```
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv('data.csv', sep=';')
```

Micro README:

- В этом ноутбуке отсутствуют графики (кол-во эл. операций; время работы). К сожалению, у меня не хватило времени, чтобы их построить, потому что **.ipynb** в PyCharm, как я понял, не очень дружит с копированием cell'ов. Получить требуемые графики можно, если провести в строке **plt.plot(x['Amount'], x['Time'], label)** следующую замену: **x['Time'] -> x['Operations']**.
- fill0To4100 - заполнение массива случайными значениями от 0 до **4100**! К сожалению, заметил ошибку в описании метода генерации массива слишком поздно, чтобы исправить **data.csv** и все cell'ы...

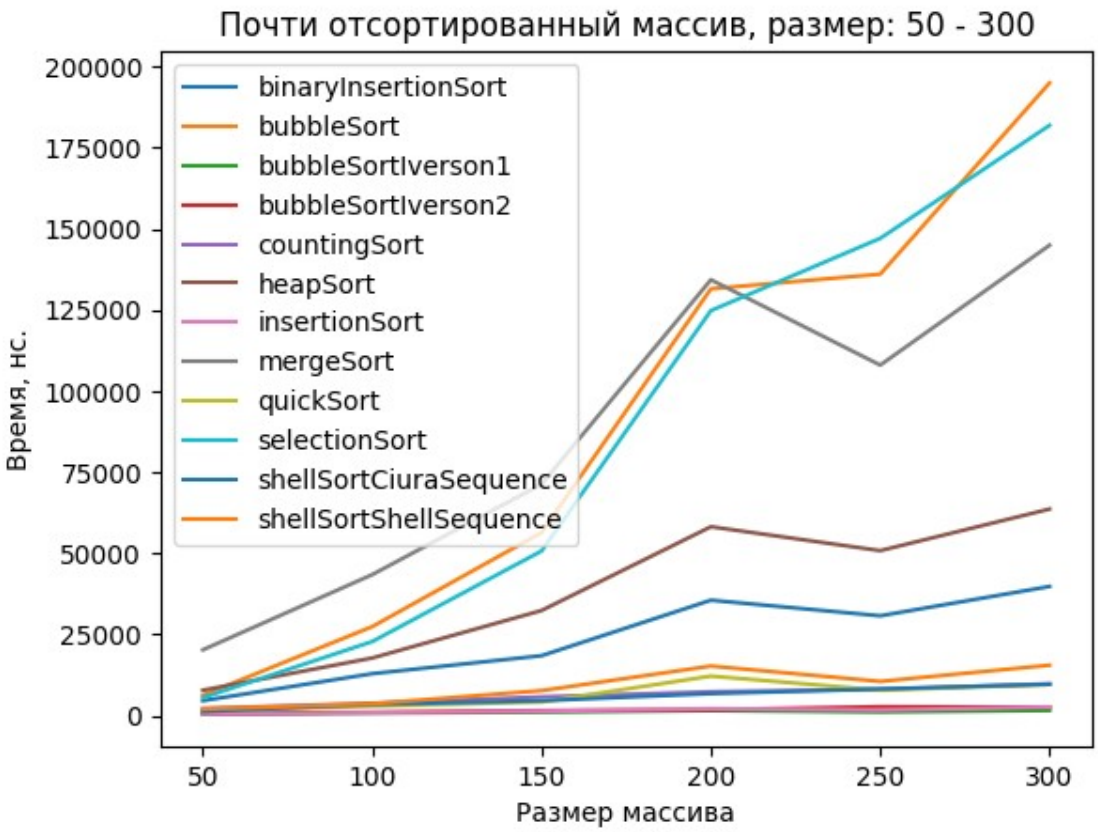
Надеюсь на Ваше понимание и спасибо за проверку!

Графики скорости выполнения сортировок в зависимости от размера сортируемых массивов, варьирующегося в диапазоне [50; 300]

Почти отсортированный массив

```
almost_sorted = data[(data['Amount'] <= 300) & (data['Array type'] ==
'almostSorted')]
sort_methods = almost_sorted['Sort method'].unique()
for method in sort_methods:
    sort = almost_sorted[almost_sorted['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Почти отсортированный массив, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

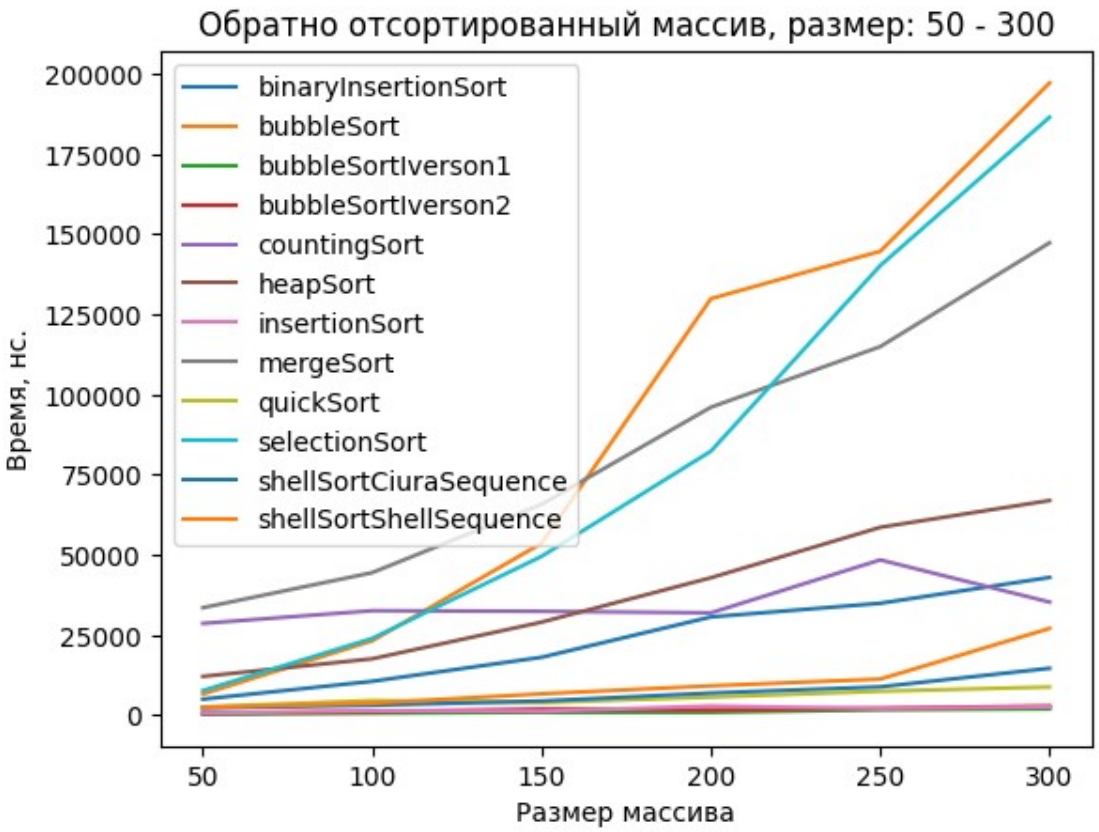


На графике все вполне очевидно: пузырьковая сортировка, как и предполагалось, работает дольше всего. Сортировка выбором догоняет пузырьковую по скорости (по медленности) выполнения, а где-то даже обгоняет ее (например, при size ~ 250). Сортировка вставками работает медленнее всех остальных сортировок при 200 элементах в исходном массиве. При размере массива равном 250 практически у всех сортировок наблюдается сокращение времени выполнения, по сравнению с сортировкой массивов размером 200.

Обратно отсортированный массив

```
backward_sorted = data[(data['Amount'] <= 300) & (data['Array type']
== 'backwardSorted')]
sort_methods = backward_sorted['Sort method'].unique()
for method in sort_methods:
    sort = backward_sorted[backward_sorted['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Обратно отсортированный массив, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

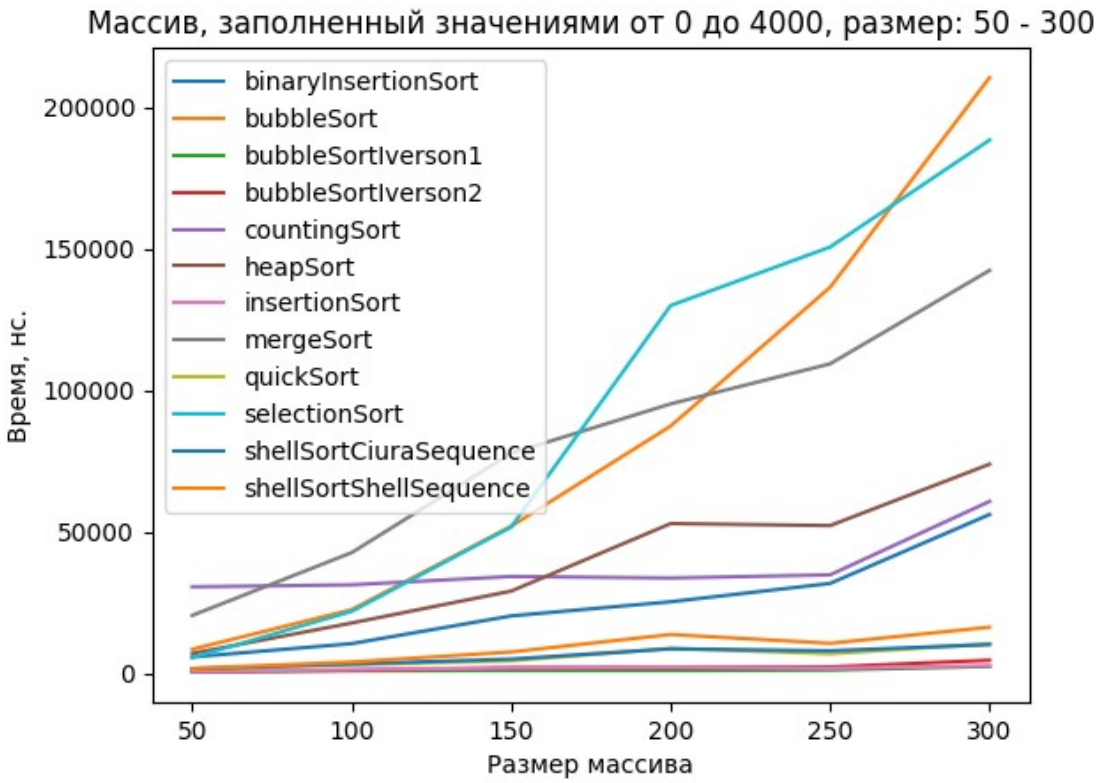


Ситуация с этим графиком аналогична предыдущей, однако теперь сортировка выбором нигде не обгоняет по времени выполнения сортировку пузырьком.

Массив, заполненный значениями от 0 до 4000

```
zero_4000 = data[(data['Amount'] <= 300) & (data['Array type'] ==
'fill0To4000')]
sort_methods = zero_4000['Sort method'].unique()
for method in sort_methods:
    sort = zero_4000[zero_4000['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Массив, заполненный значениями от 0 до 4000, размер: 50 -
300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

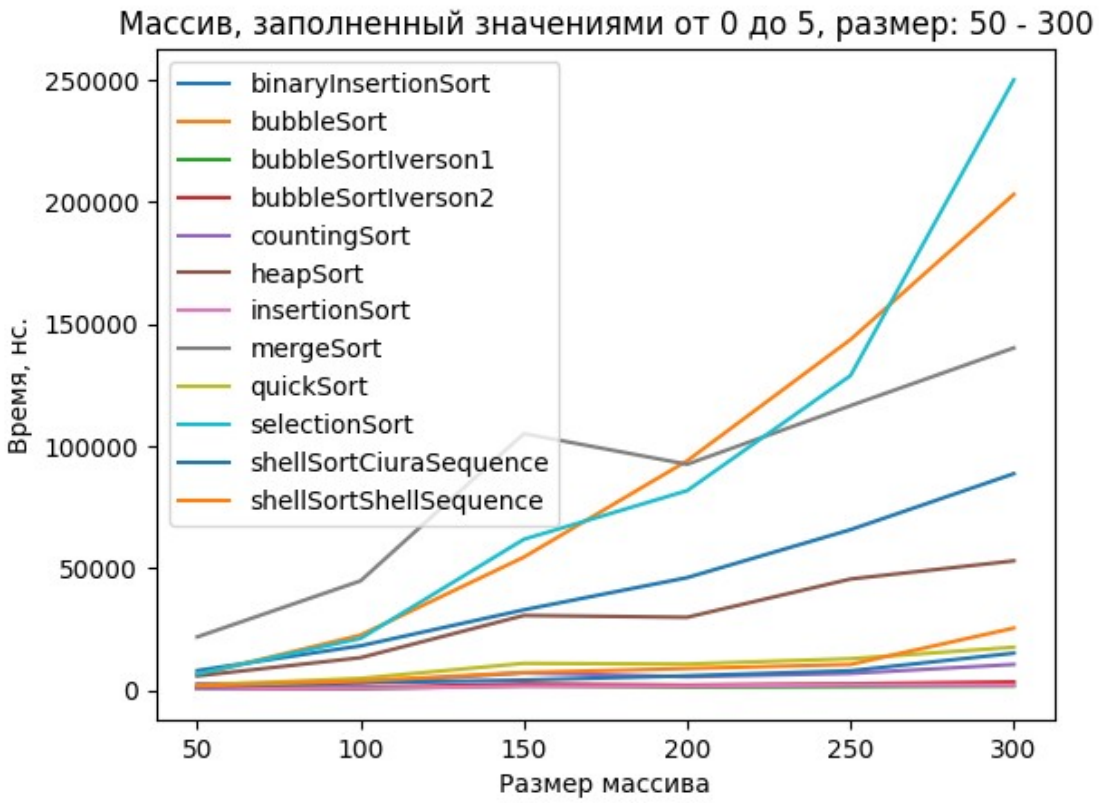


Здесь ситуация больше похожа на первый график: так же заметны просадки при размере массива 250 и наблюдается увеличенное время выполенния при размере равном 200.

Массив, заполненный значениями от 0 до 5

