SET3. Задача A3

Фролов-Буканов Виктор Дмитриевич БПИ-228

28 ноября 2023

1 Исходный код

Программа декомпозирована на 3 header-файла и 1 файл main.cpp.
Header-файл heap_sort.h содержит реализацию пирамидальной сортировки
Header-файл quick_sort.h содержит реализацию быстрой сортировки, а также гибридной
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

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

enum sort_type {
  quick,
  hybrid
};

long long mark_time(std::vector<int> &vec, sort_type value) {
  long long millisec;
  if (value == quick) {
    auto start = std::chrono::high_resolution_clock::now();
    quick_sort(vec, 0, static_cast<int>(vec.size()) - 1);
    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()) - 1);
    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_quick.txt)");
std::ofstream fout_rev(R"(C:\Users\frolo\CLionProjects\
    assemblyTestProgram\reversed_quick.txt)");
std::ofstream fout_sor(R"(C:\Users\frolo\CLionProjects\
    assemblyTestProgram\alm_sorted_quick.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(quick);
  fout_abs.close();
  fout_rev.close();
  fout_sor.close();

  fout_abs.open(R"(C:\Users\frolo\CLionProjects\assemblyTestProgram\
      abs_random_hybrid50.txt)");
  fout_rev.open(R"(C:\Users\frolo\CLionProjects\assemblyTestProgram\
      reversed_hybrid50.txt)");
  fout_sor.open(R"(C:\Users\frolo\CLionProjects\assemblyTestProgram\
      alm_sorted_hybrid50.txt)");

  test_sort(hybrid);

  return 0;
}
```

heap_sort.h

```cpp
#pragma once

#include <iostream>
#include <vector>

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

int left(int i) {
  return 2 * (i + 1) - 1;
}

int right(int i) {
  return 2 * (i + 1);
}

void max_heapify(std::vector<int> &vec, int i, int size, int start) {
    // NOLINT
  int l = left(i - start) + start;
  int r = right(i - start) + start;
  int max = i;
  if (r < start + size && vec[r] > vec[i]) {
    max = r;
  }
  if (l < start + size && vec[l] > vec[max]) {
    max = l;
  }

  if (i != max) {
    std::swap(vec[i], vec[max]);
    max_heapify(vec, max, size, start);
  }
}

void build_heap(std::vector<int> &vec, int start, int end) {
  int size = static_cast<int>(end - start) + 1;
  for (int i = start + size / 2; i != start - 1; --i) {
    max_heapify(vec, i, size, start);
  }
}

void heap_sort(std::vector<int> &vec, int start, int end) {
  if (end == start) return;
  build_heap(vec, start, end);
  int size = static_cast<int>(end - start) + 1;
  for (auto i = end; i != start; --i) {
    std::swap(vec[i], vec[start]);
    --size;
    max_heapify(vec, start, size, start);
  }
}
```

quick_sort.h

```cpp
#pragma once

#include <iostream>
#include <random>
```

```cpp
#include <vector>
#include "heap_sort.h"

// A function to return a seeded random number generator.
inline std::mt19937& generator() {
  // the generator will only be seeded once (per thread) since it's static
  static thread_local std::mt19937 gen(std::random_device{}());
  return gen;
}

// A function to generate integers in the range [min, max]
template<typename T, std::enable_if_t<std::is_integral_v<T>>* = nullptr>
T my_rand(T min, T max) {
  std::uniform_int_distribution<T> dist(min, max);
  return dist(generator());
}

// A function to generate floats in the range [min, max)
template<typename T, std::enable_if_t<std::is_floating_point_v<T>>* = nullptr>
T my_rand(T min, T max) {
  std::uniform_real_distribution<T> dist(min, max);
  return dist(generator());
}


std::pair<int, int> partition(std::vector<int>& vec, int left, int right, int pivot) {
  int e = left, g = left;
  for (int i = left; i <= right; i++) {
    if (vec[i] > pivot) continue;
    if (vec[i] < pivot) {
      std::swap(vec[e], vec[i]);
      if (e != g) {
        std::swap(vec[g], vec[i]);
      }
      ++e;
      ++g;
      continue;
    }
    std::swap(vec[g], vec[i]);
    ++g;
  }

  return std::make_pair(e, g);
}

void quick_sort(std::vector<int>& vec, int left, int right) { // NOLINT
  if (left < right) {
    if (right - left == 1 && vec[left] > vec[right]) {
      std::swap(vec[left], vec[right]);
      return;
    }
    int pvt = vec[my_rand(left, right)];
    auto pair = partition(vec, left, right, pvt);
    quick_sort(vec, left, pair.first - 1);
    quick_sort(vec, pair.second, right);
```

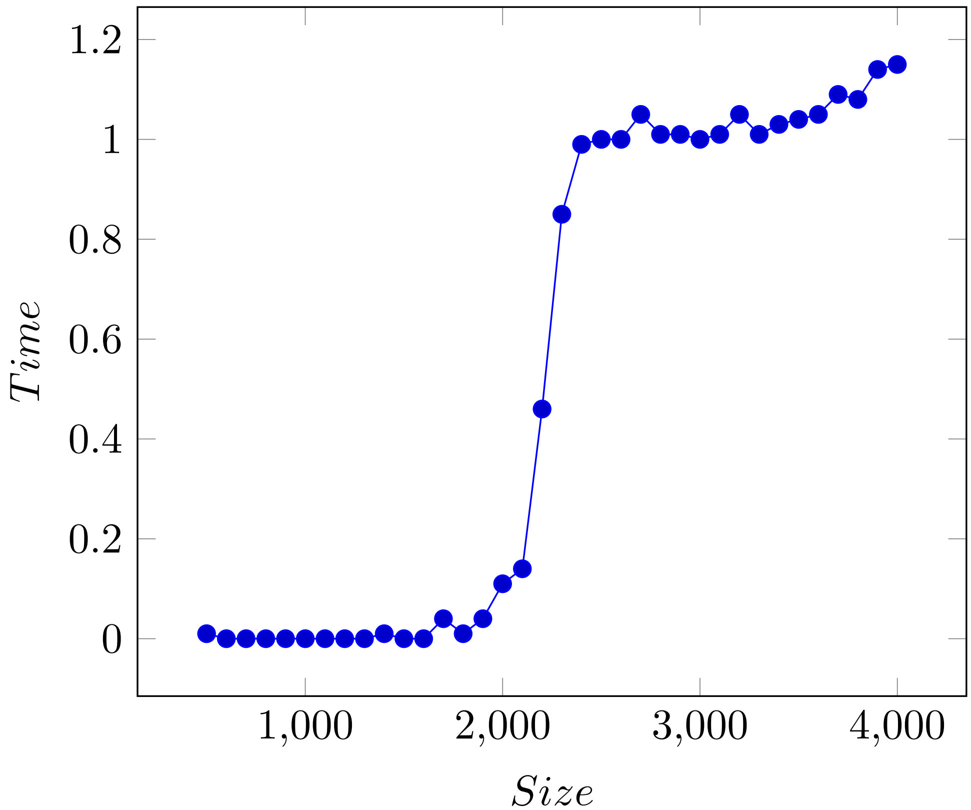
```
  }
}

void hybrid_sort(std::vector<int>& vec, int left, int right) { //
    NOLINT
  if (left < right) {
    if (right - left <= 50) {
      heap_sort(vec, left, right);
      return;
    }
    int pvt = vec[my_rand(left, right)];
    auto pair = partition(vec, left, right, pvt);
    hybrid_sort(vec, left, pair.first - 1);
    hybrid_sort(vec, pair.second, right);
  }
}
```

random_vec.h

