МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Тема: «Алгоритмы сортировки»

Студент гр. 2302	______________________	Фролов А. Э.
Преподаватель:	______________________	Пестерев Д. О.

Санкт-Петербург
2023

1. Постановка задачи

1. Реализовать следующие алгоритмы сортировки:
   - Сортировка выбором
   - Сортировка вставками
   - Сортировка пузырьком
   - Сортировка слиянием
   - Быстрая сортировка
   - Сортировка Шелла (не менее трех различных последовательностей, желательно приводящих к разным асимптотикам)
   - Пирамидальная сортировка
   - Timsort
   - IntroSort
2. Экспериментально определить время работы алгоритмов при различных размерах массива для:
   - Отсортированного массива
   - Почти отсортированного массива
   - Обратно отсортированного массива
   - Массива, о структуре которого дополнительных данных не дано

Постараться определить асимптотику для лучшего/среднего/худшего случая путем анализа экспериментальных данных и применения к ним нелинейной регрессии.

2. Оценка алгоритмов.

2.1. SelectionSort

```
void SortFunctions::SelectionSort(vector<int>& A, int low, int high)
{
    int maxInd;

    for (int i = low; i <= high; i++)
    {
        maxInd = low;
        for (int j = low + 1; j <= (high - i); j++)
        {
            if (A[j] > A[maxInd])
                maxInd = j;
        }
        swap(A[maxInd], A[high - i]);
    }
}
```

Временная сложность: Сортировка состоит из базовых операций и двух вложенных циклов for, соответственно сложность алгоритма:

$$T(n) = \Theta(n * n) = \Theta(n^2)$$

Лучший случай: $\Theta(n^2)$

Средний случай: $\Theta(n^2)$

Худший случай: $\Theta(n^2)$

Дополнительная память: $\Theta(1)$

Устойчивость: может быть как устойчивой, так и не устойчивой.

2.2. InsertionSort

```
void SortFunctions::InsertionSort(vector<int>& A, int low, int high)
{
    for (int i = low + 1; i <= high; i++)
    {
        for (int j = i; j > low; j--)
        {
            if (A[j] < A[j - 1])
            {
                swap(A[j], A[j - 1]);
            }
            else
            {
                break;
            }
        }
    }
}
```

Временная сложность: Сортировка состоит из базовых операций и двух вложенных циклов for. Количество итераций второго цикла зависит от состояния массива.

Лучший случай (Sorted array): $\Theta(\mathrm{n})$

Средний случай (Random Array): $\Theta(n^2)$

Худший случай (BackSorted array): $\Theta(n^2)$

Дополнительная память: $\Theta(1)$

Устойчивость: может быть как устойчивой, так и не устойчивой.

2.3. BubbleSort

```
void SortFunctions::BubbleSort(vector<int>& A, int low, int high)
{
    for (int i = low; i <= high; i++)
    {
        for (int j = low; j < (high - i); j++)
        {
            if (A[j] > A[j + 1])
            {
                swap(A[j], A[j+1]);
            }
        }
    }
}
```

Временная сложность: Сортировка состоит из базовых операций и двух вложенных циклов for. Количество итераций второго цикла зависит от состояния массива.

Лучший случай: $\Theta(n^2)$

Средний случай: $\Theta(n^2)$

Худший случай: $\Theta(n^2)$

Дополнительная память: $\Theta(1)$

Устойчивость: может быть как устойчивым, так и не устойчивым.

2.4. MergeSort

```
void MSMerge(vector<int>& A,
     int low, int mid, int high)
{
    int indA1 = low, indA2 = mid + 1, indR = 0;
    int resultLength = high - low + 1;
    vector<int> result(resultLength);

    while (indA1 < (mid + 1) && indA2 < (high +
1))
    {
        if (A[indA1] > A[indA2])
        {
            result[indR] = A[indA2];
            indA2++;
        }
        else
```

```
void SortFunctions::MergeSort
    (vector<int>& A, int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort(A, low, mid);
        MergeSort(A, mid + 1, high);

        MSMerge(A, low, mid, high);
    }
}
```

```
        {
            result[indR] = A[indA1];
            indA1++;
        }
        indR++;
    }
    if (indA1 == (mid + 1))
    {
        while (indR < resultLength)
        {
            result[indR] = A[indA2];
            indA2++; indR++;
        }
    }
    else // if (indA2 == high+1)
    {
        while (indR < resultLength)
        {
            result[indR] = A[indA1];
            indA1++; indR++;
        }
    }

    // ПРЕОБРАЗУЕМ ИСХОДНЫЙ МАССИВ,
   // ВМЕСТО СОЗДАНИЯ ВСПОМОГАТЕЛЬНЫХ
    for (int i = 0; i < resultLength; i++)
    {
        A[low + i] = result[i];
    }
}
```

Временная сложность: очевидно, что сложность функции слияния равна $\Theta(n)$, а количество вызовов рекурсивной функции - $\Theta(\log(n))$. Соответственно временная сложность алгоритма:

$$T(n) = \Theta(n) * \Theta(\log(n)) = \Theta(\mathrm{n} * \log(\mathrm{n}))$$

Лучший случай: $\Theta(\mathrm{n} * log(n))$

Средний случай: $\Theta(\mathrm{n} * log(n))$

Худший случай: $\Theta(\mathrm{n} * log(n))$

Дополнительная память: $\Theta(\mathrm{n})$

Устойчивость: может быть как устойчивым, так и не устойчивым.

2.5. QuickSort

```
int QSPartition(std::vector<int>& A, int
low, int high)
{
    std::swap(A[(high + low) / 2],
A[high]);

    int pivot = A[high];
    int i = low;
    int j = high - 1;

    while (true) {
        while (i <= high && A[i] < pivot) {
            i++;
        }

        while (j >= low && A[j] >= pivot) {
            j--;
        }
```

```
void SortFunctions::QuickSort(vector<int>& A,
 int low, int high) {
    if (low < high) {
        int pivotIndex = QSPartition(A, low,
       high);
        QuickSort(A, low, pivotIndex - 1);
        QuickSort(A, pivotIndex + 1, high);
    }
}
```

```
        if (i <= j)
            std::swap(A[i], A[j]);
        else
            break;
    }

    std::swap(A[i], A[high]);
    return i;
}
```

Временная сложность: очевидно, что сложность функции Partition равна $\Theta(\mathrm{n})$. Сложность рекурсивной функции зависит от того, насколько близко к центру выбран pivot.

Лучший случай: $\Theta(\mathrm{n} * \log(\mathrm{n}))$

Средний случай: $\Theta(\mathrm{n} * \log(\mathrm{n}))$

Худший случай: $\Theta(n^2)$

Дополнительная память: $\Theta(1)$

Устойчивость: не устойчивый.

2.6. ShellSort1 (классический метод)

```
void SortFunctions::ShellSort1(vector<int>& A, int low, int high)
{
    int size = (high - low + 1);
    int d = size / 2;
    while (d > 0)
    {
        for (int i = low; i <= (high - d); i++)
        {
            if (A[i] > A[i + d])
            {
                swap(A[i], A[i+d]);
            }
        }
        d /= 2;
    }
    InsertionSort(A, low, high);
}
```

Временная сложность: дать конкретную сложность данному алгоритму сложно, поэтому оценим его в соответствии с графиками.

Лучший случай: $\Theta(n * \log(n))$

Средний случай: $\Theta(n * (\log(n))^2)$

Худший случай: $\Theta(n * (\log(n))^2)$

Дополнительная память: $\Theta(1)$

Устойчивость: не устойчивый.

2.7. ShellSort2 (метод Седжвика)

```
void SortFunctions::ShellSort2(vector<int>& A, int low, int high)
{
	const int SIZE = 28;
	int steps[] = {
		1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001,
		36289, 64769, 146305, 260609, 587521, 1045505, 2354689,
		4188161, 9427969, 16764929, 37730305, 67084289, 150958081,
		268386305, 603906049, 1073643521
	};
	/*
		if (steps.size() % 2 == 0)
			d = 9 * pow(2, steps.size()) - 9 * pow(2, steps.size() / 2) + 1;
		else
			d = 8 * pow(2, steps.size()) - 6 * pow(2, (steps.size() + 1) / 2) + 1;
	*/
	int ind = -1, d;
	for (int i = 0; (ind+1) < SIZE; ind++)
	{
		if (steps[ind+1] > (high - low + 1))
			break;
	}
	while (ind > 0)
	{
		d = steps[ind];
		for (int i = low; i <= (high - d); i++)
		{
			if (A[i] > A[i + d])
			{
				swap(A[i], A[i + d]);
			}
		}
		ind--;
	}
	InsertionSort(A, low, high);
}
```

Временная сложность: дать конкретную сложность данному алгоритму сложно, поэтому оценим его в соответствии с графиками.

Лучший случай: $\Theta(n * \log(n))$

Средний случай: $\Theta(n * (\log(n))^2)$

Худший случай: $\Theta(n * (\log(n))^2)$

Дополнительная память: $\Theta(1)$

Устойчивость: не устойчивый.

2.8. ShellSort3 (последовательность Марцина Циура)

```
void SortFunctions::ShellSort3(vector<int>& A, int low, int high)
{
	const int SIZE = 9;
	int steps[SIZE] = { 1, 4, 10, 23, 57, 132, 301, 701, 1750 };
	int ind = -1, d;
	for (int i = 0; (ind + 1) < SIZE; ind++)
	{
		if (steps[ind + 1] > (high - low + 1))
			break;
	}

	while (ind > 0)
	{
		d = steps[ind];
```

```
                for (int i = low; i <= (high - d); i++)
                {
                        if (A[i] > A[i + d])
                        {
                                swap(A[i], A[i + d]);
                        }
                }
                ind--;
        }
        InsertionSort(A, low, high);
}
```

Временная сложность: дать конкретную сложность данному алгоритму сложно, поэтому оценим его в соответствии с графиками.