```
zero_5 = data[(data['Amount'] <= 300) & (data['Array type'] ==
'fill0To5')]
sort_methods = zero_5['Sort method'].unique()
for method in sort_methods:
    sort = zero_5[zero_5['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Массив, заполненный значениями от 0 до 5, размер: 50 -
300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```



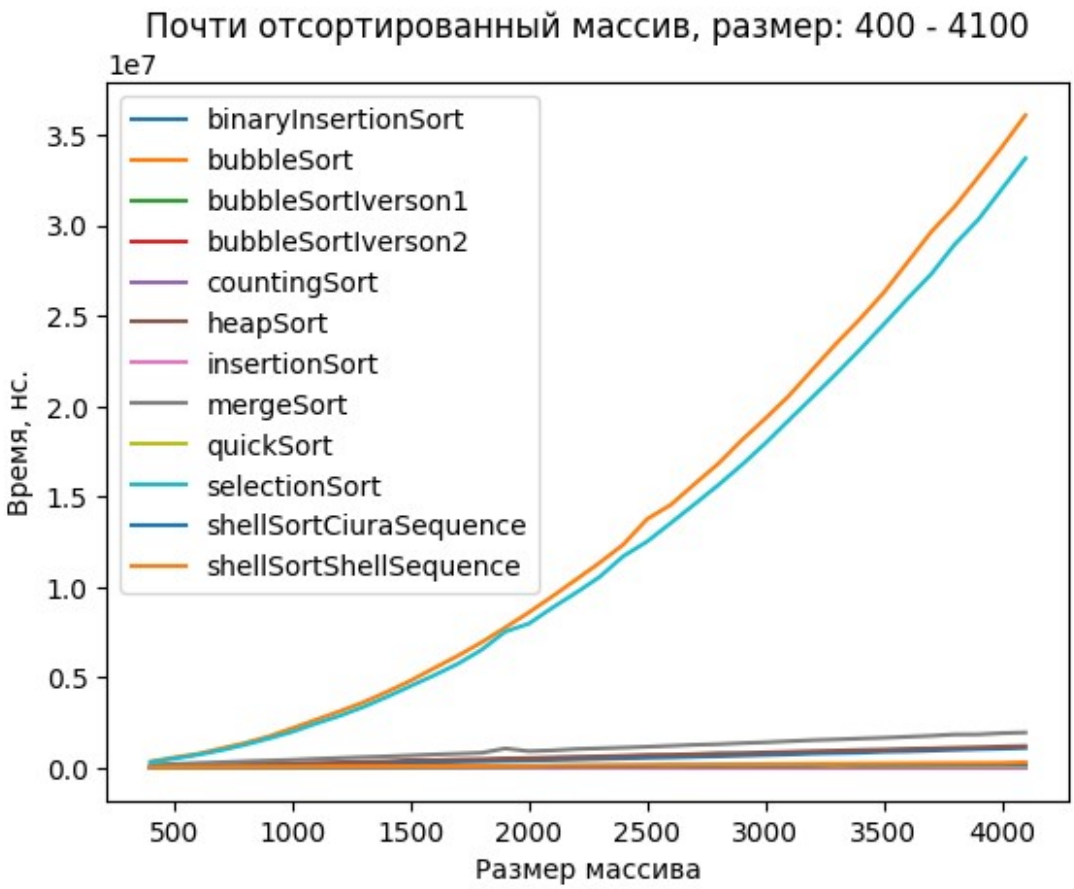
На данном графике заметно, как нестабильно работает сортировка слиянием. Время выполнения некоторых других сортировок так же резко увеличиваются в зависимости от размера массива (например, сортировка пузырьком при size > 250 или сортировка последовательностью Циура при size > 100).

Графики скорости выполнения сортировок в зависимости от размера сортируемых массивов, варьирующегося в диапазоне [400; 4100]

Почти отсортированный массив

```
almost_sorted = data[(data['Amount'] > 300) & (data['Array type'] ==
'almostSorted')]
sort_methods = almost_sorted['Sort method'].unique()
for method in sort_methods:
    sort = almost_sorted[almost_sorted['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Почти отсортированный массив, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

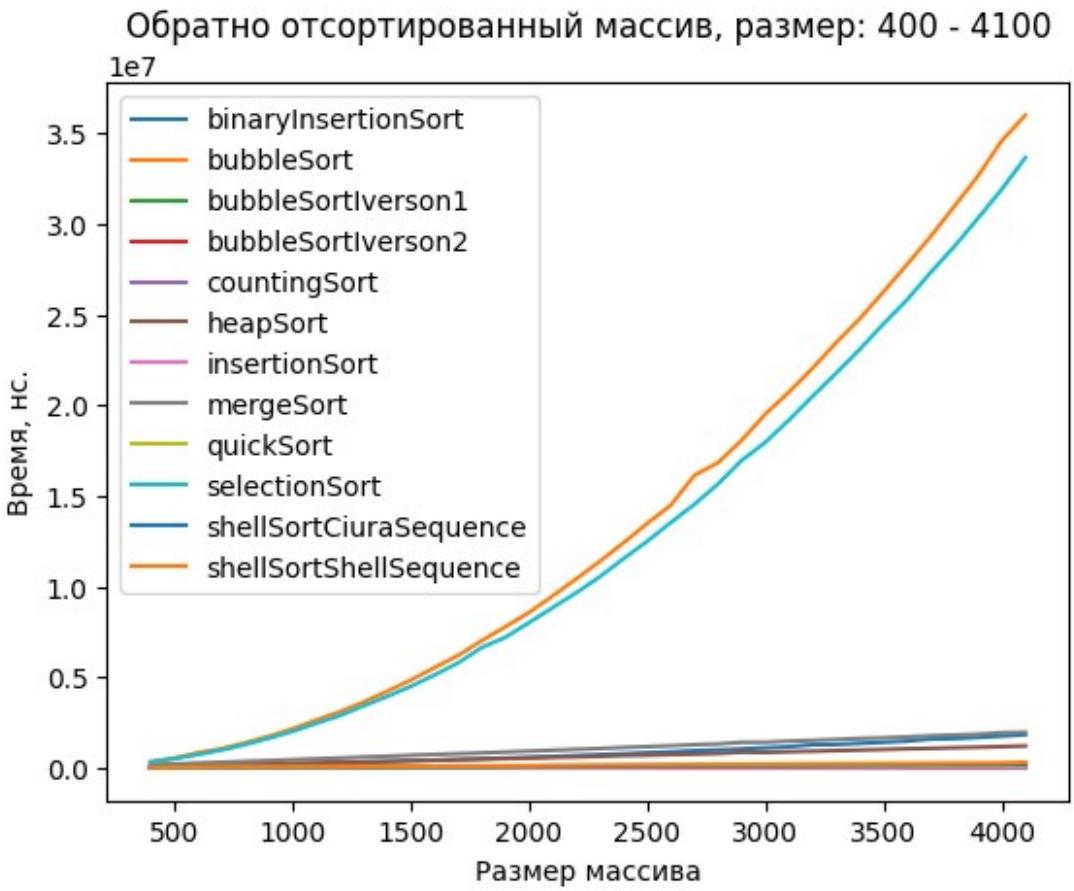


Почти идеально ровные графики, красиво :).

Обратно отсортированный массив

```
backward_sorted = data[(data['Amount'] > 300) & (data['Array type'] ==
'backwardSorted')]
sort_methods = backward_sorted['Sort method'].unique()
for method in sort_methods:
    sort = backward_sorted[backward_sorted['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Обратно отсортированный массив, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

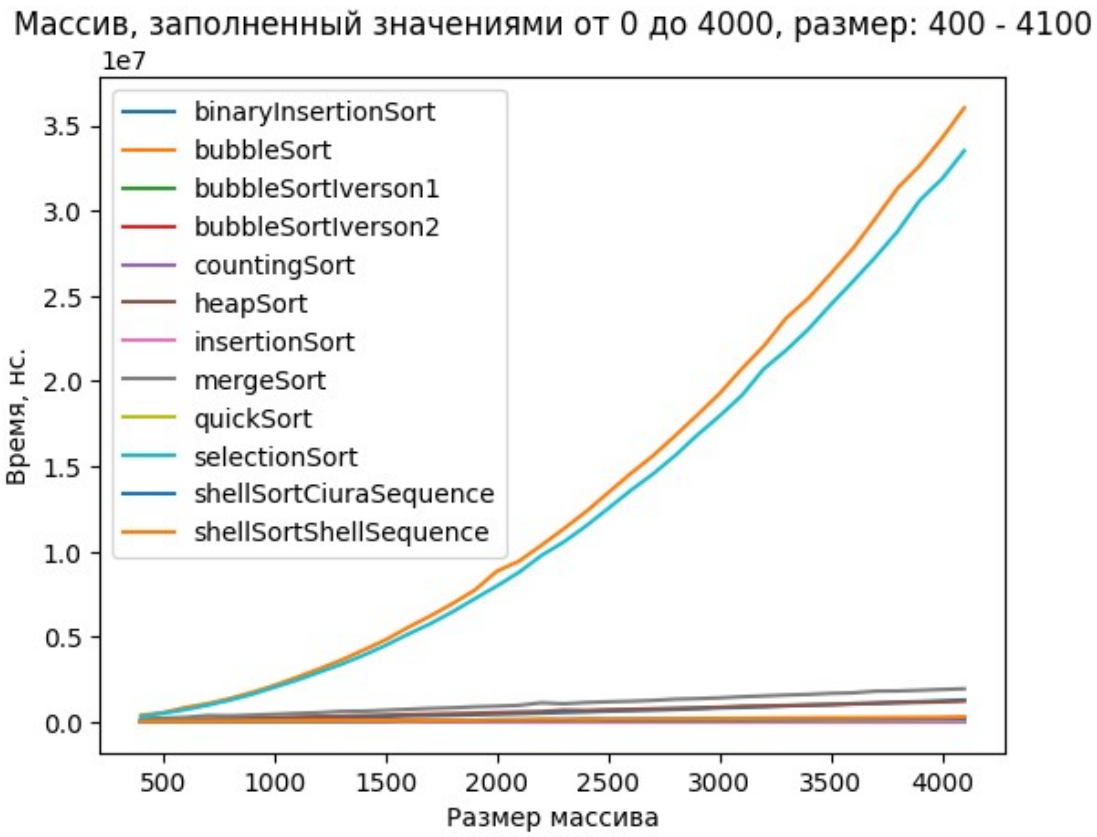


И здесь все точно так же. Сортировка пузырьком и выбором работают медленнее всех остальных.

Массив, заполненный значениями от 0 до 4000

```
zero_4000 = data[(data['Amount'] > 300) & (data['Array type'] ==
'fill0To4000')]
sort_methods = zero_4000['Sort method'].unique()
for method in sort_methods:
    sort = zero_4000[zero_4000['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Массив, заполненный значениями от 0 до 4000, размер: 400 -
4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

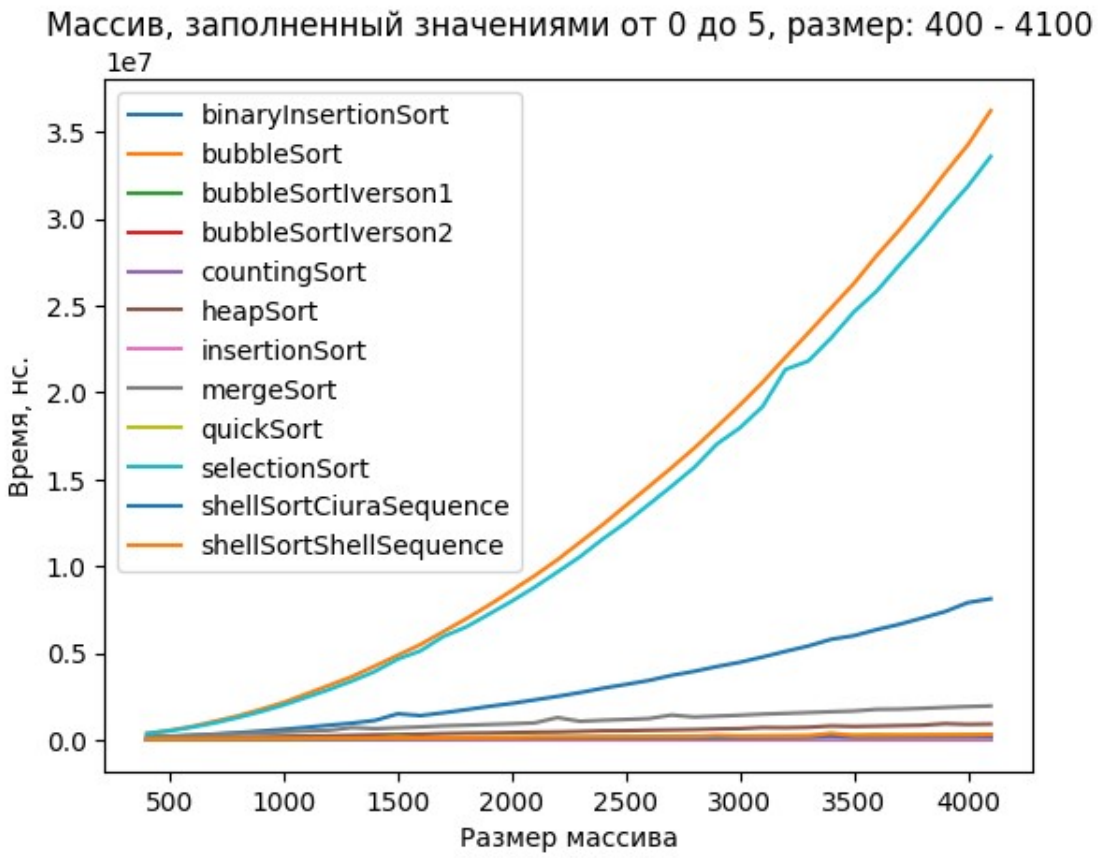


И снова ничего нового: сортировка пузырьком и выбором самые долгие.

Массив, заполненный значениями от 0 до 5

