SET3. Задача A2

Фролов-Буканов Виктор Дмитриевич БПИ-228

28 ноября 2023

1 Исходный код

Программа декомпозирована на 2 header-файла и 1 файл main.cpp.
Header-файл sort_algorithms.h содержит 3 реализации сортировок согласно заданию: merge, insertion, hybrid
Header-файл random_vec.h содержит кастомные генераторы рандомных векторов из 3 групп согласно условию
main.cpp содержит основную программу, которая, собственно и выполняет замер времени на каждый вид сортировки. Там реализовано вспомогательное перечисление (для определения вида сортировки при передаче в функцию), а также 2 вспомогательные функции. Результаты измерений программа записывает в файлы, чтобы данные из них можно было потом использовать для построения графиков
Число в гибридной сортировке тут 50, но оно каждый раз менялось по ходу выполнения эксперимента (5, 10, 20, 50)

main.cpp

```cpp
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include "sort_algorithms.h"
#include "random_vec.h"

enum sort_type {
  merge,
  hybrid
};


long long mark_time(std::vector<int> &vec, sort_type value) {
  long long millisec;
  if (value == merge) {
    auto start = std::chrono::high_resolution_clock::now();
    merge_sort(vec, 0, static_cast<int>(vec.size()));
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    millisec = std::chrono::duration_cast<std::chrono::milliseconds>(
        elapsed).count();
  } else {
    auto start = std::chrono::high_resolution_clock::now();
    hybrid_sort(vec, 0, static_cast<int>(vec.size()));
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    millisec = std::chrono::duration_cast<std::chrono::milliseconds>(
        elapsed).count();
  }

  return millisec;
}
```

```
std::ofstream fout_abs(R"(C:\Users\frolo\CLionProjects\
    assemblyTestProgram\abs_random_merge.txt)");
std::ofstream fout_rev(R"(C:\Users\frolo\CLionProjects\
    assemblyTestProgram\reversed_merge.txt)");
std::ofstream fout_sor(R"(C:\Users\frolo\CLionProjects\
    assemblyTestProgram\alm_sorted_merge.txt)");

void test_sort(sort_type value) {
  for (auto size = 500; size <= 4000; size += 100) {
    long long total_time = 0;
    for (auto i = 0; i < 100; ++i) {
      auto vec = get_random_vector();
      auto sub_vec = get_random_subvector(vec, size);
      total_time += mark_time(sub_vec, value);
    }
    fout_abs << "(" << size << ", " << static_cast<double>(total_time)
        / 100 << ") ";

    total_time = 0;
    for (auto i = 0; i < 100; ++i) {
      auto vec = get_reversed_vector();
      auto sub_vec = get_random_subvector(vec, size);
      total_time += mark_time(sub_vec, value);
    }
    fout_rev << "(" << size << ", " << static_cast<double>(total_time)
        / 100 << ") ";

    total_time = 0;
    for (auto i = 0; i < 100; ++i) {
      auto vec = get_random_vector();
      auto sub_vec = get_random_subvector(vec, size);
      make_almost_sorted_vector(sub_vec);
      total_time += mark_time(sub_vec, value);
    }
    fout_sor << "(" << size << ", " << static_cast<double>(total_time)
        / 100 << ") ";
  }
}

int main() {
  test_sort(merge);

  fout_abs.close();
  fout_rev.close();
  fout_sor.close();

  fout_abs.open(R"(C:\Users\frolo\CLionProjects\assemblyTestProgram\
      abs_random_hybrid.txt)");
  fout_rev.open(R"(C:\Users\frolo\CLionProjects\assemblyTestProgram\
      reversed_hybrid.txt)");
  fout_sor.open(R"(C:\Users\frolo\CLionProjects\assemblyTestProgram\
      alm_sorted_hybrid.txt)");

  test_sort(hybrid);

  return 0;
}
```

random_vec.h

```cpp
#pragma once

#include <iostream>
#include <algorithm>
#include <vector>

std::vector<int> get_random_vector() {
  std::vector<int> vec;
  vec.reserve(4000);
  for (auto i = 0; i < 4000; ++i) {
    vec.push_back(rand() % 3001); // NOLINT
  }

  return vec;
}

std::vector<int> get_random_subvector(std::vector<int>& src, int size)
    {
  std::vector<int> vec;
  vec.reserve(size);
  if (size == 4000) {
    vec = src;
    return vec;
  }
  int start = rand() % (4000 - size); // NOLINT

  for (auto i = start; i < start + size; ++i) {
    vec.push_back(src[i]);
  }

  return vec;
}

std::vector<int> get_reversed_vector() {
  auto vec = get_random_vector();
  std::sort(vec.begin(), vec.end(), std::greater());
  return vec;
}

void make_almost_sorted_vector(std::vector<int>& src) {
  std::sort(src.begin(), src.end());
  int size = static_cast<int>(src.size());
  int pairs_to_swap = rand() % 4 + 1; // NOLINT
  for (auto i = 0; i < pairs_to_swap; ++i) {
    std::swap(src[rand() % size], src[rand() % size]); // NOLINT
  }
}
```