Лучший случай: $\Theta(n * \log(n))$

Средний случай: $\Theta(n * (\log(n))^2)$

Худший случай: $\Theta(n * (\log(n))^2)$

Дополнительная память: $\Theta(1)$

Устойчивость: не устойчивый.

2.9. HeapSort

```
void Heapify(vector<int>& A, int n, int i)
{
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int largest = i;

    if (left < n && A[left] > A[largest])
        largest = left;
    if (right < n && A[right] > A[largest])
        largest = right;

    if (largest != i)
    {
        swap(A[largest], A[i]);
        Heapify(A, n, largest);
    }
}
```

```
void SortFunctions::HeapSort(vector<int>& A,
int low, int high)
{
    if (low >= high)
        return;
    int size = (high - low + 1);
    // перемещение максимального элемента в
корень кучи
    for (int i = (high + 1) / 2 - 1; i >= low;
i--)
    {
        Heapify(A, size, i);
    }

    // Один за другим извлекаем элементы из
кучи
    for (int i = high; i >= low; i--)
    {
        // Перемещаем текущий корень в конец
        swap(A[0], A[i]);

        // heapify на уменьшенной куче
        Heapify(A, i, 0);
    }
}
```

Временная сложность: перемещение максимального элемента в корень - $\Theta(n)$, дальнейшая сортировка - $\Theta(\log(n))$. Итоговая сложность:

$$T(n) = \Theta(n) * \Theta(\log(n)) = \Theta(\mathrm{n} * \log(\mathrm{n}))$$

Лучший случай: $\Theta(n * \log(n))$

Средний случай: $\Theta(n * \log(n))$

Худший случай: $\Theta(n * \log(n))$

Дополнительная память: $\Theta(1)$

Устойчивость: не устойчивый.

2.10. TimSort

```
int GetMinRun(int n)
{
    int r = 0;
    while (n >= 64) {
        r |= n & 1;
        n >>= 1;
    }
    return n + r;
}

void TSMerge(vector<int>& A, int low, int mid,
int high)
{
    int sizeLeft = mid - low + 1;
    int sizeRight = high - mid;

    vector<int> tempLeft(sizeLeft);
    vector<int> tempRight(sizeRight);

    for (int i = 0; i < sizeLeft; i++)
        tempLeft[i] = A[low + i];
    for (int i = 0; i < sizeRight; i++)
        tempRight[i] = A[mid + 1 + i];

    int i = 0, j = 0, k = low;
    while (i < sizeLeft && j < sizeRight)
    {
        if (tempLeft[i] <= tempRight[j])
            A[k++] = tempLeft[i++];
        else
            A[k++] = tempRight[j++];
    }
    while (i < sizeLeft)
        A[k++] = tempLeft[i++];
    while (j < sizeRight)
        A[k++] = tempRight[j++];
}
```

```
void SortFunctions::TimSort(vector<int>& A,
int low, int high)
{
    if (low >= high)
        return;
    int size = high - low + 1;

    int minRun = GetMinRun(size);
    for (int i = 0; i < size; i += minRun)
        InsertionSort(A, i, min(minRun, size -
i) - 1);

    for (int sizeRun = minRun; sizeRun < size;
sizeRun *= 2)
    {
        for (int i = 0; i < size; i += 2 *
sizeRun)
        {
            int mid = min(i + sizeRun - 1,
size - 1);
            int right = min(i + 2 * sizeRun -
1, size - 1);
            TSMerge(A, i, mid, right);
        }
    }
}
```

Временная сложность: слияние выполняется за n операций, а кол-во таких операций – log(n). Значит, итоговая сложность:

$$T(n) = \Theta(n) * \Theta(\log(n)) = \Theta(\mathrm{n} * \log(\mathrm{n}))$$

Лучший случай: $\Theta(n)$

Средний случай: $\Theta(n * \log(n))$

Худший случай: $\Theta(n * \log(n))$

Дополнительная память: $\Theta(\mathrm{n})$

Устойчивость: может быть как устойчивым, так и не устойчивым.

2.11. IntroSort

```
// Функция для выбора опорного элемента
```

```
// Функция для выполнения сортировки IntroSort
```

```
int choosePivot(vector<int>& A, int low, int
high) {
    int mid = (low + high) / 2;

    if (A[low] > A[high]) {
        swap(A[low], A[high]);
    }
    if (A[low] > A[mid]) {
        swap(A[low], A[mid]);
    }
    if (A[mid] > A[high]) {
        swap(A[mid], A[high]);
    }

    swap(A[mid], A[high - 1]);
    return high - 1;
}

// Функция для разделения массива и выполнения
сортировки "Хоара"
int ISpartition(vector<int>& A, int low, int
high) {
    int pivotIndex = choosePivot(A, low,
high);
    int pivot = A[pivotIndex];

    int i = low;
    int j = high - 1;

    while (true) {
        while (A[++i] < pivot) {}
        while (A[--j] > pivot) {}
        if (i < j) {
            swap(A[i], A[j]);
        }
        else {
            break;
        }
    }

    swap(A[i], A[high - 1]);
    return i;
}
```

```
void introSort(vector<int>& A, int low, int
high, int depthLimit) {
    while (high - low > INSERTION_THRESHOLD) {
        if (depthLimit == 0) {
            SortFunctions::HeapSort(A, low,
high);
            return;
        }
        depthLimit--;

        int pivotIndex = partition(A, low,
high);
        introSort(A, pivotIndex + 1, high,
depthLimit);

        high = pivotIndex;
    }

    SortFunctions::InsertionSort(A, low,
high);
}

// Функция для вызова IntroSort
void SortFunctions::IntroSort(vector<int>& A,
int low, int high) {
    int depthLimit = 2 * log(high - low + 1);
    introSort(A, 0, high - low, depthLimit);
}
```

Временная сложность: кол-во вызовов рекурсии – log(n), сложность каждой из которых - $\Theta(n)$. Значит, итоговая сложность:

$$T(n) = \Theta(n) * \Theta(\log(n)) = \Theta(\mathrm{n} * \log(\mathrm{n}))$$

Лучший случай: $\Theta(n * \log(n))$

Средний случай: $\Theta(n * \log(n))$

Худший случай: $\Theta(n * \log(n))$

Дополнительная память: $\Theta(1)$

Устойчивость: не устойчивый.

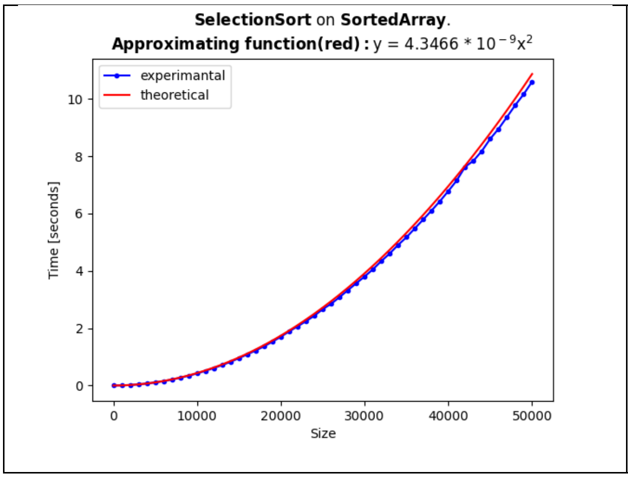
3. Экспериментальные графики и вывод о работе алгоритмов.

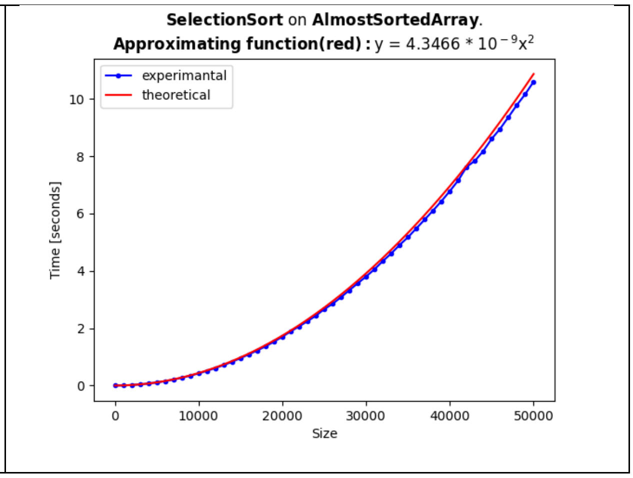
Ниже представлены экспериментальные графики зависимости времени (в секундах) от размера массива (от 1 до 50000 элементов) с шагом в 1% = 500 элементов. Проще говоря, рассматривались следующие размеры массива:

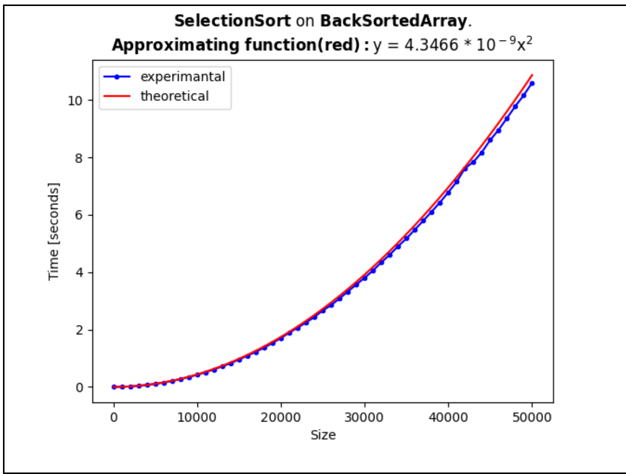
$$[1, 500, 1000, 1500, \dots, 49500, 50000]$$