```
zero_5 = data[(data['Amount'] > 300) & (data['Array type'] ==
'fill0To5')]
sort_methods = zero_5['Sort method'].unique()
for method in sort_methods:
    sort = zero_5[zero_5['Sort method'] == method]
    plt.plot(sort['Amount'], sort['Time'], label=method)

plt.title('Массив, заполненный значениями от 0 до 5, размер: 400 -
4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```



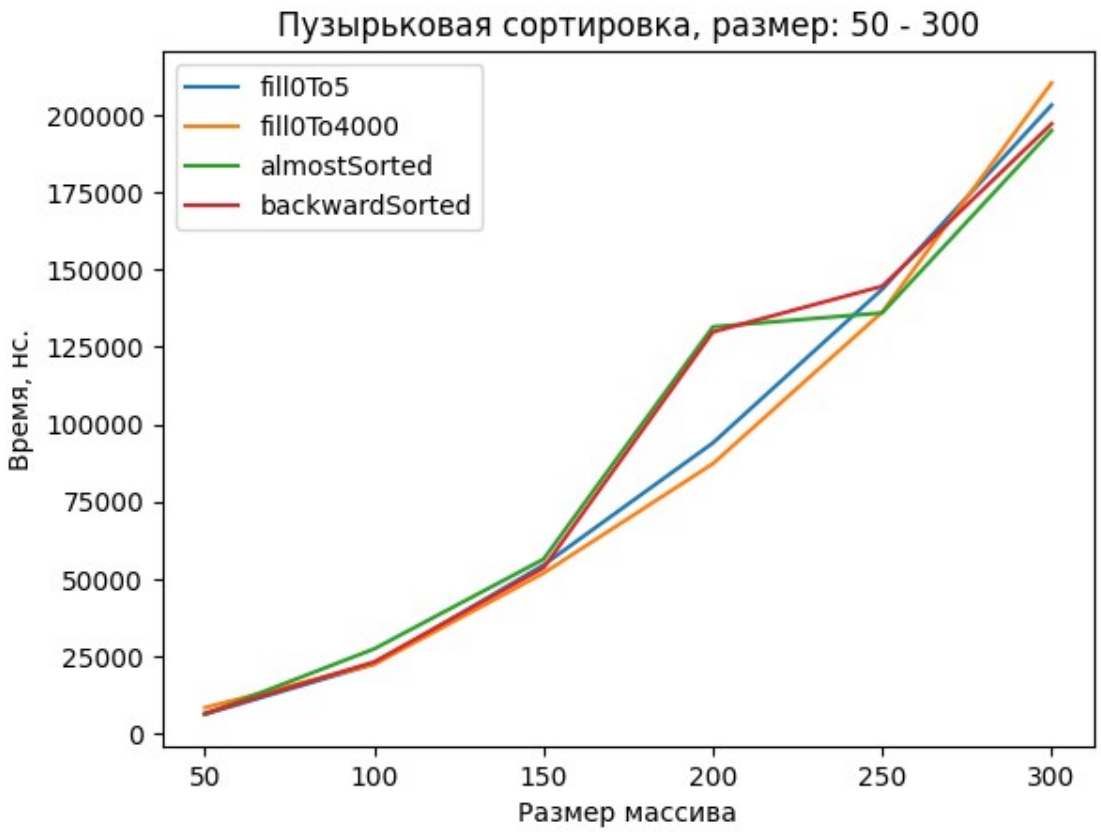
Снова ничего нового. Выбросов не видно, графики почти идеально. Исключение, наверное, составляет только сортировка вставками - у нее заметны небольшие повышения времени сортировки.

Графики скорости выполнения сортировок в зависимости типа генерации массива и размера, варьирующегося в диапазоне [50; 300]

Сортировка пузырьком

```
bubble_sort = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'bubbleSort')]
types = bubble_sort['Array type'].unique()
for gen_type in types:
    plot_type = bubble_sort[bubble_sort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пузырьковая сортировка, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

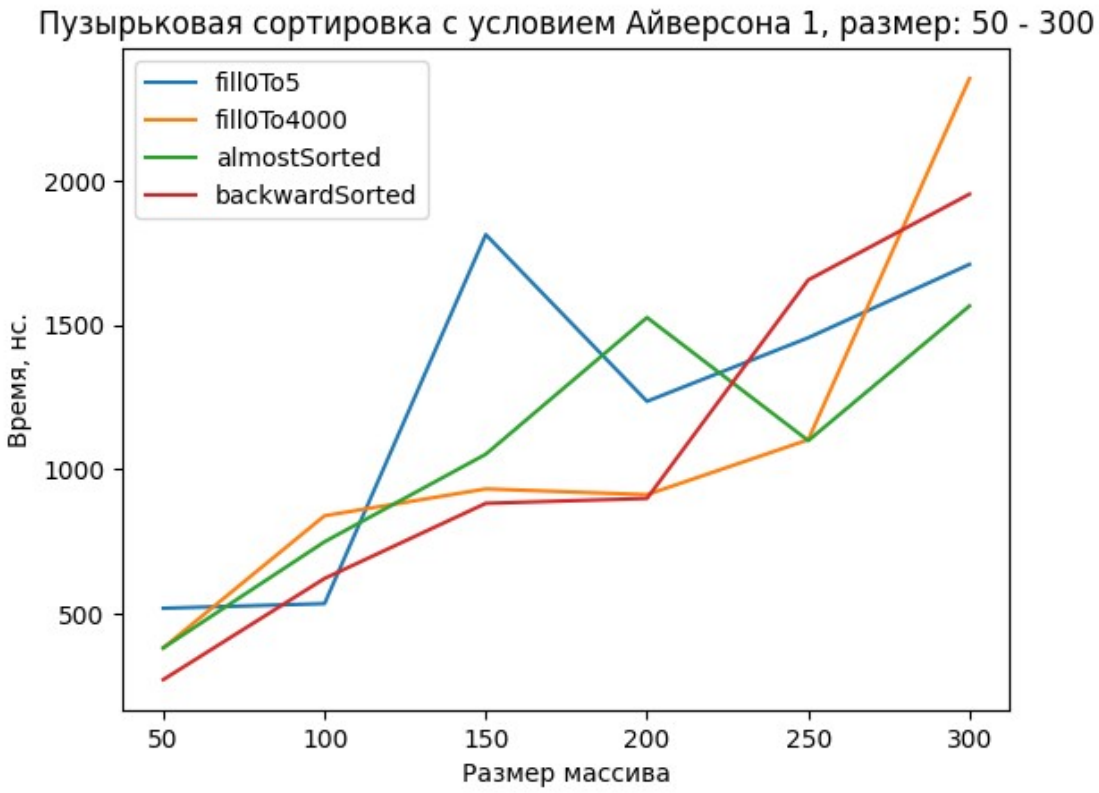


Как видно, для пузырьковой сортировки не очень важен тип генерации массива и она показывает одни и те же результаты на всех видах заполнения, однако видны повышения времени выполнения алгоритма на массивах с типами генерации "almostSorted" и "backwardSorted".

Сортировка пузырьком с условием Айверсона 1

```
iverson_1 = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'bubbleSortIverson1')]
types = iverson_1['Array type'].unique()
for gen_type in types:
    plot_type = iverson_1[iverson_1['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пузырьковая сортировка с условием Айверсона 1, размер: 50 -
300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

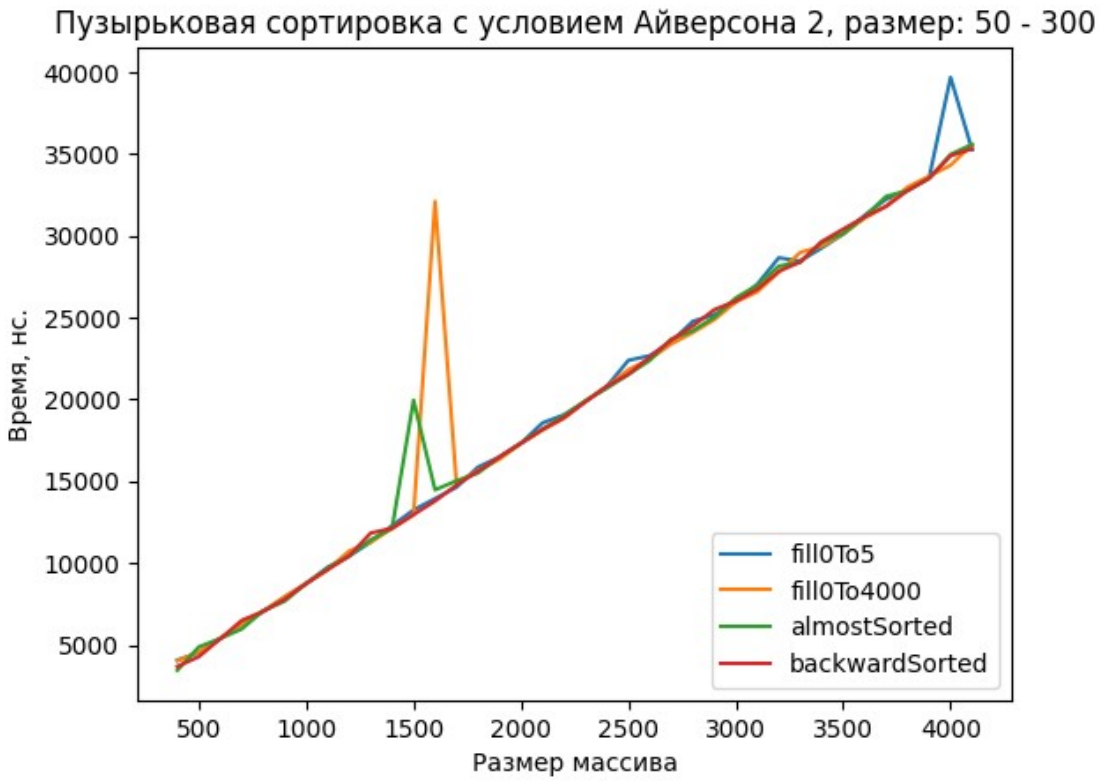


Здесь уже другая ситуация - на графиках очень много пиков, а сами графики не такие ровные, как в случае пузырьковой сортировки. Самым трудным для сортировки оказался массив, заполненный случайными числами от 0 до 4000, а вот почти отсортированный массив при количестве элементов равному 300 показал самый быстрый результат.

Сортировка пузырьком с условием Айверсона 2

```
iverson_2 = data[(data['Amount'] > 300) & (data['Sort method'] ==
'bubbleSortIverson2')]
types = iverson_2['Array type'].unique()
for gen_type in types:
    plot_type = iverson_2[iverson_2['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пузырьковая сортировка с условием Айверсона 2, размер: 50 -
300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

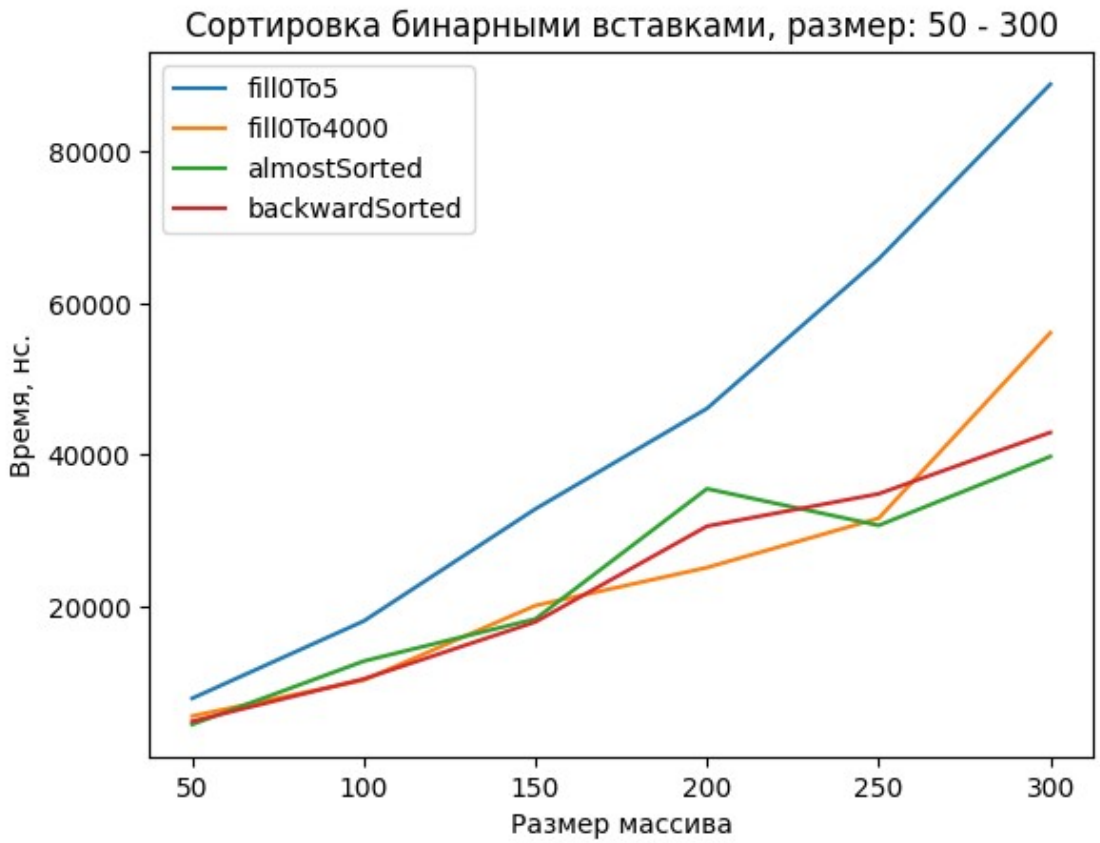


На этом графике ничего сильно примечательного нет, однако можно выделить довольно крутой пик, который демонстрирует повышение времени выполнения пузыречной сортировки с условием Айверсона 2 для массива заполненного по правилу "fill0To4000" для размера примерно равного 1700. Так же на графике есть еще два пика - для почти отсортированного массива размера 1500 и для массива, заполненного случайными значениями от 0 до 5 и размером около 4000 единиц.

Сортировка бинарными вставками

