Реализация алгоритма Merge + Insertion Sort

ссылка на репозиторий:
https://github.com/AnnaAstashkina/A2_set3

id посылки: 292895601

код:

```
#include <iostream>
#include <vector>

void insertionSort(std::vector<int>& vect, int start, int end) {
  for (int i = start + 1; i <= end; ++i) {
    int key = vect[i];
    int j = i - 1;
    while (j >= start && vect[j] > key) {
      vect[j + 1] = vect[j];
      j--;
    }
    vect[j + 1] = key;
  }
}

void merge(std::vector<int>& vect, int start, int middle, int end) {
  int left_half_size = middle - start + 1;
  int right_half_size = end - middle;
  std::vector<int> left_half(left_half_size);
  std::vector<int> right_half(right_half_size);
  int i = 0;
  int j = 0;
  for (i = 0; i < left_half_size; ++i) {
    left_half[i] = vect[start + i];
  }
  for (i = 0; i < right_half_size; ++i) {
    right_half[i] = vect[middle + 1 + i];
  }
  i = 0;
  int curr = start;
  while (i < left_half_size && j < right_half_size) {
```

```
    if (left_half[i] <= right_half[j]) {
      vect[curr] = left_half[i];
      i++;
      curr++;
    } else {
      vect[curr] = right_half[j];
      j++;
      curr++;
    }
  }
  for (i; i < left_half_size; ++i) {
    vect[curr] = left_half[i];
    curr++;
  }
  for (j; j < right_half_size; ++j) {
    vect[curr] = right_half[j];
    curr++;
  }
}

void mergeSort(std::vector<int>& vect, int start, int end) {
  if (end - start + 1 <= 15) {
    insertionSort(vect, start, end);
    return;
  }
  if (start >= end) {
    return;
  }
  int middle = start + (end - start) / 2;
  mergeSort(vect, start, middle);
  mergeSort(vect, middle + 1, end);
  merge(vect, start, middle, end);
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n = 0;
  std::cin >> n;
  std::vector<int> vect(n);
  for (int i = 0; i < n; ++i) {
    std::cin >> vect[i];
  }
```

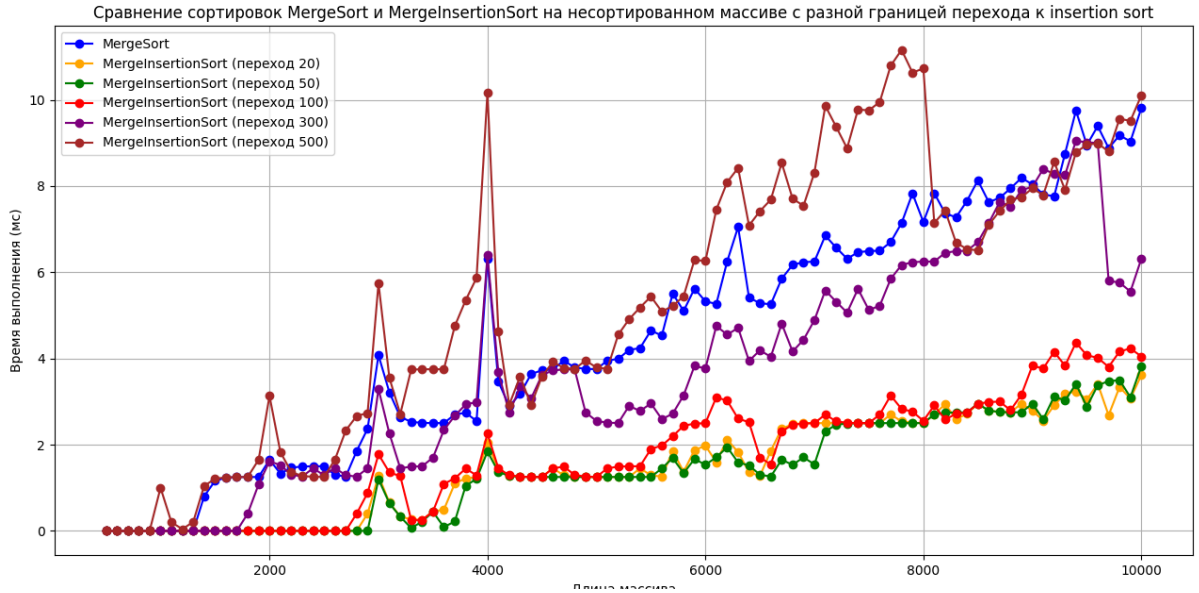
```
  mergeSort(vect, 0, n - 1);
  for (int i = 0; i < n; ++i) {
    std::cout << vect[i] << " ";
  }
  return 0;
}
```

Графики и анализ

Графики для удобства анализа построены на одной картинке!

Графики времени выполнения (в мс) MergeSort и MergeInsertionSort для разных границ перехода к InsertionSort для несортированного массива

Выводы о границе перехода будут сделаны на основе этого графика т.к. далее идут не случайные массивы => выводы лучше всего делать на основе данных графиков.