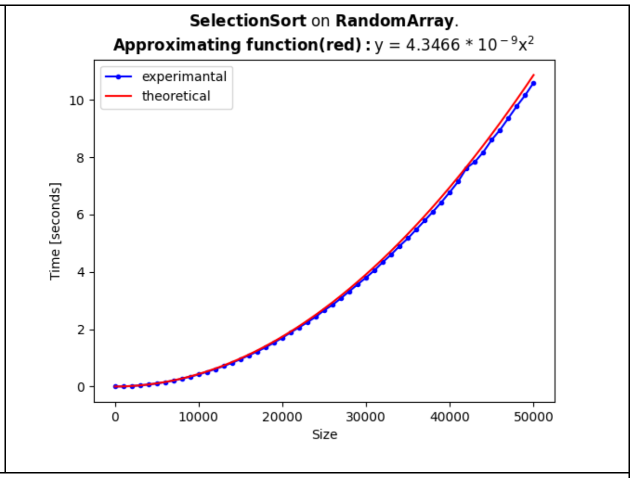
Чтобы получить наиболее достоверные данные (избавиться от резкого возрастания и моментального падения времени работы), в программе имеется переменная **repeat**, отвечающая за количество повторов сортировки массива одного и того же размера, чтобы впоследствии сохранить только наилучшее время работы. Значение переменной **repeat** устанавливалось уникально для каждого алгоритма (поэтому в приложенной программе сохранен только один вариант, использованный для алгоритма QuickSort), но чтобы было не менее 2 повторов. Также переменная **repeat** выбиралась так, чтобы количество повторов было больше маленьких размеров массива. Таким образом были получены наиболее достоверные графики.

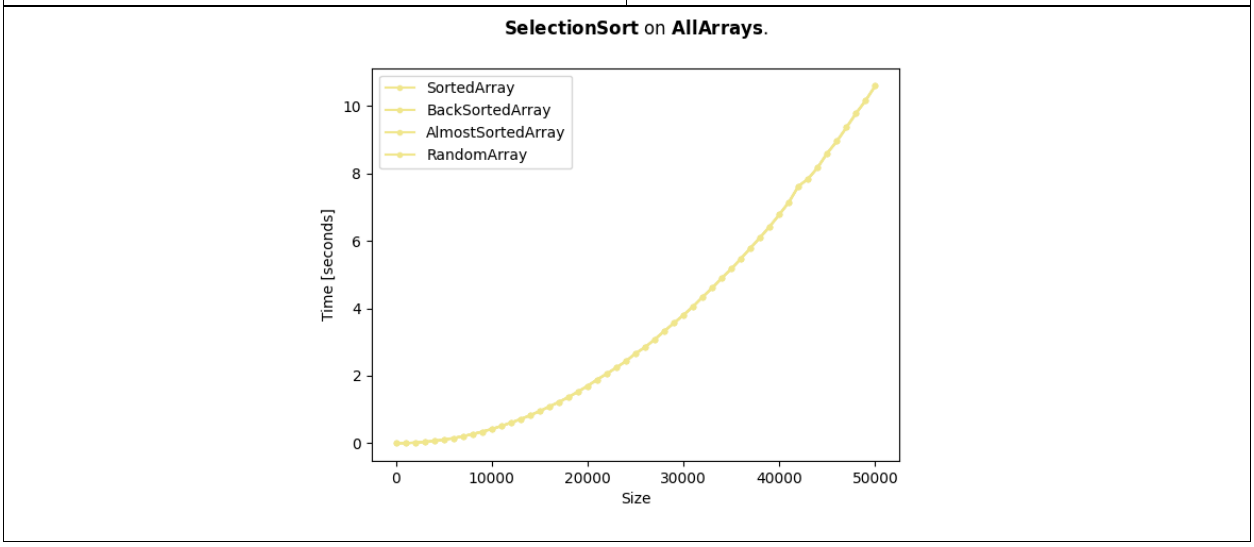
3.1. SelectionSort





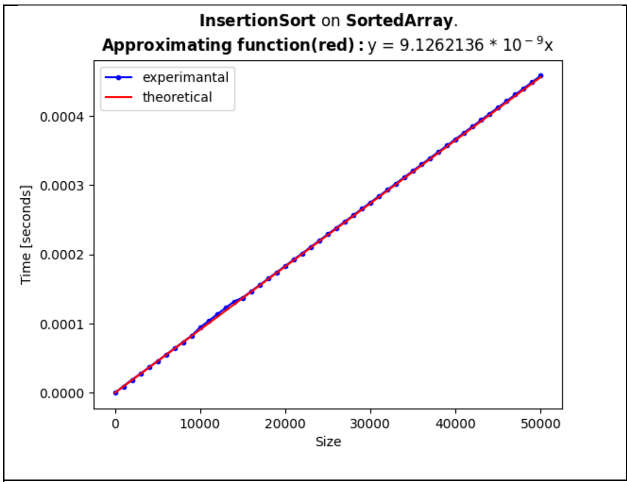


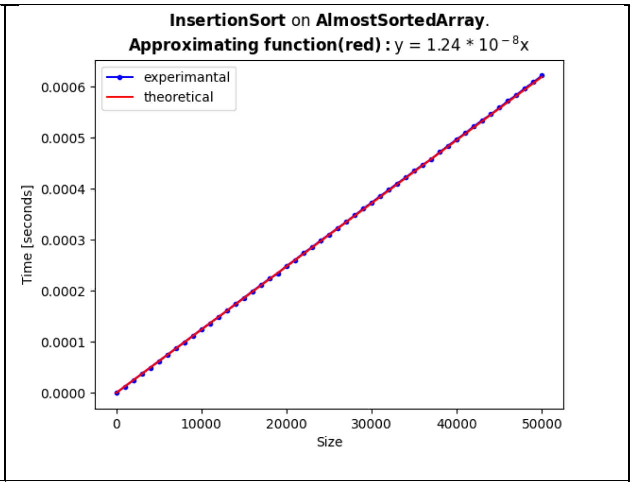


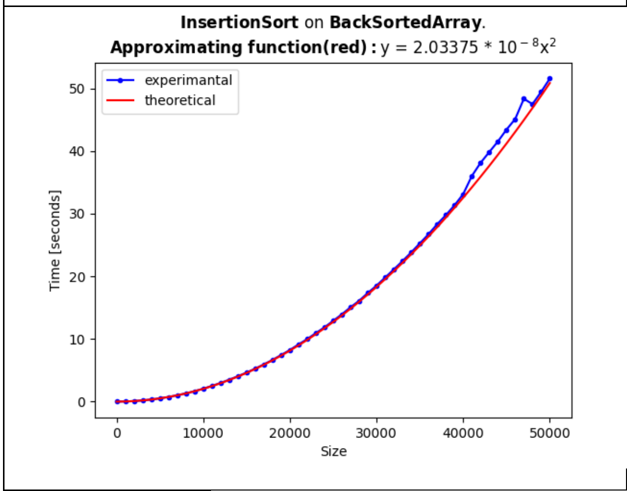


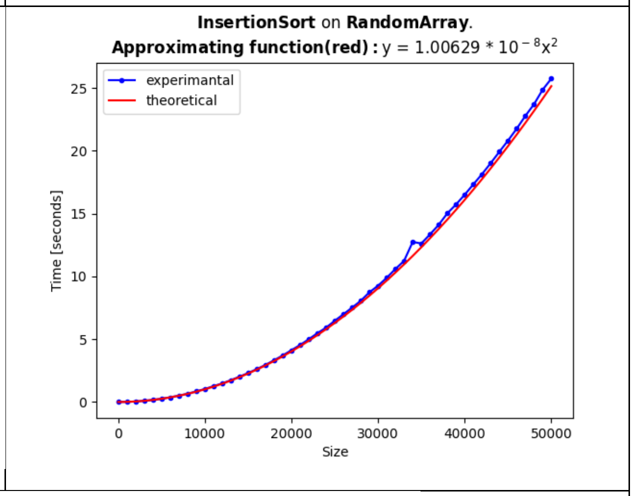
Вывод: алгоритм подходит для маленьких массивов.