```
bin_sort = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'binaryInsertionSort')]
types = bin_sort['Array type'].unique()
for gen_type in types:
    plot_type = bin_sort[bin_sort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка бинарными вставками, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

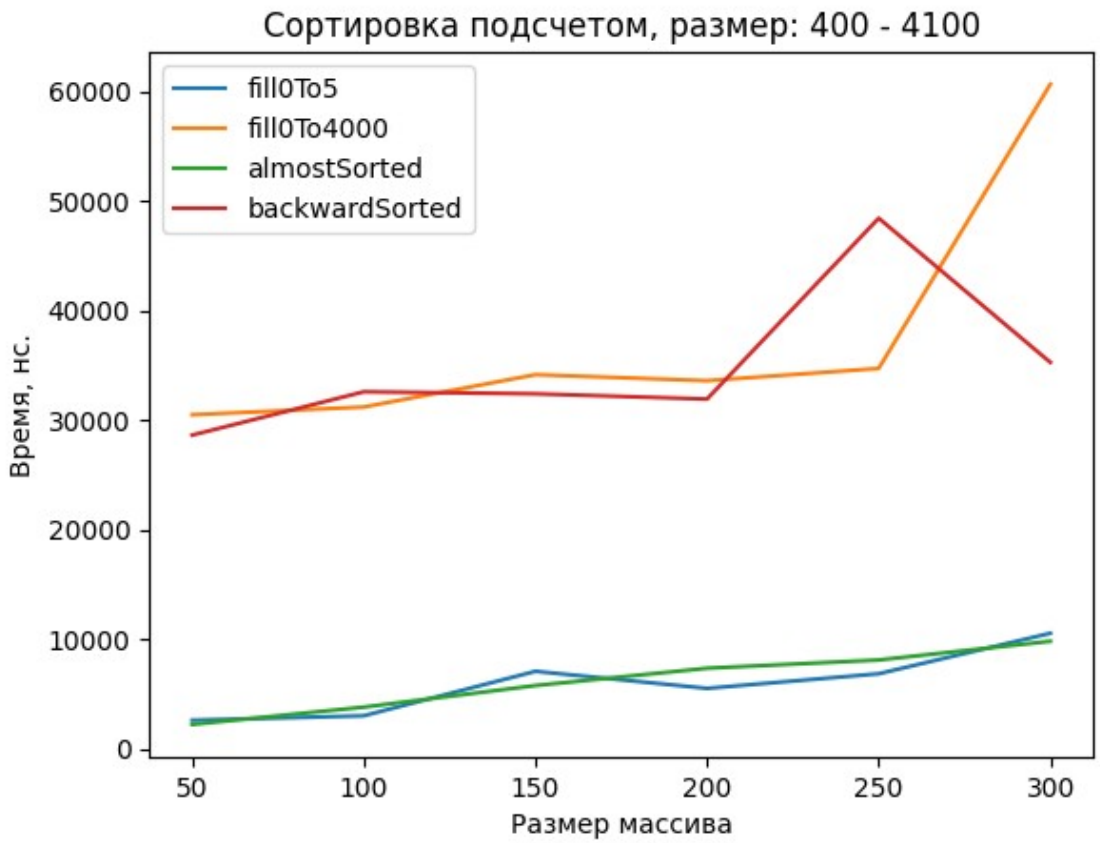


Худшую асимптотику показывает вектор, заполненный случайными значениями от 0 до 5. Остальные виды массивов сортируются примерно за одинаковое время. На самом большом размере диапазона (300) хуже остальных 3 видов генерации массива сортируется вектор, заполненный элементами от 0 до 4000.

Сортировка подсчетом

```
counting = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'countingSort')]
types = counting['Array type'].unique()
for gen_type in types:
    plot_type = counting[counting['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка подсчетом, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

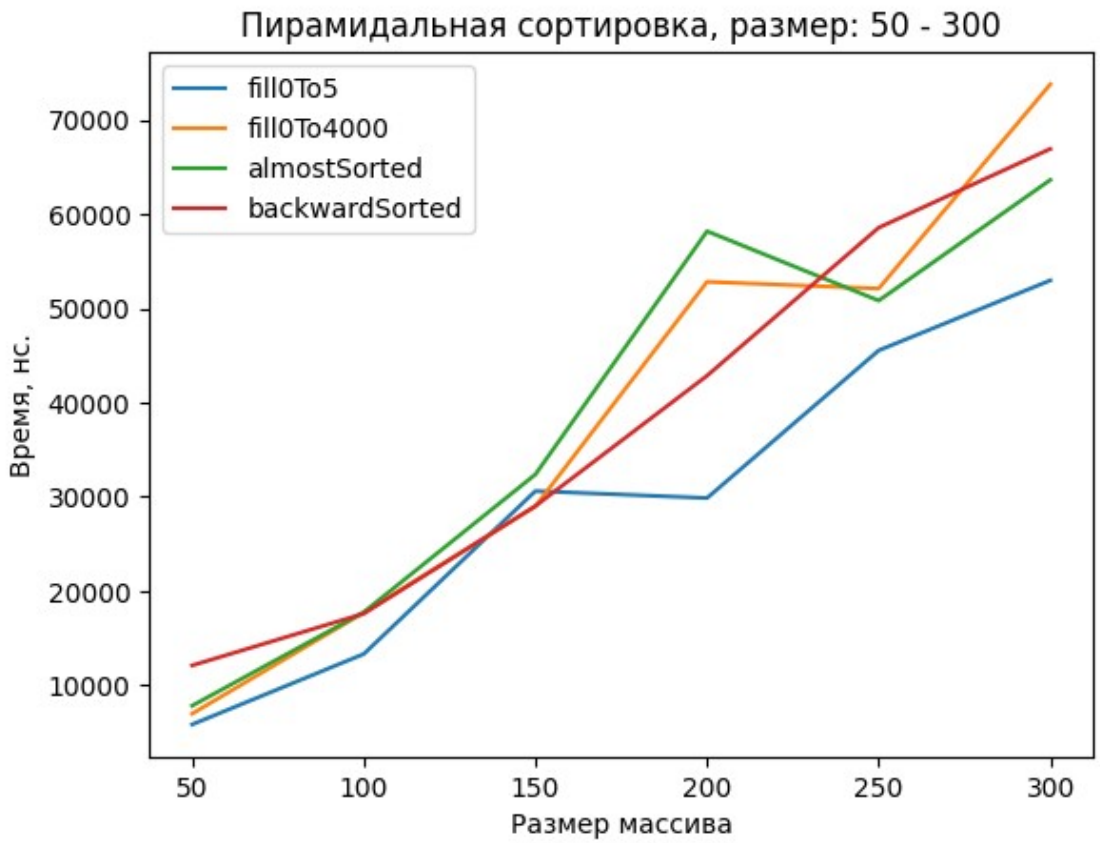


Здесь уже нечто интересное - заполнение вектора значениями от 0 до 5 и почти отсортированный массив сортируются довольно быстро, а вот заполнение массива случайными значениями от 0 до 4000 и "backwardSorted" сортируются в несколько раз медленнее двух вышеописанных видов.

Пирамидальная сортировка

```
heap = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'heapSort')]
types = heap['Array type'].unique()
for gen_type in types:
    plot_type = heap[heap['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пирамидальная сортировка, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

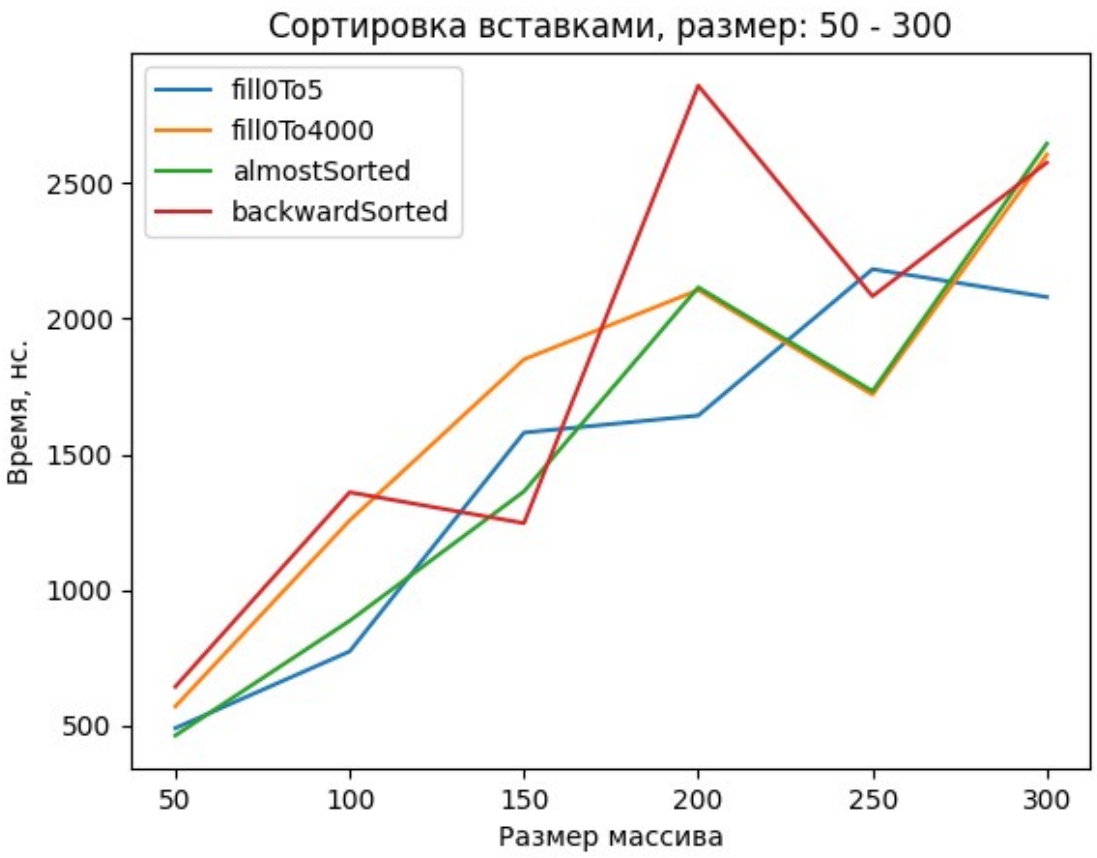


Сказать особо нечего - при размере массива <= 150 все типы генерации вектора сортируются примерно за одинаковое время, однако потом идут расхождения, в результате которых заполнение случайными значениями от 0 до 5 оказывается эффективнее остальных видов заполнения.

Сортировка вставками

```
insertionSort = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'insertionSort')]
types = insertionSort['Array type'].unique()
for gen_type in types:
    plot_type = insertionSort[insertionSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка вставками, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

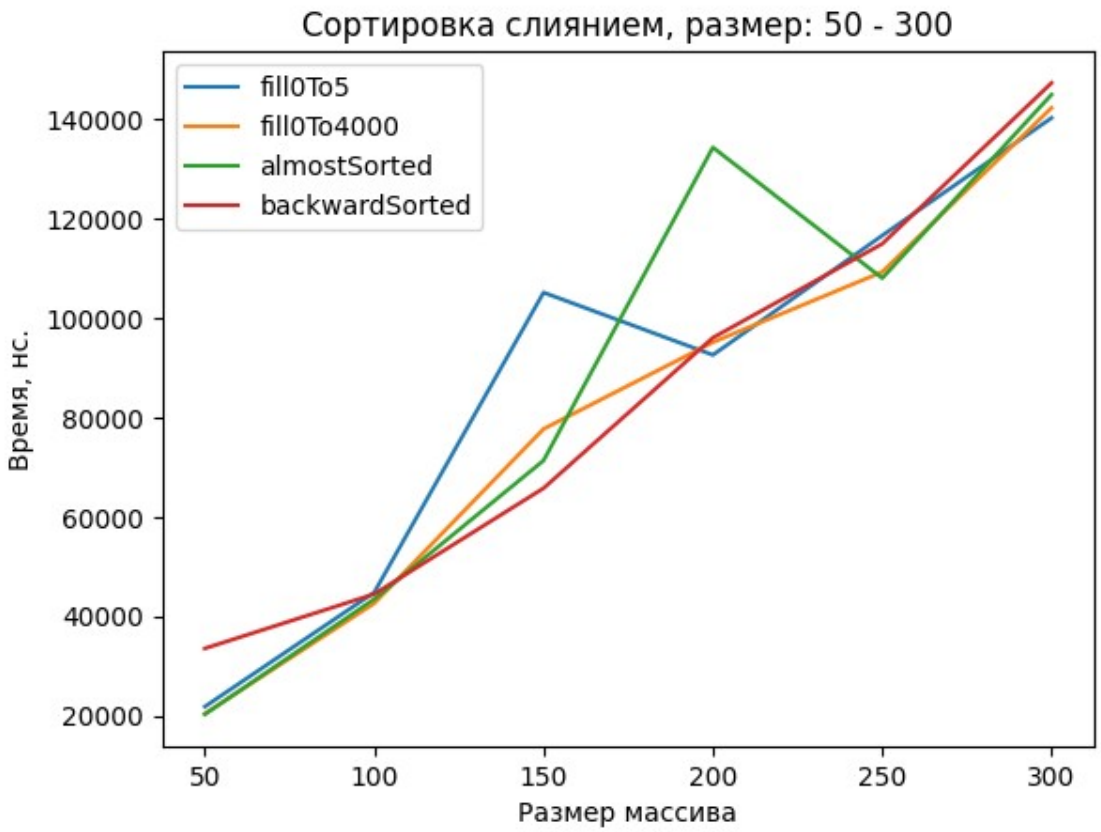


Из интересного: наблюдается резкое повышение времени сортировки массива, заполненным по правилу "backwardSorted" при размере массива = 150. При size = 300 все типы массивов, кроме заполнения от 0 до 5, работают примерно за одинаковое время.

Сортировка слиянием

```
mergeSort = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'mergeSort')]
types = mergeSort['Array type'].unique()
for gen_type in types:
    plot_type = mergeSort[mergeSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка слиянием, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

