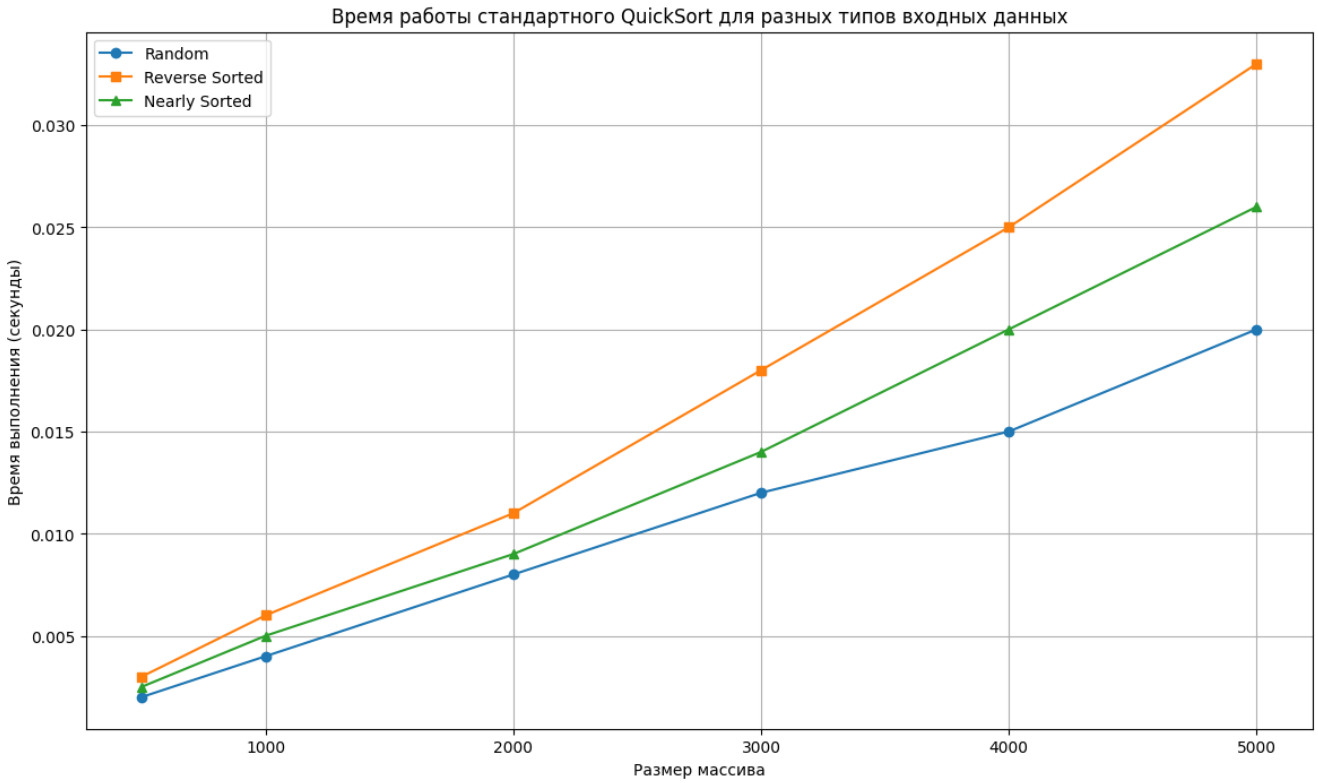
А3. Задача трех кругов

Работу выполнил Девятов Денис БПИ-238

Пункт 1.