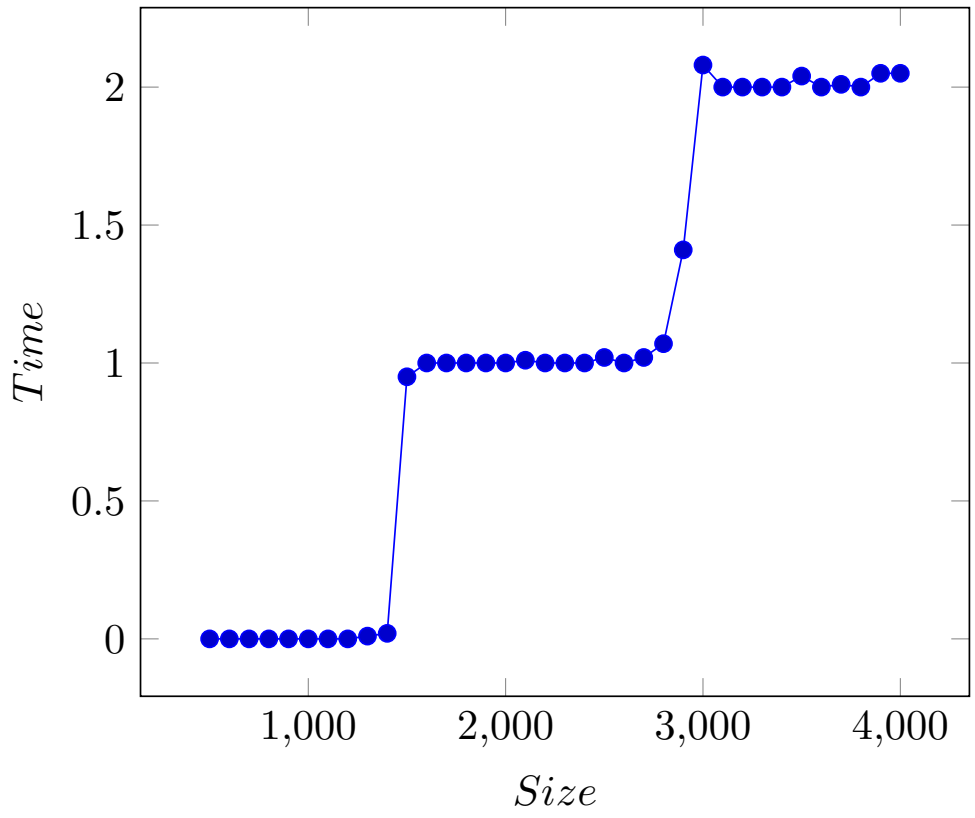
sort_algorithms.h

```cpp
#pragma once

#include <iostream>
#include <vector>

void insertion_sort(std::vector<int> &vec, int start, int end) {
  int i, j, key;
  for (i = start + 1; i < end; i++) {
```

```
    key = vec[i];
    j = i - 1;
    while (j >= 0 && vec[j] > key) {
      vec[j + 1] = vec[j];
      j = j - 1;
    }
    vec[j + 1] = key;
  }
}

void merge_sort(std::vector<int> &vec, int start, int end) { // NOLINT
  if (end - start <= 1) return;

  if (end - start == 2) {
    if (vec[start] > vec[start + 1]) {
      std::swap(vec[start], vec[start + 1]);
      return;
    }
  }

  merge_sort(vec, start, start + (end - start) / 2);
  merge_sort(vec, start + (end - start) / 2, end);

  std::vector<int> tmp;
  int i = start, j = start + (end - start) / 2;

  while (i < start + (end - start) / 2 && j < end) {
    if (vec[i] < vec[j]) {
      tmp.push_back(vec[i]);
      ++i;
    } else {
      tmp.push_back(vec[j]);
      ++j;
    }
  }

  if (i == start + (end - start) / 2) {
    for (auto t = j; t < end; ++t) tmp.push_back(vec[t]);
  } else {
    for (auto t = i; t < start + (end - start) / 2; ++t) tmp.push_back(
        vec[t]);
  }

  for (auto t = start; t < end; ++t) {
    vec[t] = tmp[t - start];
  }
}

void hybrid_sort(std::vector<int> &vec, int start, int end) { // NOLINT
  if (end - start <= 50) {
    insertion_sort(vec, start, end);
    return;
  }

  merge_sort(vec, start, start + (end - start) / 2);
  merge_sort(vec, start + (end - start) / 2, end);

  std::vector<int> tmp;
  int i = start, j = start + (end - start) / 2;
```

```
  while (i < start + (end - start) / 2 && j < end) {
    if (vec[i] < vec[j]) {
      tmp.push_back(vec[i]);
      ++i;
    } else {
      tmp.push_back(vec[j]);
      ++j;
    }
  }

  if (i == start + (end - start) / 2) {
    for (auto t = j; t < end; ++t) tmp.push_back(vec[t]);
  } else {
    for (auto t = i; t < start + (end - start) / 2; ++t) tmp.push_back(
        vec[t]);
  }

  for (auto t = start; t < end; ++t) {
    vec[t] = tmp[t - start];
  }
}
```

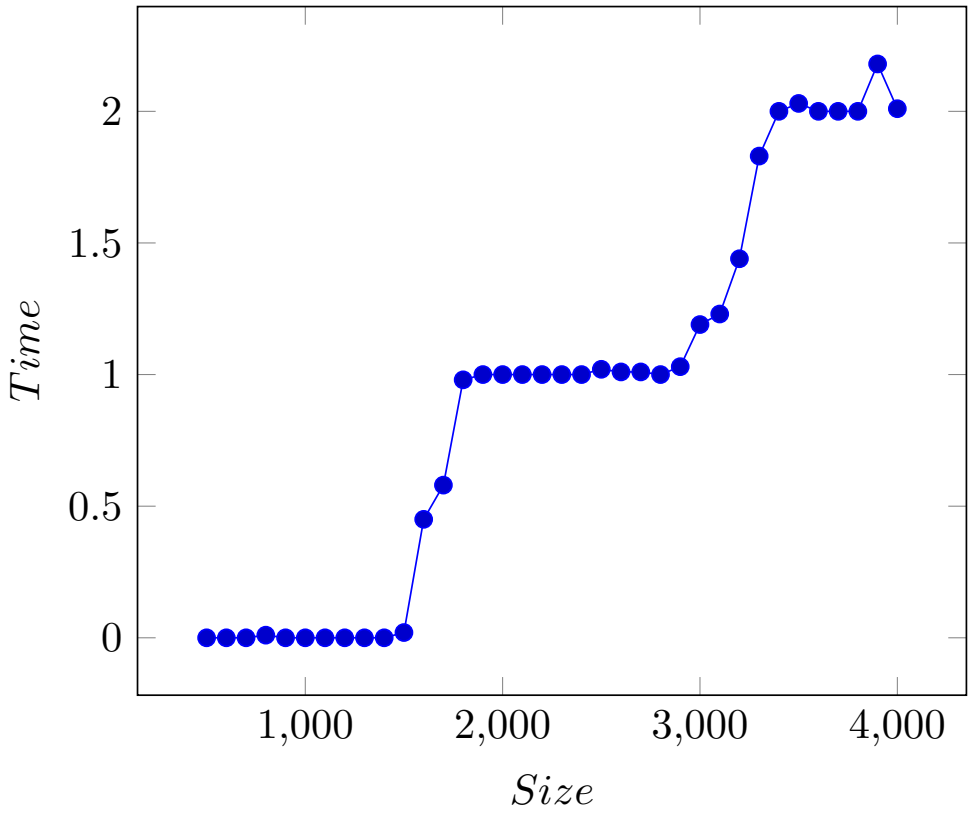
2 График №1 (время работы сортировки merge sort для рандомных векторов)