```
#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
  }
}
```

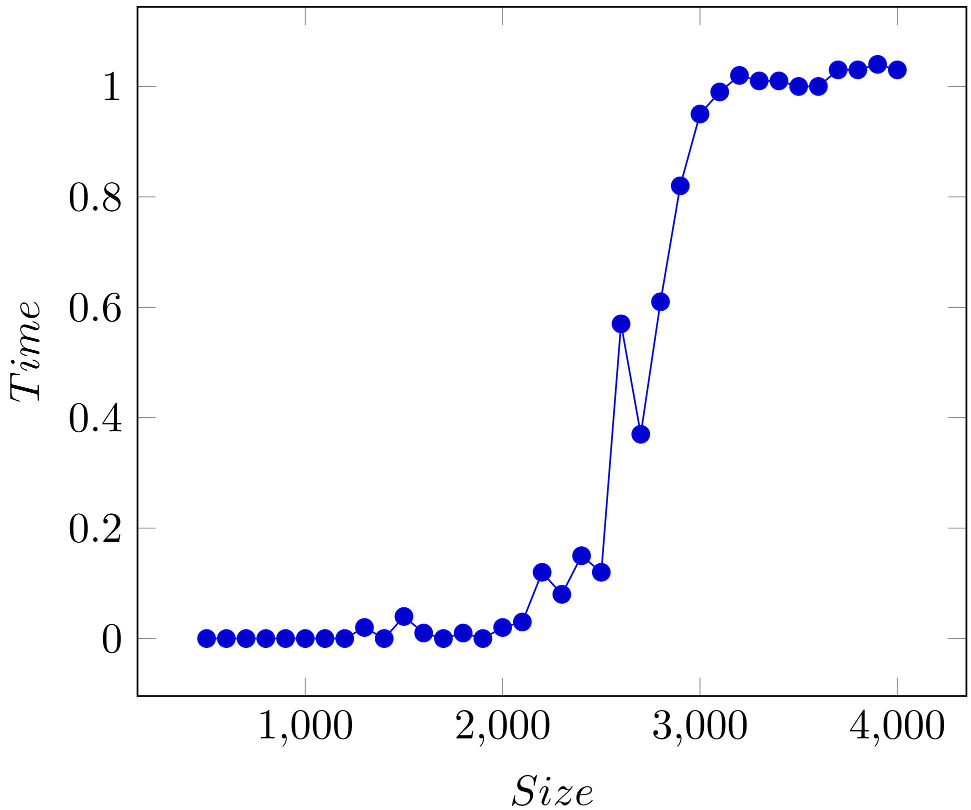
2 График №1 (время работы сортировки quick sort для рандомных векторов)



3 Исходные данные для построение графика №1

```
(500, 0.01) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0)
    (1200, 0) (1300, 0) (1400, 0.01) (1500, 0) (1600, 0) (1700, 0.04)
    (1800, 0.01) (1900, 0.04) (2000, 0.11) (2100, 0.14) (2200, 0.46)
    (2300, 0.85) (2400, 0.99) (2500, 1) (2600, 1) (2700, 1.05) (2800,
    1.01) (2900, 1.01) (3000, 1) (3100, 1.01) (3200, 1.05) (3300, 1.01)
    (3400, 1.03) (3500, 1.04) (3600, 1.05) (3700, 1.09) (3800, 1.08)
    (3900, 1.14) (4000, 1.15)
```

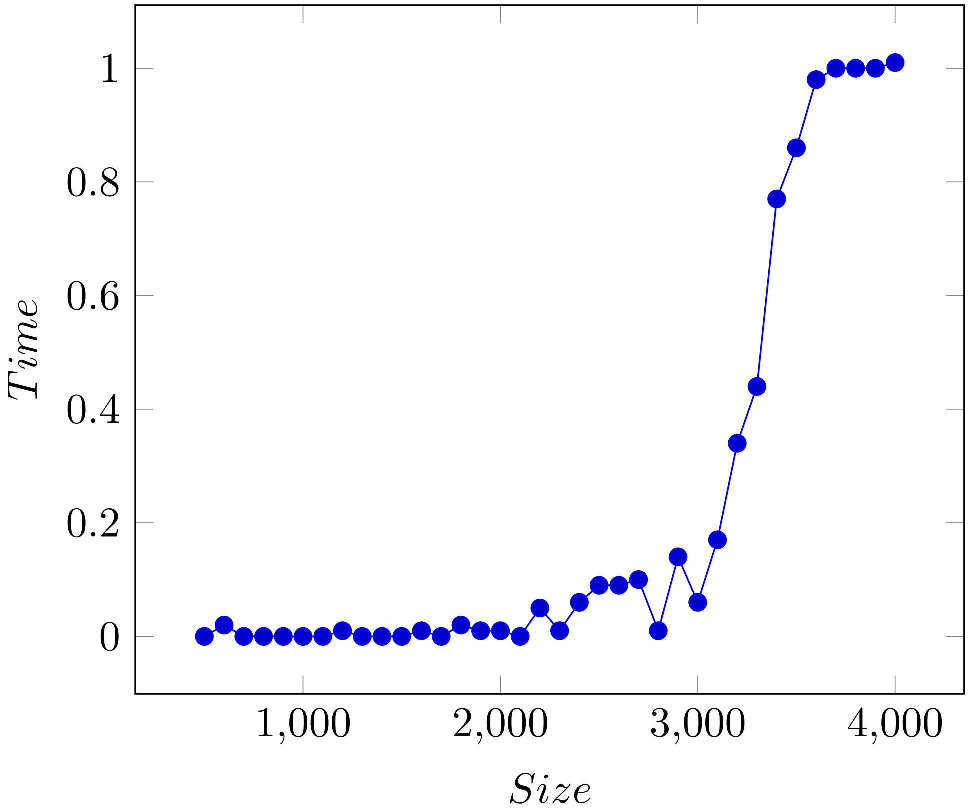
4 График №2 (время работы сортировки quick sort для векторов, отсортированных по невозрастанию)



5 Исходные данные для построение графика №2

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0.02) (1400, 0) (1500, 0.04) (1600, 0.01) (1700, 0)
    (1800, 0.01) (1900, 0) (2000, 0.02) (2100, 0.03) (2200, 0.12) (2300,
    0.08) (2400, 0.15) (2500, 0.12) (2600, 0.57) (2700, 0.37) (2800,
    0.61) (2900, 0.82) (3000, 0.95) (3100, 0.99) (3200, 1.02) (3300,
    1.01) (3400, 1.01) (3500, 1) (3600, 1) (3700, 1.03) (3800, 1.03)
    (3900, 1.04) (4000, 1.03)
```