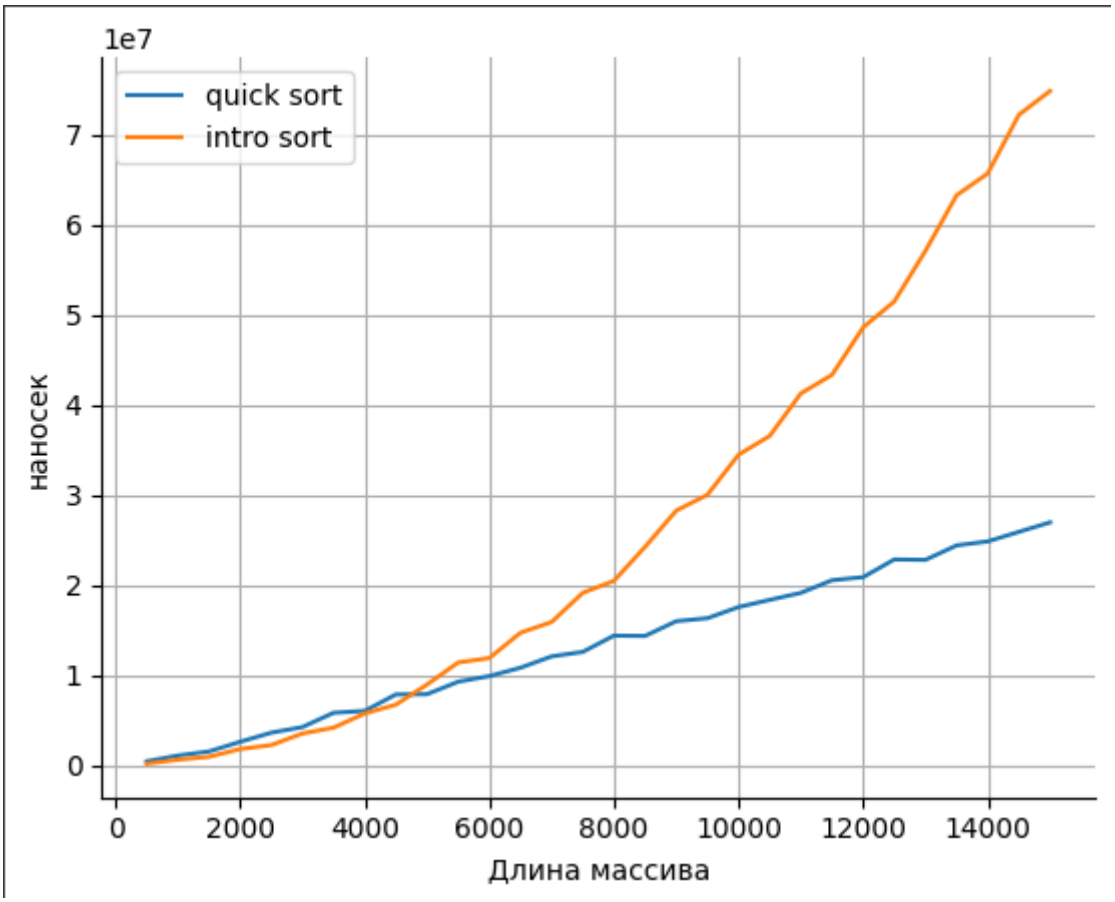
293153584 – id задачи