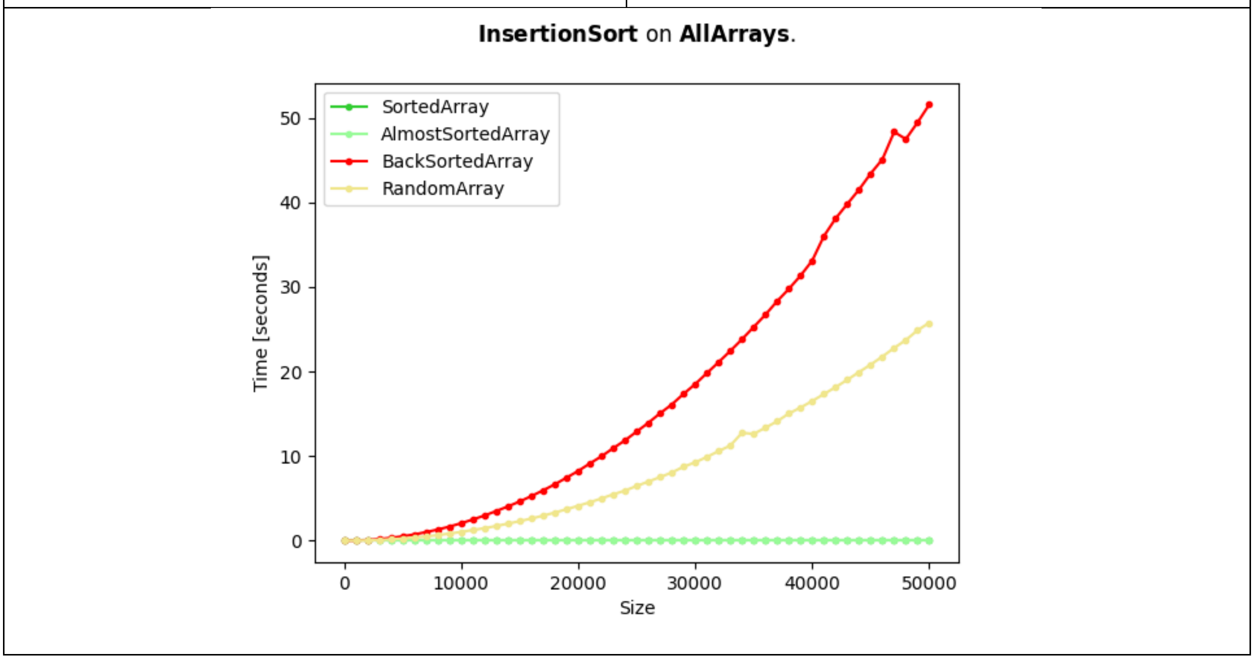
3.2. InsertionSort





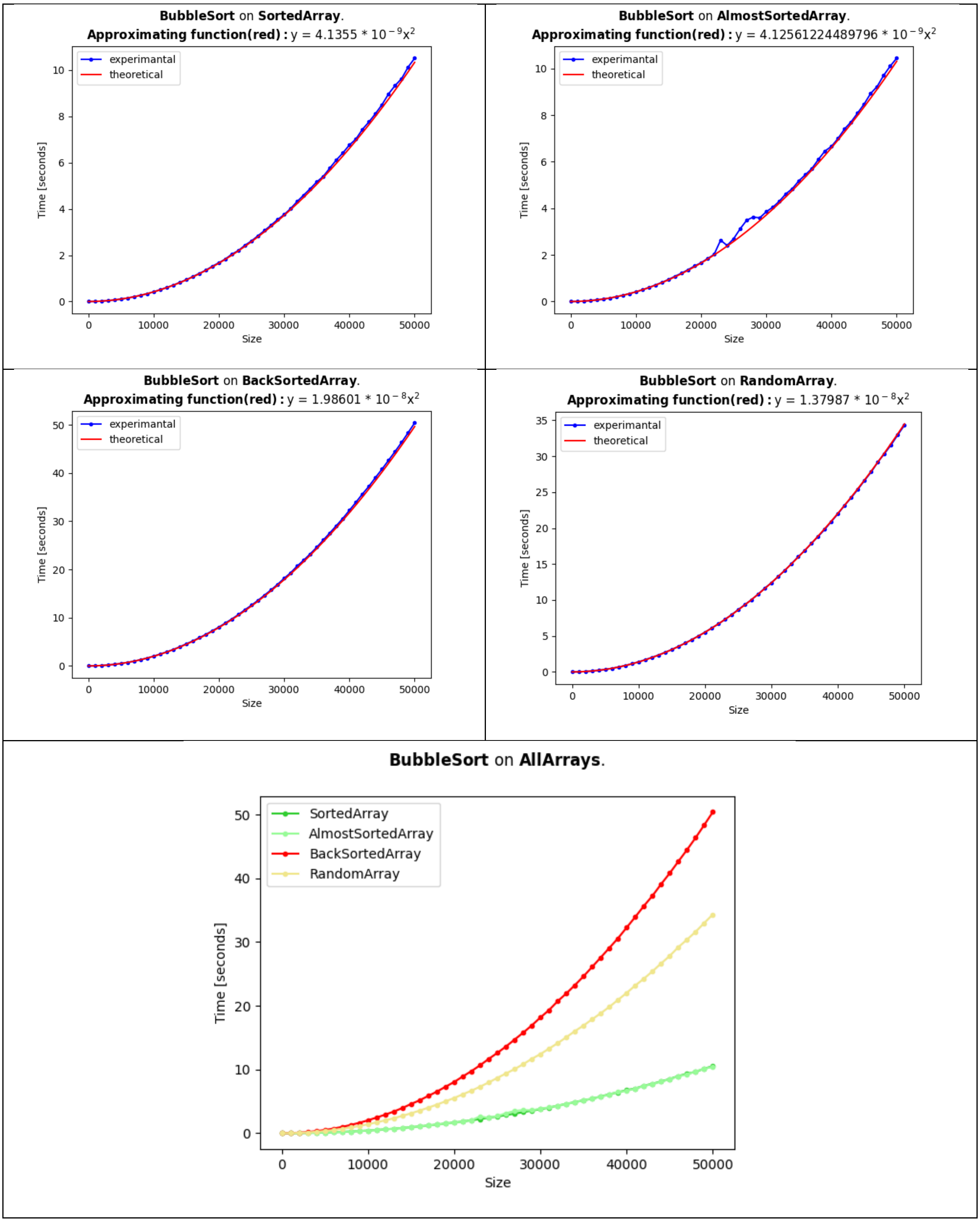






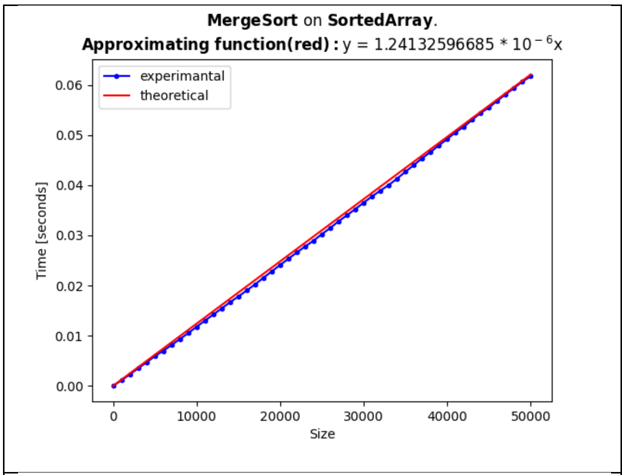
Вывод: алгоритм подходит для маленьких массивов или отсортированных (почти отсортированных) массивов.

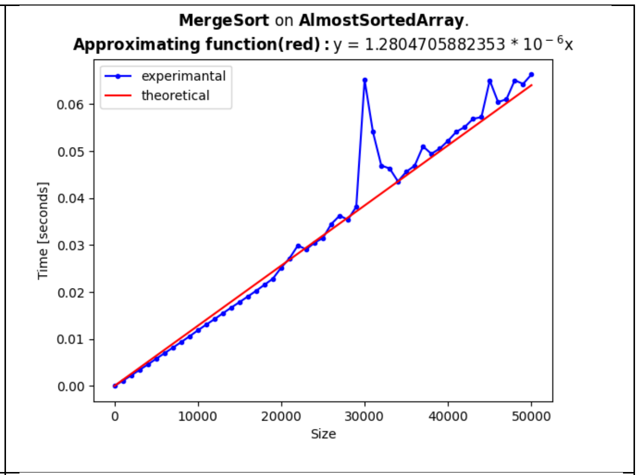
3.3. BubbleSort

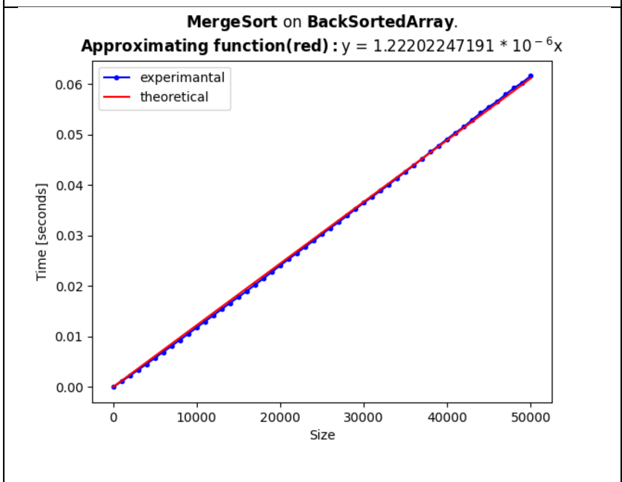


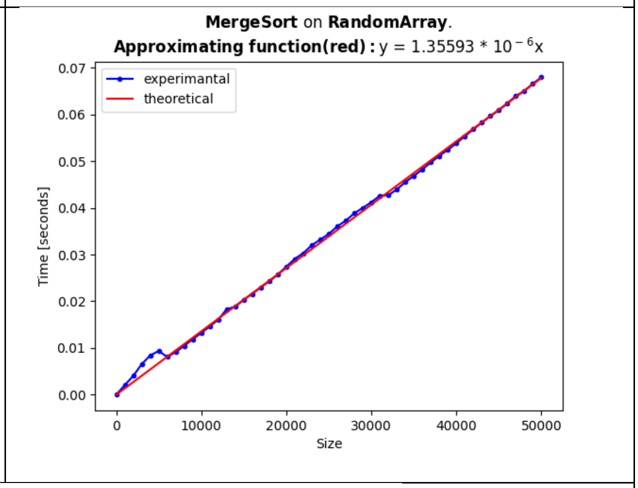
<u>Вывод: алгоритм подходит для маленьких массивов.</u>