3 Исходные данные для построение графика №1

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0.01) (1400, 0.02) (1500, 0.95) (1600, 1) (1700, 1)
   (1800, 1) (1900, 1) (2000, 1) (2100, 1.01) (2200, 1) (2300, 1)
   (2400, 1) (2500, 1.02) (2600, 1) (2700, 1.02) (2800, 1.07) (2900,
   1.41) (3000, 2.08) (3100, 2) (3200, 2) (3300, 2) (3400, 2) (3500,
   2.04) (3600, 2) (3700, 2.01) (3800, 2) (3900, 2.05) (4000, 2.05)
```

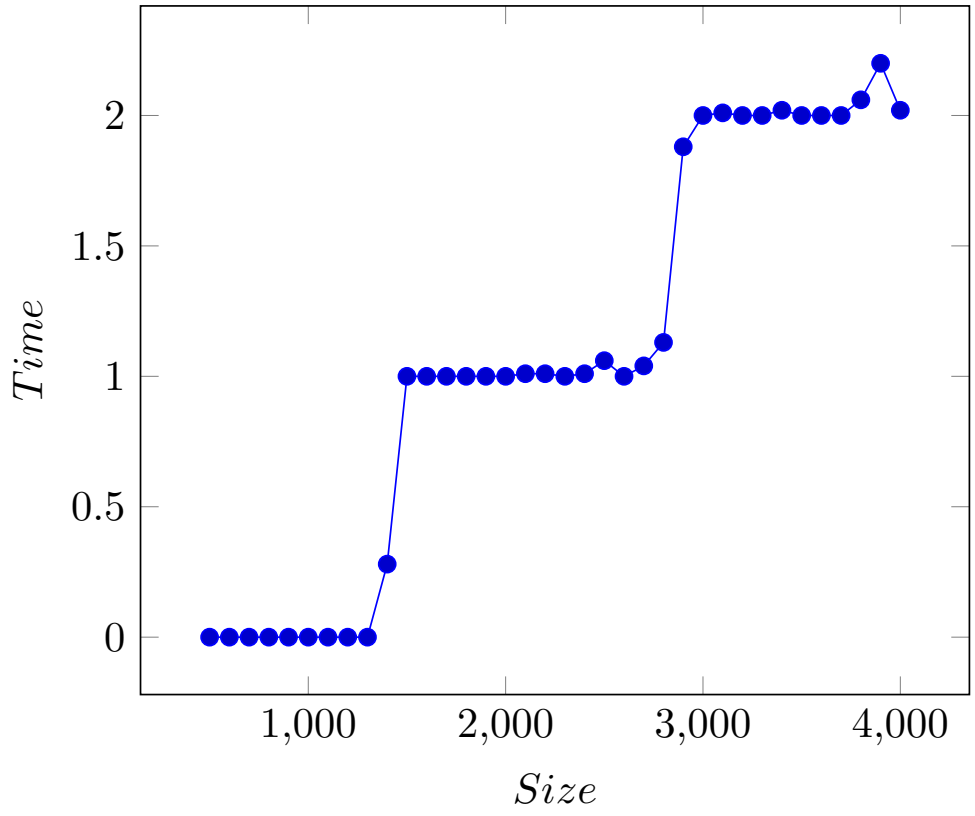
4 График №2 (время работы сортировки merge sort для векторов, отсортированных по невозрастанию)



5 Исходные данные для построение графика №2

```
(500, 0) (600, 0) (700, 0) (800, 0.01) (900, 0) (1000, 0) (1100, 0)
   (1200, 0) (1300, 0) (1400, 0) (1500, 0.02) (1600, 0.45) (1700, 0.58)
    (1800, 0.98) (1900, 1) (2000, 1) (2100, 1) (2200, 1) (2300, 1)
   (2400, 1) (2500, 1.02) (2600, 1.01) (2700, 1.01) (2800, 1) (2900,
   1.03) (3000, 1.19) (3100, 1.23) (3200, 1.44) (3300, 1.83) (3400, 2)
   (3500, 2.03) (3600, 2) (3700, 2) (3800, 2) (3900, 2.18) (4000, 2.01)
```

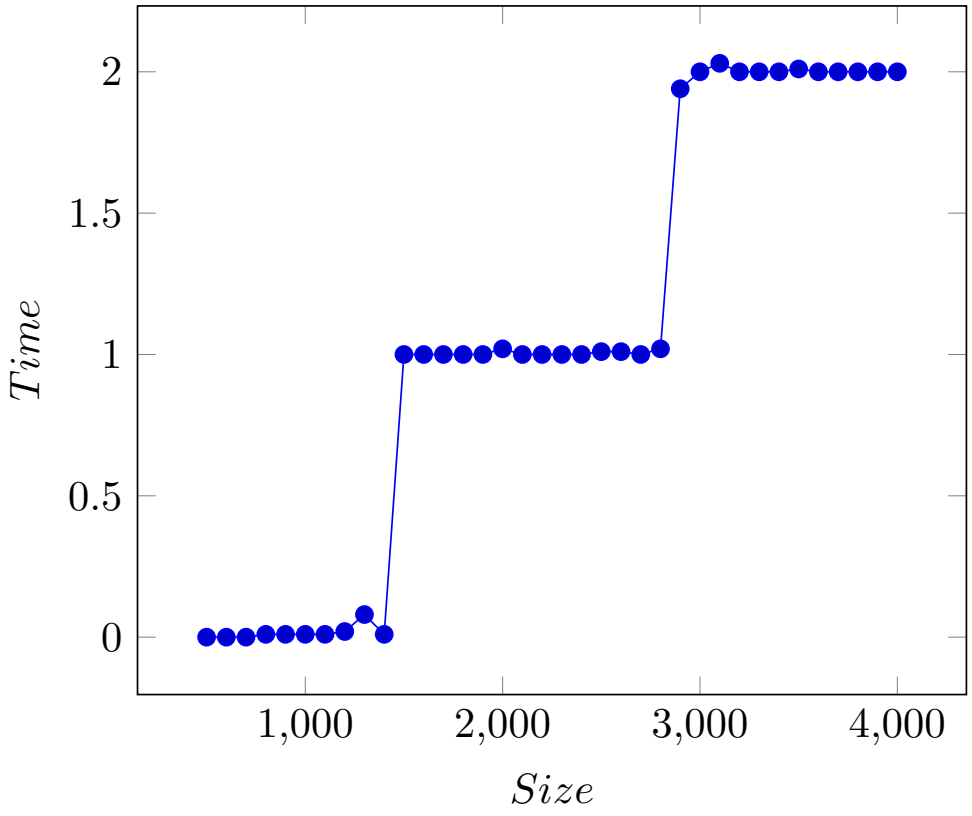
6 График №3 (время работы сортировки merge sort для почти отсортированных векторов)



7 Исходные данные для построение графика №3

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0) (1400, 0.28) (1500, 1) (1600, 1) (1700, 1) (1800, 1)
   (1900, 1) (2000, 1) (2100, 1.01) (2200, 1.01) (2300, 1) (2400, 1.01)
    (2500, 1.06) (2600, 1) (2700, 1.04) (2800, 1.13) (2900, 1.88)
   (3000, 2) (3100, 2.01) (3200, 2) (3300, 2) (3400, 2.02) (3500, 2)
   (3600, 2) (3700, 2) (3800, 2.06) (3900, 2.2) (4000, 2.02)
```

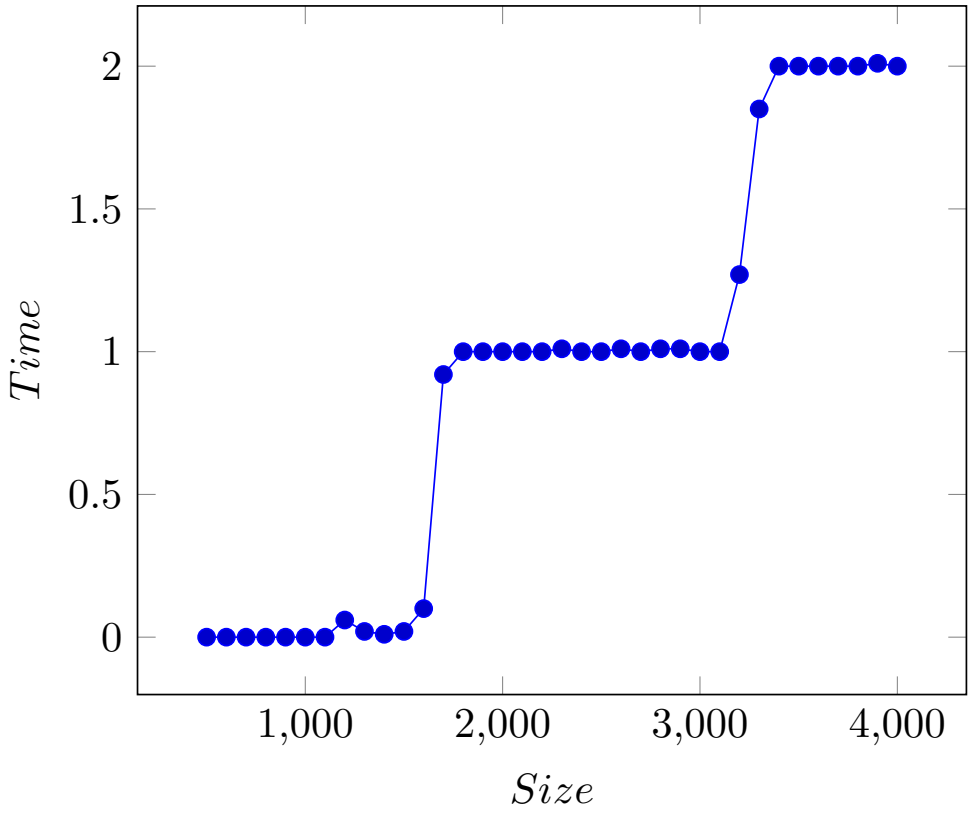
8 График №4 (время работы сортировки hybrid5 sort для рандомных векторов)