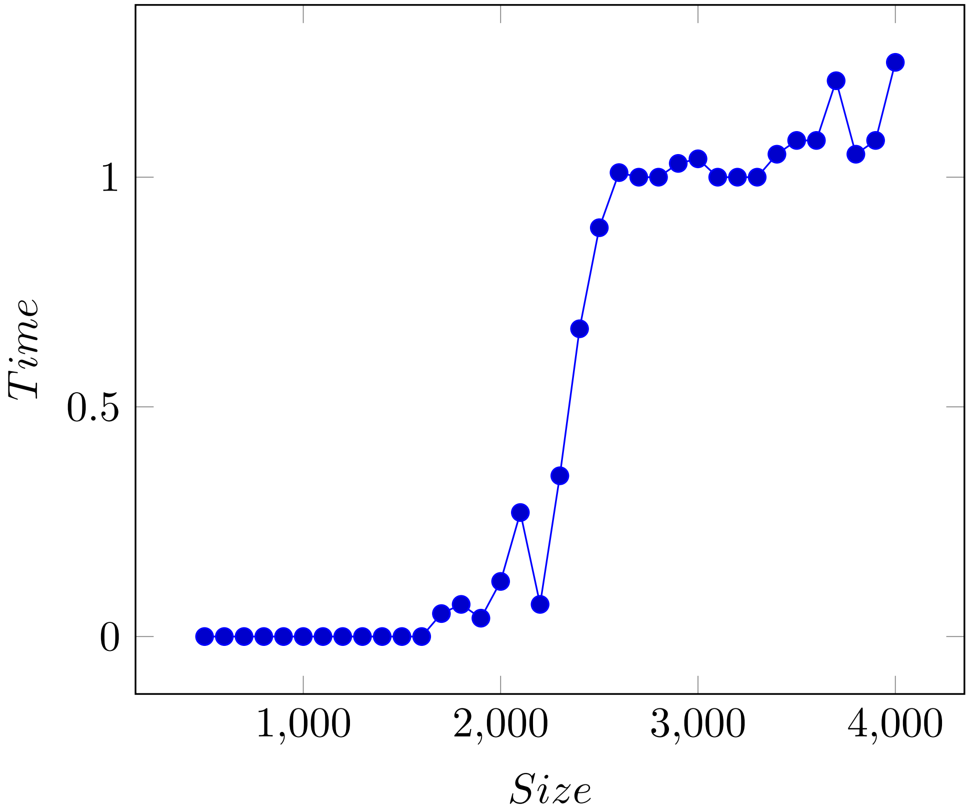
6 График №3 (время работы сортировки quick sort для почти отсортированных векторов)



7 Исходные данные для построение графика №3

```
(500, 0) (600, 0.02) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0)
    (1200, 0.01) (1300, 0) (1400, 0) (1500, 0) (1600, 0.01) (1700, 0)
    (1800, 0.02) (1900, 0.01) (2000, 0.01) (2100, 0) (2200, 0.05) (2300,
     0.01) (2400, 0.06) (2500, 0.09) (2600, 0.09) (2700, 0.1) (2800,
    0.01) (2900, 0.14) (3000, 0.06) (3100, 0.17) (3200, 0.34) (3300,
    0.44) (3400, 0.77) (3500, 0.86) (3600, 0.98) (3700, 1) (3800, 1)
    (3900, 1) (4000, 1.01)
```

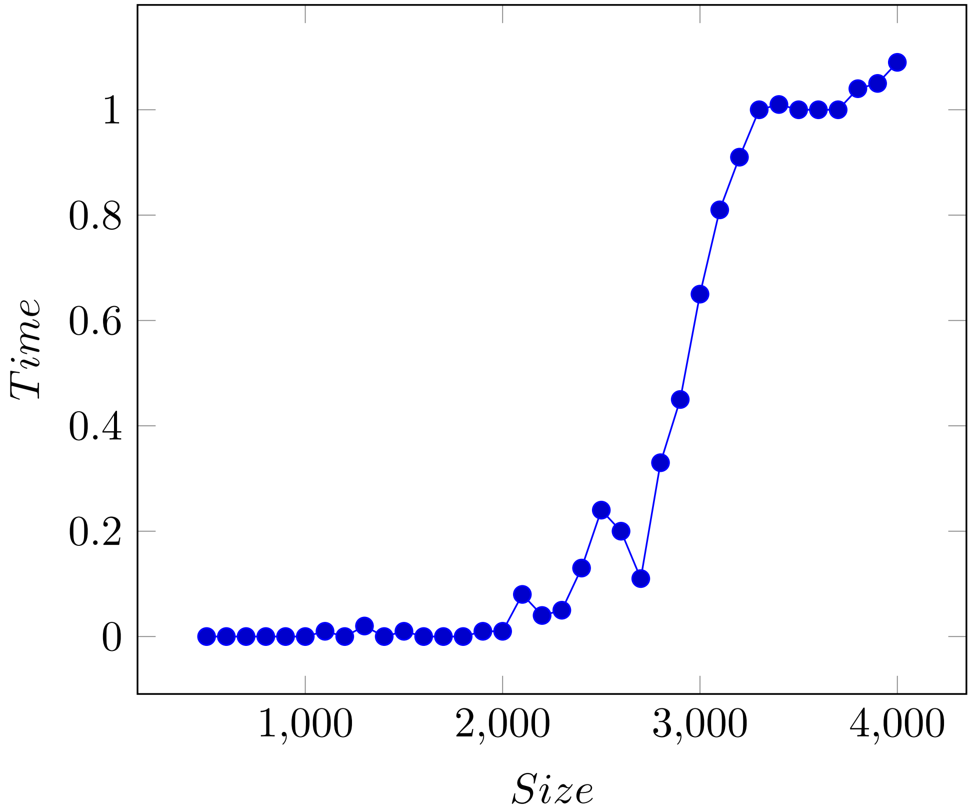
8 График №4 (время работы сортировки hybrid5 sort для рандомных векторов)