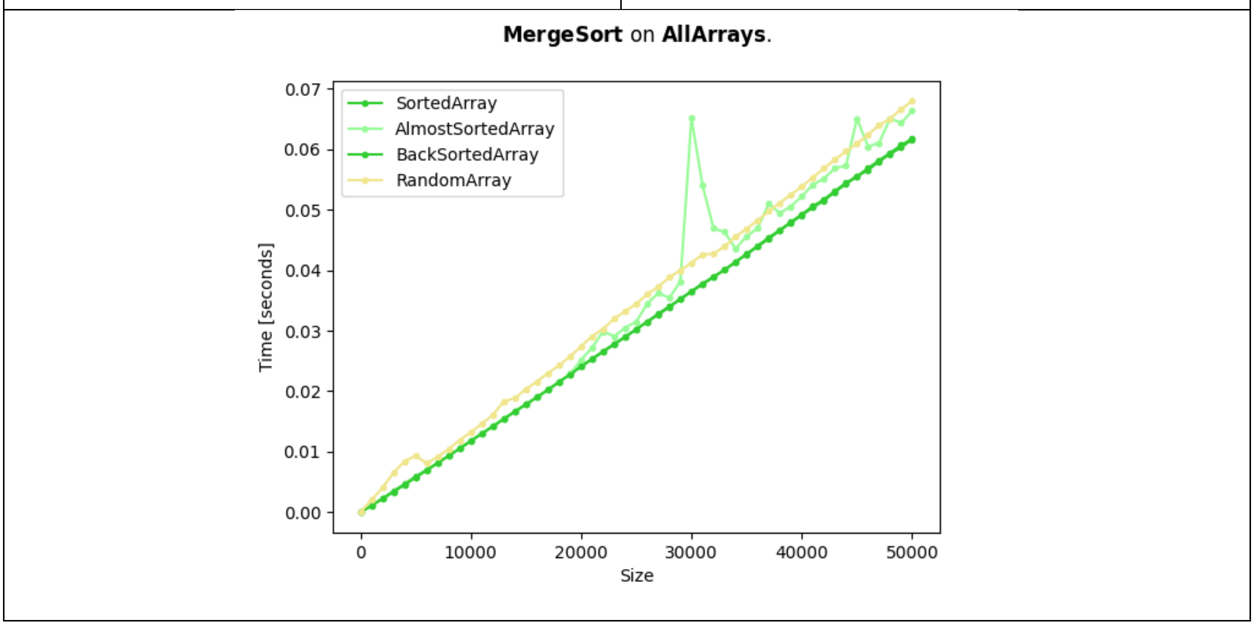
3.4. MergeSort





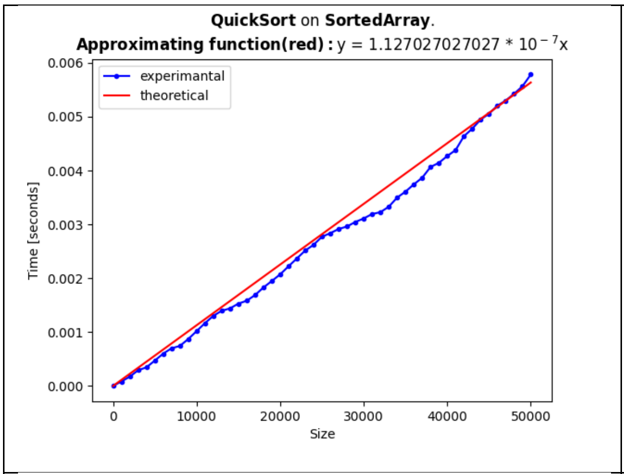


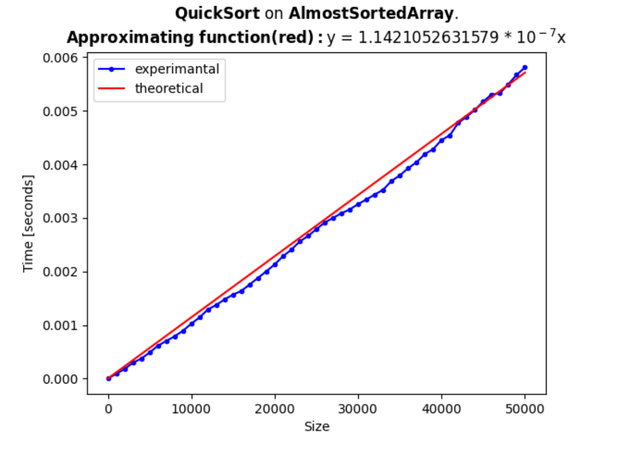


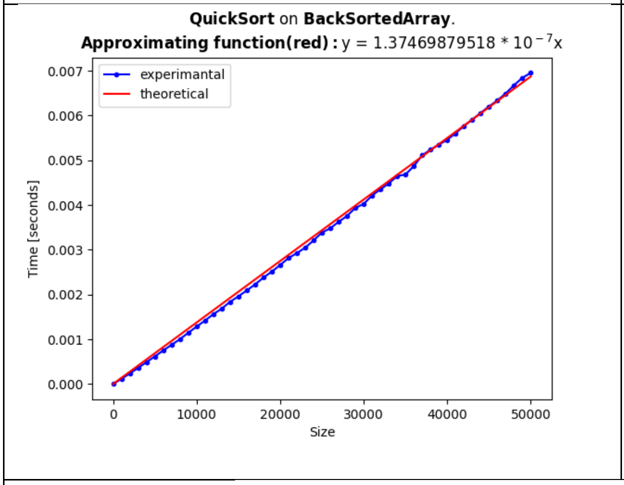


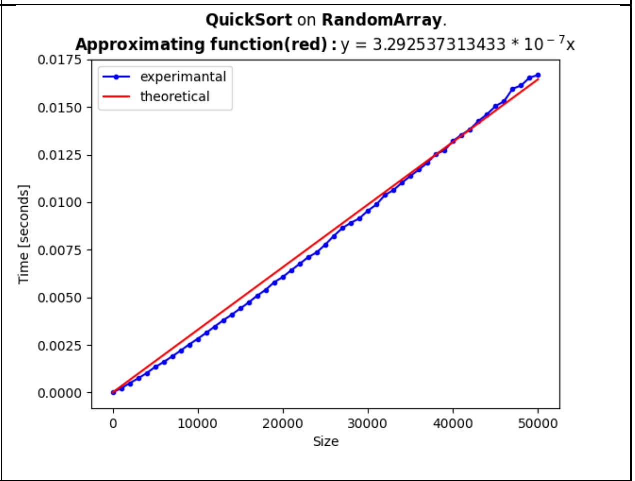
Вывод: алгоритм подходит для любого массива.