9 Исходные данные для построение графика №4

```
(500, 0) (600, 0) (700, 0) (800, 0.01) (900, 0.01) (1000, 0.01) (1100,
    0.01) (1200, 0.02) (1300, 0.08) (1400, 0.01) (1500, 1) (1600, 1)
    (1700, 1) (1800, 1) (1900, 1) (2000, 1.02) (2100, 1) (2200, 1)
    (2300, 1) (2400, 1) (2500, 1.01) (2600, 1.01) (2700, 1) (2800, 1.02)
     (2900, 1.94) (3000, 2) (3100, 2.03) (3200, 2) (3300, 2) (3400, 2)
    (3500, 2.01) (3600, 2) (3700, 2) (3800, 2) (3900, 2) (4000, 2)
```

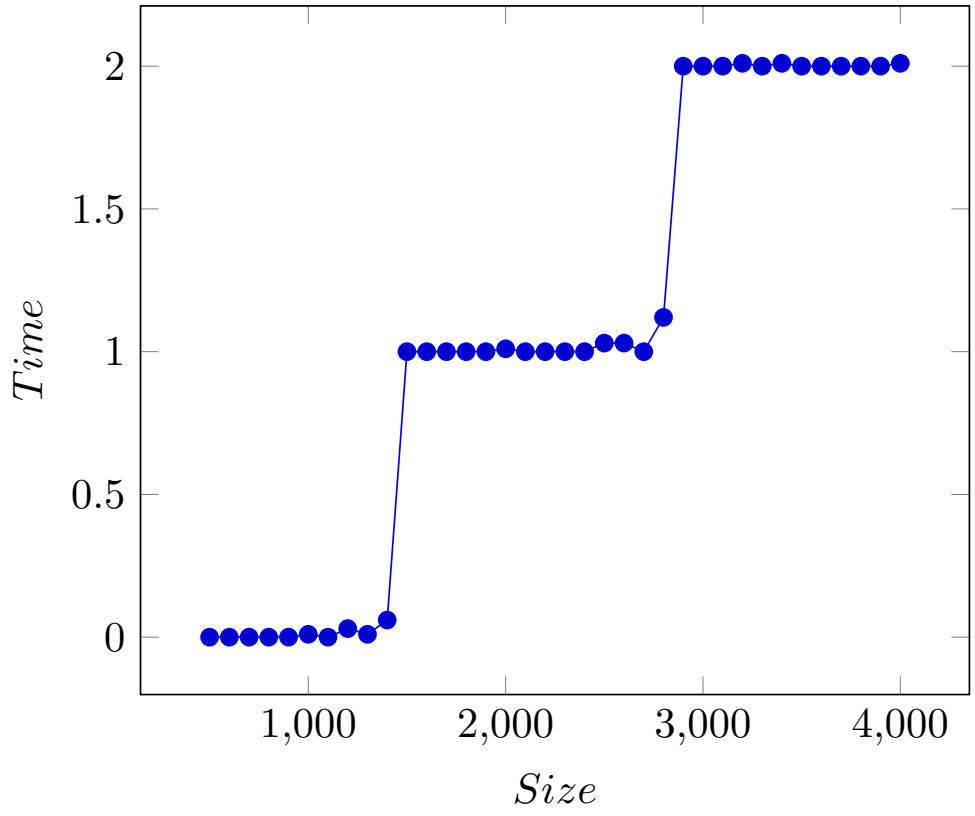
10 График №5 (время работы сортировки hybrid5 sort для векторов, отсортированных по невозрастанию)



11 Исходные данные для построение графика №5

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0.06) (1300, 0.02) (1400, 0.01) (1500, 0.02) (1600, 0.1) (1700,
    0.92) (1800, 1) (1900, 1) (2000, 1) (2100, 1) (2200, 1) (2300, 1.01)
    (2400, 1) (2500, 1) (2600, 1.01) (2700, 1) (2800, 1.01) (2900,
    1.01) (3000, 1) (3100, 1) (3200, 1.27) (3300, 1.85) (3400, 2) (3500,
    2) (3600, 2) (3700, 2) (3800, 2) (3900, 2.01) (4000, 2)
```

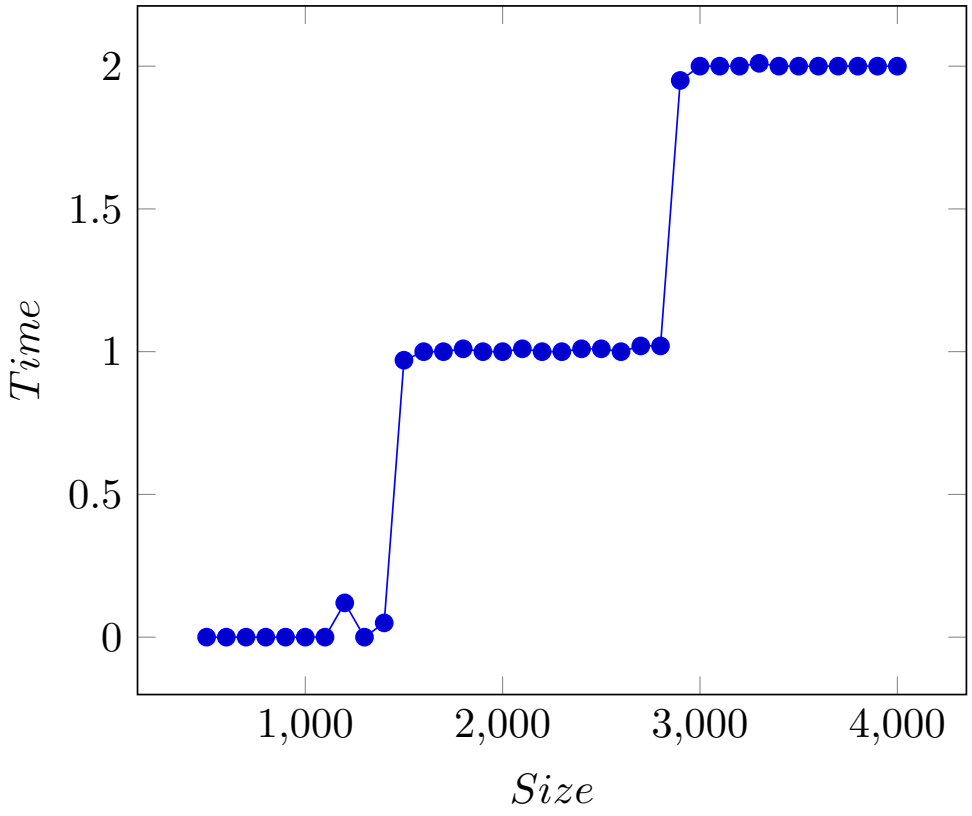
12 График №6 (время работы сортировки hybrid5 sort для почти отсортированных векторов)



13 Исходные данные для построение графика №6

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0.01) (1100, 0)
    (1200, 0.03) (1300, 0.01) (1400, 0.06) (1500, 1) (1600, 1) (1700, 1)
     (1800, 1) (1900, 1) (2000, 1.01) (2100, 1) (2200, 1) (2300, 1)
    (2400, 1) (2500, 1.03) (2600, 1.03) (2700, 1) (2800, 1.12) (2900, 2)
     (3000, 2) (3100, 2) (3200, 2.01) (3300, 2) (3400, 2.01) (3500, 2)
    (3600, 2) (3700, 2) (3800, 2) (3900, 2) (4000, 2.01)
```