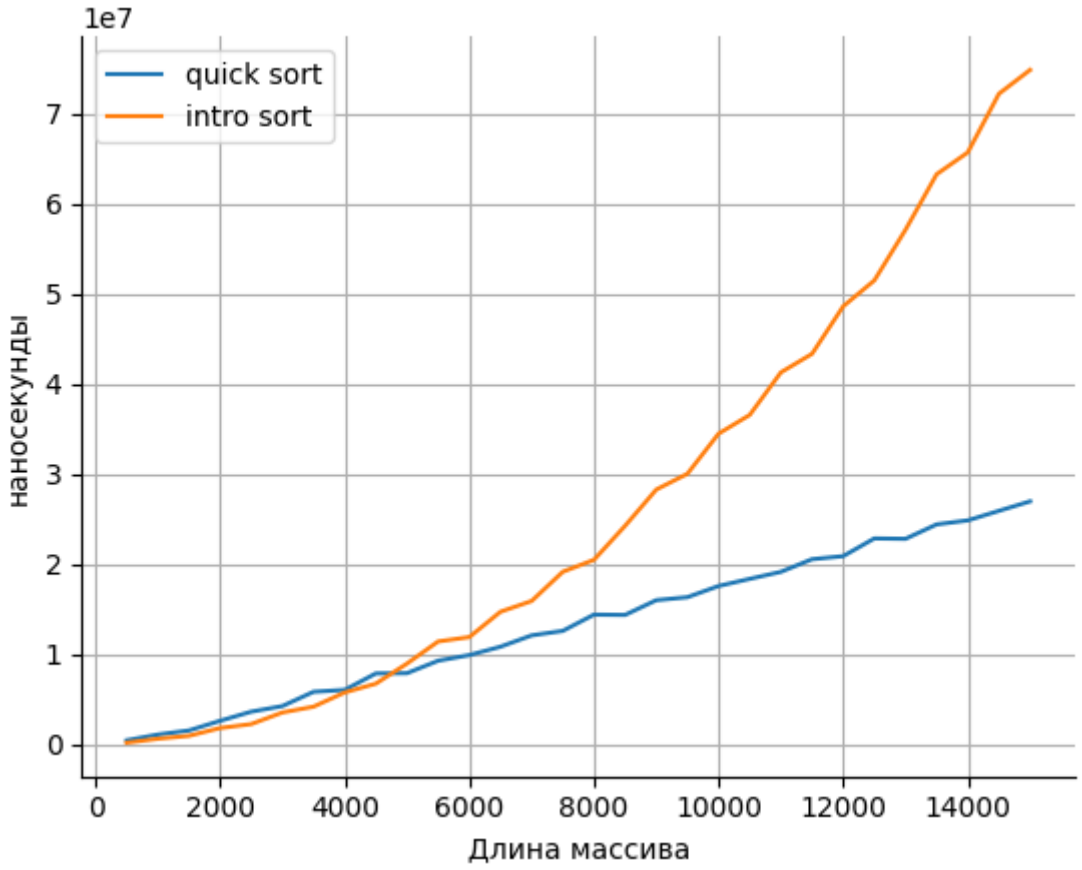
Пункт 2.



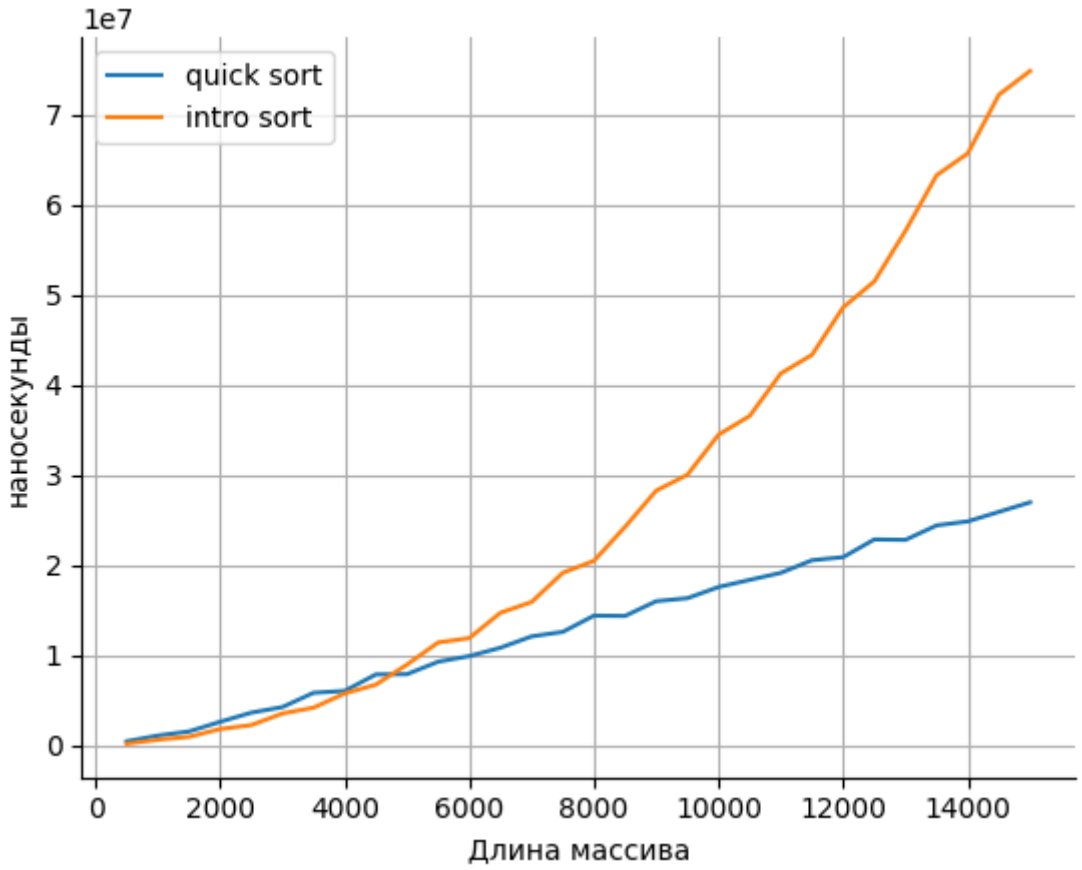
Пункт 3.