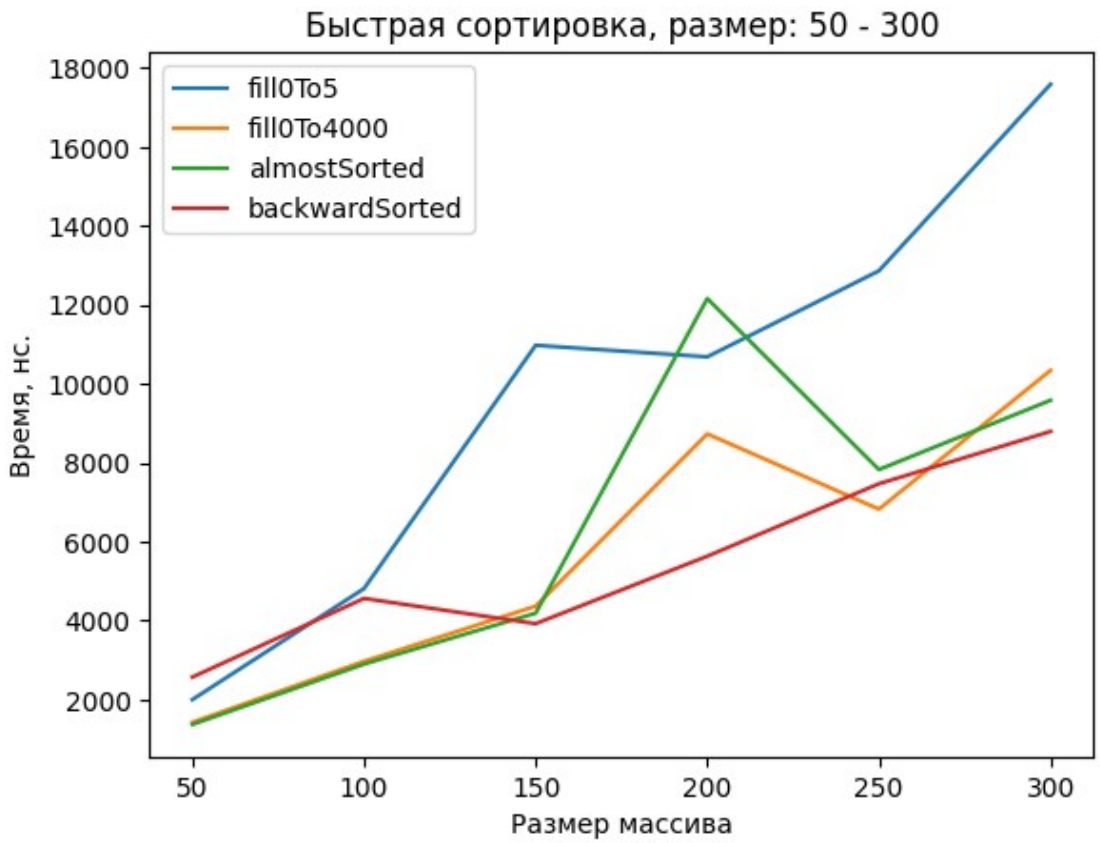


Два ярко выраженных пика - ("fill0To5", size = 150) и ("almostSorted", size = 200). На максимальном размере массива из диапазона тип генерации очень незначительно влияет на скорость выполнения сортировки.

Быстрая сортировка

```
quickSort = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'quickSort')]
types = quickSort['Array type'].unique()
for gen_type in types:
    plot_type = quickSort[quickSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Быстрая сортировка, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

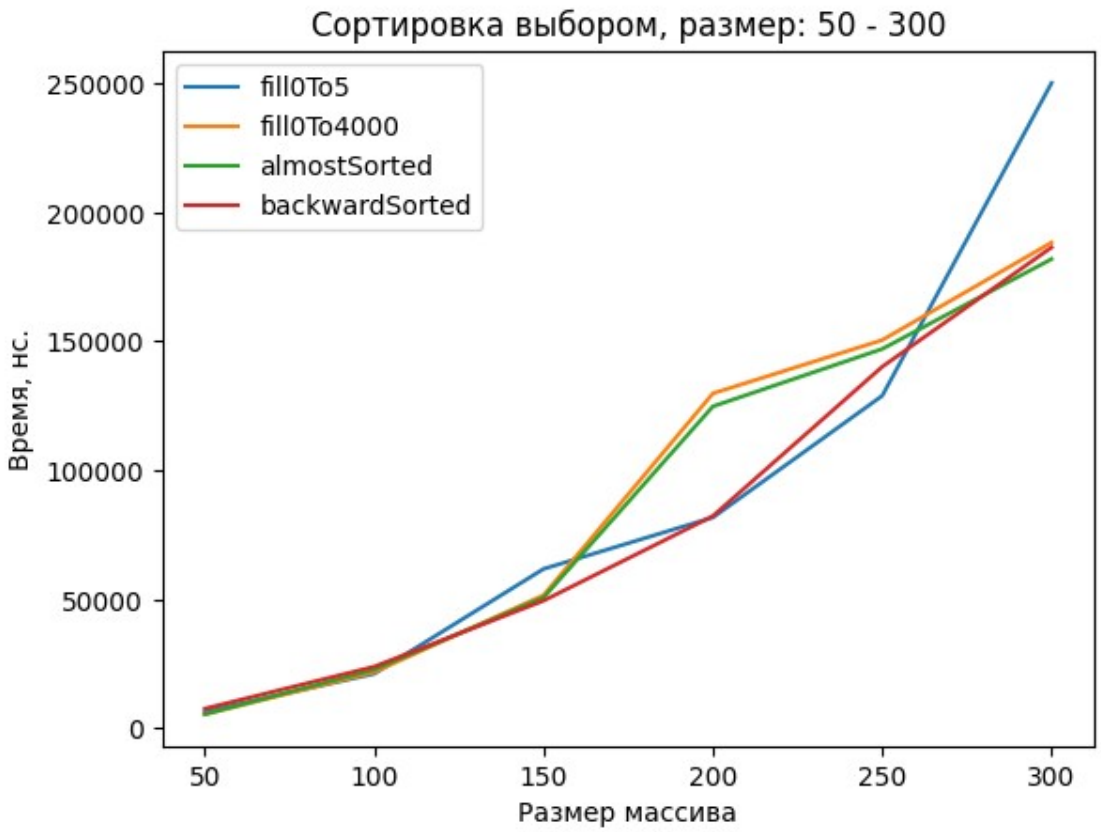


Два пика: ("almostSorted", size = 200), ("fill0To4000", size = 200). Сортировка массива, заполненного числами от 0 до 5 в перспективе показывает наихудшую асимптотику сортировки методом быстрой сортировки. Остальные 3 типа генерации показывают примерно одинаковое время сортировки на размере массива равном 300.

Сортировка выбором

```
selectionSort = data[(data['Amount'] <= 300) & (data['Sort method'] ==
'selectionSort')]
types = selectionSort['Array type'].unique()
for gen_type in types:
    plot_type = selectionSort[selectionSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка выбором, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

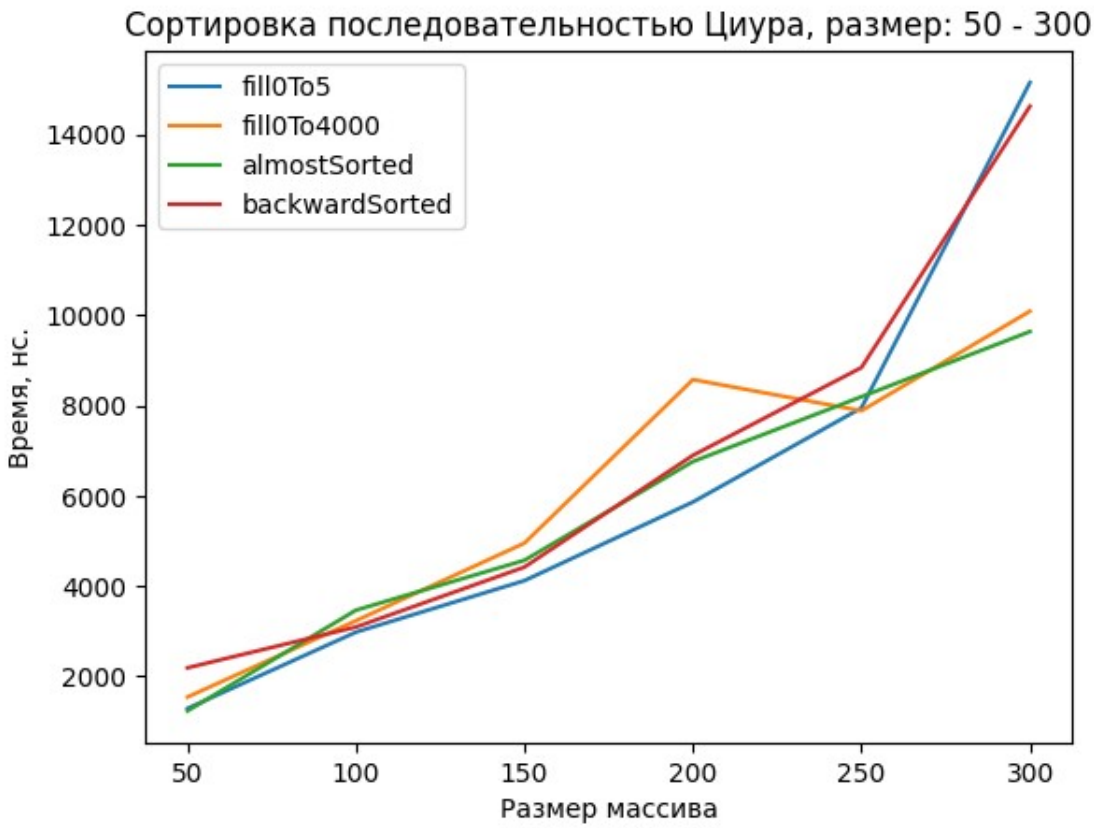


Сортировка массива, заполненного по правилу "fill0To5", начинает стремительно набирать время сортировки выбором, когда размер массива становится больше 250.

Сортировка последовательностью Циура

```
shellSortCiuraSequence = data[(data['Amount'] <= 300) & (data['Sort
method'] == 'shellSortCiuraSequence')]
types = shellSortCiuraSequence['Array type'].unique()
for gen_type in types:
    plot_type = shellSortCiuraSequence[shellSortCiuraSequence['Array
type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка последовательностью Циура, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

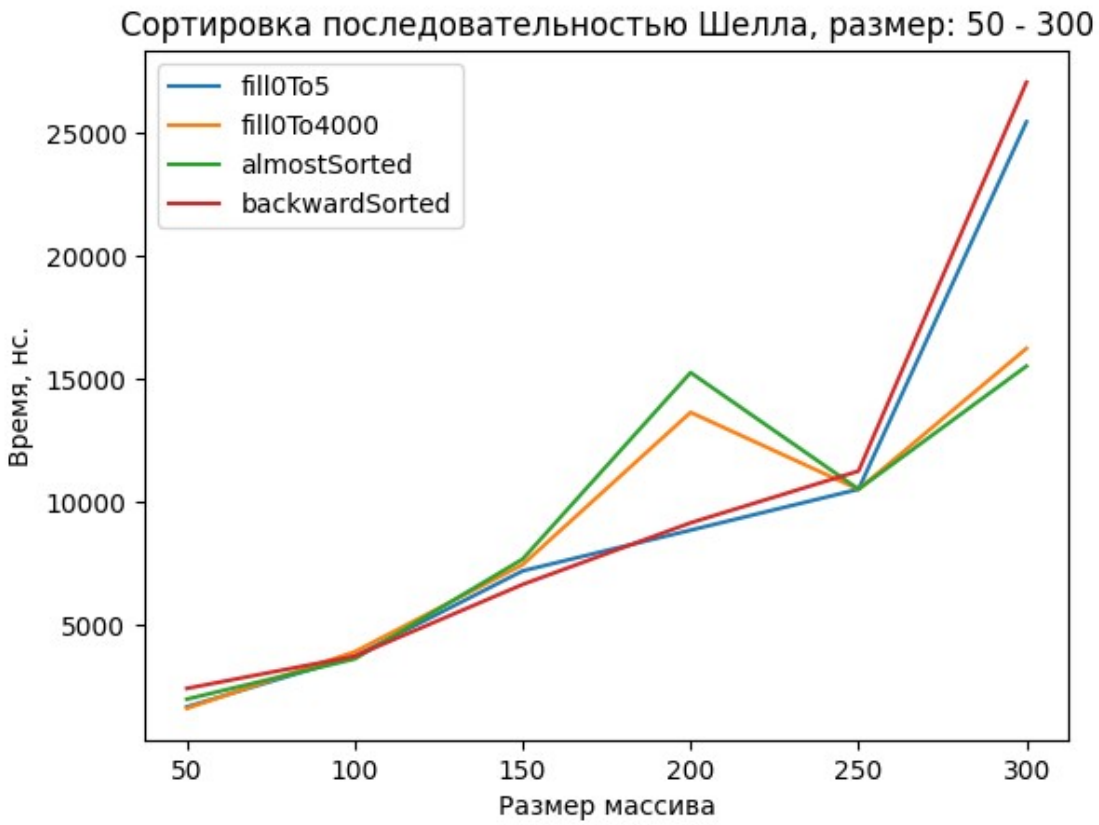


Заполнение массива числами от 0 до 5 начинает снижать скорость сортировки при размере массива равном 250. На том же значении размера генерация по правилу "backwardSorted" тоже начинает увеличивать время работы сортировки.

Сортировка последовательностью Шелла

```
shellSortShellSequence = data[(data['Amount'] <= 300) & (data['Sort
method'] == 'shellSortShellSequence')]
types = shellSortShellSequence['Array type'].unique()
for gen_type in types:
    plot_type = shellSortShellSequence[shellSortShellSequence['Array
type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка последовательностью Шелла, размер: 50 - 300')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```