9 Исходные данные для построение графика №4

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0) (1400, 0) (1500, 0) (1600, 0) (1700, 0.05) (1800,
   0.07) (1900, 0.04) (2000, 0.12) (2100, 0.27) (2200, 0.07) (2300,
   0.35) (2400, 0.67) (2500, 0.89) (2600, 1.01) (2700, 1) (2800, 1)
   (2900, 1.03) (3000, 1.04) (3100, 1) (3200, 1) (3300, 1) (3400, 1.05)
    (3500, 1.08) (3600, 1.08) (3700, 1.21) (3800, 1.05) (3900, 1.08)
   (4000, 1.25)
```

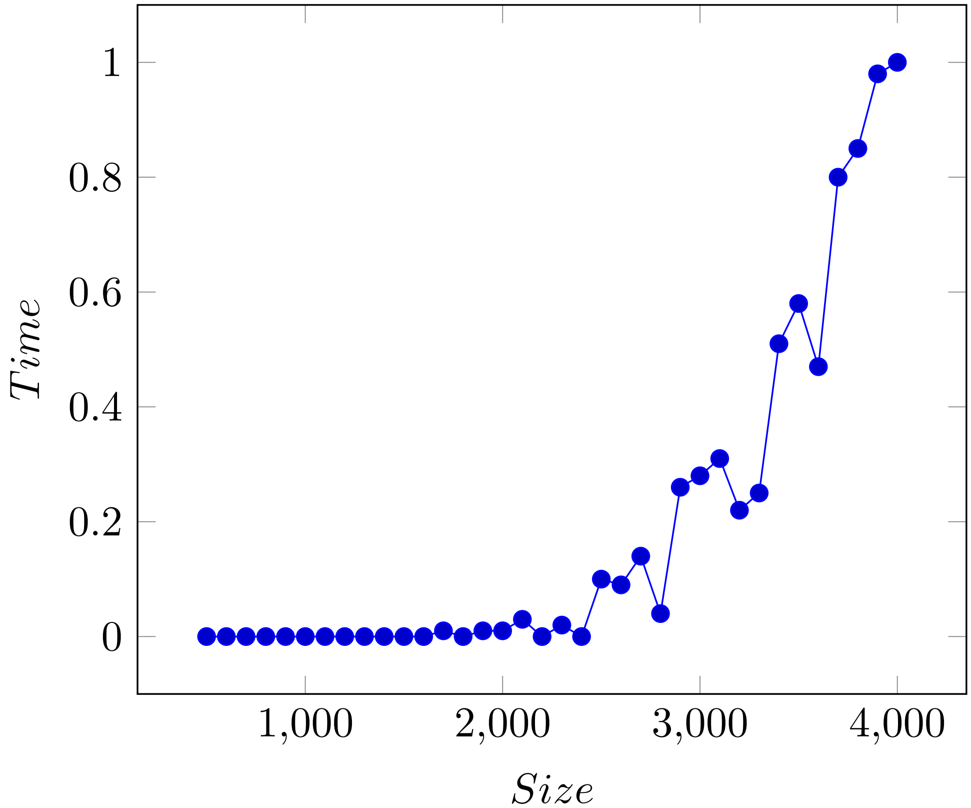
10 График №5 (время работы сортировки hybrid5 sort для векторов, отсортированных по невозрастанию)



11 Исходные данные для построение графика №5

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0.01)
   (1200, 0) (1300, 0.02) (1400, 0) (1500, 0.01) (1600, 0) (1700, 0)
   (1800, 0) (1900, 0.01) (2000, 0.01) (2100, 0.08) (2200, 0.04) (2300,
    0.05) (2400, 0.13) (2500, 0.24) (2600, 0.2) (2700, 0.11) (2800,
   0.33) (2900, 0.45) (3000, 0.65) (3100, 0.81) (3200, 0.91) (3300, 1)
   (3400, 1.01) (3500, 1) (3600, 1) (3700, 1) (3800, 1.04) (3900, 1.05)
    (4000, 1.09)
```

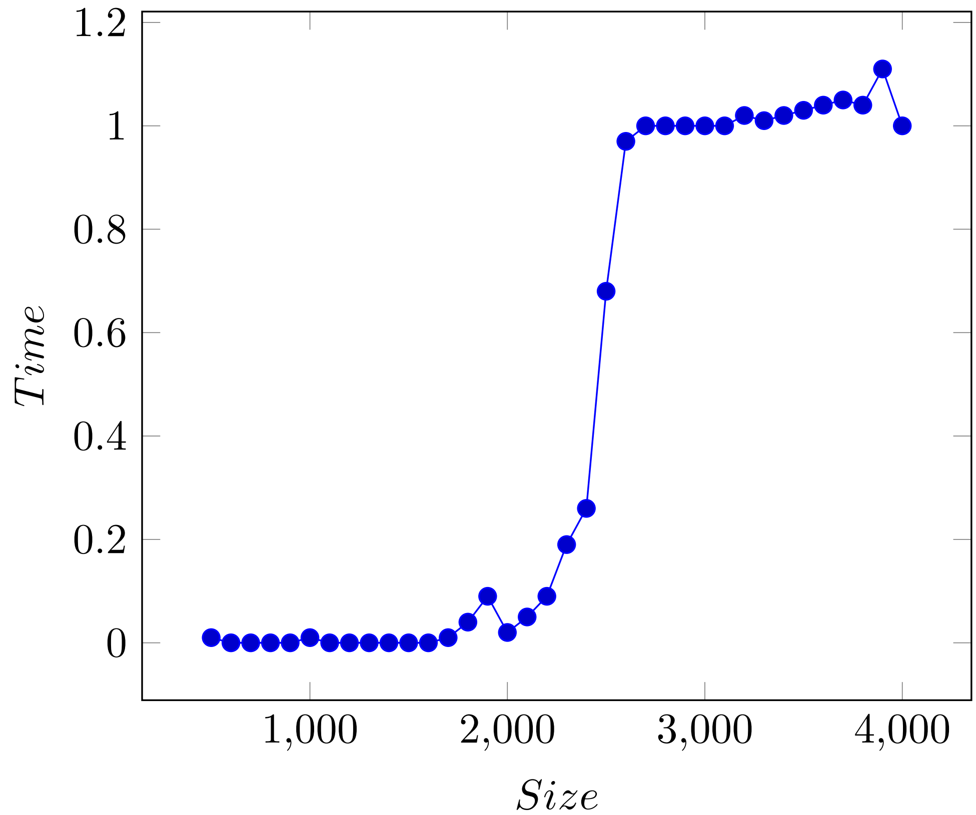
12 График №6 (время работы сортировки hybrid5 sort для почти отсортированных векторов)



13 Исходные данные для построение графика №6

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0) (1400, 0) (1500, 0) (1600, 0) (1700, 0.01) (1800, 0)
   (1900, 0.01) (2000, 0.01) (2100, 0.03) (2200, 0) (2300, 0.02) (2400,
    0) (2500, 0.1) (2600, 0.09) (2700, 0.14) (2800, 0.04) (2900, 0.26)
   (3000, 0.28) (3100, 0.31) (3200, 0.22) (3300, 0.25) (3400, 0.51)
   (3500, 0.58) (3600, 0.47) (3700, 0.8) (3800, 0.85) (3900, 0.98)
   (4000, 1)
```

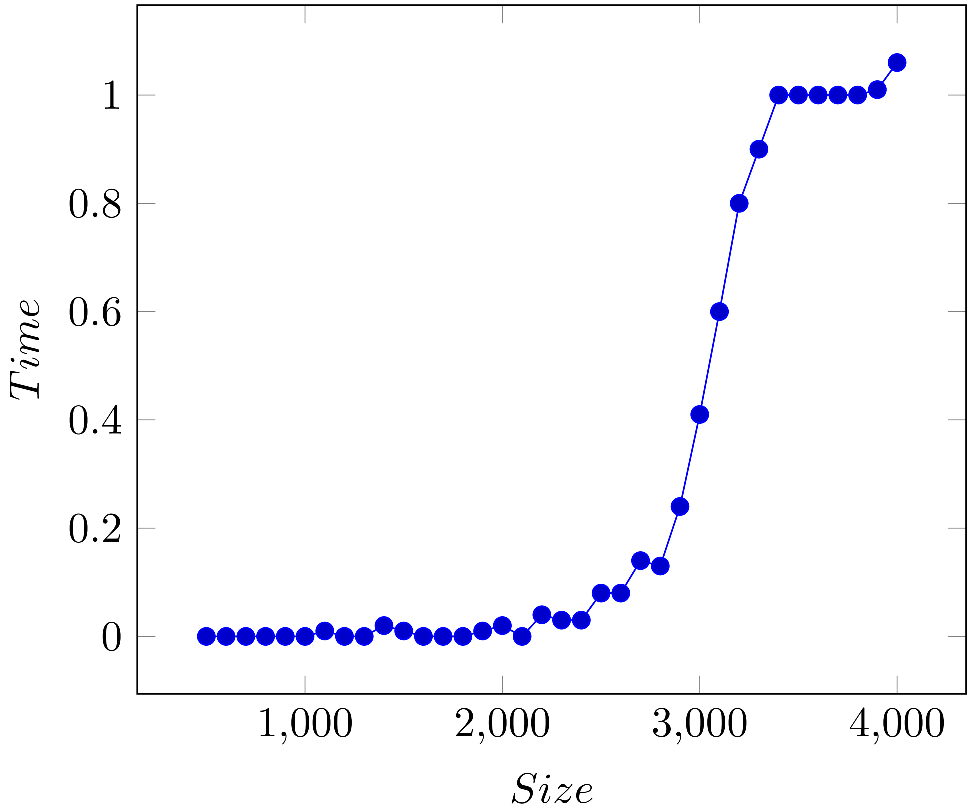
14 График №7 (время работы сортировки hybrid10 sort для рандомных векторов)



15 Исходные данные для построение графика №7