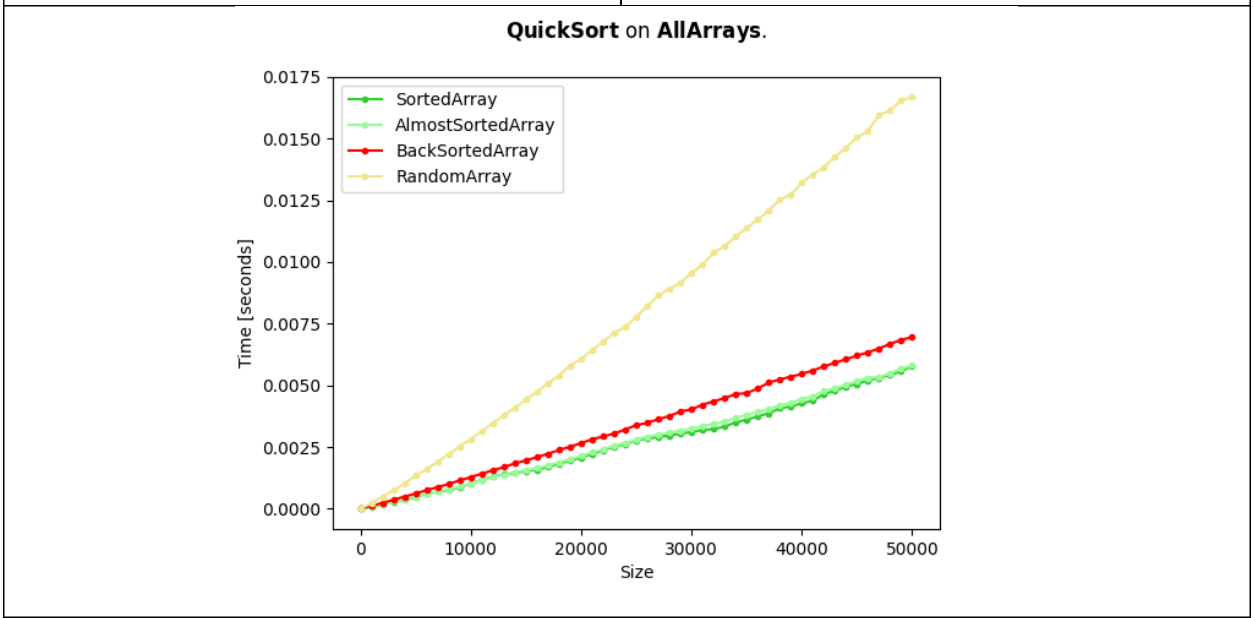
3.5. QuickSort





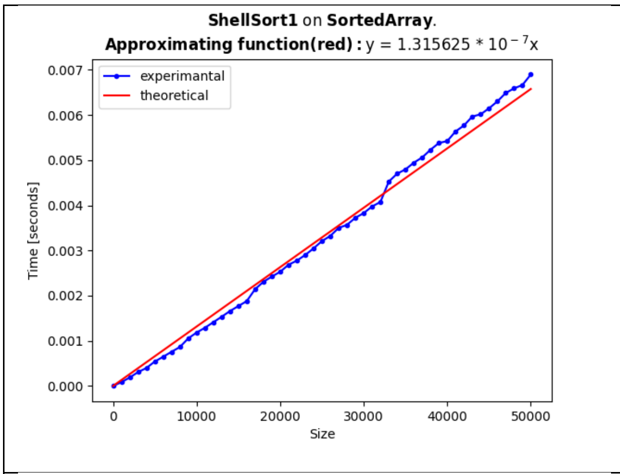


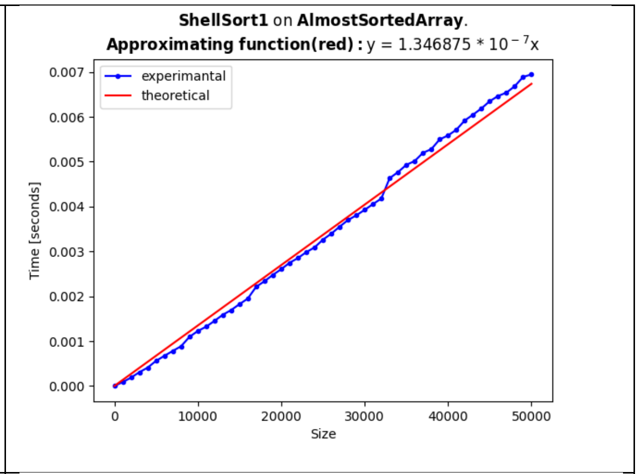


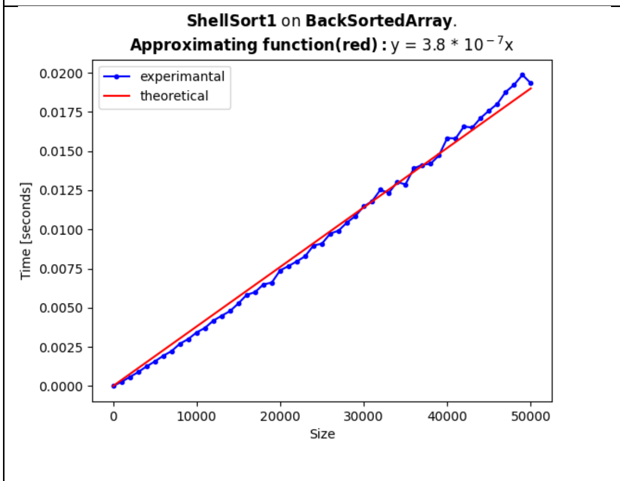


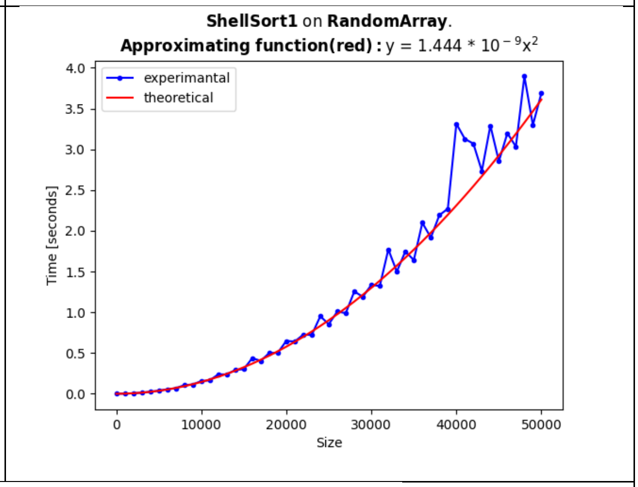
Вывод: алгоритм подходит для любого массива.

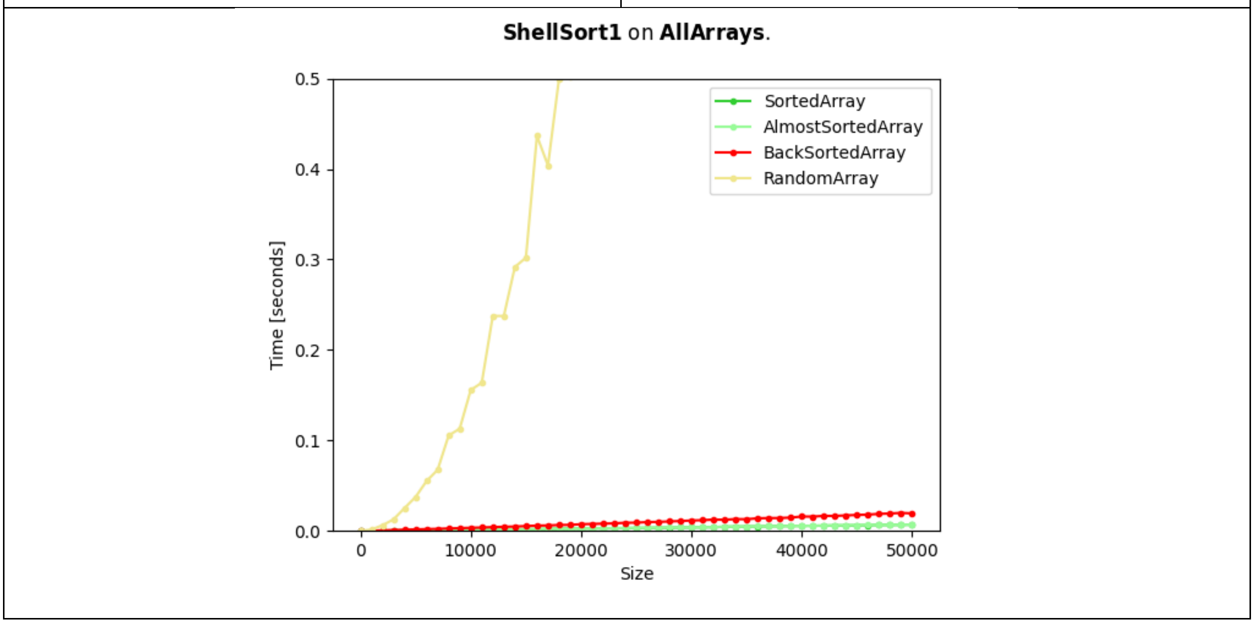
3.6. ShellSort1 (классический метод)





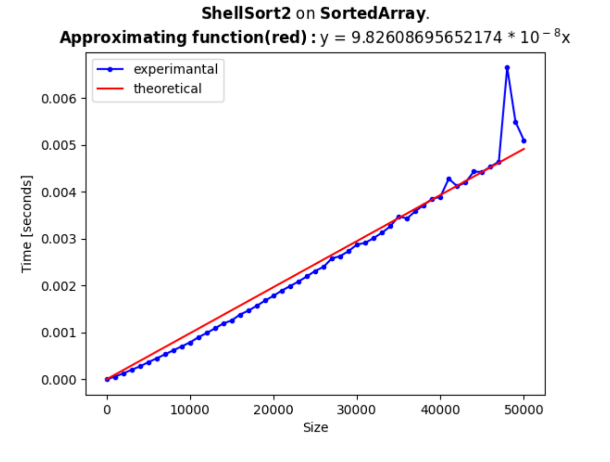


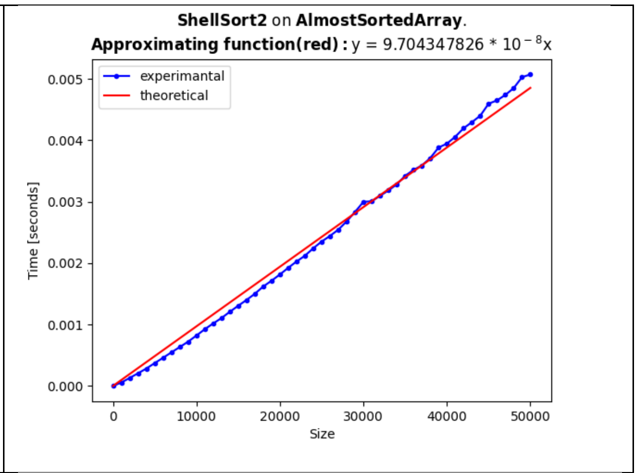


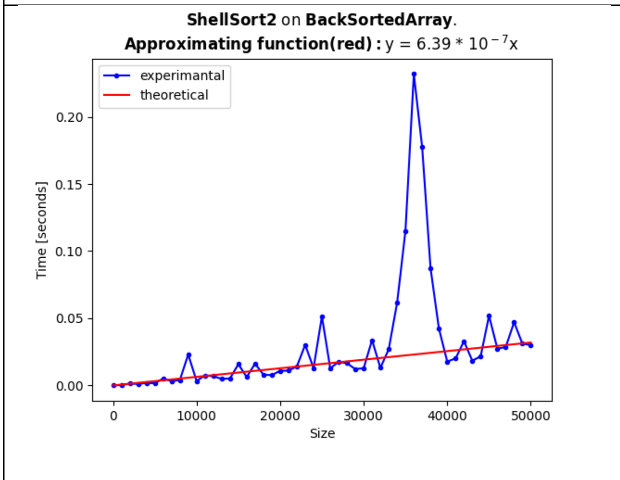


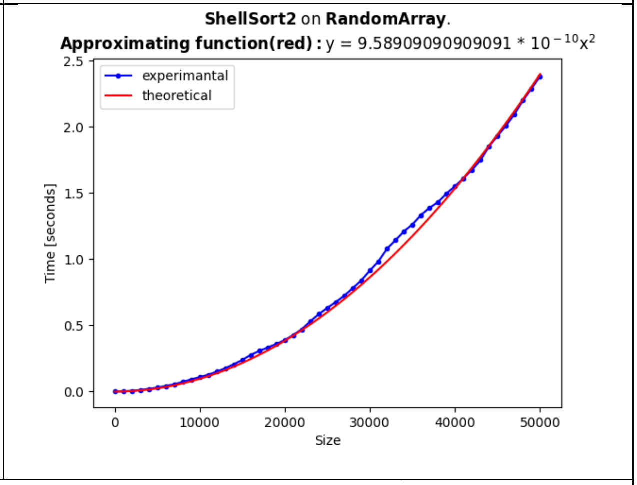
Вывод: алгоритм подходит для маленьких массивов или отсортированного (в том числе почти отсортированного или обратно отсортированного) массива.