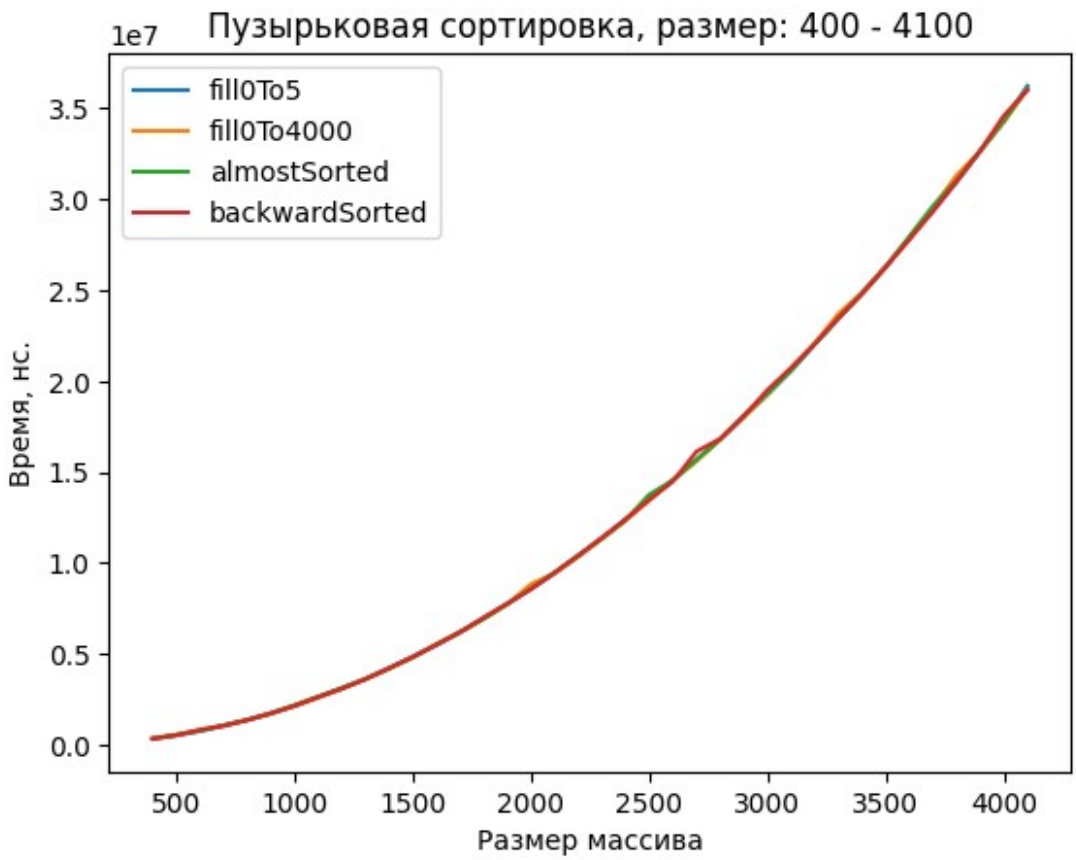
Два пика при размере массива равном 200 у массивов "almostSorted" и "fill0To4000", которые затем собираются в одной точке со всеми остальными способами генерации (size = 250). В перспективе худшую асимптотику алгоритм сортировки покажет на массивах "backwardSorted" и "fill0To5".

Графики скорости выполнения сортировок в зависимости типа генерации массива и размера, варьирующегося в диапазоне [400; 4100]

Сортировка пузырьком

```
bubble_sort = data[(data['Amount'] > 300) & (data['Sort method'] ==
'bubbleSort')]
types = bubble_sort['Array type'].unique()
for gen_type in types:
    plot_type = bubble_sort[bubble_sort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пузырьковая сортировка, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

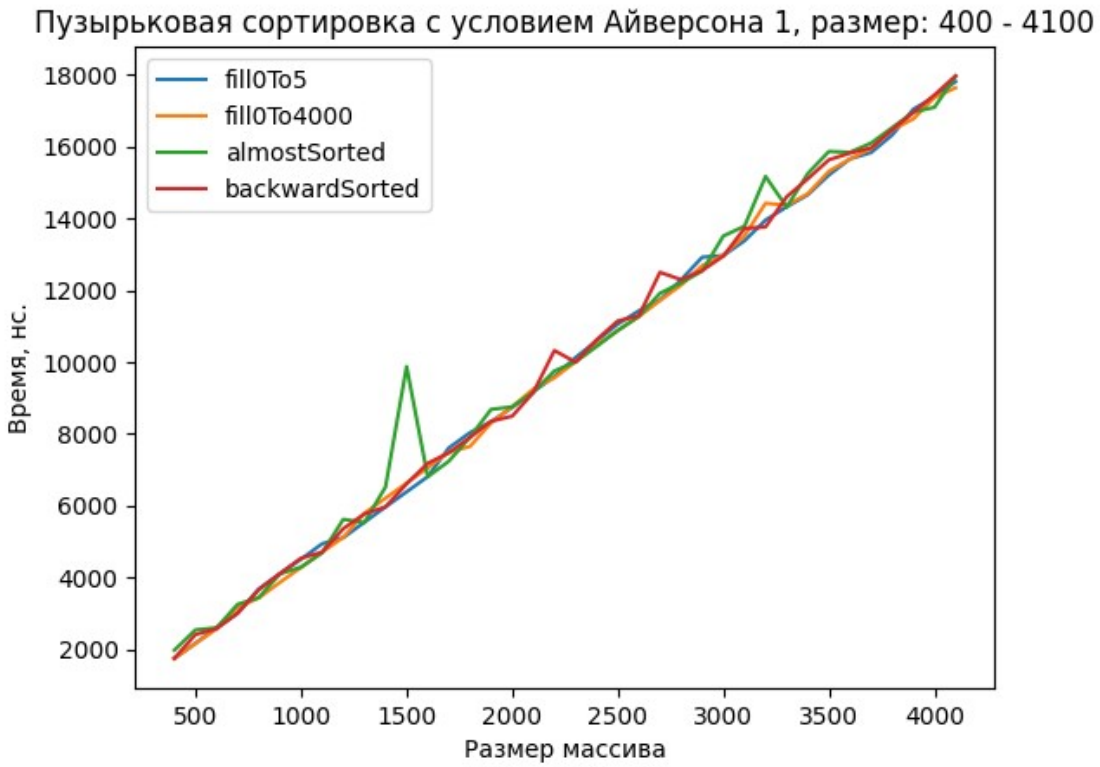


Как видно, для пузырьковой сортировки не очень важен тип генерации массива и она показывает одни и те же результаты на всех видах заполнения.

Сортировка пузырьком с условием Айверсона 1

```
iverson_1 = data[(data['Amount'] > 300) & (data['Sort method'] ==
'bubbleSortIverson1')]
types = iverson_1['Array type'].unique()
for gen_type in types:
    plot_type = iverson_1[iverson_1['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пузырьковая сортировка с условием Айверсона 1, размер: 400
- 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

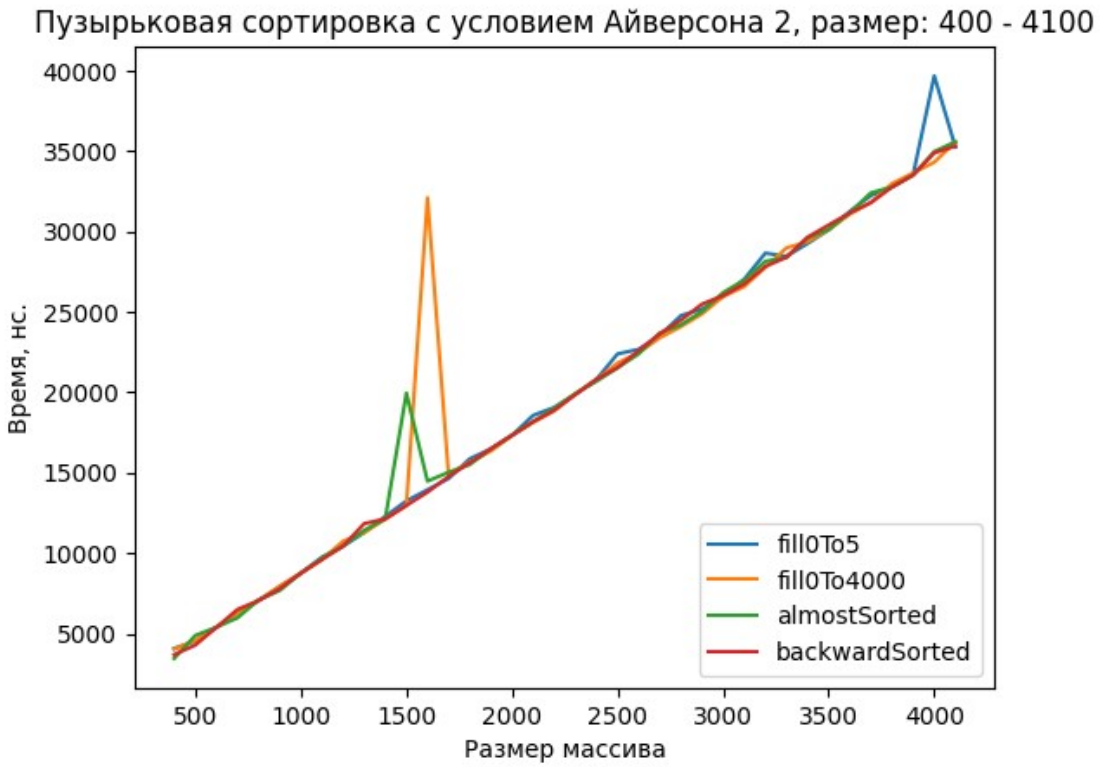


А вот здесь уже ситуация немного другая - заметны небольшие выбросы на прямых и один большой пик при типе генерации массива "почти отсортированный" на размере 1500. Различие результатов на различных типах заполнения массива не видны.

Сортировка пузырьком с условием Айверсона 2

```
iverson_2 = data[(data['Amount'] > 300) & (data['Sort method'] ==
'bubbleSortIverson2')]
types = iverson_2['Array type'].unique()
for gen_type in types:
    plot_type = iverson_2[iverson_2['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пузырьковая сортировка с условием Айверсона 2, размер: 400
- 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

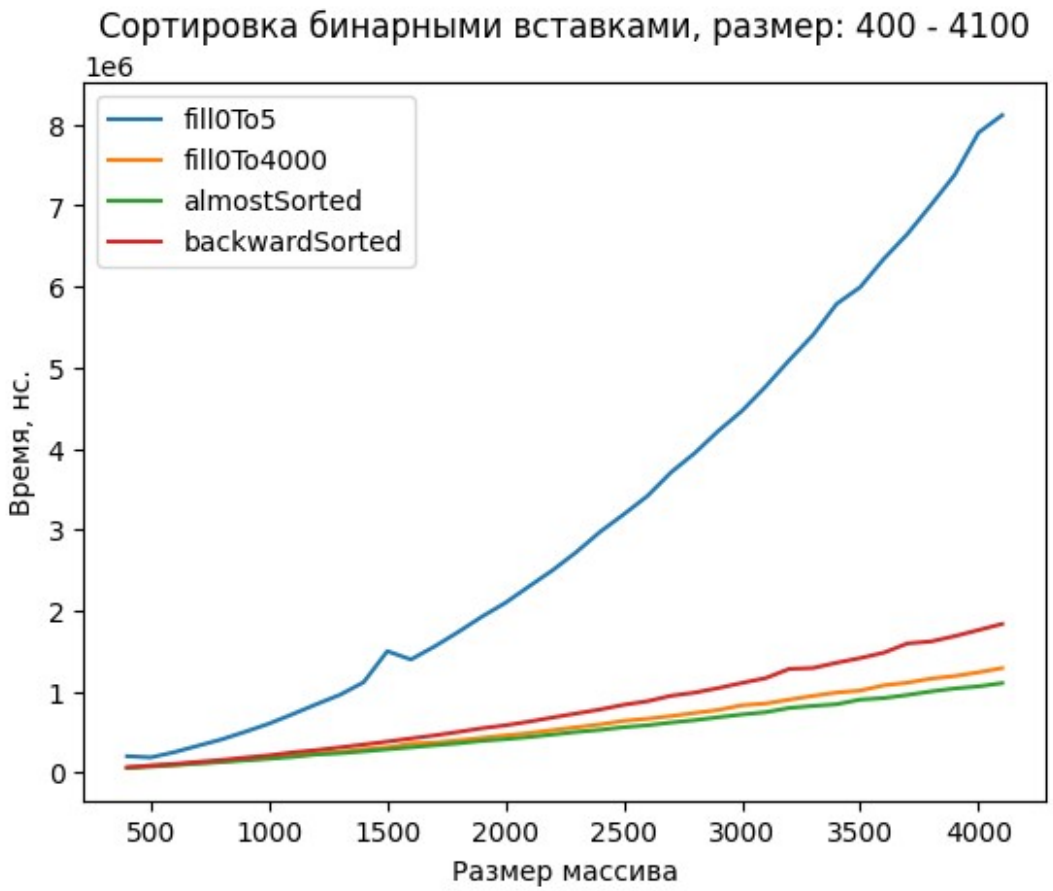


А вот тут начинается jest': мы видим очень сильные повышения скорости работы при массиве заполненным случайными значениями от 0 до 4000 с приблизительным значением размера массива равным 1600. Практически отсортированный массив также требует больше времени для сортировки при 1500 сортируемых элементах.

Сортировка бинарными вставками

```
bin_sort = data[(data['Amount'] > 300) & (data['Sort method'] ==
'binaryInsertionSort')]
types = bin_sort['Array type'].unique()
for gen_type in types:
    plot_type = bin_sort[bin_sort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка бинарными вставками, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