```
(500, 0.01) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0.01) (1100, 0)
    (1200, 0) (1300, 0) (1400, 0) (1500, 0) (1600, 0) (1700, 0.01)
    (1800, 0.04) (1900, 0.09) (2000, 0.02) (2100, 0.05) (2200, 0.09)
    (2300, 0.19) (2400, 0.26) (2500, 0.68) (2600, 0.97) (2700, 1) (2800,
     1) (2900, 1) (3000, 1) (3100, 1) (3200, 1.02) (3300, 1.01) (3400,
    1.02) (3500, 1.03) (3600, 1.04) (3700, 1.05) (3800, 1.04) (3900,
    1.11) (4000, 1)
```

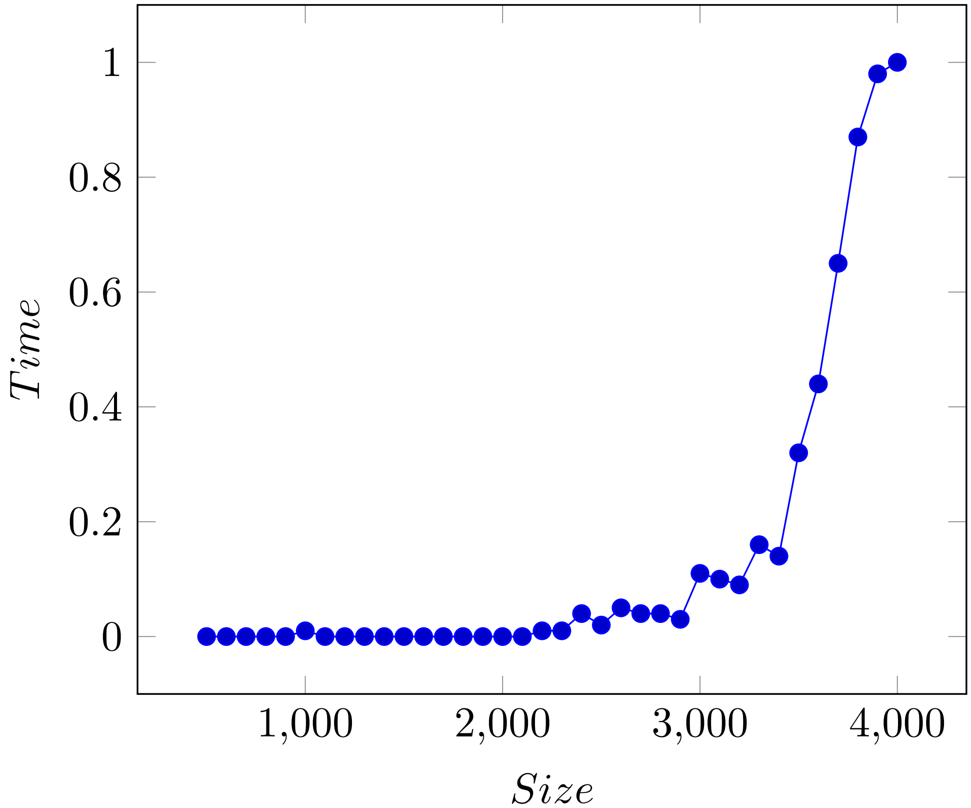
16 График №8 (время работы сортировки hybrid10 sort для векторов, отсортированных по невозрастанию)



17 Исходные данные для построение графика №8

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0.01)
    (1200, 0) (1300, 0) (1400, 0.02) (1500, 0.01) (1600, 0) (1700, 0)
    (1800, 0) (1900, 0.01) (2000, 0.02) (2100, 0) (2200, 0.04) (2300,
    0.03) (2400, 0.03) (2500, 0.08) (2600, 0.08) (2700, 0.14) (2800,
    0.13) (2900, 0.24) (3000, 0.41) (3100, 0.6) (3200, 0.8) (3300, 0.9)
    (3400, 1) (3500, 1) (3600, 1) (3700, 1) (3800, 1) (3900, 1.01)
    (4000, 1.06)
```

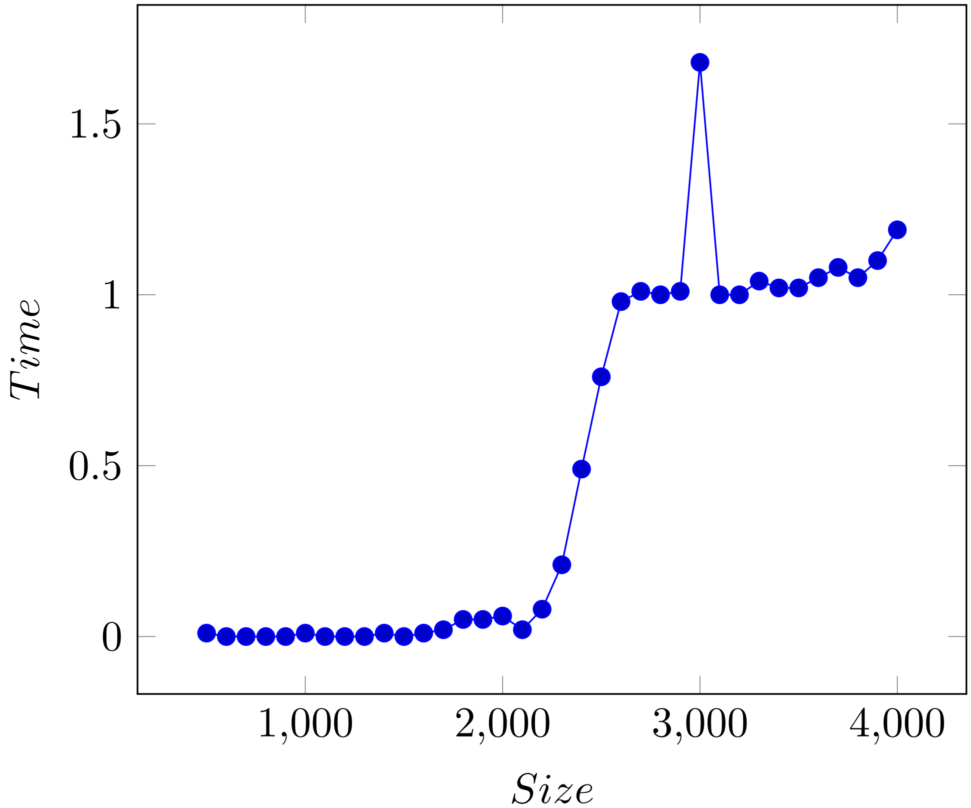
18 График №9 (время работы сортировки hybrid10 sort для почти отсортированных векторов)