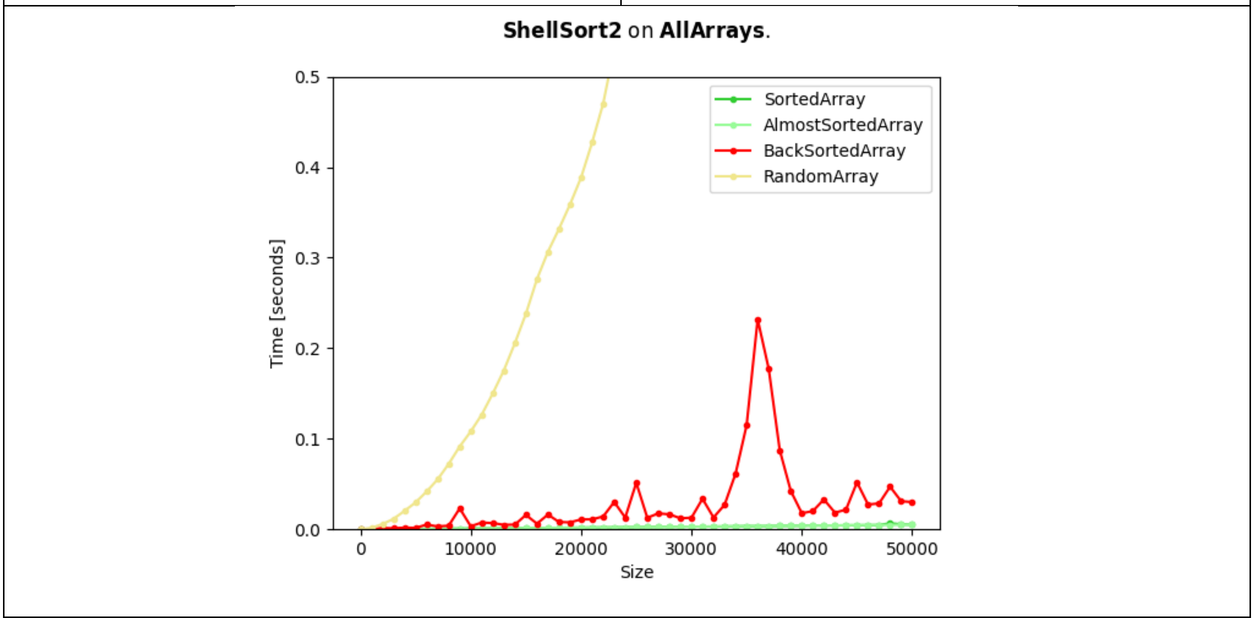
3.7. ShellSort2 (метод Седжвика)





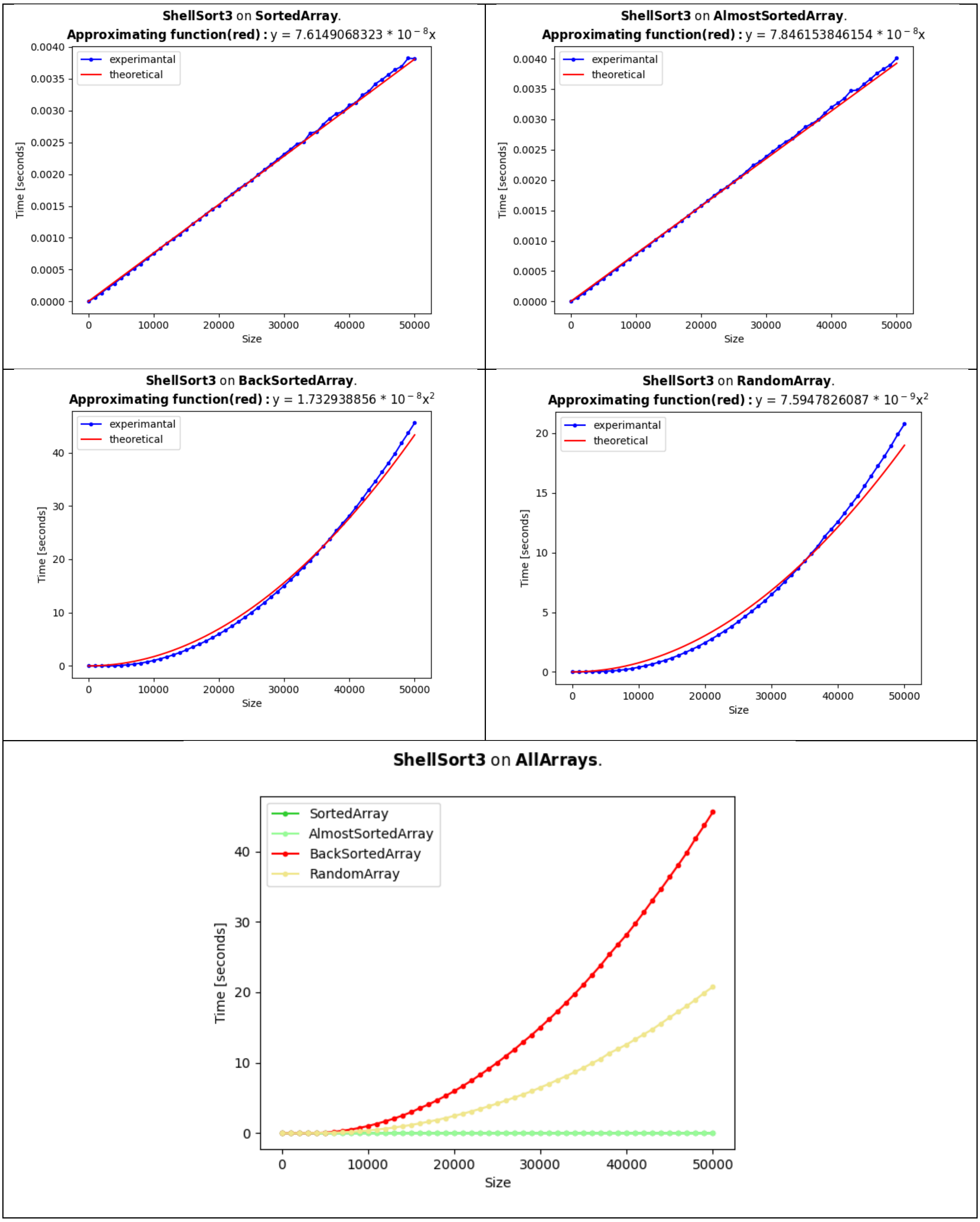






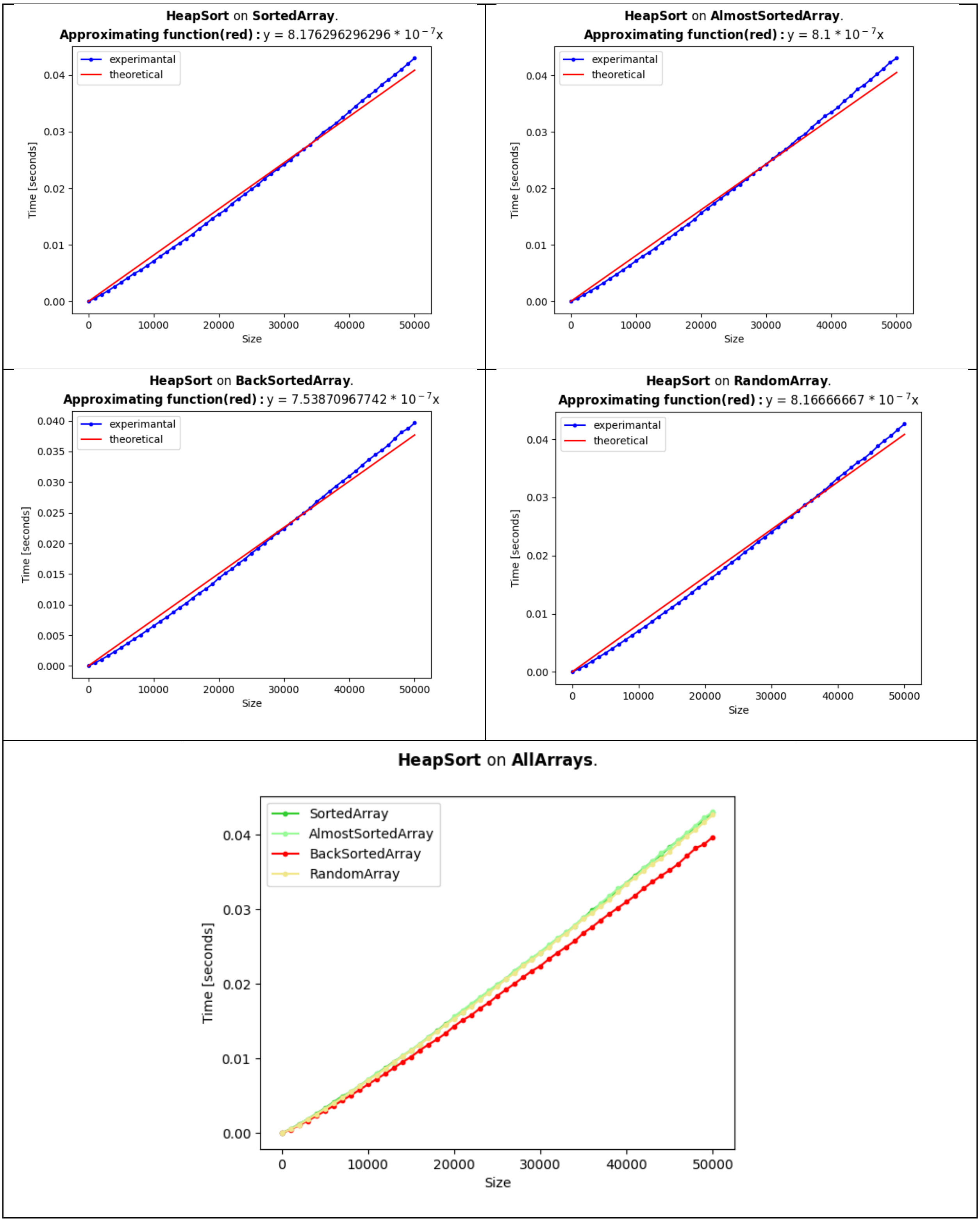
Вывод: алгоритм подходит для маленьких массивов или отсортированного (в том числе почти отсортированного или обратно отсортированного) массива.