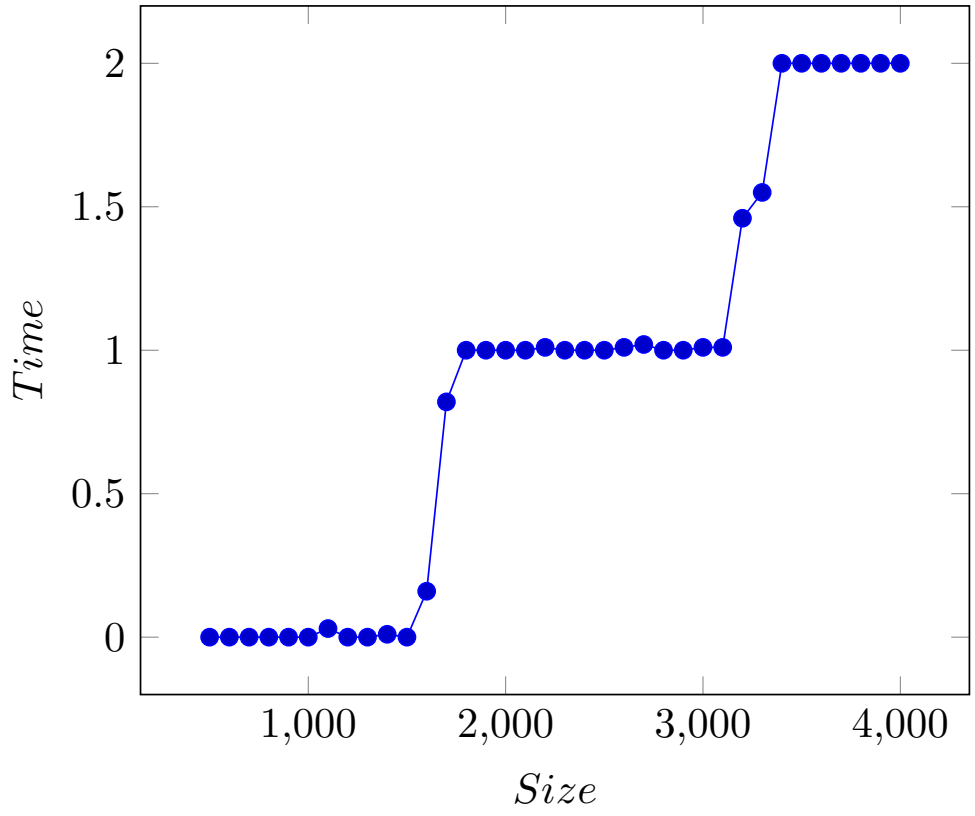
14 График №7 (время работы сортировки hybrid10 sort для рандомных векторов)



15 Исходные данные для построение графика №7

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0.12) (1300, 0) (1400, 0.05) (1500, 0.97) (1600, 1) (1700, 1)
   (1800, 1.01) (1900, 1) (2000, 1) (2100, 1.01) (2200, 1) (2300, 1)
   (2400, 1.01) (2500, 1.01) (2600, 1) (2700, 1.02) (2800, 1.02) (2900,
    1.95) (3000, 2) (3100, 2) (3200, 2) (3300, 2.01) (3400, 2) (3500,
   2) (3600, 2) (3700, 2) (3800, 2) (3900, 2) (4000, 2)
```

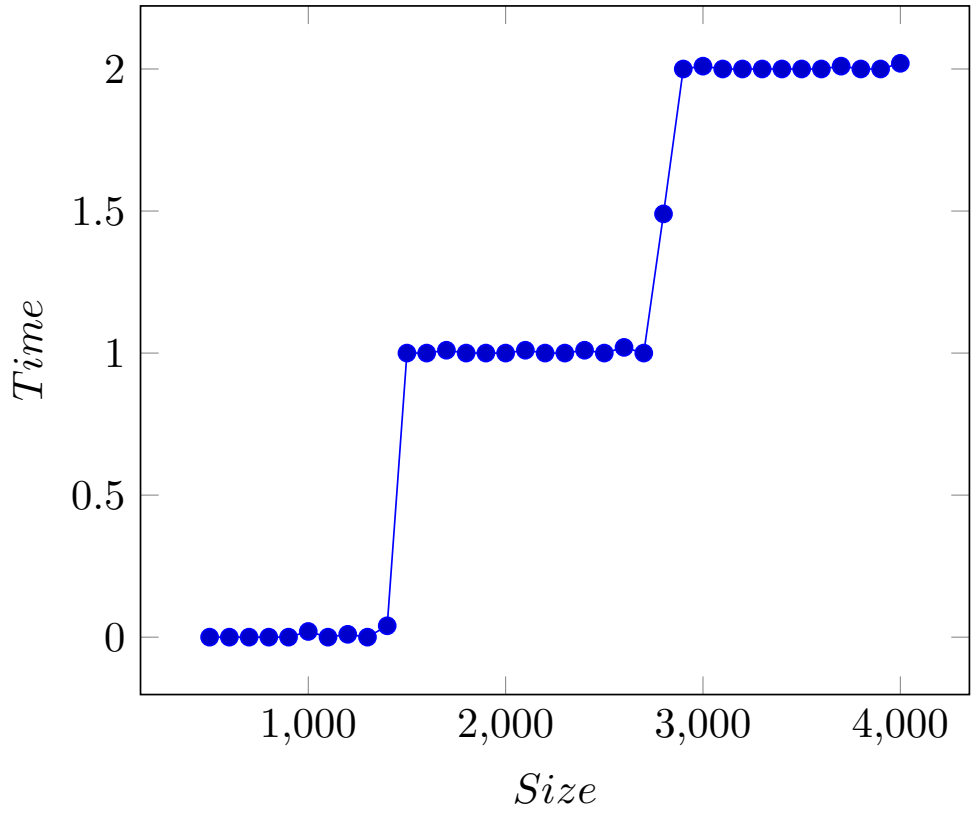
16 График №8 (время работы сортировки hybrid10 sort для векторов, отсортированных по невозрастанию)



17 Исходные данные для построение графика №8

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0.03)
    (1200, 0) (1300, 0) (1400, 0.01) (1500, 0) (1600, 0.16) (1700, 0.82)
     (1800, 1) (1900, 1) (2000, 1) (2100, 1) (2200, 1.01) (2300, 1)
    (2400, 1) (2500, 1) (2600, 1.01) (2700, 1.02) (2800, 1) (2900, 1)
    (3000, 1.01) (3100, 1.01) (3200, 1.46) (3300, 1.55) (3400, 2) (3500,
     2) (3600, 2) (3700, 2) (3800, 2) (3900, 2) (4000, 2)
```