19 Исходные данные для построение графика №9

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0.01) (1100, 0)
    (1200, 0) (1300, 0) (1400, 0) (1500, 0) (1600, 0) (1700, 0) (1800,
    0) (1900, 0) (2000, 0) (2100, 0) (2200, 0.01) (2300, 0.01) (2400,
    0.04) (2500, 0.02) (2600, 0.05) (2700, 0.04) (2800, 0.04) (2900,
    0.03) (3000, 0.11) (3100, 0.1) (3200, 0.09) (3300, 0.16) (3400,
    0.14) (3500, 0.32) (3600, 0.44) (3700, 0.65) (3800, 0.87) (3900,
    0.98) (4000, 1)
```

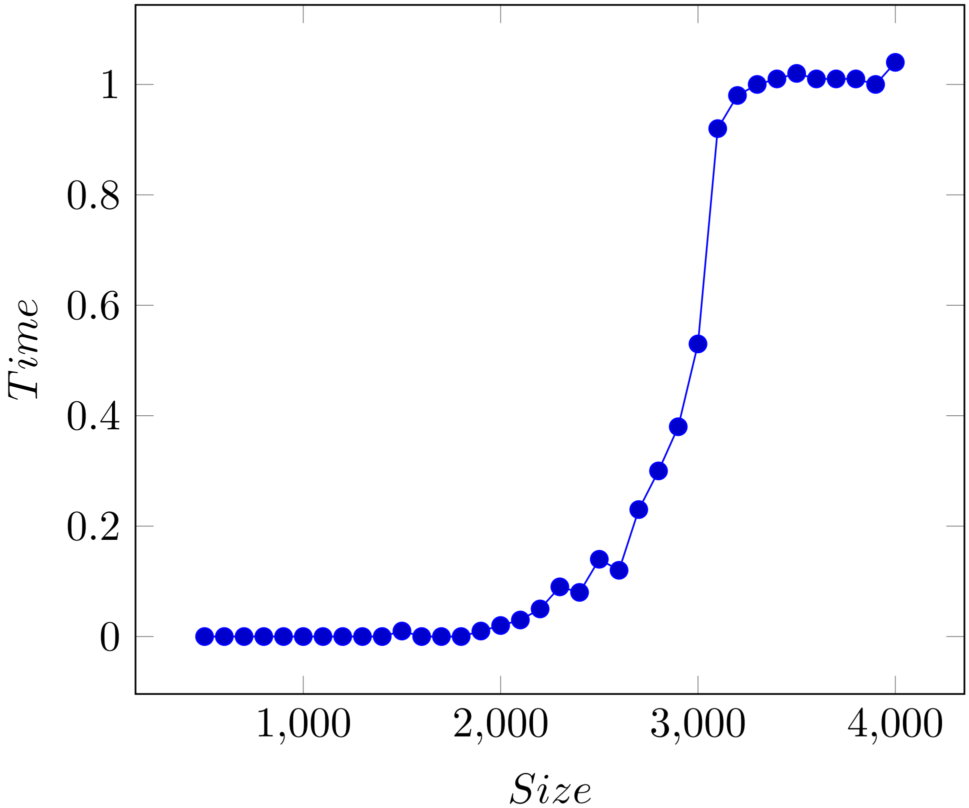
20 График №10 (время работы сортировки hybrid20 sort для рандомных векторов)



21 Исходные данные для построение графика №10

```
(500, 0.01) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0.01) (1100, 0)
    (1200, 0) (1300, 0) (1400, 0.01) (1500, 0) (1600, 0.01) (1700, 0.02)
    (1800, 0.05) (1900, 0.05) (2000, 0.06) (2100, 0.02) (2200, 0.08)
    (2300, 0.21) (2400, 0.49) (2500, 0.76) (2600, 0.98) (2700, 1.01)
    (2800, 1) (2900, 1.01) (3000, 1.68) (3100, 1) (3200, 1) (3300, 1.04)
    (3400, 1.02) (3500, 1.02) (3600, 1.05) (3700, 1.08) (3800, 1.05)
    (3900, 1.1) (4000, 1.19)
```

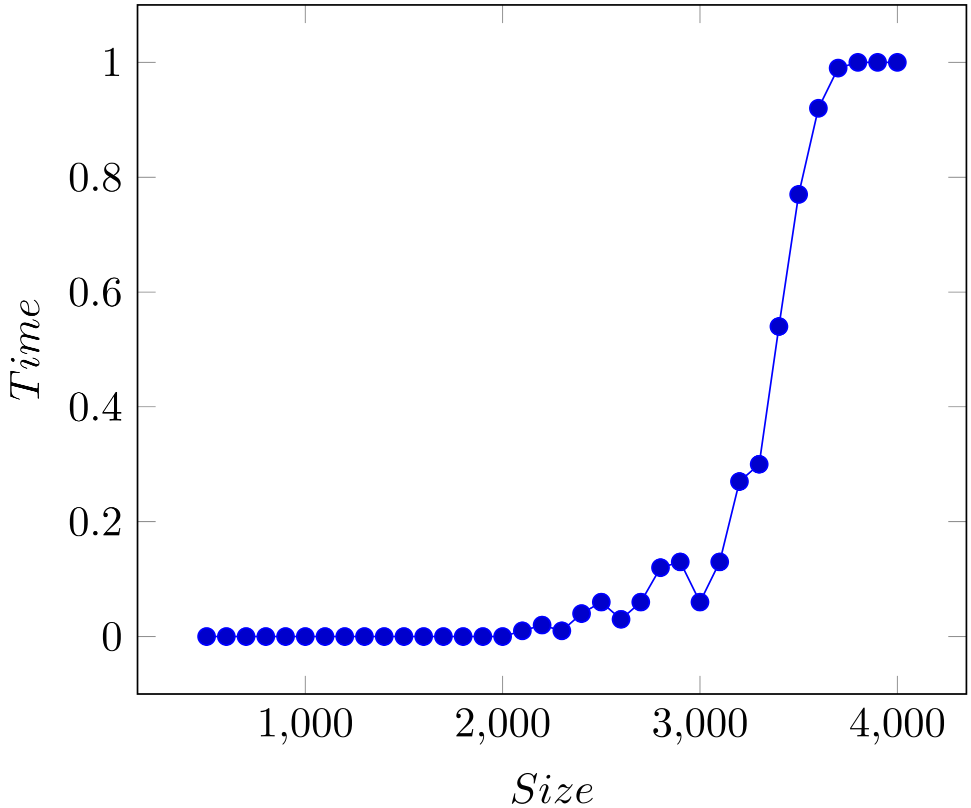
22 График №11 (время работы сортировки hybrid20 sort для векторов, отсортированных по невозрастанию)



23 Исходные данные для построение графика №11

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0) (1400, 0) (1500, 0.01) (1600, 0) (1700, 0) (1800, 0)
   (1900, 0.01) (2000, 0.02) (2100, 0.03) (2200, 0.05) (2300, 0.09)
   (2400, 0.08) (2500, 0.14) (2600, 0.12) (2700, 0.23) (2800, 0.3)
   (2900, 0.38) (3000, 0.53) (3100, 0.92) (3200, 0.98) (3300, 1) (3400,
    1.01) (3500, 1.02) (3600, 1.01) (3700, 1.01) (3800, 1.01) (3900, 1)
    (4000, 1.04)
```