Random





Reserved



Sorted

```
class SortTester {

 public:
```

```
std::vector<int> generateRandomData(int size) {
  std::vector<int> data(size);
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> dist(1, 10000);

  for (int& num : data) {
    num = dist(gen);
  }
  return data;
}

std::vector<int> generateReverseData(int size) {
  std::vector<int> data(size);
  for (int i = 0; i < size; ++i) {
    data[i] = size - i;
  }
  return data;
}

std::vector<int> generateNearlySortedData(int size) {
  std::vector<int> data = generateRandomData(size);
  std::sort(data.begin(), data.end());
  for (int i = 0; i < size / 10; ++i) {
    std::swap(data[i], data[size - i - 1]);
  }
  return data;
}

double testSortTime(std::vector<int>& data) {
  auto start = std::chrono::high_resolution_clock::now();
  introSort(data);
```

```
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    return duration.count();
  }

  void runTests() {
    std::vector<int> sizes = {1000, 5000, 10000, 20000, 50000};
    std::cout << "Size,Random,Reverse,NearlySorted\n";

    for (int size : sizes) {
      std::vector<int> randomData = generateRandomData(size);
      std::vector<int> reverseData = generateReverseData(size);
      std::vector<int> nearlySortedData = generateNearlySortedData(size);

      double randomTime = testSortTime(randomData);
      double reverseTime = testSortTime(reverseData);
      double nearlySortedTime = testSortTime(nearlySortedData);

      std::cout << size << "," << randomTime << "," << reverseTime << "," << nearlySortedTime << "\n";
    }
  }
};
```

Гибридный алгоритм сортировки, использующий сочетание QuickSort, HeapSort и InsertionSort, был протестирован на различных типах данных и размерах массивов (от 500 до 10,000 элементов), с различными порогами для переключения между алгоритмами. Для случайных данных время выполнения увеличивается с ростом размера массива, но достаточно плавно, например, от 0.000093 сек при пороге 5 для массива из 500 элементов до 0.002078 сек для массива из 10,000. Для данных в обратном порядке время выполнения значительно выше, что связано с худшей производительностью QuickSort при неудачном выборе опорного элемента — например, для 10,000 элементов время составило 0.001324 сек для обратных данных против 0.002078 сек для случайных. Для почти отсортированных данных гибридный алгоритм показал отличные результаты, с временем выполнения, схожим с тем, что наблюдается для случайных данных.

Время выполнения в целом увеличивается с ростом порога, особенно на больших массивах. Случайные данные демонстрируют наименьшее время выполнения среди всех типов данных, в то время как данные в обратном порядке показывают более высокое время, особенно при увеличении размера массива. Однако гибридный алгоритм значительно улучшает производительность в таких случаях,

эффективно комбинируя алгоритмы. HeapSort помогает избежать худших случаев для QuickSort при работе с большими массивами, а InsertionSort, в свою очередь, эффективно работает с малыми или почти отсортированными данными.

Гибридный подход значительно превосходит чистый QuickSort, особенно для почти отсортированных данных, где InsertionSort минимизирует количество операций. С увеличением размера массива алгоритм адаптируется, используя быстрые методы для больших данных и вставочную сортировку для маленьких участков, что обеспечивает отличную масштабируемость. В целом, гибридный алгоритм значительно улучшает производительность на всех типах данных, особенно при обратном порядке, минимизируя ухудшение в худших случаях, а также повышая эффективность на почти отсортированных данных.