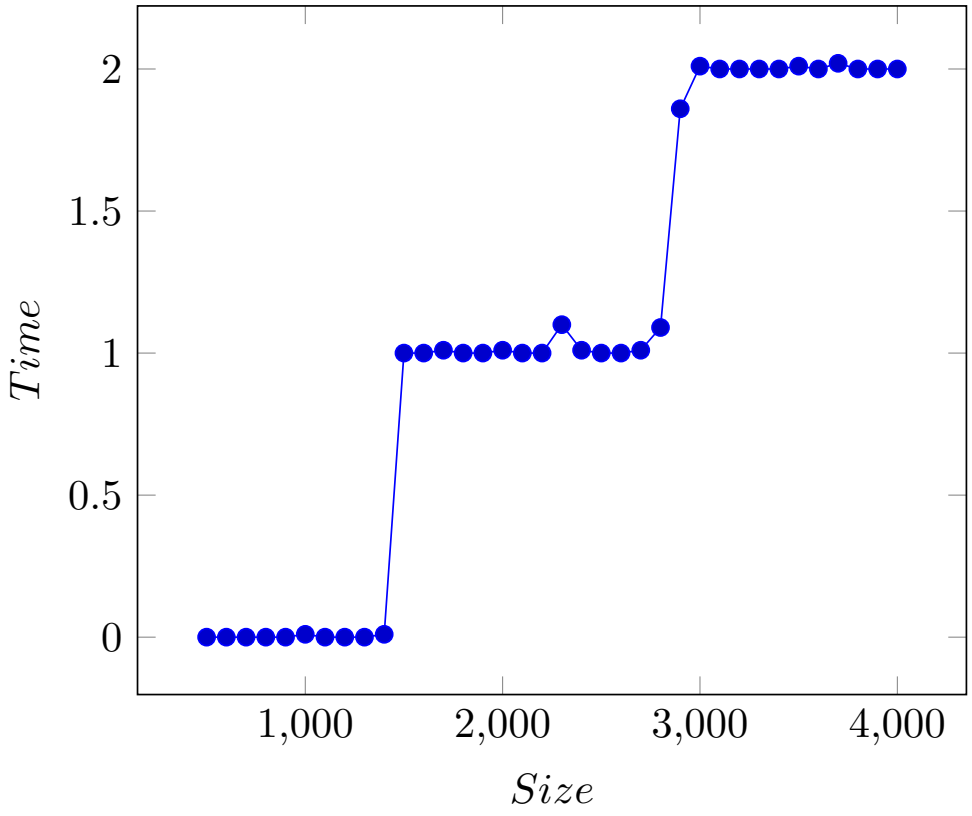
18 График №9 (время работы сортировки hybrid10 sort для почти отсортированных векторов)



19 Исходные данные для построение графика №9

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0.02) (1100, 0)
    (1200, 0.01) (1300, 0) (1400, 0.04) (1500, 1) (1600, 1) (1700, 1.01)
     (1800, 1) (1900, 1) (2000, 1) (2100, 1.01) (2200, 1) (2300, 1)
    (2400, 1.01) (2500, 1) (2600, 1.02) (2700, 1) (2800, 1.49) (2900, 2)
     (3000, 2.01) (3100, 2) (3200, 2) (3300, 2) (3400, 2) (3500, 2)
    (3600, 2) (3700, 2.01) (3800, 2) (3900, 2) (4000, 2.02)
```

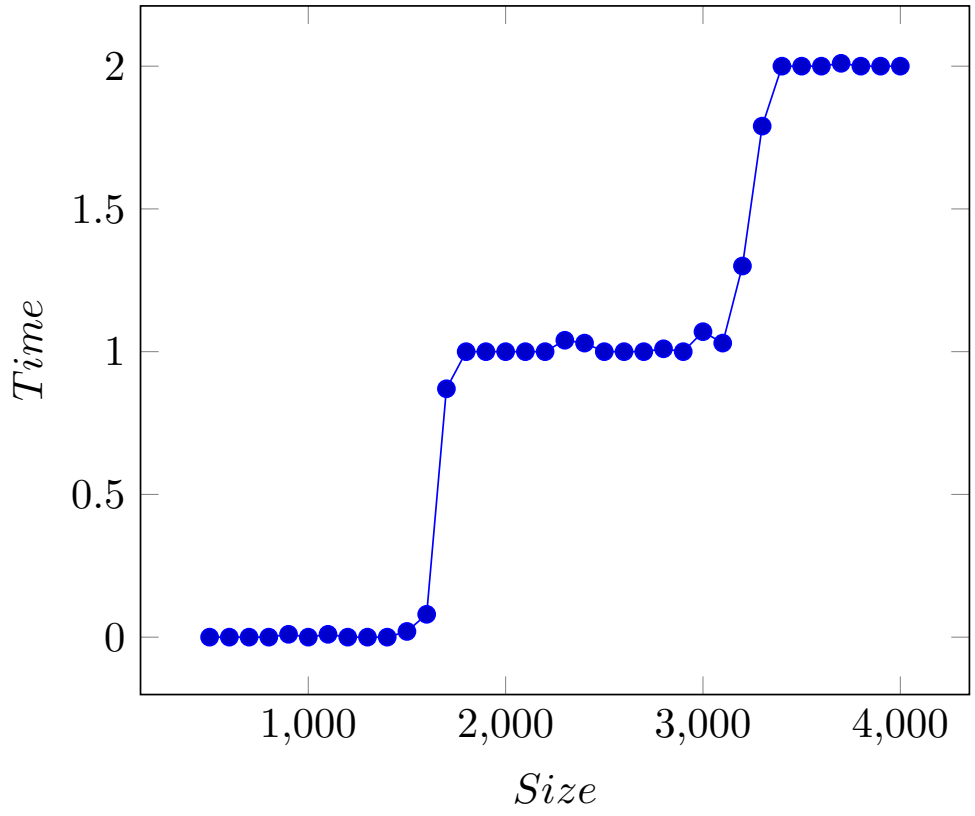
20 График №10 (время работы сортировки hybrid20 sort для рандомных векторов)



21 Исходные данные для построение графика №10

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0.01) (1100, 0)
    (1200, 0) (1300, 0) (1400, 0.01) (1500, 1) (1600, 1) (1700, 1.01)
    (1800, 1) (1900, 1) (2000, 1.01) (2100, 1) (2200, 1) (2300, 1.1)
    (2400, 1.01) (2500, 1) (2600, 1) (2700, 1.01) (2800, 1.09) (2900,
    1.86) (3000, 2.01) (3100, 2) (3200, 2) (3300, 2) (3400, 2) (3500,
    2.01) (3600, 2) (3700, 2.02) (3800, 2) (3900, 2) (4000, 2)
```