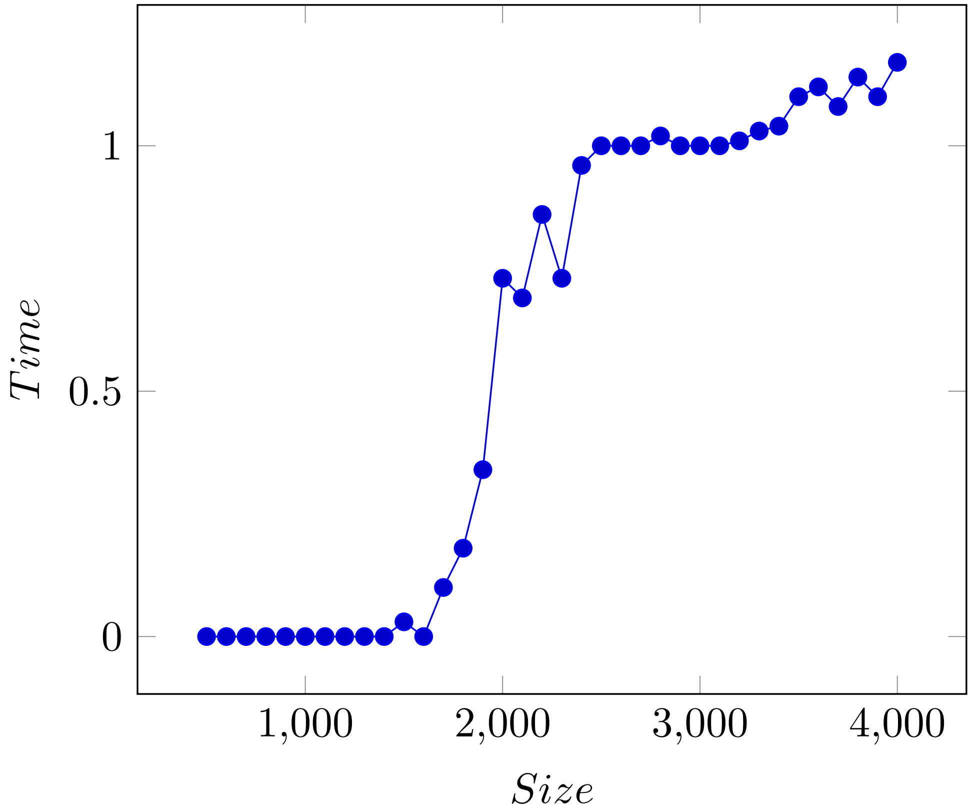
24 График №12 (время работы сортировки hybrid20 sort для почти отсортированных векторов)



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) (1400, 0) (1500, 0) (1600, 0) (1700, 0) (1800, 0)
   (1900, 0) (2000, 0) (2100, 0.01) (2200, 0.02) (2300, 0.01) (2400,
   0.04) (2500, 0.06) (2600, 0.03) (2700, 0.06) (2800, 0.12) (2900,
   0.13) (3000, 0.06) (3100, 0.13) (3200, 0.27) (3300, 0.3) (3400,
   0.54) (3500, 0.77) (3600, 0.92) (3700, 0.99) (3800, 1) (3900, 1)
   (4000, 1)
```

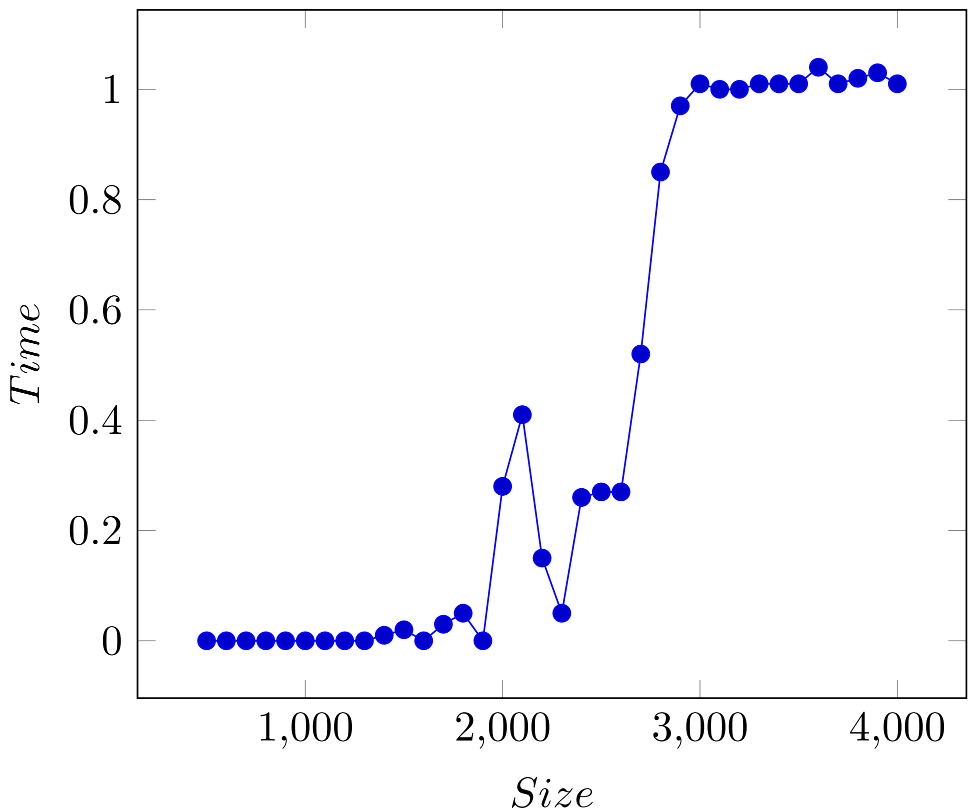
26 График №13 (время работы сортировки hybrid50 sort для рандомных векторов)



27 Исходные данные для построение графика №13

