadixSort, ориентированный на целочисленные данные, сохранял стабильную и быструю работу на больших объемах, но его применение ограничено специфическим набором типов данных.

Особое внимание уделялось встроенной сортировке языка C#, которая продемонстрировала высокую оптимизацию и стабильность в работе с разными типами и объемами данных. В большинстве случаев она превосходит самостоятельно реализованные алгоритмы по скорости и надежности, что ставит под вопрос целесообразность разработки собственных сортировок, за исключением узкоспециализированных задач.

Таким образом, выбор алгоритма сортировки должен основываться не только на теоретической сложности, но и учитывать особенности данных, их распределение, объем, а также ограничения среды выполнения. Встроенные механизмы сортировки чаще всего являются оптимальным решением для повседневных задач, однако в условиях работы с большими объемами или нестандартными типами данных могут потребоваться специализированные или итеративные алгоритмы.

В итоге:

- Лучше всего использовать встроенную сортировку C# для большинства задач

- При работе с большими целочисленными массивами лучше использовать Radix Sort

- Избегать рекурсивные реализации алгоритмов на больших объемах данных

Приложение 1. Программный код

<https://github.com/KiRilLTatar/Test_Sorting>