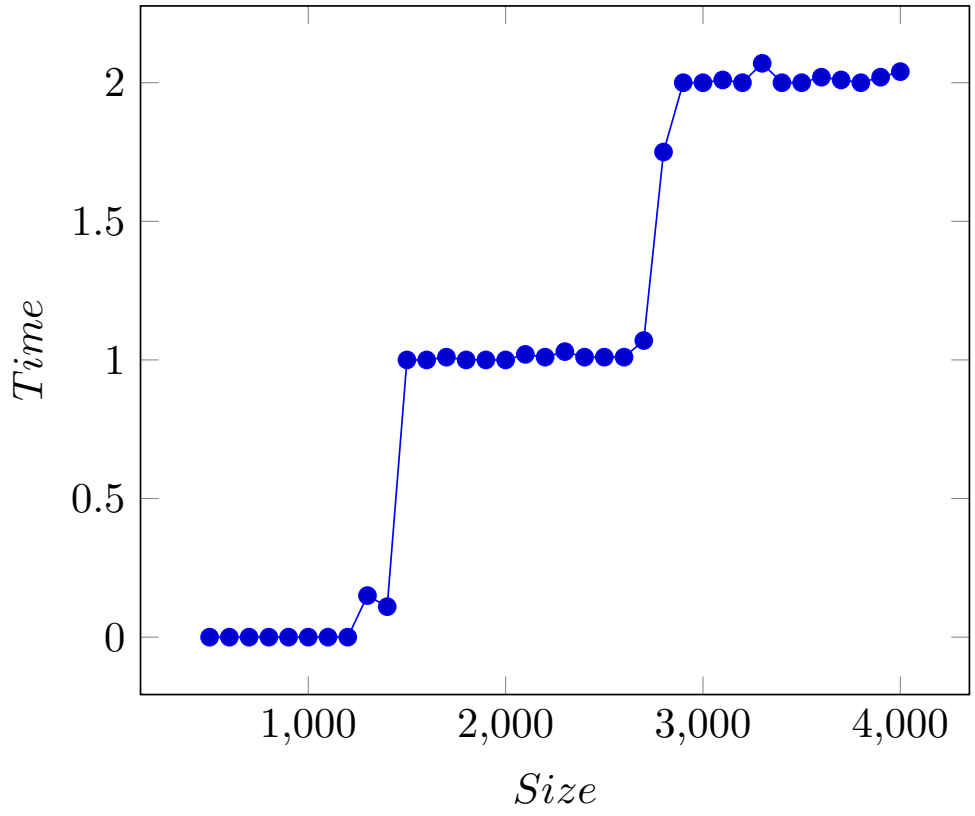
22 График №11 (время работы сортировки hybrid20 sort для векторов, отсортированных по невозрастанию)



23 Исходные данные для построение графика №11

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0.01) (1000, 0) (1100, 0.01)
    (1200, 0) (1300, 0) (1400, 0) (1500, 0.02) (1600, 0.08) (1700, 0.87)
     (1800, 1) (1900, 1) (2000, 1) (2100, 1) (2200, 1) (2300, 1.04)
    (2400, 1.03) (2500, 1) (2600, 1) (2700, 1) (2800, 1.01) (2900, 1)
    (3000, 1.07) (3100, 1.03) (3200, 1.3) (3300, 1.79) (3400, 2) (3500,
    2) (3600, 2) (3700, 2.01) (3800, 2) (3900, 2) (4000, 2)
```

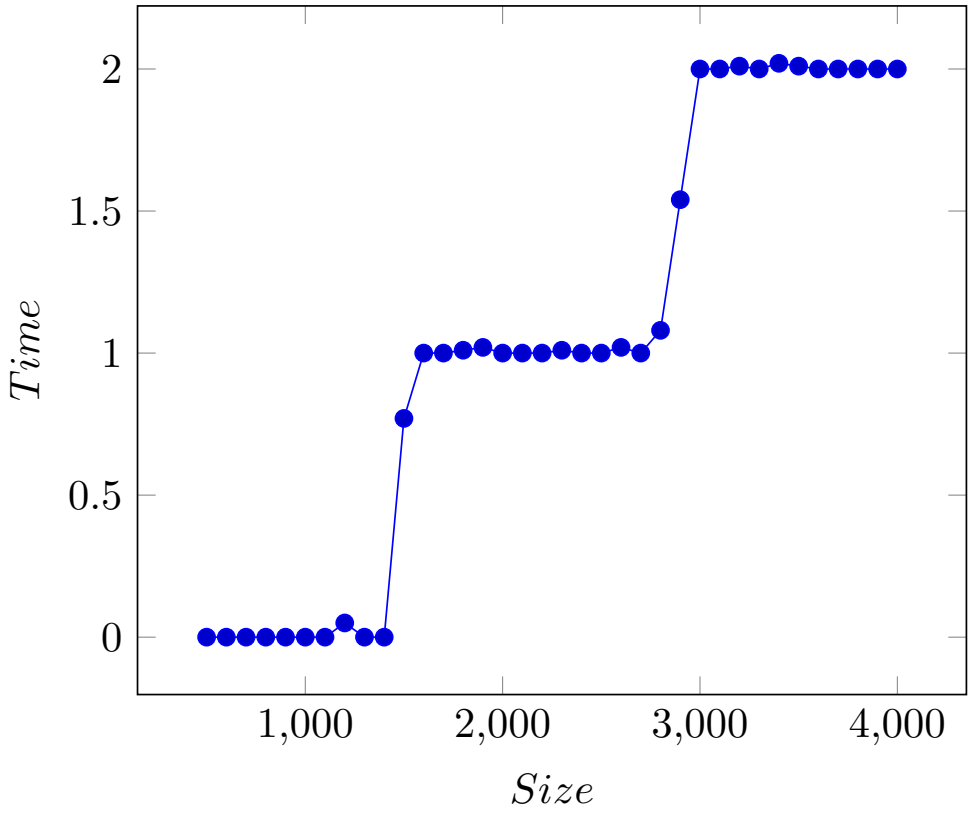
24 График №12 (время работы сортировки hybrid20 sort для почти отсортированных векторов)



25 Исходные данные для построение графика №12

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0.15) (1400, 0.11) (1500, 1) (1600, 1) (1700, 1.01)
   (1800, 1) (1900, 1) (2000, 1) (2100, 1.02) (2200, 1.01) (2300, 1.03)
    (2400, 1.01) (2500, 1.01) (2600, 1.01) (2700, 1.07) (2800, 1.75)
   (2900, 2) (3000, 2) (3100, 2.01) (3200, 2) (3300, 2.07) (3400, 2)
   (3500, 2) (3600, 2.02) (3700, 2.01) (3800, 2) (3900, 2.02) (4000,
   2.04)
```