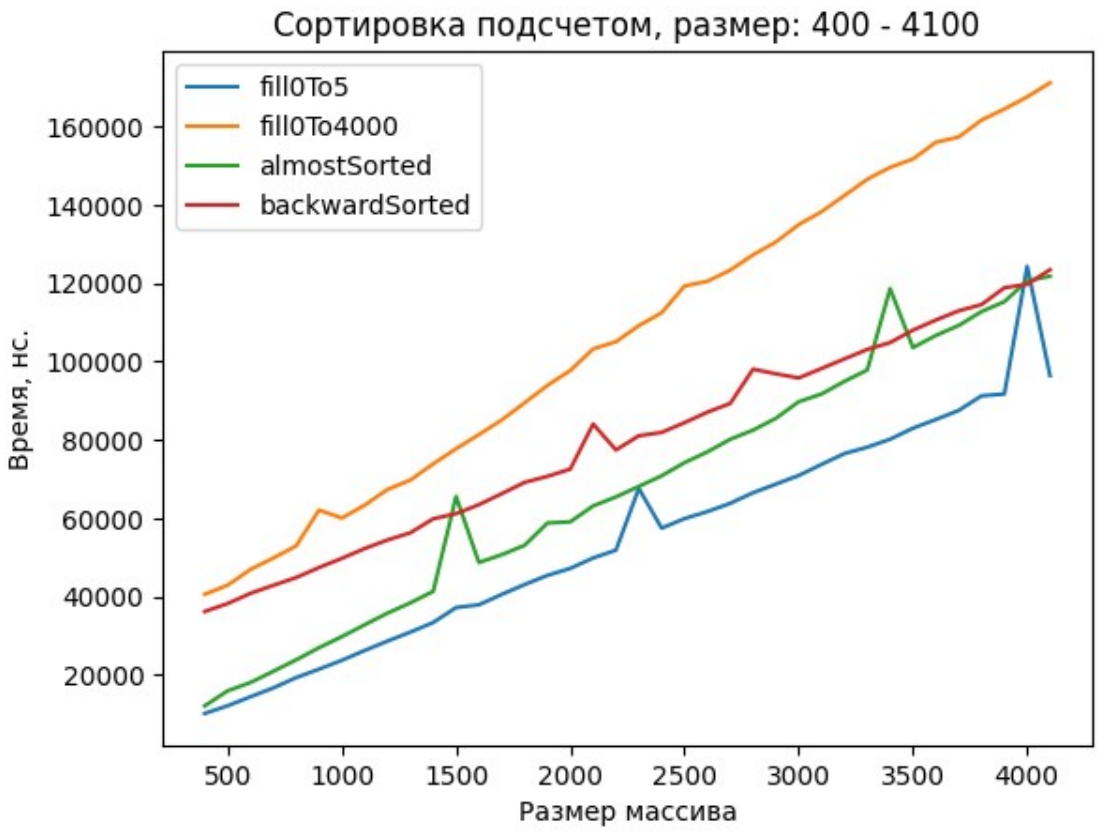


Можно отметить - небольшое повышение времени выполнения при генерации массива размером 1500 типом заполнения от 0 до 5. В этой точке на графике виден небольшой бугорок. Помимо этого, в отличие от предыдущих графиков, по этому можно сделать вывод, что сортировка бинарными вставками хуже всего справляется с массивом заполненным случайными числами от 0 до 5, что довольно странно.

Сортировка подсчетом

```
counting = data[(data['Amount'] > 300) & (data['Sort method'] ==
'countingSort')]
types = counting['Array type'].unique()
for gen_type in types:
    plot_type = counting[counting['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка подсчетом, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

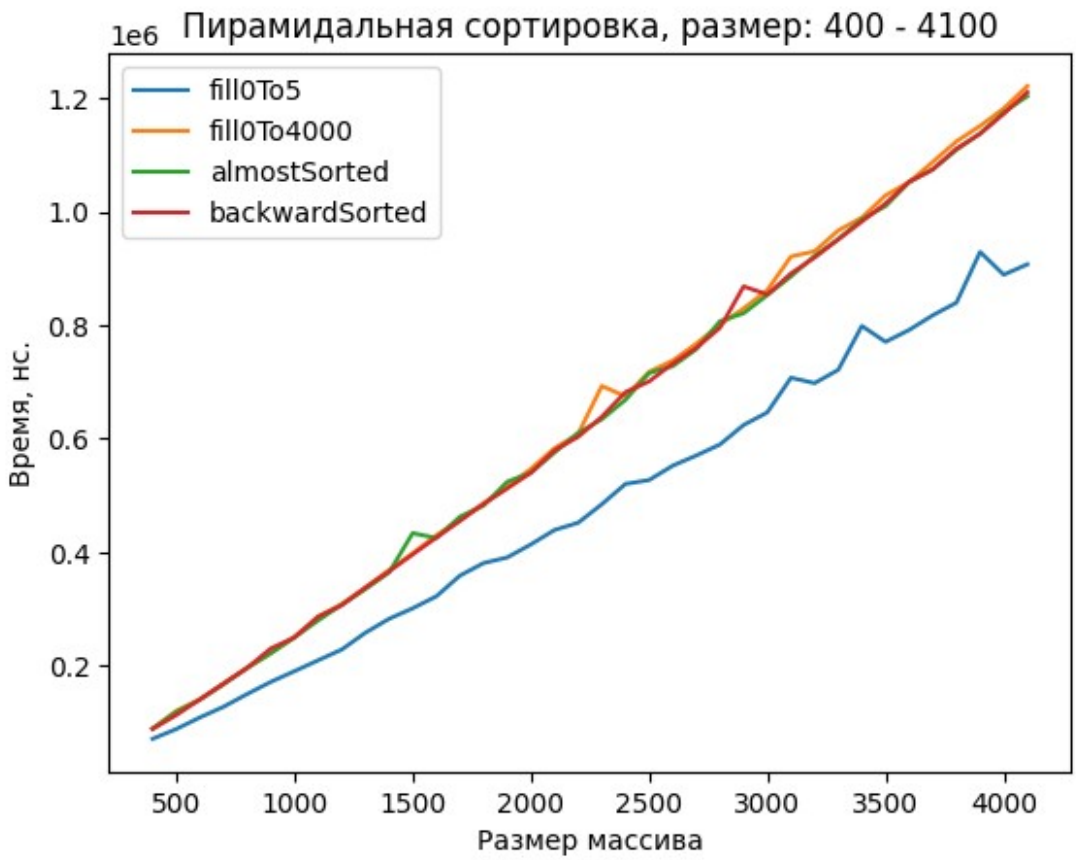


Здесь уже видны ярко выраженные пики повышения времени выполнения. Самый крутой - размер массива = 4000, заполнение - от 0 до 5. Можно отметить, что в отличие от предыдущей сортировки, сортировка подсчетом хуже всего справляется с массивом, заполненным случайными значениями от 0 до 4000, а если изменить случайные значения на числа из диапазона [0; 5), то сортировка покажет наилучшую асимптотику.

Пирамидальная сортировка

```
heap = data[(data['Amount'] > 300) & (data['Sort method'] ==
'heapSort')]
types = heap['Array type'].unique()
for gen_type in types:
    plot_type = heap[heap['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Пирамидальная сортировка, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

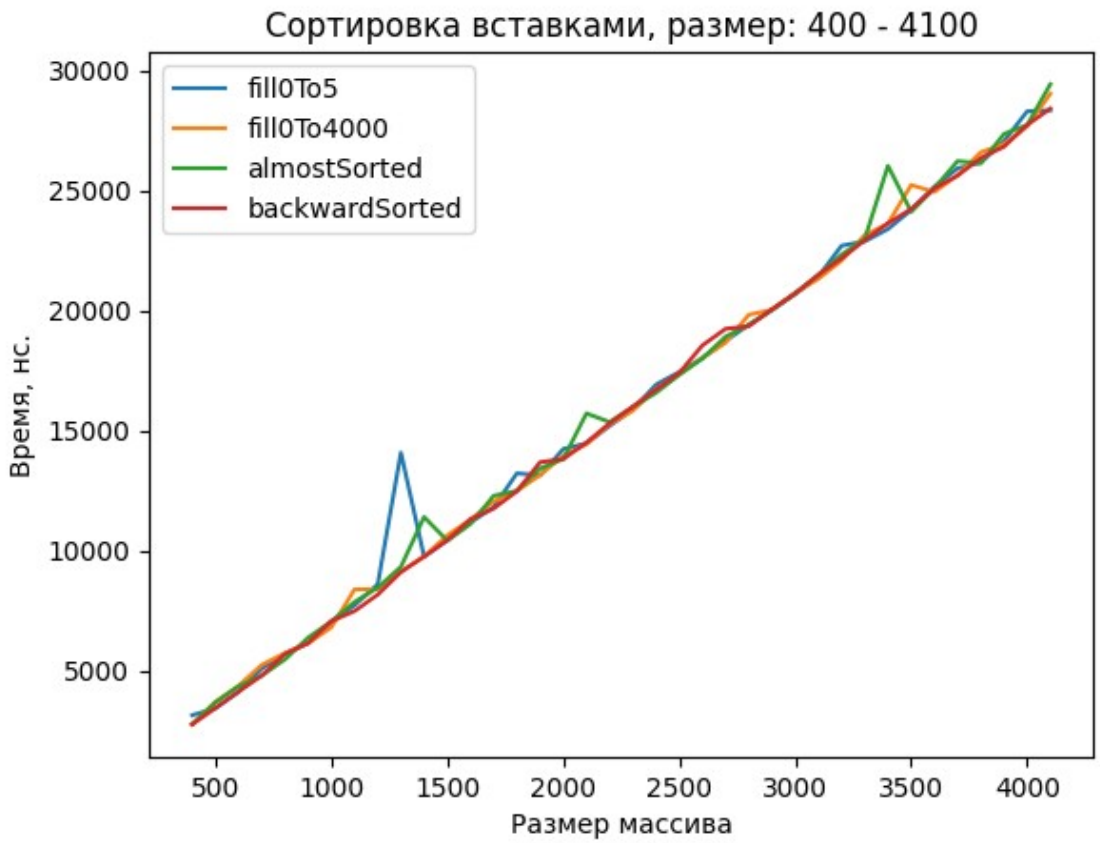


Здесь тоже ничего интересного. Тип массива, при котором сортировка покажет наихудшую асимптотику по графику определить непросто, однако можно выделить наилучший тип генерации согласно времени выполнения алгоритма - заполнение случайными числами от 0 до 5.

Сортировка вставками

```
insertionSort = data[(data['Amount'] > 300) & (data['Sort method'] ==
'insertionSort')]
types = insertionSort['Array type'].unique()
for gen_type in types:
    plot_type = insertionSort[insertionSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка вставками, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

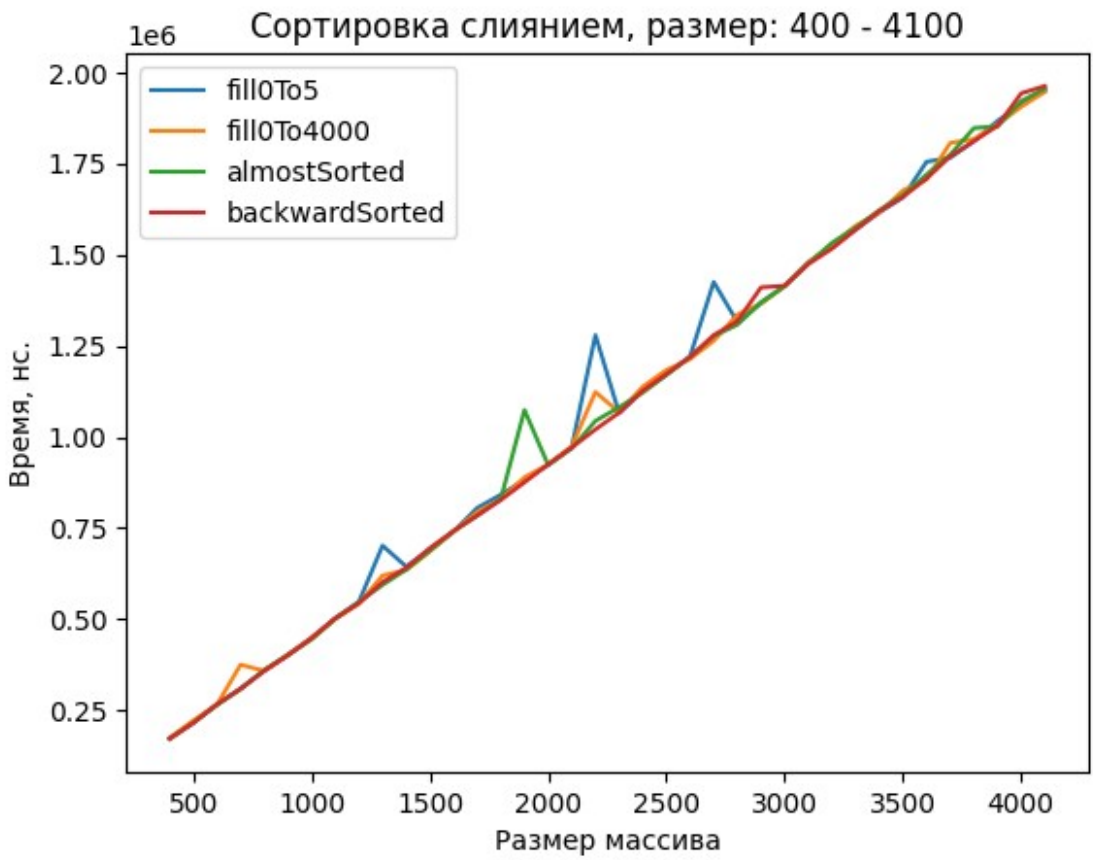


Определить наилучший и наихудший для асимптотики выполнения алгоритма тип заполнения массива тяжело. Тут ситуация аналогично пузырьковой сортировки - сортировка вставками с одинаковым временем исполнения справляется с любым типом генерации вектора. Наблюдается один четкий пик - заполнение от 0 до 5, размер массива равный ~1200.

Сортировка слиянием

```
mergeSort = data[(data['Amount'] > 300) & (data['Sort method'] ==
'mergeSort')]
types = mergeSort['Array type'].unique()
for gen_type in types:
    plot_type = mergeSort[mergeSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка слиянием, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