```
(500, 0) (600, 0) (700, 0) (800, 0) (900, 0) (1000, 0) (1100, 0) (1200,
    0) (1300, 0) (1400, 0) (1500, 0.03) (1600, 0) (1700, 0.1) (1800,
   0.18) (1900, 0.34) (2000, 0.73) (2100, 0.69) (2200, 0.86) (2300,
   0.73) (2400, 0.96) (2500, 1) (2600, 1) (2700, 1) (2800, 1.02) (2900,
    1) (3000, 1) (3100, 1) (3200, 1.01) (3300, 1.03) (3400, 1.04)
   (3500, 1.1) (3600, 1.12) (3700, 1.08) (3800, 1.14) (3900, 1.1)
   (4000, 1.17)
```

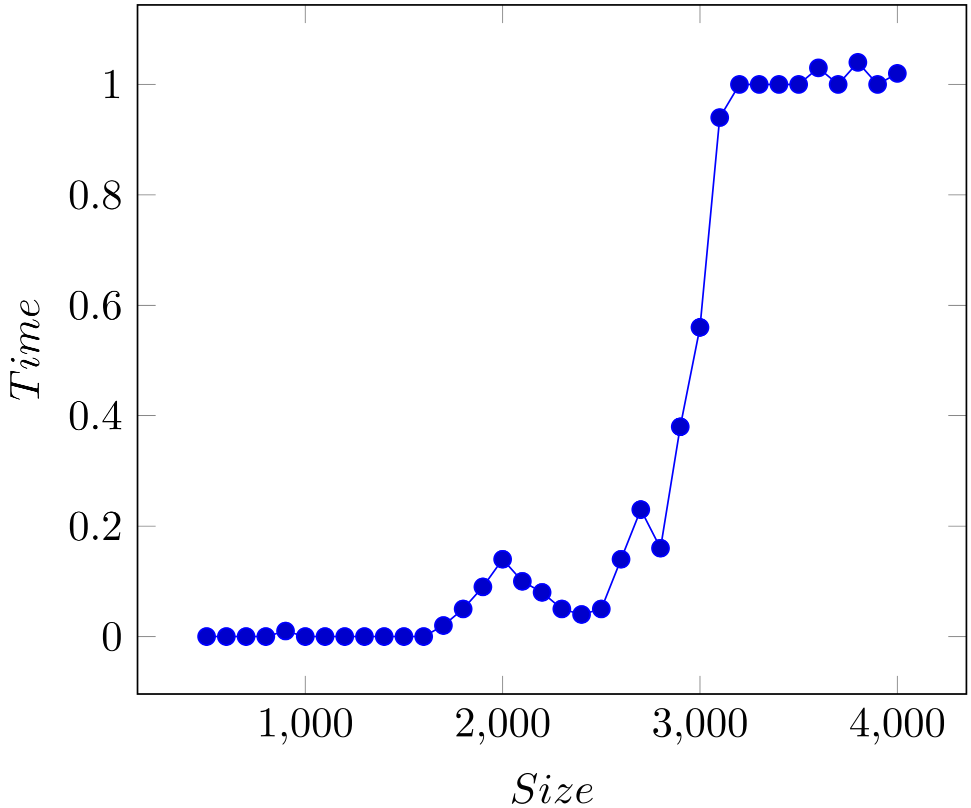
28 График №14 (время работы сортировки hybrid50 sort для векторов, отсортированных по невозрастанию)



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) (1400, 0.01) (1500, 0.02) (1600, 0) (1700, 0.03)
   (1800, 0.05) (1900, 0) (2000, 0.28) (2100, 0.41) (2200, 0.15) (2300,
    0.05) (2400, 0.26) (2500, 0.27) (2600, 0.27) (2700, 0.52) (2800,
   0.85) (2900, 0.97) (3000, 1.01) (3100, 1) (3200, 1) (3300, 1.01)
   (3400, 1.01) (3500, 1.01) (3600, 1.04) (3700, 1.01) (3800, 1.02)
   (3900, 1.03) (4000, 1.01)
```

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, 0.01) (1000, 0) (1100, 0)
   (1200, 0) (1300, 0) (1400, 0) (1500, 0) (1600, 0) (1700, 0.02)
   (1800, 0.05) (1900, 0.09) (2000, 0.14) (2100, 0.1) (2200, 0.08)
   (2300, 0.05) (2400, 0.04) (2500, 0.05) (2600, 0.14) (2700, 0.23)
   (2800, 0.16) (2900, 0.38) (3000, 0.56) (3100, 0.94) (3200, 1) (3300,
    1) (3400, 1) (3500, 1) (3600, 1.03) (3700, 1) (3800, 1.04) (3900,
   1) (4000, 1.02)
```

32 Выводы о проделанной работе

Хоть графиков и очень много, однако можно проследить, что в такой конфигурации сортировок прирост производительности уже наблюдается. Можно также заметить, что гибридная сортировка с параметром 10 ведет себя более стабильно, а также существенно быстрее на рандомных векторах (1мс в среднем против 1.15, 1.25, 1.19 и 1.17 для quick, hybrid5, hybrid20 и hybrid50 соответственно). Так что, выбирая из всех предложенных вариантов, стоит выбрать именно её

33 Доказательство того, что построение кучи это $O(n)$ по времени

Этот параграф не имеет непосредственного отношения к задаче A3, но на 7 лекции была предложена задача о доказательстве алгоритмической сложности build_heap за $O(n)$, а так как здесь мы использовали эту функцию, то имеет смысл привести решение здесь, в конце

файла
Итак, прежде всего заметим что самый длинный путь, который может пройти элемент кучи при восстановлении свойства max-кучи, равен высоте дерева. То есть это $log(n)$. Попробуем чуть улучшить эту оценку. Проиндексируем элементы кучи, начиная с 1, причем элемент в вершине имеет индекс 1, а дальше заполнение индексов происходит слева направо. Функция max_heapify принимает индекс элемента i, а также размер массива n. Тогда максимальный путь, который может пройти элемент с индексом i при восстановлении свойства max-кучи, равен длине пути от текущего элемента в дереве до его конца. Это длина равна высоте дерева минус высоте над выбранным элементом (то есть числом уровней выше текущего элемента). Высота дерева это $log(n)$. Высота над выбранным элементом это $\lfloor log(i) \rfloor$. Тогда максимальный путь, который может пройти элемент с индексом i составляет $log(n) - \lfloor log(i) \rfloor$. Метод build_heap восстанавливает свойства max-кучи для $n/2$ элементов, начиная с конца. То есть функция max_heapify вызывается $n/2$ раз для элементов с индексами 1,2,3,...,n/2. Запишем функцию временной сложности и оценим её сверху:

$$T(n) = \sum_{i=1}^{n/2} log(n) - \lfloor log(i) \rfloor < \sum_{i=1}^{n/2} log(n) - log(i) + 1 = \frac{n}{2} + \sum_{i=1}^{n/2} log(\frac{n}{i}) < \frac{n}{2} + \sum_{i=1}^{n} log(\frac{n}{i}) = \frac{n}{2} + log(\frac{n^n}{n!})$$

Далее воспользуемся разложением в ряд Тейлора для экспоненты:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \Rightarrow e^x > \frac{x^n}{n!} \Rightarrow e^n > \frac{n^n}{n!} \Rightarrow n^n < e^n n!$$

Тогда:

$$T(n) < \frac{n}{2} + log(\frac{n^n}{n!}) < \frac{n}{2} + log(\frac{e^n n!}{n!}) = \frac{n}{2} + log(e^n) = n(\frac{1}{2} + log(e)) = nlog(e\sqrt{2}) \Rightarrow \exists c : T(n) < cn(c = log(e\sqrt{2})) \Rightarrow$$

$$\Rightarrow T(n) = O(n), q.e.d.$$