**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: «Исследование различных алгоритмов сортировок»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 2309 |  | Савин П. А. |
| Преподаватель |  | Пестерев Д. О. |

Санкт-Петербург

2023

1. Постановка задачи

Реализовать известные алгоритмы сортировки данных и исследовать их временные сложности для разных случаев.

1. Описание функций, теоретическая оценка временной сложности и устойчивости алгоритмов  
     
   1) SelectionSort(int\* arr, int length)  
   Функция, реализующая алгоритм сортировки выбором для переданного массива arr длиной length.  
   Временная сложность: лучший случай — O(n²), средний случай — O(n²), худший случай — O(n²), n – длина массива. Алгоритм нестабилен, так как самый первый дубль будет определен как минимальный в процессе, а все остальные дубли относительно него будут переставлены — нарушен порядок появления дублей.

2) InsertionSort(int\* arr, int begin, int end)  
Функция, реализующая алгоритм сортировки вставками для переданного массива arr по срезу [begin:end].  
Временная сложность: лучший случай — O(n), средний случай — O(n²), худший случай — O(n²), n – длина массива. Алгоритм стабилен, так как дубли будут перемещаться на свои места в том порядке, в котором они представлены изначально.

3) BubbleSort(int\* arr, int length)  
Функция, реализующая алгоритм сортировки пузырьком для переданного массива arr длиной length.  
Временная сложность: лучший случай — O(n), средний случай — O(n²), худший случай — O(n²), n – длина массива. Алгоритм стабилен, так как дубли не будут обмениваться друг с другом, сохраняя их исходный порядок появления и при сортировке.

4) MergeSort(int\* arr, int begin, int end)  
Функция, реализующая алгоритм сортировки слиянием для переданного массива arr по срезу [begin:end].  
Временная сложность: лучший случай — O(nlogn), средний случай — O(nlogn), худший случай — O(nlogn), n – длина массива. Алгоритм стабилен, так как даже в подмассивах они будут сохранять свой порядок, аналогично и при слияниях порядок будет сохранен.

5) QuickSort(int\* arr, int begin, int end)  
Функция, реализующая алгоритм быстрой сортировки для переданного массива arr по срезу [begin:end].  
Временная сложность: лучший случай — O(nlogn), средний случай — O(nlogn), худший случай — O(n²), n – длина массива. Алгоритм нестабилен, так как в качестве опорного элемента алгоритм может выбрать любой из дублей и начать перемешивать элементы вокруг него — нарушается исходный порядок появления дублей.

6) ShellSort(int\* arr, int length)  
Функция, реализующая алгоритм сортировки Шелла для переданного массива arr длиной length.  
Временная сложность зависит от выбранной последовательности промежутков.

Последовательность Шелла: лучший случай — O(nlog²n), средний случай — O(), худший случай — O(n²).

Последовательность Хиббарда ([OEIS A000225](https://oeis.org/A000225)): лучший случай — O(nlogn), средний случай — O(), худший случай — O().

Последовательность Седжвика ([OEIS A036562](https://oeis.org/A036562)): лучший случай — O(nlogn), средний случай — O(), худший случай — O().

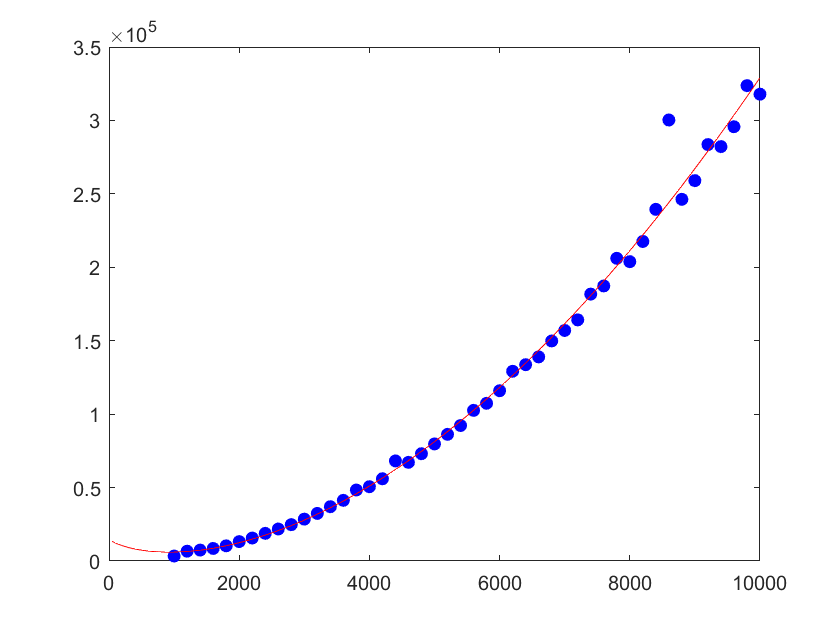
Алгоритм нестабилен, так как дубли выбираются для перестановки независимо друг от друга — исходный порядок появления дублей теряется.

7) HeapSort(int\* arr, int begin, int end)  
Функция, реализующая алгоритм пирамидальной сортировки для переданного массива arr по срезу [begin:end].  
Временная сложность: лучший случай — O(nlogn), средний случай — O(nlogn), худший случай — O(nlogn), n – длина массива. Алгоритм нестабилен, так как при образовании дерева исходный порядок появления дублей теряется.

8) TimSort(int\* arr, int length)  
Функция, реализующая алгоритм сортировки Тима для переданного массива arr длиной length.  
Временная сложность: лучший случай — O(n), средний случай — O(nlogn), худший случай — O(nlogn), n – длина массива. Алгоритм стабилен, так как сочетает в себе стабильные сортировки вставками и слиянием.

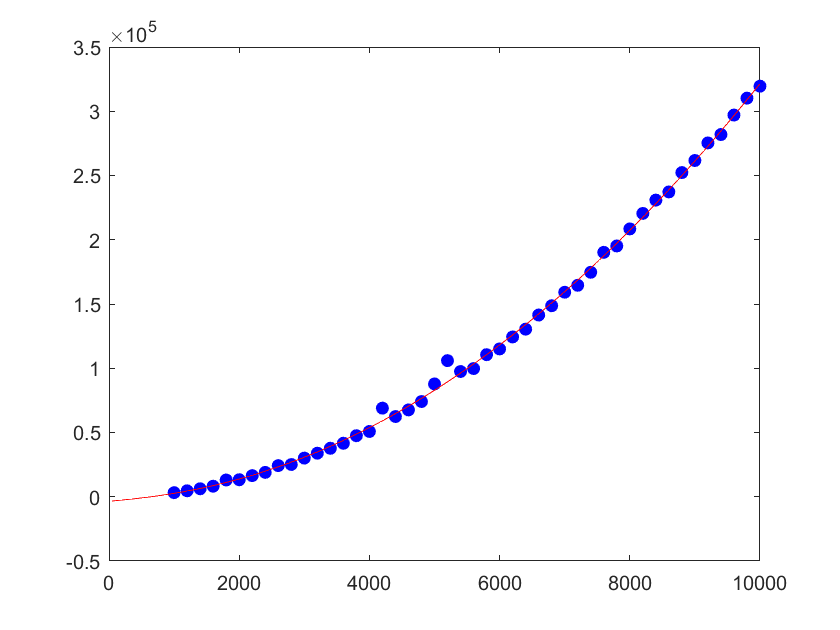
9) IntroSort(int\* arr, int begin, int end, int max\_depth)  
Функция, реализующая алгоритм интроспективной сортировки для переданного массива arr по срезу [begin:end] с начальной максимальной глубиной рекурсии max\_depth.  
Временная сложность: лучший случай — O(nlogn), средний случай — O(nlogn), худший случай — O(nlogn), n – длина массива. Алгоритм нестабилен, поскольку первый шаг задействует нестабильную быструю сортировку.

1. Временная и практическая сложность  
   Реализованные функции были воспроизведены на массивах длиной от 1000 до 10000 элементов с шагом 200, и для каждого размера было посчитано время выполнения. Более того, было создано четыре набора массивов — отсортированные, почти отсортированные, обратно отсортированные и произвольные, и для каждого набора были составлены аппроксимированные графики зависимости времени от размера. Оси абсцисс соответствуют размеры массивов, оси ординат — время выполнения для них в микросекундах. Уравнение регрессионной кривой — .
2. SelectionSort  
   Отсортированный массив



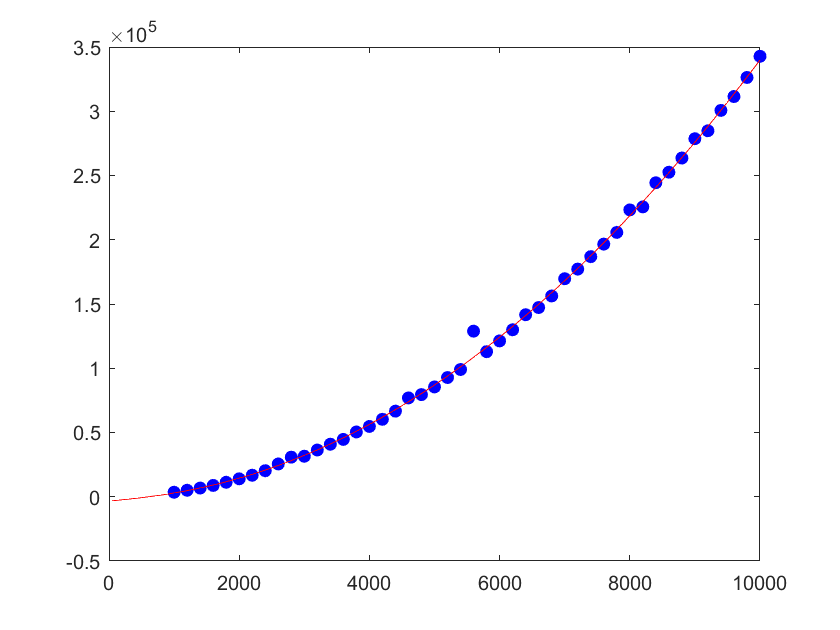
Уравнение регрессионной кривой:

Почти отсортированный массив



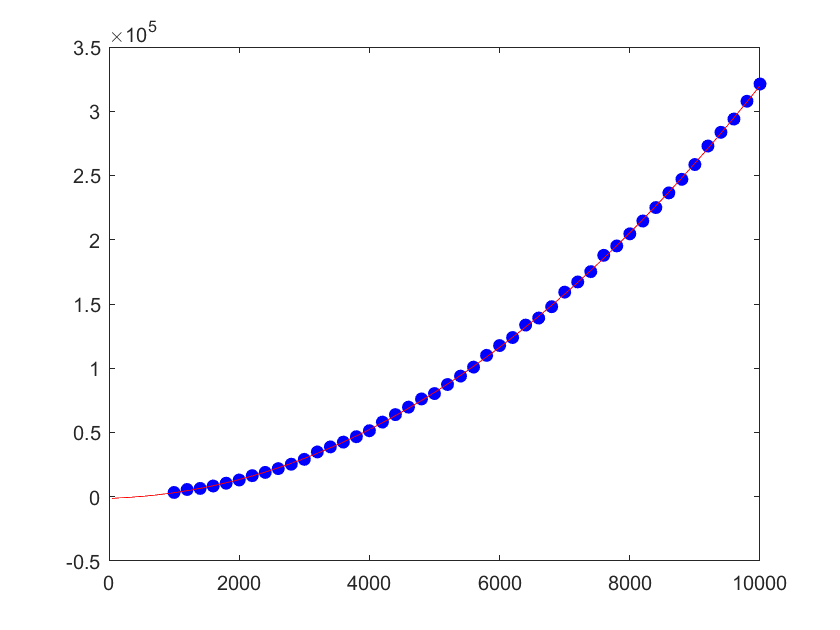
Уравнение регрессионной кривой:

Обратно отсортированный массив



Уравнение регрессионной кривой:

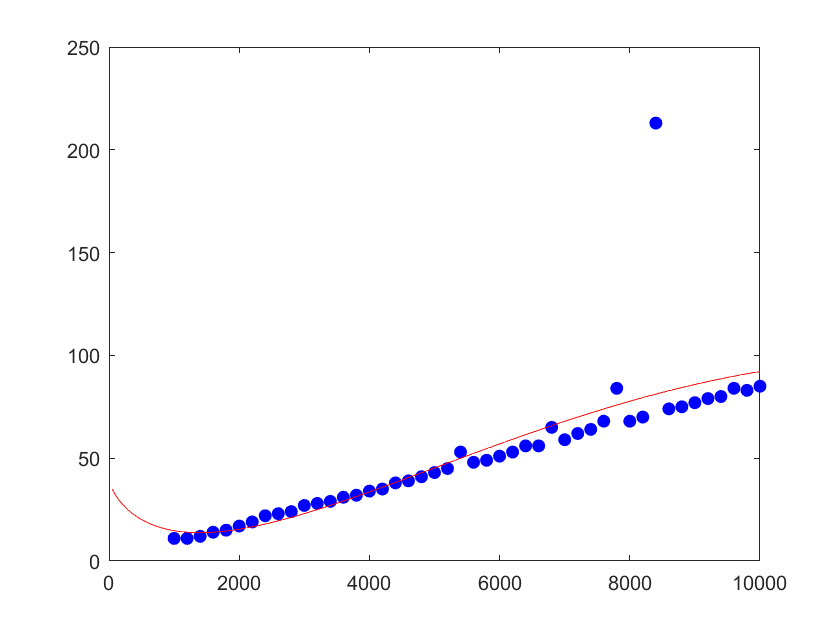
Произвольный массив



Уравнение регрессионной кривой:

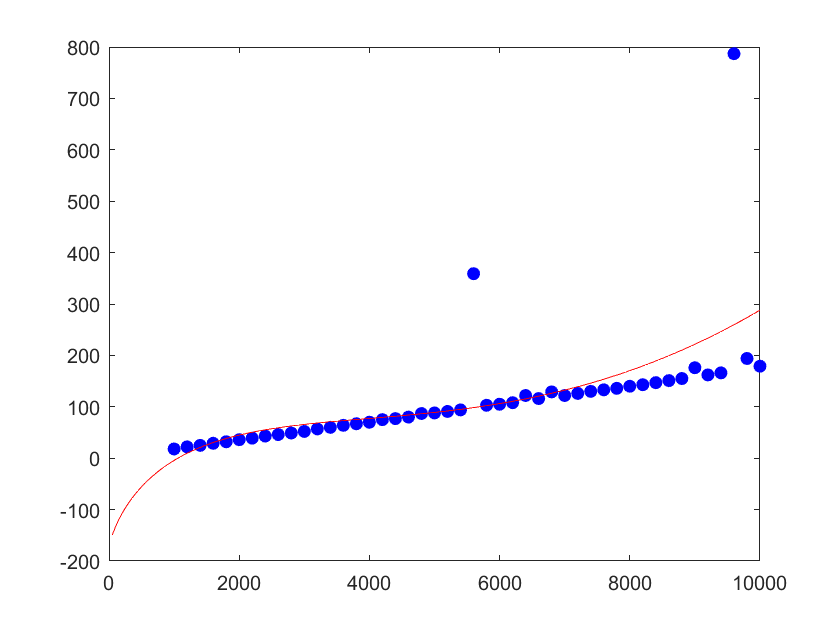
1. InsertionSort

Отсортированный массив



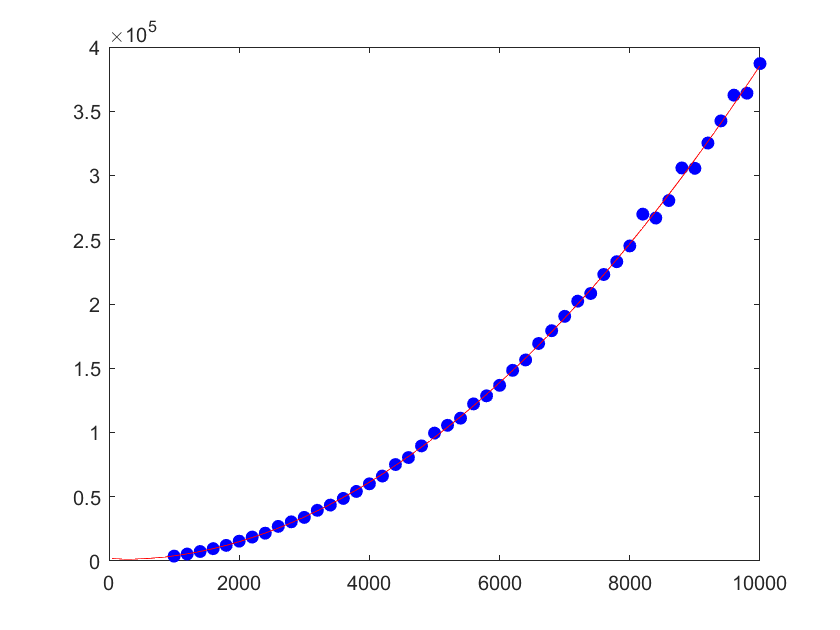
Уравнение регрессионной кривой:

Почти отсортированный массив

**

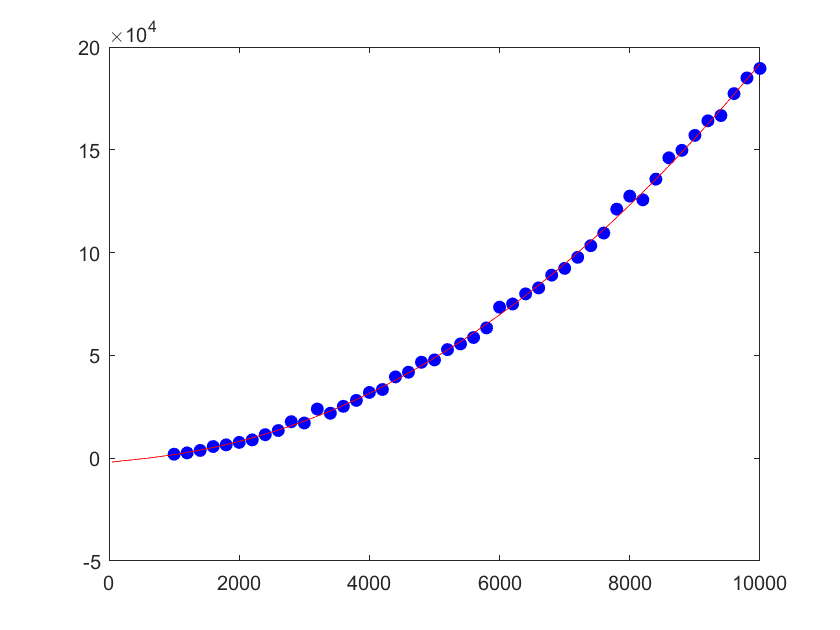
Уравнение регрессионной кривой:

Обратно отсортированный массив



Уравнение регрессионной кривой:

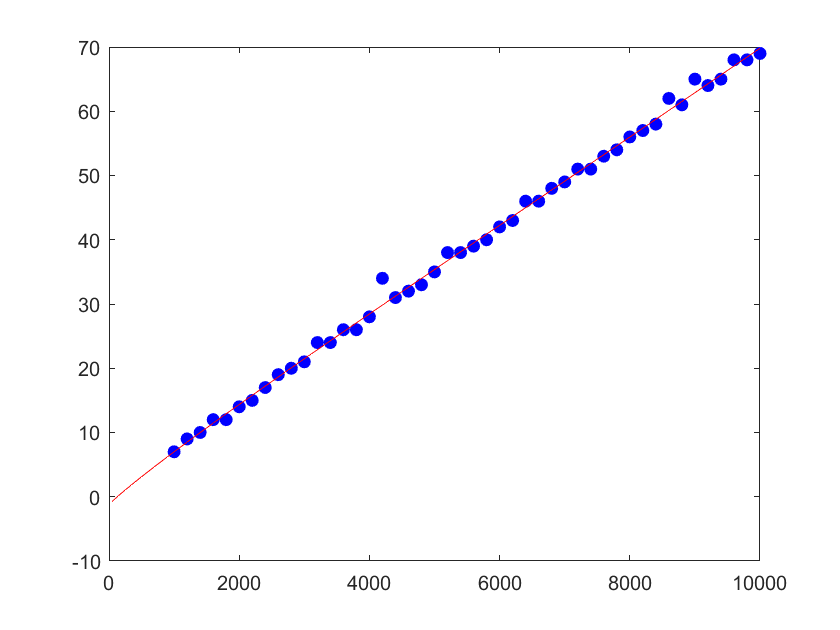
Произвольный массив



Уравнение регрессионной кривой:

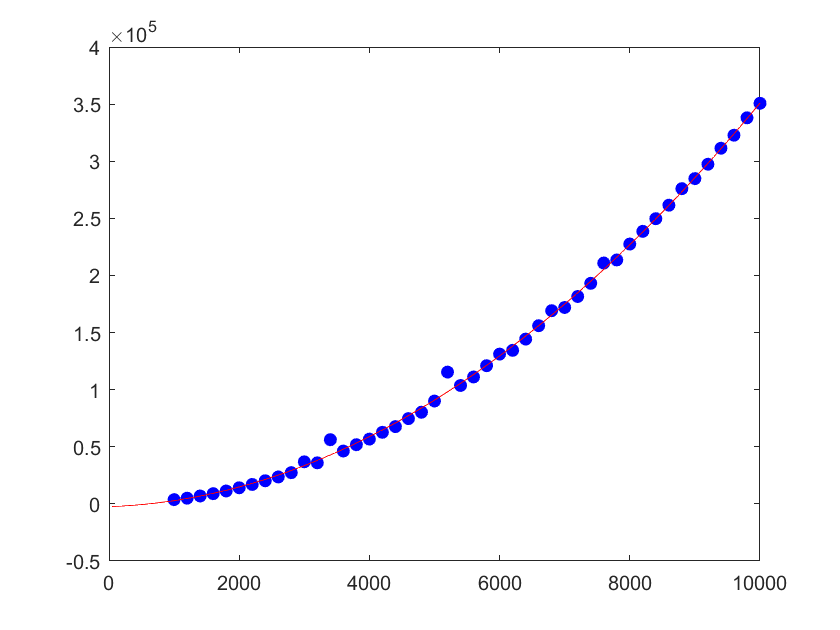
1. BubbleSort

Отсортированный массив



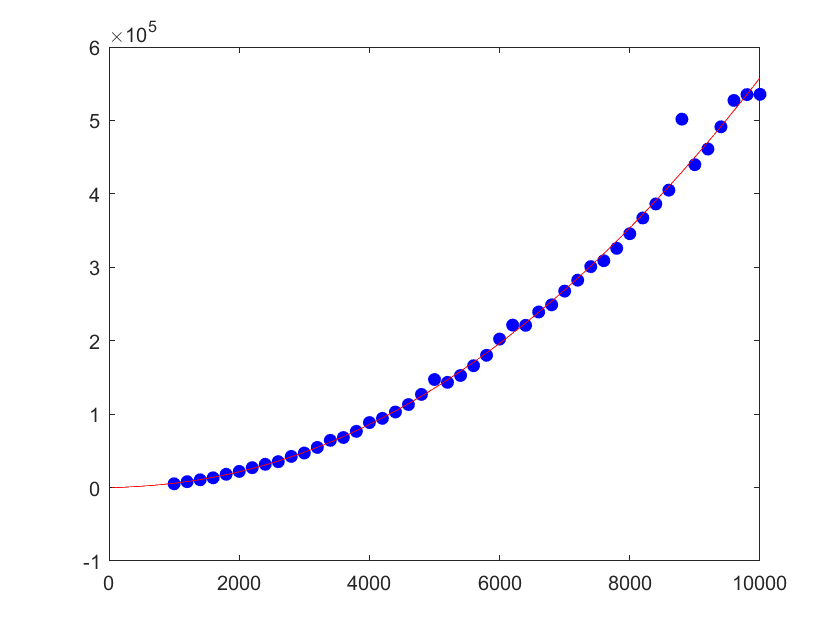
Уравнение регрессионной кривой:

Почти отсортированный массив



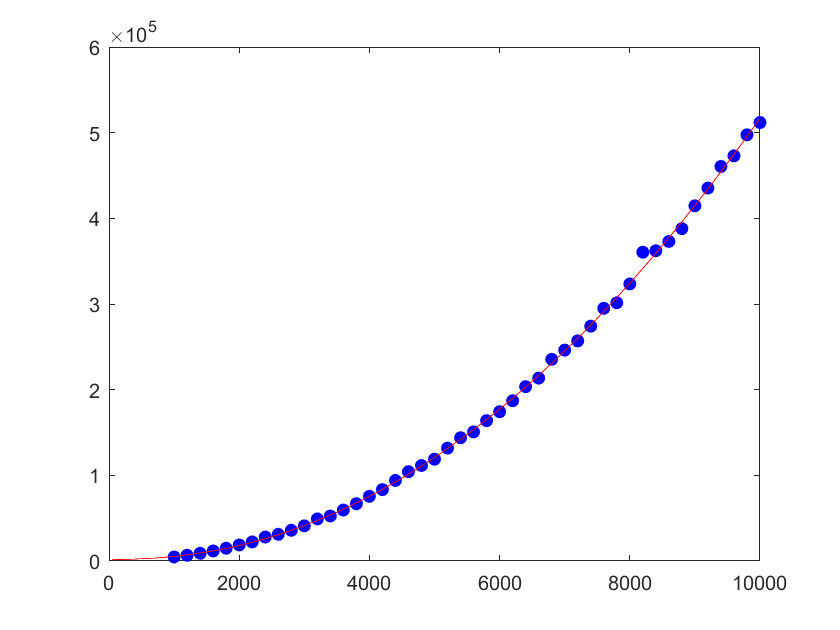
Уравнение регрессионной кривой:

Обратно отсортированный массив



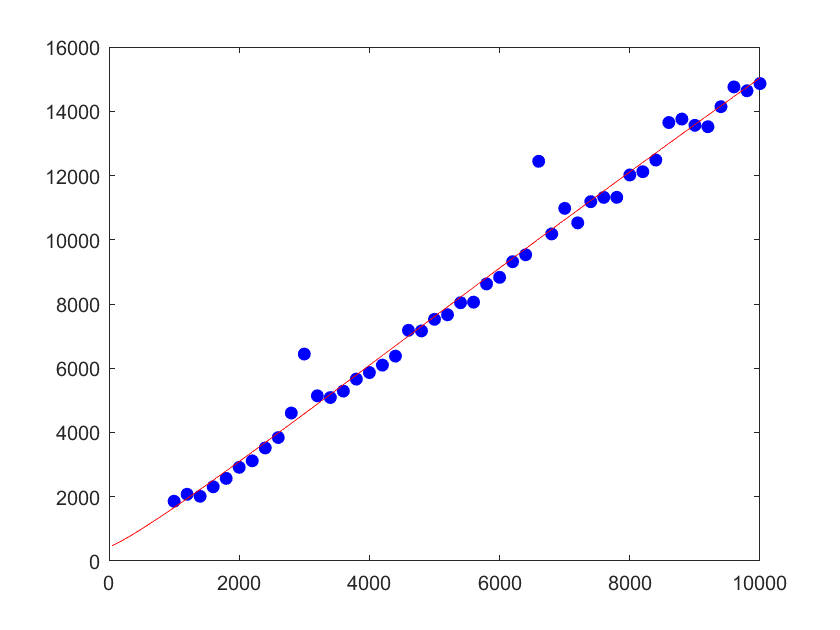
Уравнение регрессионной кривой:

Произвольный массив



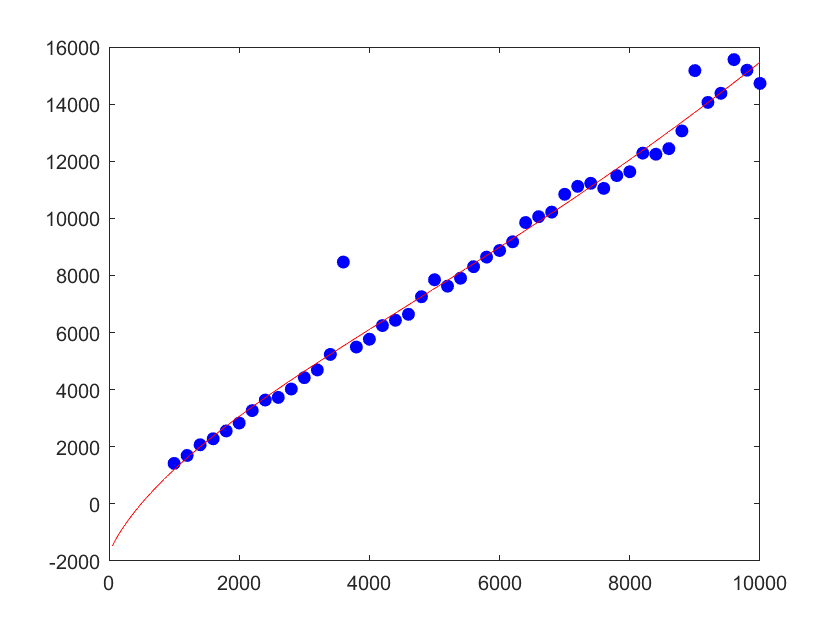
Уравнение регрессионной кривой:

1. MergeSort  
   Отсортированный массив



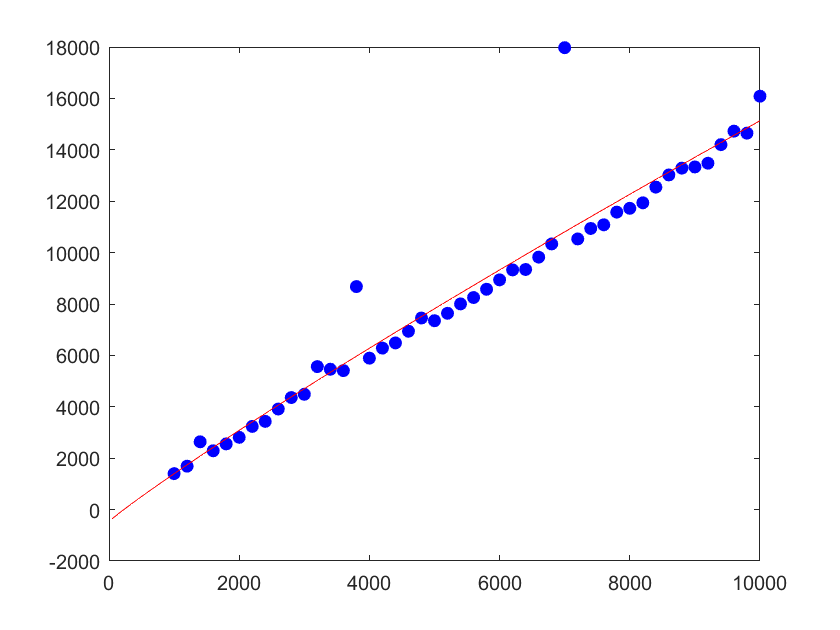
Уравнение регрессионной кривой:

Почти отсортированный массив



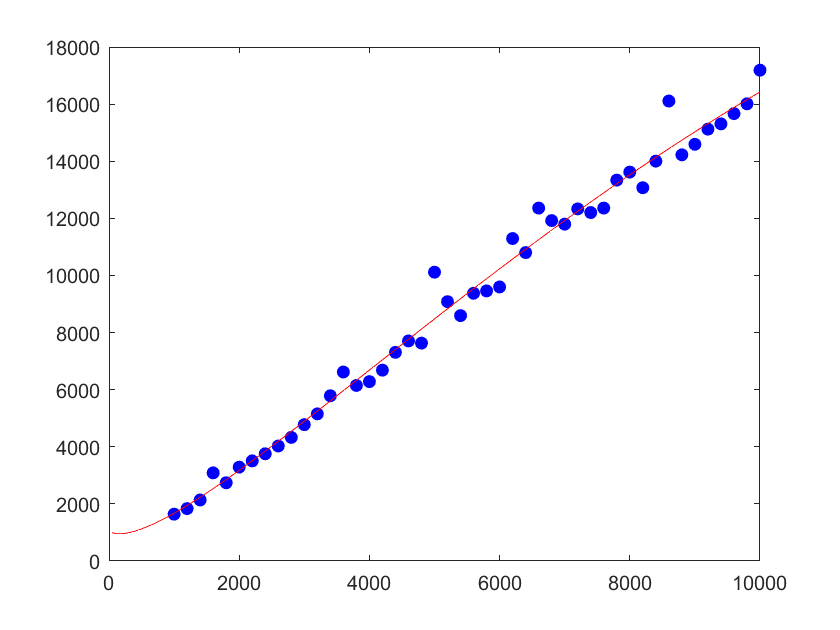
Уравнение регрессионной кривой:

Обратно отсортированный массив



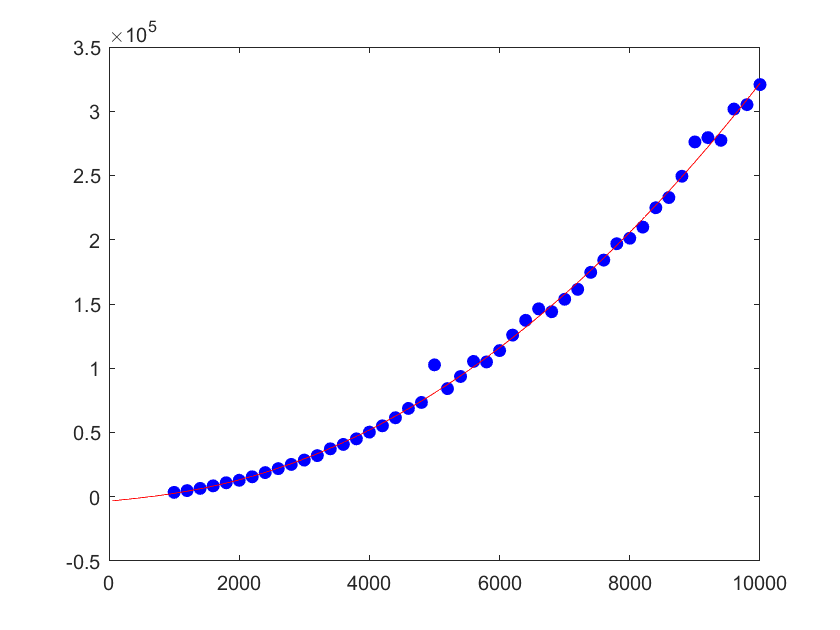
Уравнение регрессионной кривой:

Произвольный массив



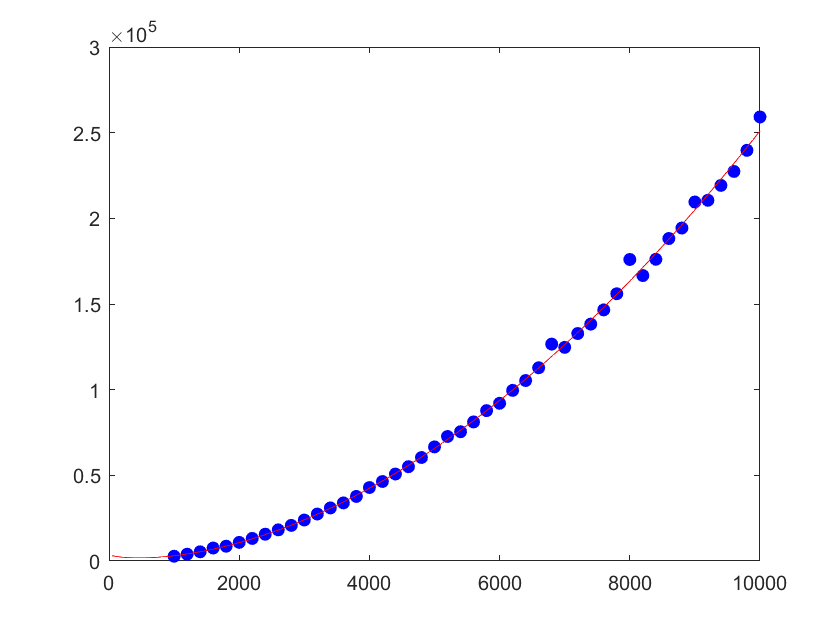
Уравнение регрессионной кривой:

1. QuickSort  
   Отсортированный массив



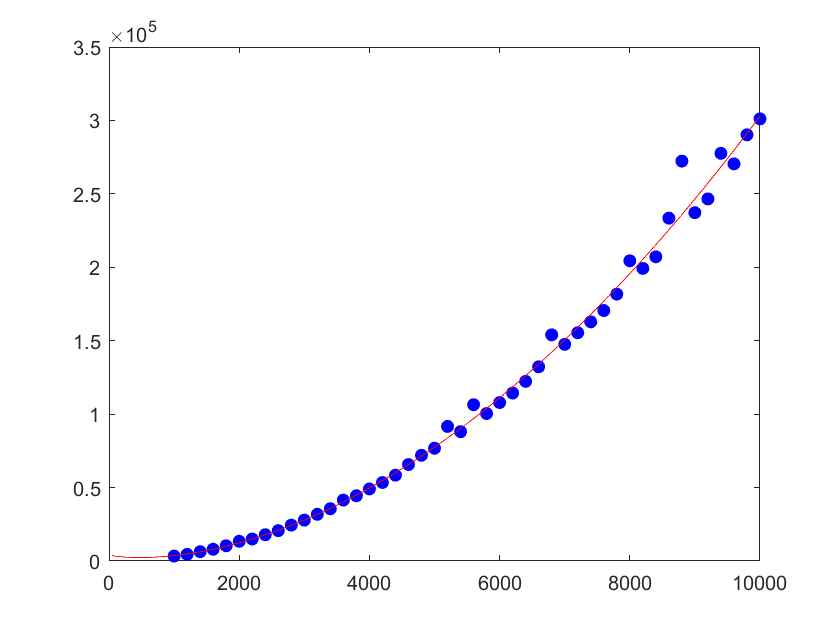
Уравнение регрессионной кривой:

Почти отсортированный массив



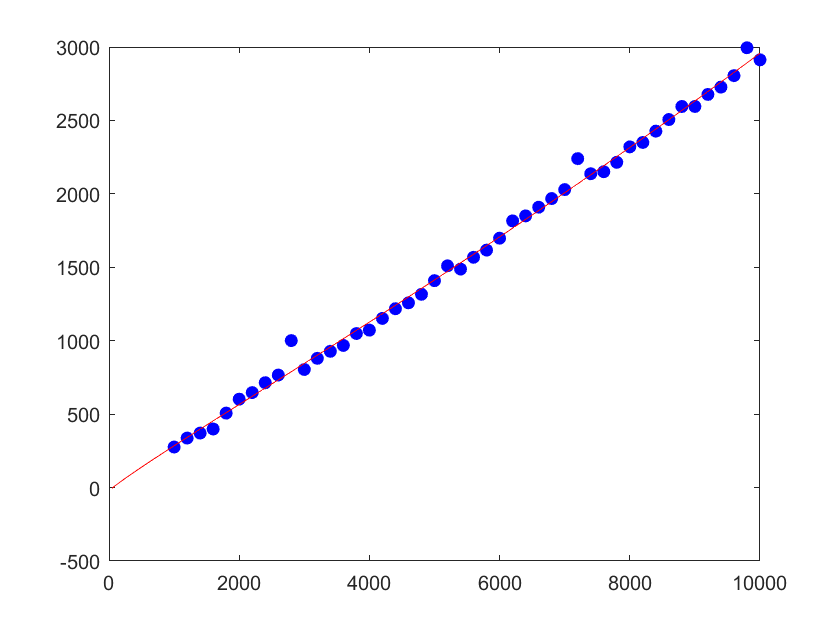
Уравнение регрессионной кривой:

Обратно отсортированный массив



Уравнение регрессионной кривой:

Произвольный массив

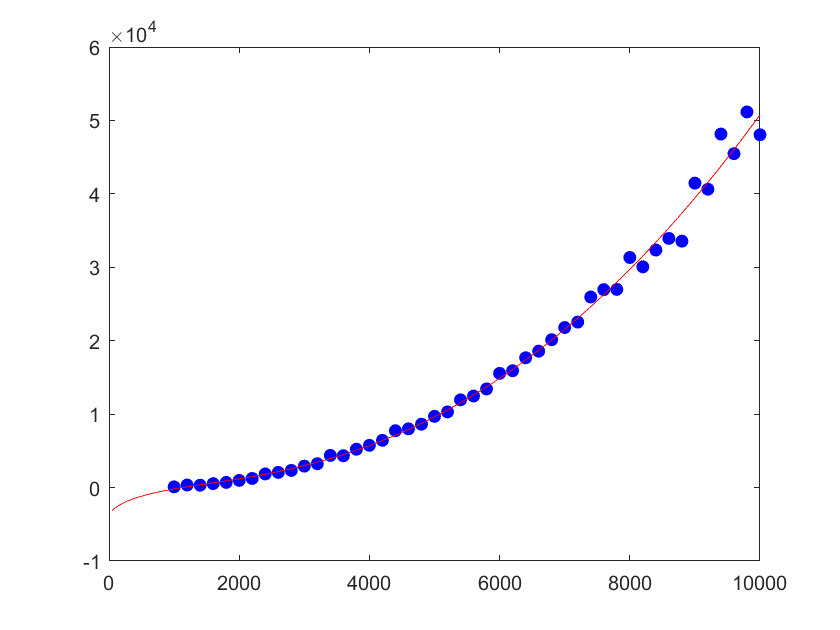


Уравнение регрессионной кривой:

1. ShellSort

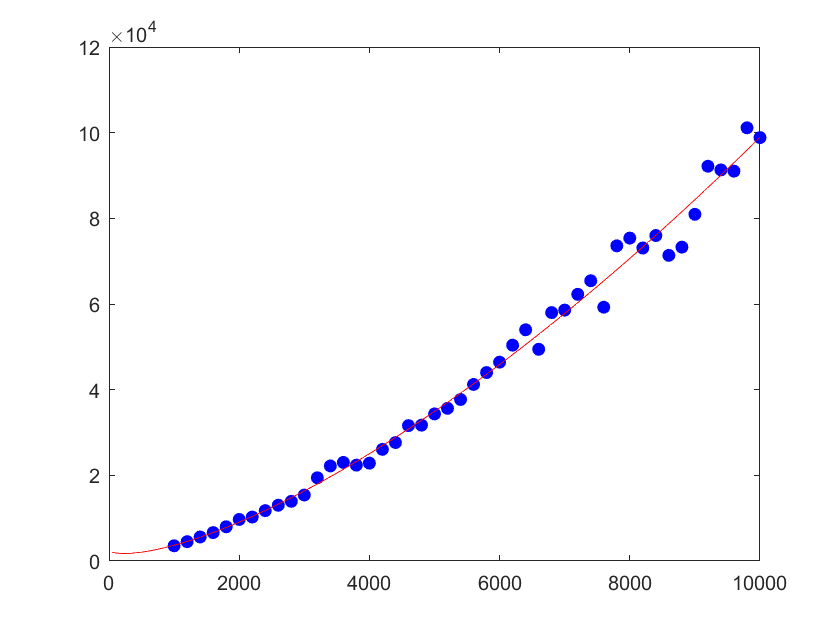
Последовательность Шелла:

Отсортированный массив



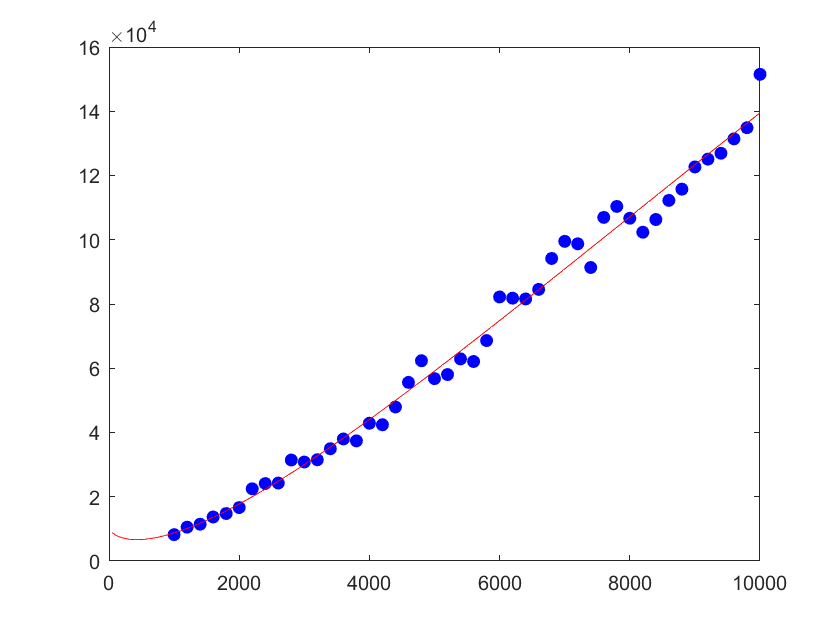
Уравнение регрессионной кривой:

Почти отсортированный массив



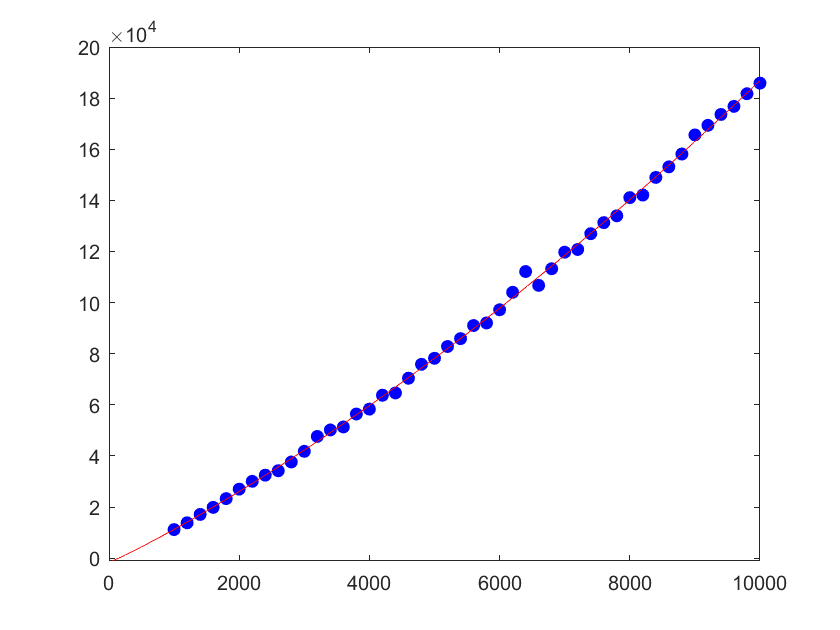
Уравнение регрессионной кривой:

Обратно отсортированный массив



Уравнение регрессионной кривой:

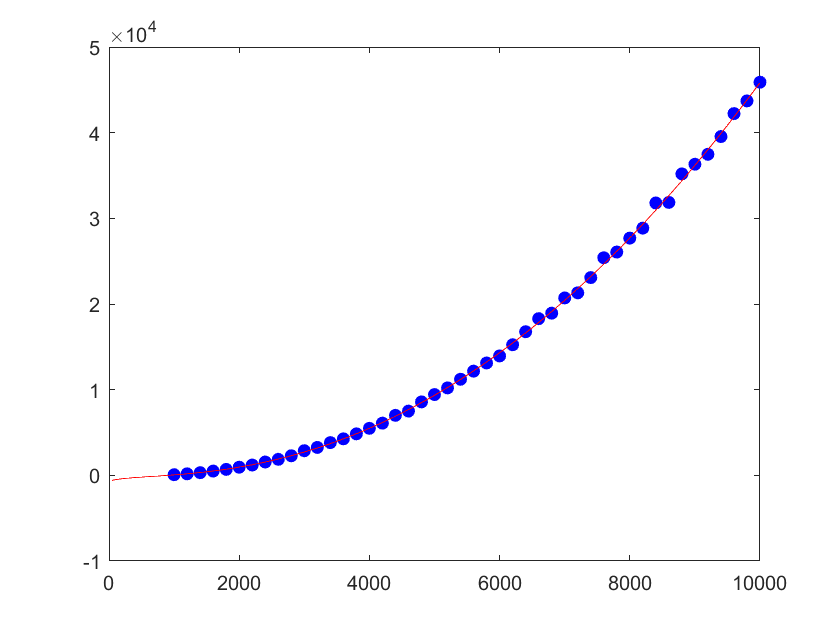
Произвольный массив



Уравнение регрессионной кривой:

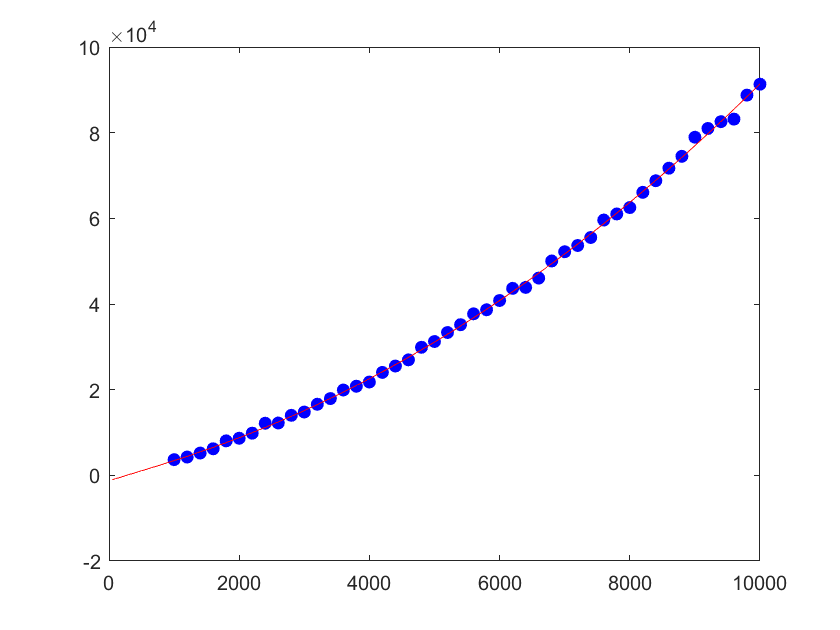
Последовательность Хиббарда:

Отсортированный массив



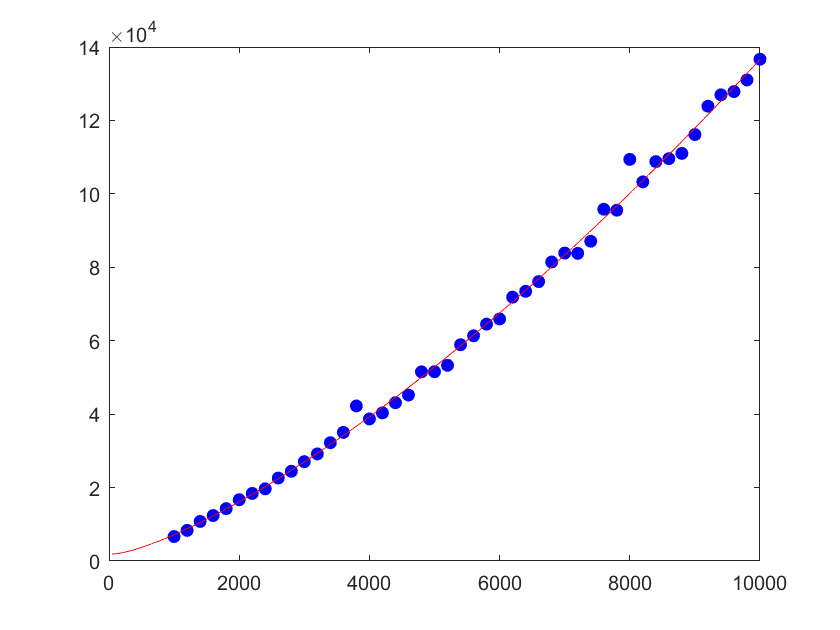
Уравнение регрессионной кривой:

Почти отсортированный массив



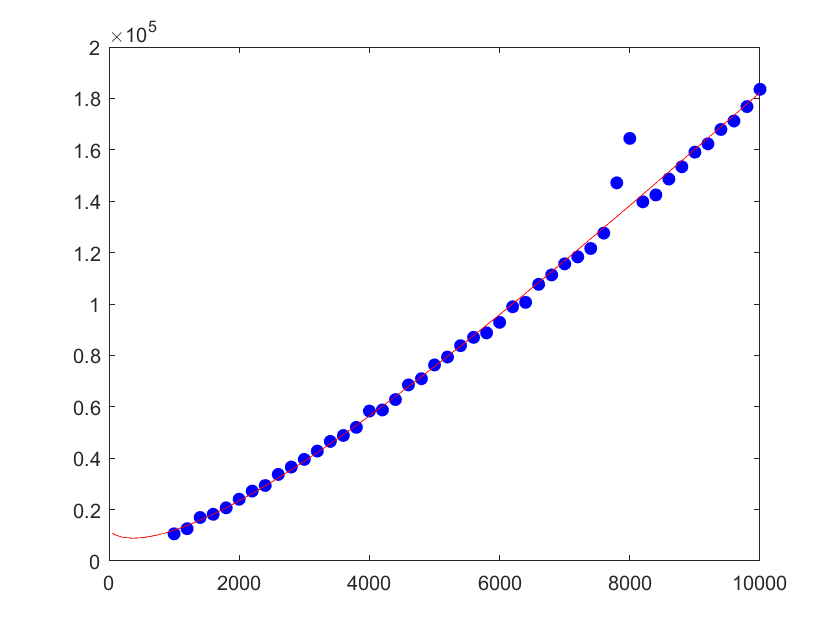
Уравнение регрессионной кривой:

Обратно отсортированный массив



Уравнение регрессионной кривой:

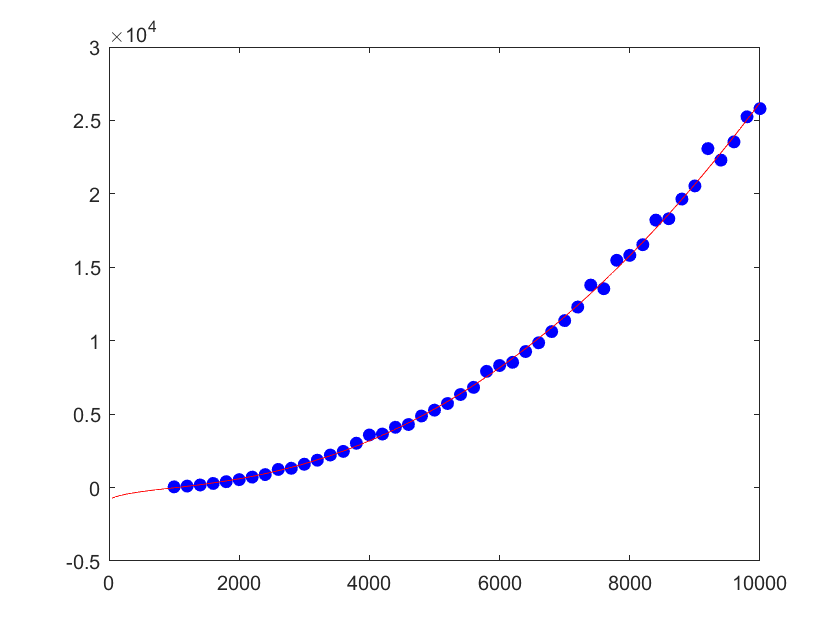
Произвольный массив



Уравнение регрессионной кривой:

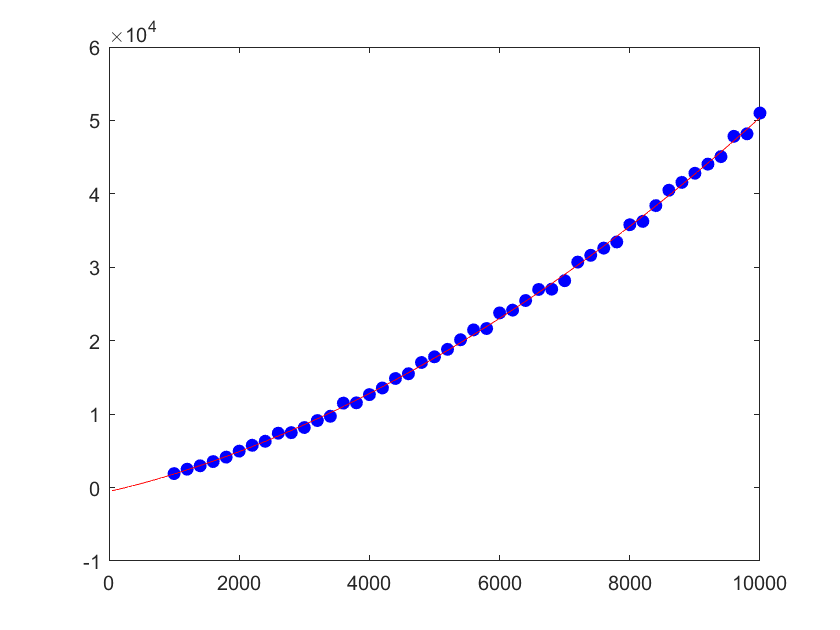
Последовательность Седжвика:

Отсортированный массив



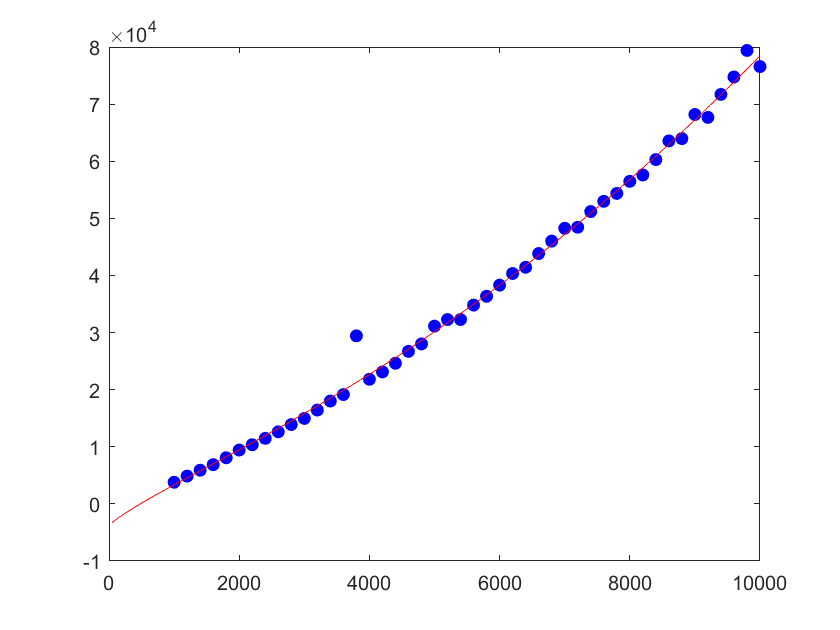
Уравнение регрессионной кривой:

Почти отсортированный массив



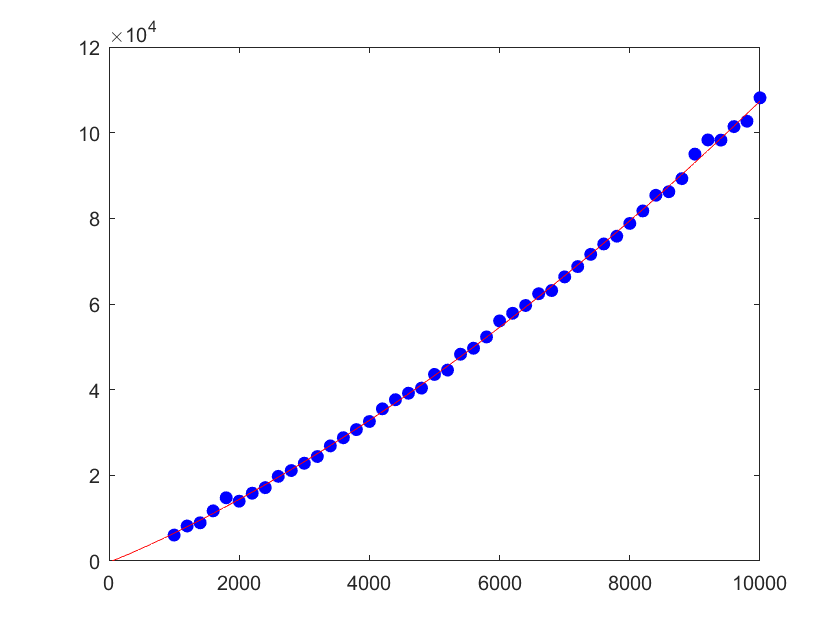
Уравнение регрессионной кривой:

Обратно отсортированный массив



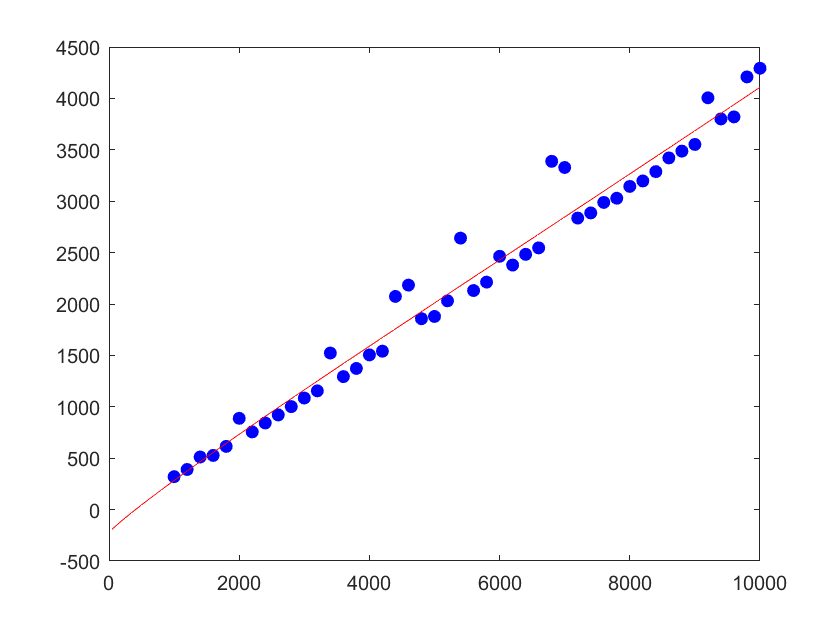
Уравнение регрессионной кривой:

Произвольный массив



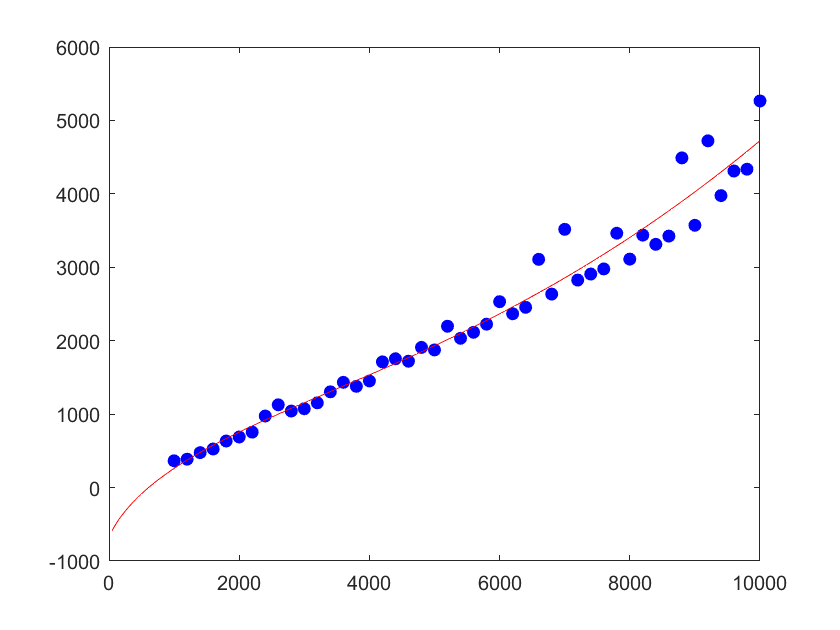
Уравнение регрессионной кривой:

1. HeapSort  
   Отсортированный массив



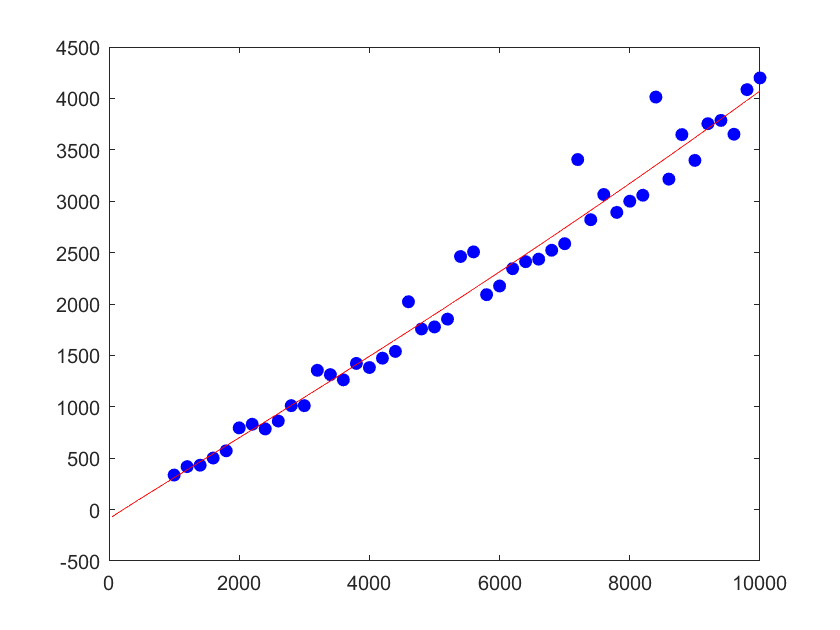
Уравнение регрессионной кривой:

Почти отсортированный массив



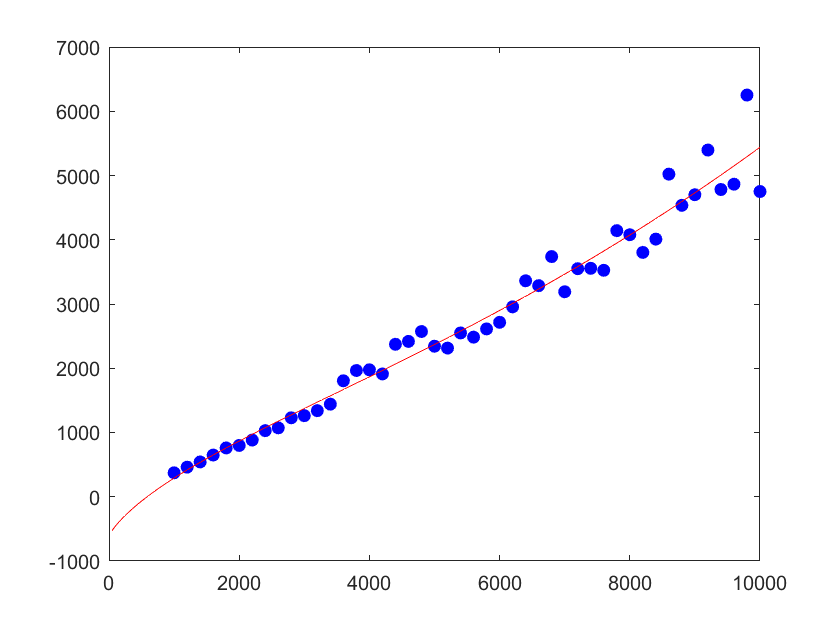
Уравнение регрессионной кривой:

Обратно отсортированный массив



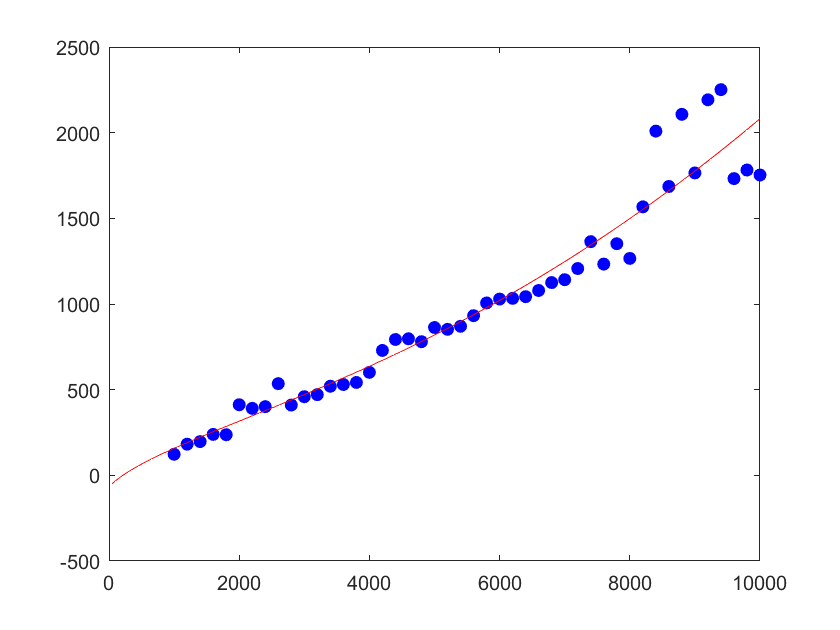
Уравнение регрессионной кривой:

Произвольный массив



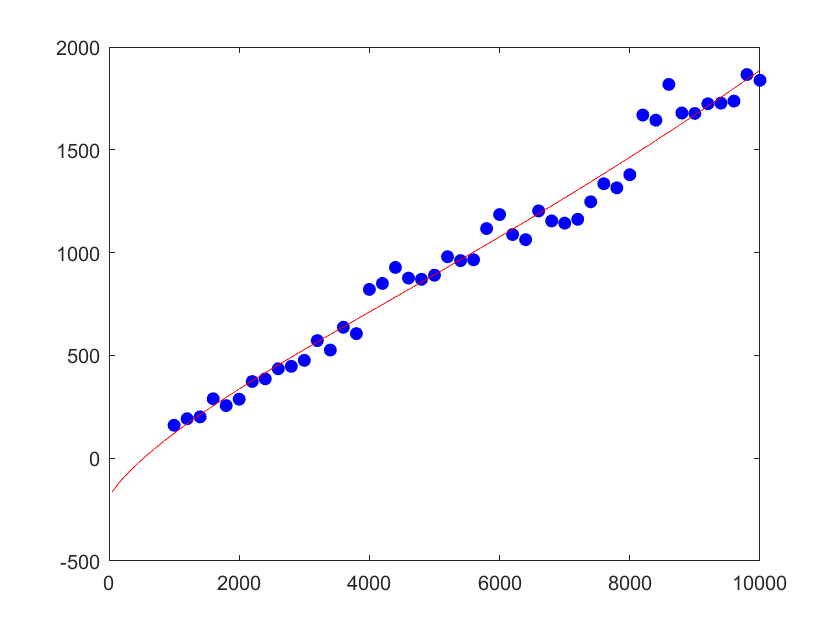
Уравнение регрессионной кривой:

1. TimSort  
   Отсортированный массив



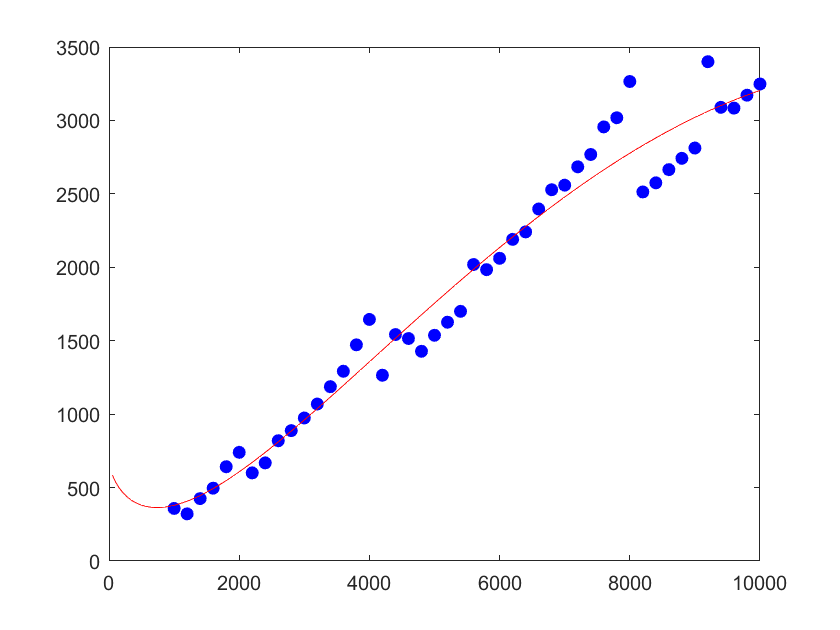
Уравнение регрессионной кривой:

Почти отсортированный массив



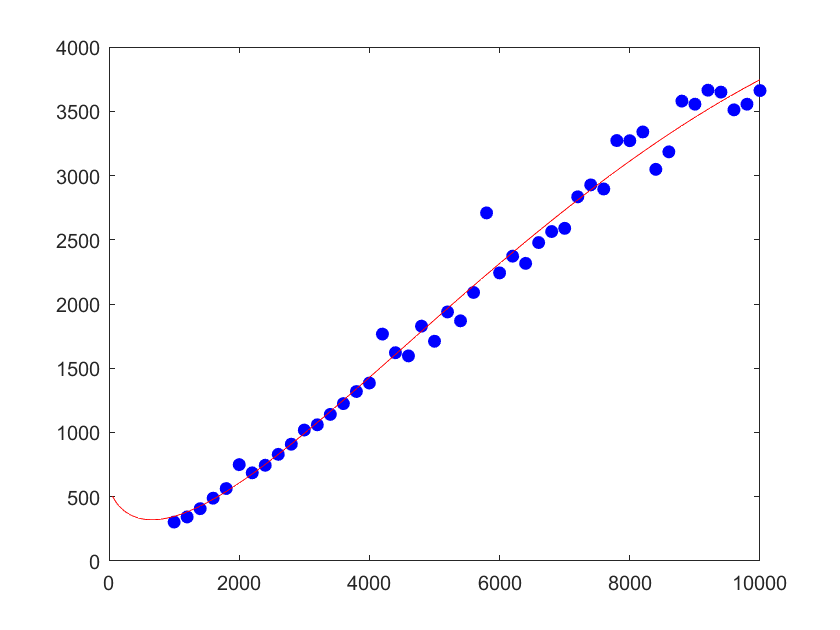
Уравнение регрессионной кривой:

Обратно отсортированный массив



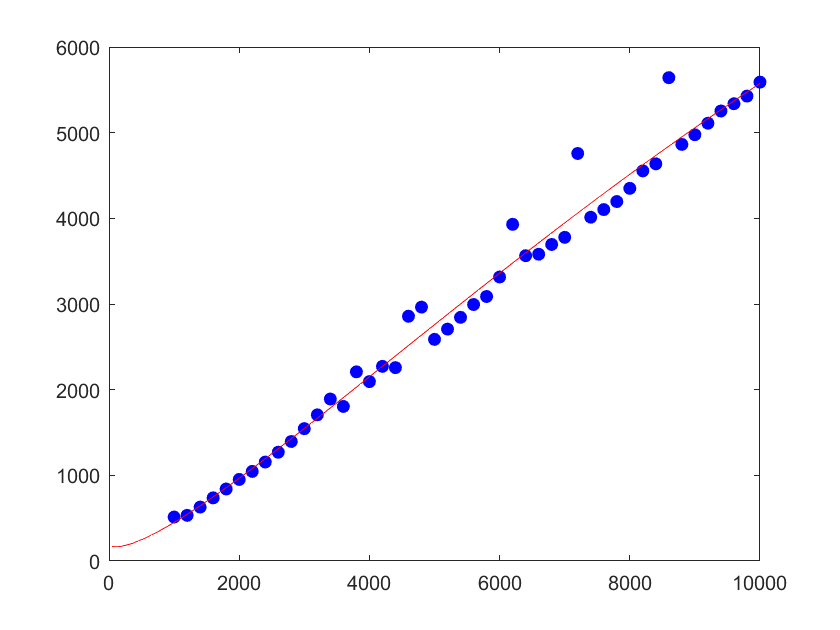
Уравнение регрессионной кривой:

Произвольный массив



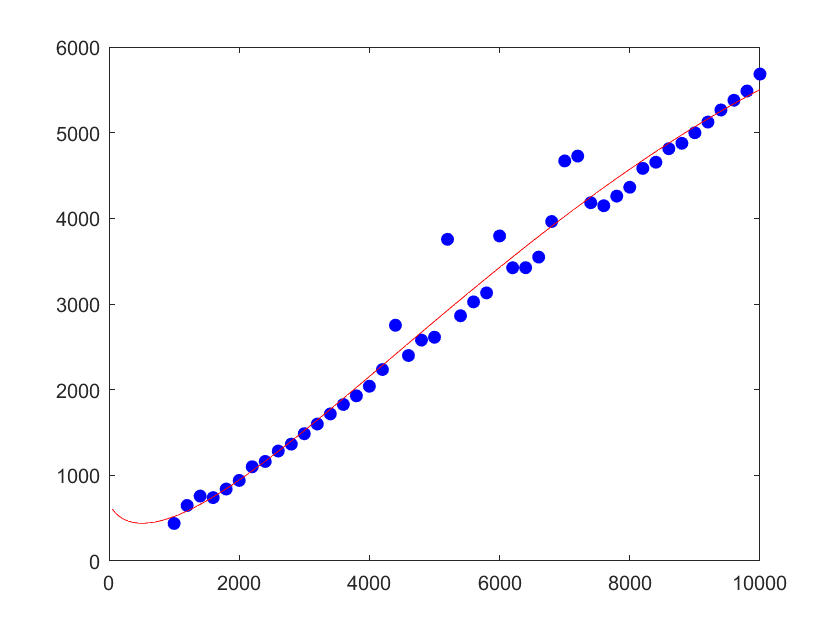
Уравнение регрессионной кривой:

1. IntroSort  
   Отсортированный массив



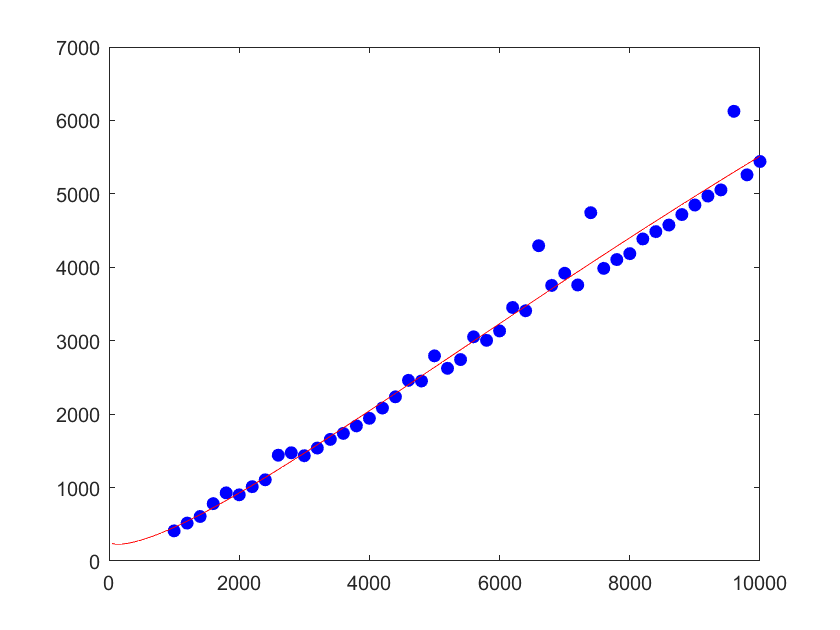
Уравнение регрессионной кривой:

Почти отсортированный массив



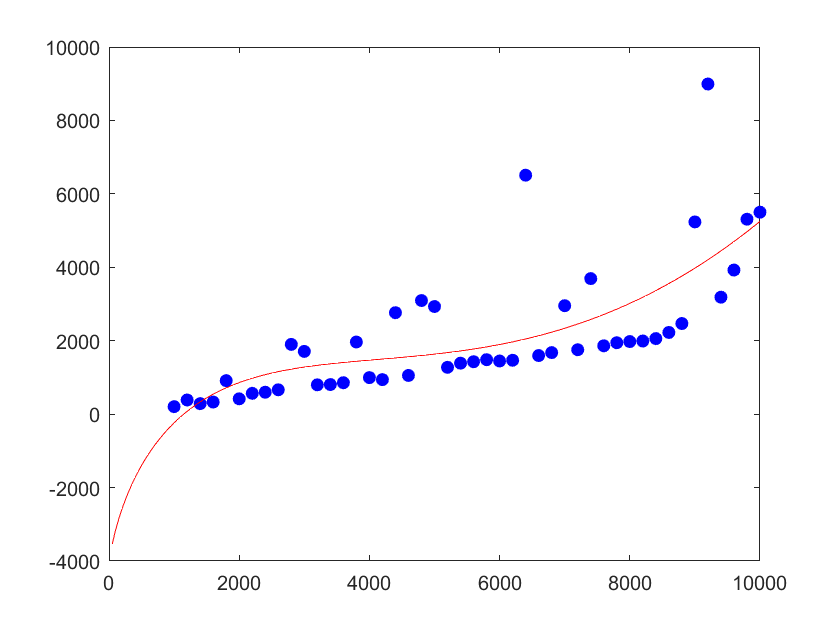
Уравнение регрессионной кривой:

Обратно отсортированный массив



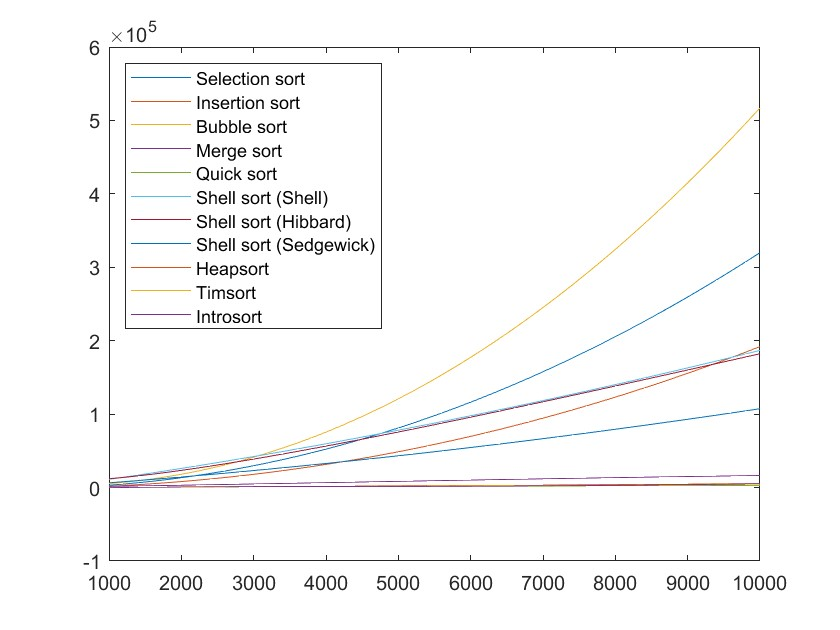
Уравнение регрессионной кривой:

Произвольный массив

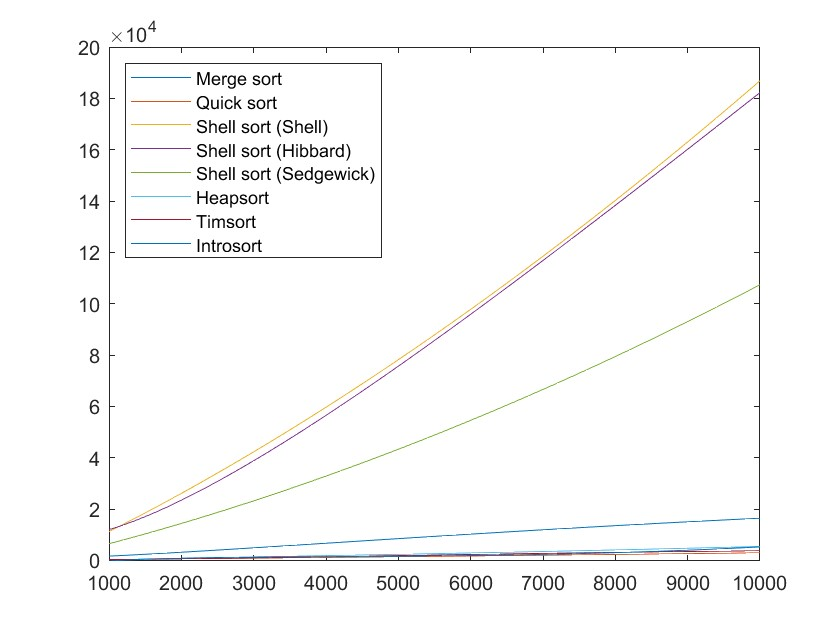


Уравнение регрессионной кривой:

Сравнение средних случаев всех сортировок:



Сравнение средних случаев сортировок без квадратичных:



Самое быстрое исполнение показали (от более быстрой): быстрая сортировка, пирамидальная сортировка, сортировка Тима и интроспективная сортировка.

1. **Оценка эффективности использования алгоритмов**

Сортировки выбором и пузырьком — самые легкие в понимании из представленных, но на практике самые медленные и неэффективные, поэтому их практическая польза только в учебных программах. Сортировка вставками также неэффективна сама по себе, но имеет преимущество на малых размерах данных, поэтому она применяется в сочетании с другими алгоритмами. Сортировка слияниями хороша в своей устойчивости и стабильной асимптотике, но ее недостаток — затраты памяти в O(n), что может приводить к переполнениям стека/буфера и замедлению/падению программы, поэтому она также используется в сочетании с другими алгоритмами, а также в модификациях с параллельными вычислениями. Быстрая сортировка и по сей день является одним из самых быстрых алгоритмов и даже является стандартным алгоритмом для различных языков программирования (например, qsort в C), но эта скорость очень неустойчива по асимптотике, и также имеются проблемы с затратами памяти в O(n) (вариант Седжвика с O(log n) решает эту проблему, но все еще не решает быстро растущую сложность), поэтому сейчас используются более устойчивые по асимптотике Timsort и Introsort. Сортировка Шелла хоть и не такая быстрая, но относительно устойчивая по асимптотике и не затратная по памяти, поэтому нашла практическое применение, например, в ядре Linux — она является итеративной и этим фактом помогает избегать переполнение стека и замедление сортировки при слишком глубоком дне рекурсии. Пирамидальная сортировка также является очень быстрым и не затратным по памяти алгоритмом, но теряет свою эффективность на очень больших размерах данных или на почти отсортированных наборах данных. Timsort и Introsort, на данный момент, являются самыми оптимальными и эффективными — быстрые, устойчивые по асимптотике на любом размере данных, поэтому и снискали практическое применение (например, Introsort в std::sort C++ и Timsort в sort() и sorted() Python).

1. **Листинг**

#include <iostream>

#include <vector>

#include <algorithm>

#include <stdlib.h>

#include <time.h>

#include <chrono>

using namespace std::chrono;

enum SEQS { Shell, Hibbard, Sedgewick };

const int MINRUN = 64;

std::vector<int> GAPS;

int MAX\_DEPTH;

// utilities

void PrintArray(int\* arr, int length) {

int i;

for (i = 0; i < length; ++i)

std::cout << arr[i] << " ";

std::cout << '\n';

}

void Merge(int\* arr, int left, int mid, int right) {

int i, j, k, n1, n2;

int\* left\_arr, \* right\_arr;

n1 = mid - left + 1;

n2 = right - mid;

left\_arr = new int[n1];

right\_arr = new int[n2];

for (i = 0; i < n1; ++i)

left\_arr[i] = arr[left + i];

for (j = 0; j < n2; ++j)

right\_arr[j] = arr[mid + 1 + j];

i = 0; j = 0; k = left;

while (i < n1 && j < n2) {

if (left\_arr[i] <= right\_arr[j]) {

arr[k] = left\_arr[i];

++i;

}

else {

arr[k] = right\_arr[j];

++j;

}

++k;

}

while (i < n1) {

arr[k] = left\_arr[i];

++i; ++k;

}

while (j < n2) {

arr[k] = right\_arr[j];

++j; ++k;

}

delete[] left\_arr;

delete[] right\_arr;

}

int Partition(int\* arr, int left, int right) {

int i, j, temp, pivot;

pivot = arr[left];

i = left - 1;

j = right + 1;

while (true) {

while (++i < right && arr[i] < pivot);

while (--j > left && arr[j] > pivot);

if (i < j) {

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

else {

return j;

}

}

}

void MakeSequence(int length, int seq) {

int i, temp;

switch (seq) {

case Shell: // Shell gap sequence

temp = length / 2;

while (temp > 0) {

GAPS.push\_back(temp);

temp /= 2;

}

break;

case Hibbard: // Hibbard gap sequence (OEIS A000225)

i = 1;

temp = pow(2, i) - 1;

while (temp < length) {

GAPS.push\_back(temp);

++i; temp = pow(2, i) - 1;

}

std::reverse(GAPS.begin(), GAPS.end());

break;

case Sedgewick: // Sedgewick gap sequence (OEIS A036562)

i = 1;

GAPS.push\_back(1);

temp = pow(4, i) + 3 \* pow(2, i - 1) + 1;

while (temp < length) {

GAPS.push\_back(temp);

++i; temp = pow(4, i) + 3 \* pow(2, i - 1) + 1;

}

std::reverse(GAPS.begin(), GAPS.end());

break;

default: // default is Shell sequence

temp = length / 2;

while (temp > 0) {

GAPS.push\_back(temp);

temp /= 2;

}

break;

}

}

void Heapify(int\* arr, int length, int root) {

int left, right, largest, temp;

largest = root;

left = 2 \* root + 1;

right = 2 \* root + 2;

if (left < length && arr[left] > arr[largest])

largest = left;

if (right < length && arr[right] > arr[largest])

largest = right;

if (largest != root) {

temp = arr[root];

arr[root] = arr[largest];

arr[largest] = temp;

Heapify(arr, length, largest);

}

}

int CalculateMinrun(int length) {

int flag;

flag = 0;

while (length >= MINRUN) {

flag |= length & 1;

length >>= 1;

}

return length + flag;

}

void CalculateMaxDepth(int length) {

MAX\_DEPTH = 2 \* (log(length) / log(2));

}

// sorting algorithms

void SelectionSort(int\* arr, int length) {

int i, j, min\_index, temp;

for (i = 0; i < length - 1; ++i) {

min\_index = i;

for (j = i + 1; j < length; ++j) {

if (arr[j] < arr[min\_index]) {

min\_index = j;

}

}

if (min\_index != i) {

temp = arr[min\_index];

arr[min\_index] = arr[i];

arr[i] = temp;

}

}

}

void InsertionSort(int\* arr, int begin, int end) {

int i, j, key;

for (i = begin + 1; i <= end; ++i) {

key = arr[i];

j = i - 1;

while (j >= begin && arr[j] > key) {

arr[j + 1] = arr[j];

--j;

}

arr[j + 1] = key;

}

}

void BubbleSort(int\* arr, int length) {

int i, j, temp;

bool swapped;

for (i = 0; i < length - 1; ++i) {

swapped = false;

for (j = 0; j < length - i - 1; ++j) {

if (arr[j] > arr[j + 1]) {

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

}

}

if (swapped == false) {

return;

}

}

}

void MergeSort(int\* arr, int begin, int end) {

if (begin < end) {

int mid;

mid = begin + (end - begin) / 2;

MergeSort(arr, begin, mid);

MergeSort(arr, mid + 1, end);

Merge(arr, begin, mid, end);

}

}

void QuickSort(int\* arr, int begin, int end) {

int mid;

while (begin < end) {

mid = Partition(arr, begin, end);

if (mid - begin <= end - mid + 1) {

QuickSort(arr, begin, mid);

begin = mid + 1;

}

else {

QuickSort(arr, mid + 1, end);

end = mid;

}

}

}

void ShellSort(int\* arr, int length) {

int i, j, k, gap, temp;

for (i = 0; i < GAPS.size(); ++i) {

gap = GAPS[i];

for (j = gap; j < length; ++j) {

temp = arr[j];

for (k = j; k >= gap && arr[k - gap] > temp; k -= gap)

arr[k] = arr[k - gap];

arr[k] = temp;

}

}

}

void HeapSort(int\* arr, int begin, int end) {

int i, temp;

for (i = end / 2 - 1; i >= 0; i--)

Heapify(arr, end, i);

for (i = end - 1; i >= 0; i--) {

temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

Heapify(arr, i, 0);

}

}

void TimSort(int\* arr, int length) {

int i, size, left, mid, right, min\_run;

min\_run = CalculateMinrun(length);

for (i = 0; i < length; i += min\_run)

InsertionSort(arr, i, std::min((i + min\_run - 1), (length - 1)));

for (size = min\_run; size < length; size \*= 2) {

for (left = 0; left < length; left += 2 \* size) {

mid = left + size - 1;

right = std::min((left + 2 \* size - 1), (length - 1));

if (mid < right)

Merge(arr, left, mid, right);

}

}

}

void IntroSort(int\* arr, int begin, int end, int max\_depth) {

int length;

length = end - begin + 1;

if (length < 16) {

InsertionSort(arr, begin, end);

}

else if (max\_depth == 0) {

HeapSort(arr, begin, end);

}

else {

int mid;

mid = Partition(arr, begin, end);

IntroSort(arr, begin, mid, max\_depth - 1);

IntroSort(arr, mid + 1, end, max\_depth - 1);

}

}

int main() {

srand(time(0));

int i, j, min, max, length;

int\* arr;

std::cout << "Input array length: ";

std::cin >> length;

std::cout << "Input minimum value in range: ";

std::cin >> min;

std::cout << "Input maximum value in range: ";

std::cin >> max;

arr = new int[length];

for (i = 0; i < length; ++i)

arr[i] = (min + rand() % (max - min + 1));

std::cout << "Your array: ";

PrintArray(arr, length);

auto start = high\_resolution\_clock::now();

BubbleSort(arr, length);

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

std::cout << "Your sorted array: ";

PrintArray(arr, length);

std::cout << "Algorithm execution time (in microseconds): " << duration.count();

delete[] arr;

return 0;

}

Ссылка на репозиторий: <https://github.com/pashkev14/A_DS_Lab2>