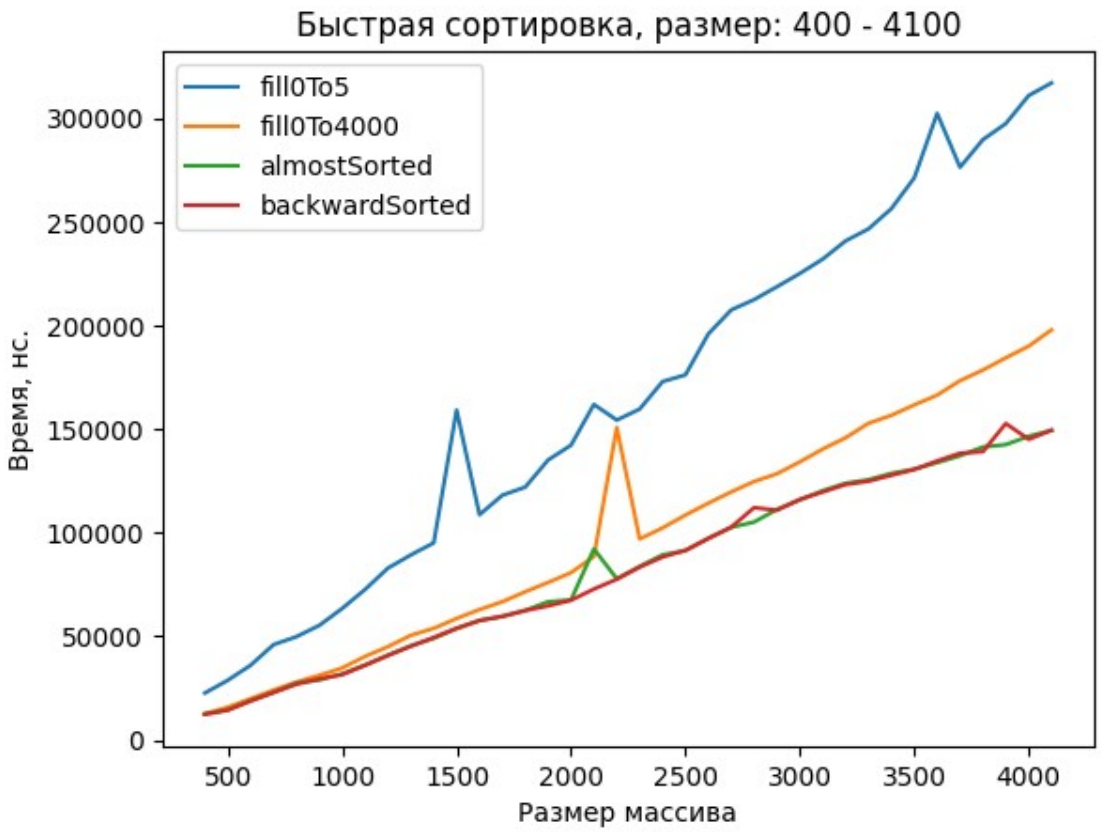


Тоже ничего удивительного. Сортировка слиянием показывает одинаковые результаты вне зависимости от типа генерации массива. Наблюдаются небольшие увеличения времени выполнения алгоритмы в случае заполнения массива случайными элементами в диапазоне от 0 до 5 и при почти отсортированном векторе.

Быстрая сортировка

```
quickSort = data[(data['Amount'] > 300) & (data['Sort method'] ==
'quickSort')]
types = quickSort['Array type'].unique()
for gen_type in types:
    plot_type = quickSort[quickSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Быстрая сортировка, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

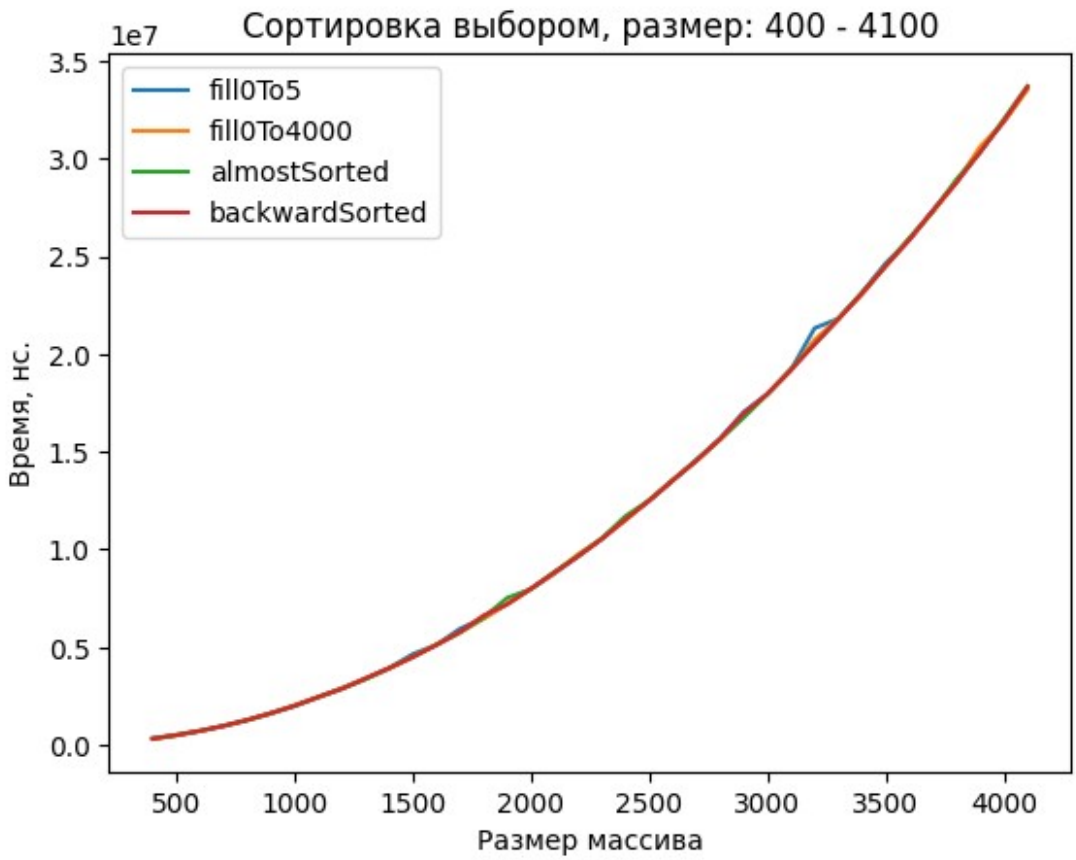


Тут уже заметны отличия от предыдущих сортировок. Быстрая сортировка хуже справляется с типом генерации массива fill0To5. Чуть лучше она сортирует массив со значениями от 0 до 4000, а для остальных двух типов заполнения показывает одинаковые результаты.

Сортировка выбором

```
selectionSort = data[(data['Amount'] > 300) & (data['Sort method'] ==
'selectionSort')]
types = selectionSort['Array type'].unique()
for gen_type in types:
    plot_type = selectionSort[selectionSort['Array type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка выбором, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

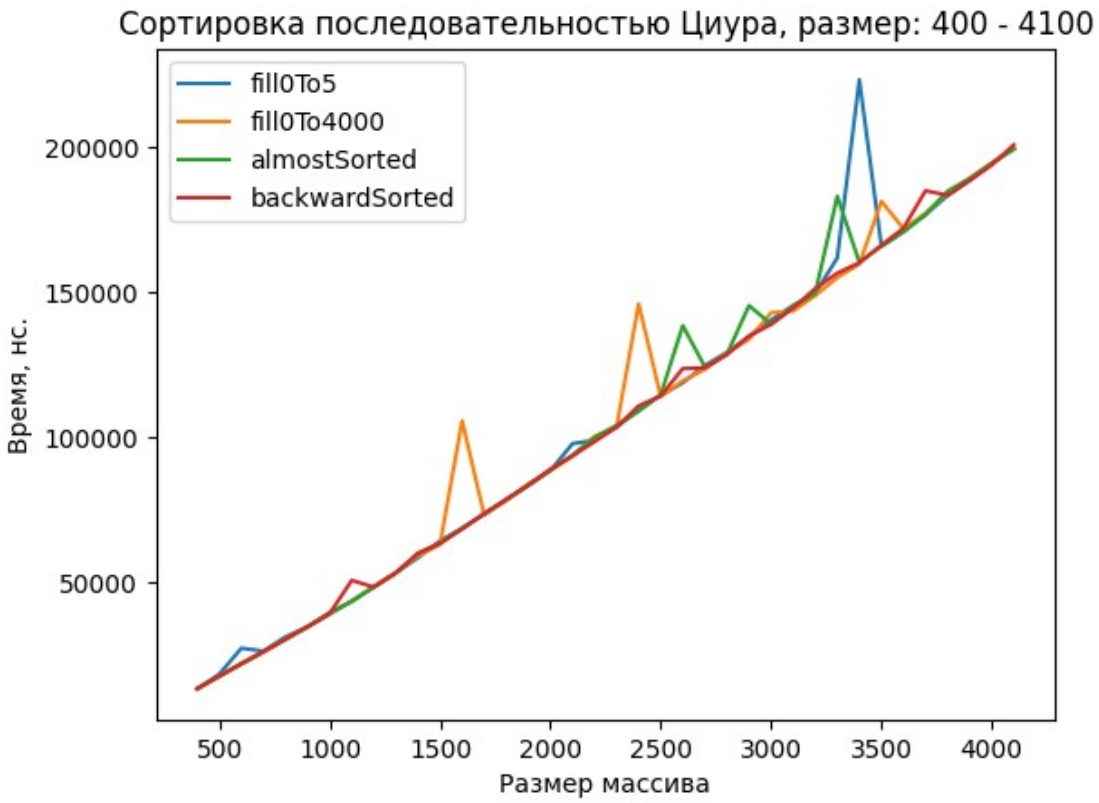


Красиво. Сказать нечего. Сортировка выбором одинаково справляется со всеми видами массивов. Заметен один маленький пик для заполнения числами от 0 до 5 при количестве элементов ~3250.

Сортировка последовательностью Циура

```
shellSortCiuraSequence = data[(data['Amount'] > 300) & (data['Sort
method'] == 'shellSortCiuraSequence')]
types = shellSortCiuraSequence['Array type'].unique()
for gen_type in types:
    plot_type = shellSortCiuraSequence[shellSortCiuraSequence['Array
type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка последовательностью Циура, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```

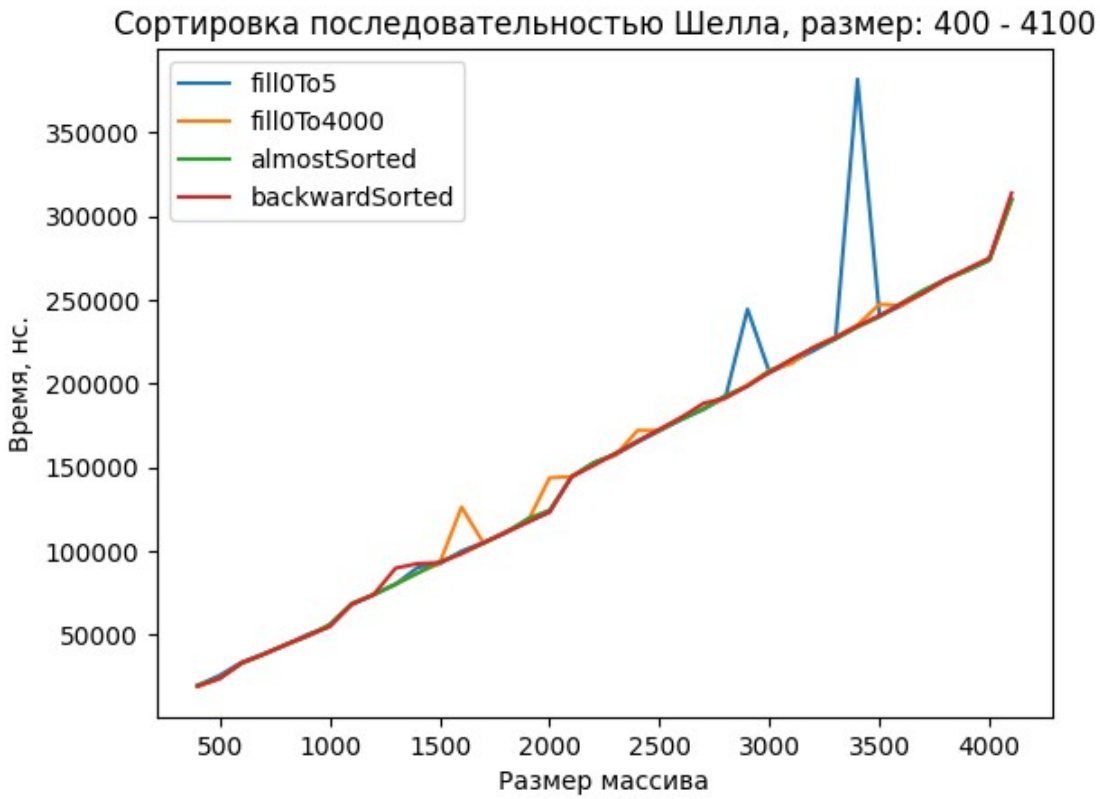


Здесь уже присутствуют ярко выраженные пики, самый крутой из которых в случае заполенния массив значениями в диапазоне от 0 до 5

Сортировка последовательностью Шелла

```
shellSortShellSequence = data[(data['Amount'] > 300) & (data['Sort
method'] == 'shellSortShellSequence')]
types = shellSortShellSequence['Array type'].unique()
for gen_type in types:
    plot_type = shellSortShellSequence[shellSortShellSequence['Array
type'] == gen_type]
    plt.plot(plot_type['Amount'], plot_type['Time'], label=gen_type)

plt.title('Сортировка последовательностью Шелла, размер: 400 - 4100')
plt.xlabel('Размер массива')
plt.ylabel('Время, нс.')
plt.legend()
plt.show()
```



Похожая ситуация, однако видно, что при размере массива около 3500 единиц время работы сортировки значений от 0 до 5 резко возрастает.