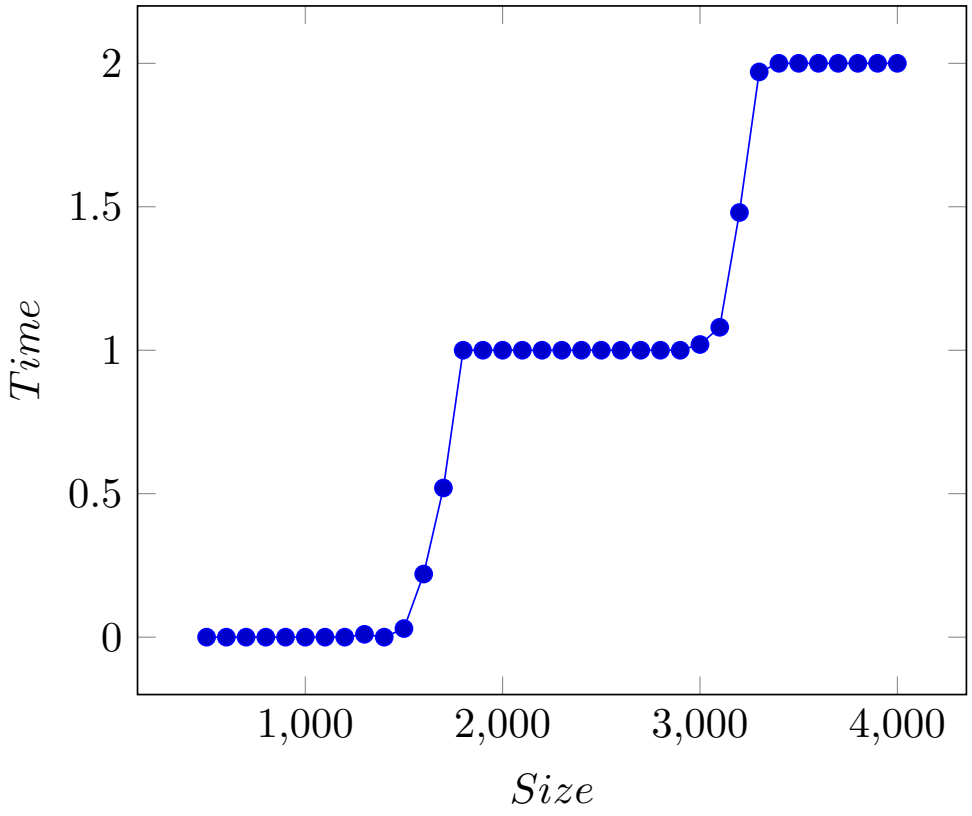
26 График №13 (время работы сортировки hybrid50 sort для рандомных векторов)



27 Исходные данные для построение графика №13

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0.05) (1300, 0) (1400, 0) (1500, 0.77) (1600, 1) (1700, 1) (1800,
    1.01) (1900, 1.02) (2000, 1) (2100, 1) (2200, 1) (2300, 1.01) (2400,
    1) (2500, 1) (2600, 1.02) (2700, 1) (2800, 1.08) (2900, 1.54)
    (3000, 2) (3100, 2) (3200, 2.01) (3300, 2) (3400, 2.02) (3500, 2.01)
    (3600, 2) (3700, 2) (3800, 2) (3900, 2) (4000, 2)
```

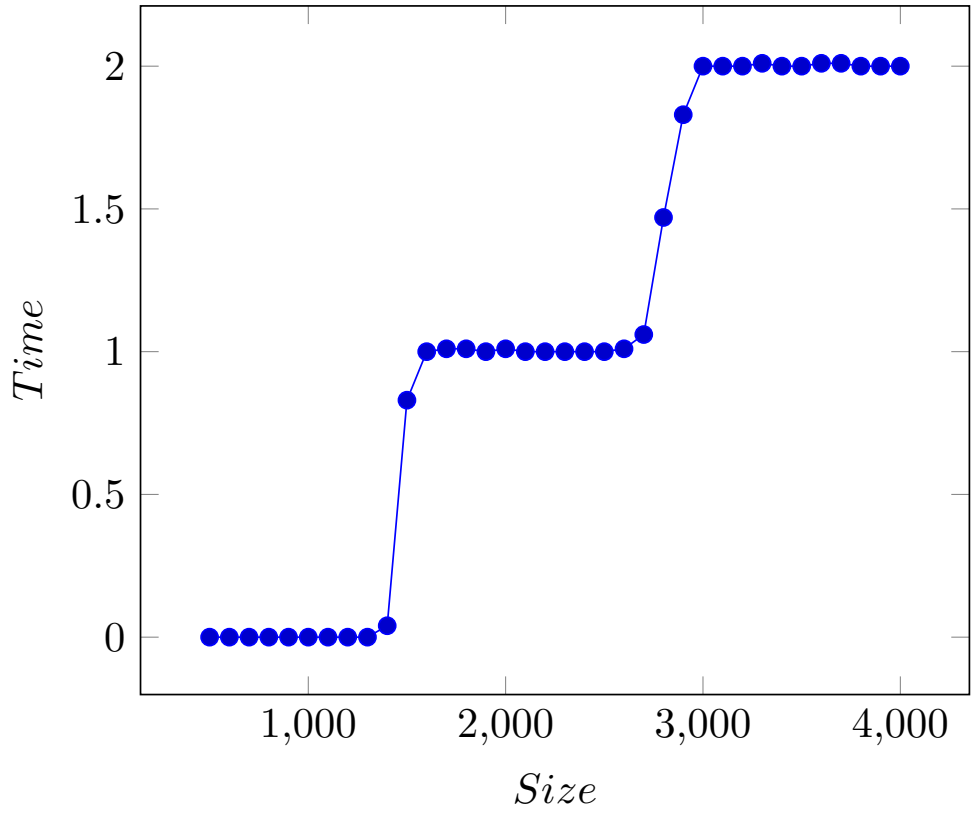
28 График №14 (время работы сортировки hybrid50 sort для векторов, отсортированных по невозрастанию)



29 Исходные данные для построение графика №14

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0.01) (1400, 0) (1500, 0.03) (1600, 0.22) (1700, 0.52)
   (1800, 1) (1900, 1) (2000, 1) (2100, 1) (2200, 1) (2300, 1) (2400,
   1) (2500, 1) (2600, 1) (2700, 1) (2800, 1) (2900, 1) (3000, 1.02)
   (3100, 1.08) (3200, 1.48) (3300, 1.97) (3400, 2) (3500, 2) (3600, 2)
    (3700, 2) (3800, 2) (3900, 2) (4000, 2)
```

30 График №15 (время работы сортировки hybrid50 sort для почти отсортированных векторов)



31 Исходные данные для построение графика №15

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0) (1400, 0.04) (1500, 0.83) (1600, 1) (1700, 1.01)
   (1800, 1.01) (1900, 1) (2000, 1.01) (2100, 1) (2200, 1) (2300, 1)
   (2400, 1) (2500, 1) (2600, 1.01) (2700, 1.06) (2800, 1.47) (2900,
   1.83) (3000, 2) (3100, 2) (3200, 2) (3300, 2.01) (3400, 2) (3500, 2)
    (3600, 2.01) (3700, 2.01) (3800, 2) (3900, 2) (4000, 2)
```

32 Выводы о проделанной работе

Хоть графиков и очень много, однако можно проследить, что существенных приростов производительности не наблюдается, однако можно заметить, что гибридная сортировка с параметром 50 ведет себя более стабильно, так что, выбирая из всех реализованных в этом задании сортировок, я бы выбрал именно её