3.8. ShellSort3 (последовательность Марцина Циура)



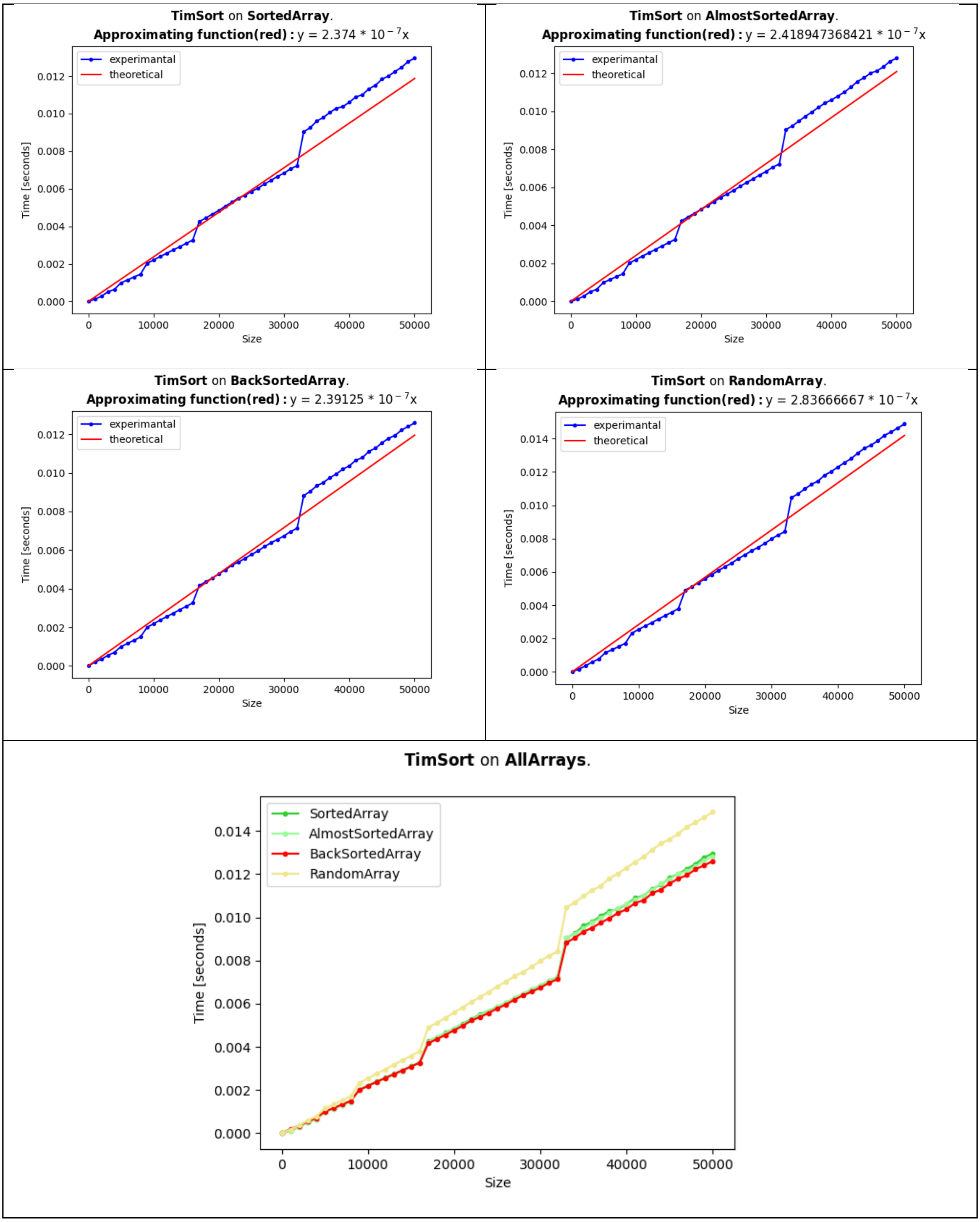
Вывод: алгоритм подходит для маленьких массивов или отсортированного (в том числе почти отсортированного или обратно отсортированного) массива.

3.9. HeapSort



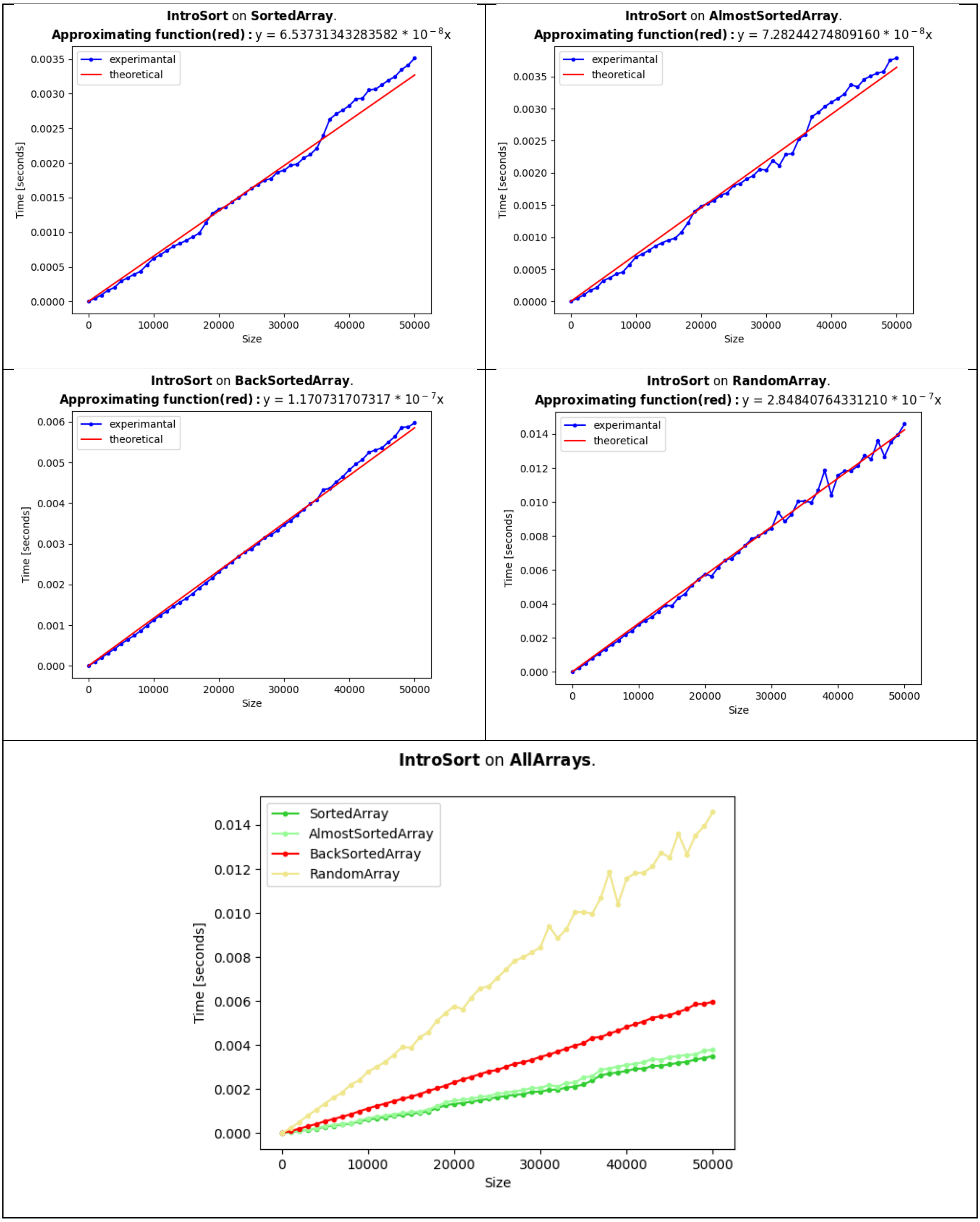
Вывод: алгоритм подходит для любого массива

3.10. TimSort



<u>Вывод: алгоритм подходит для любого массива</u>

3.11. IntroSort



Вывод: алгоритм подходит для любого массива.

4. Ссылка на репозиторий

https://github.com/afrlfff/university-student-tasks/tree/master/algoritms-and-data-structures/lab2-sorting-algs