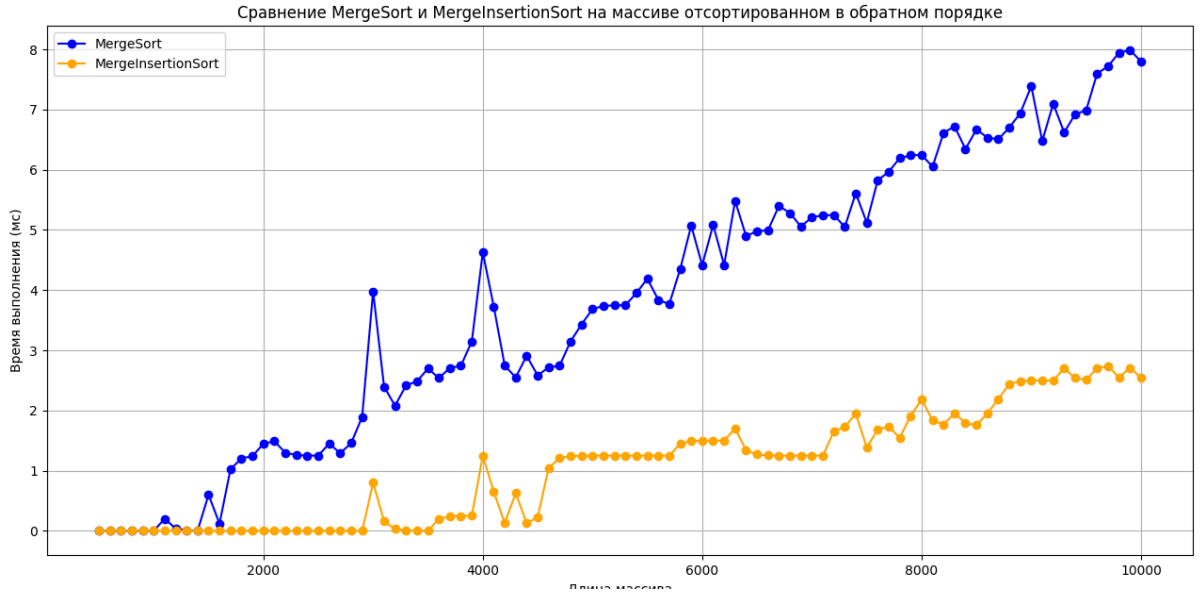


Вывод на несортированном массиве

Из графиков видно, что Merge + Insertion Sort в общем работает быстрее, чем Merge Sort, т.к. для анализа используются массивы, заполненные случайными числами в случайном порядке. Особенно это видно на больших данных, т.к. на маленьких размерах Merge Sort практически не вносит свой вклад и почти сразу переходит к Insertion Sort. **Кроме того несложно отследить примерную хорошую границу перехода к Insertion Sort: длина сортируемой части должна быть примерно 20-50 (Далее в замерах используется граница 20)**. При таких данных алгоритм Merge + Insertion Sort показывает себя с наилучшей стороны. Также можно

увидеть, что при переходе ~300 комбинированный алгоритм работает примерно с той же скоростью, как и обычный Merge Sort => переход не имеет смысла, а при переходе >=500 комбинированный алгоритм начинает заметно проигрывать в скорости MergeSort.

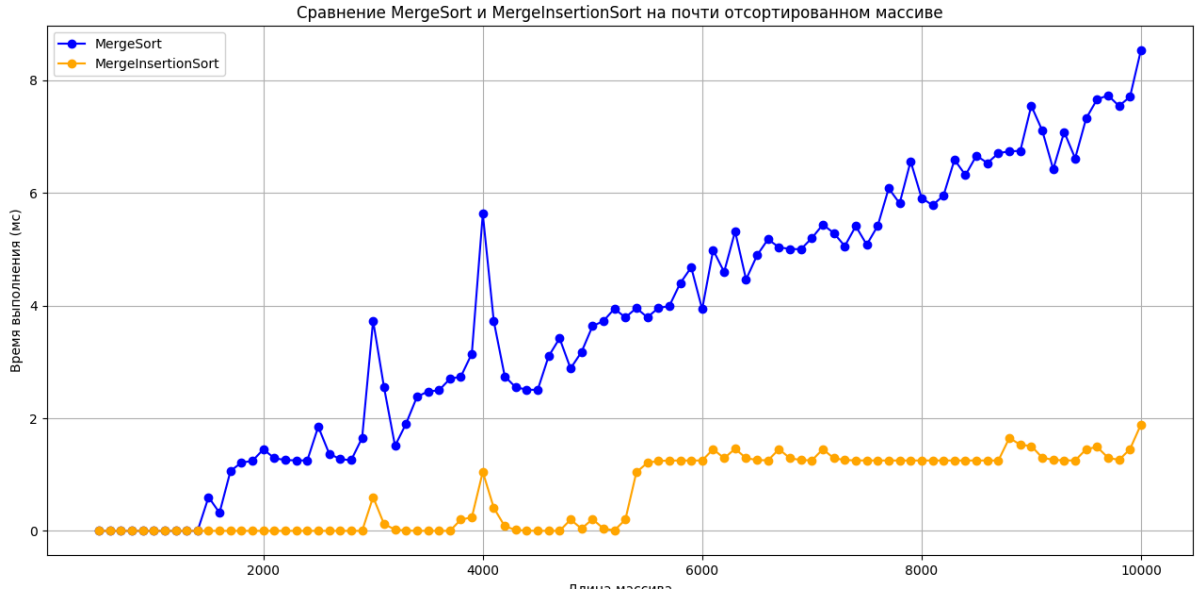
Графики времени выполнения (в мс) MergeSort и MergeInsertionSort для разных границ перехода к InsertionSort для массива сортированного в обратном порядке



Вывод на обратно-сортированном массиве

Из графиков видно, что на массиве, отсортированном в обратном порядке комбинированный алгоритм с переходом 20 (вычислено из предыдущего пункта) показывает себя значительно лучше. чем обычный MergeSort. Выводы о влиянии массива между одним и тем же алгоритмом будут сделаны на последующих графиках.

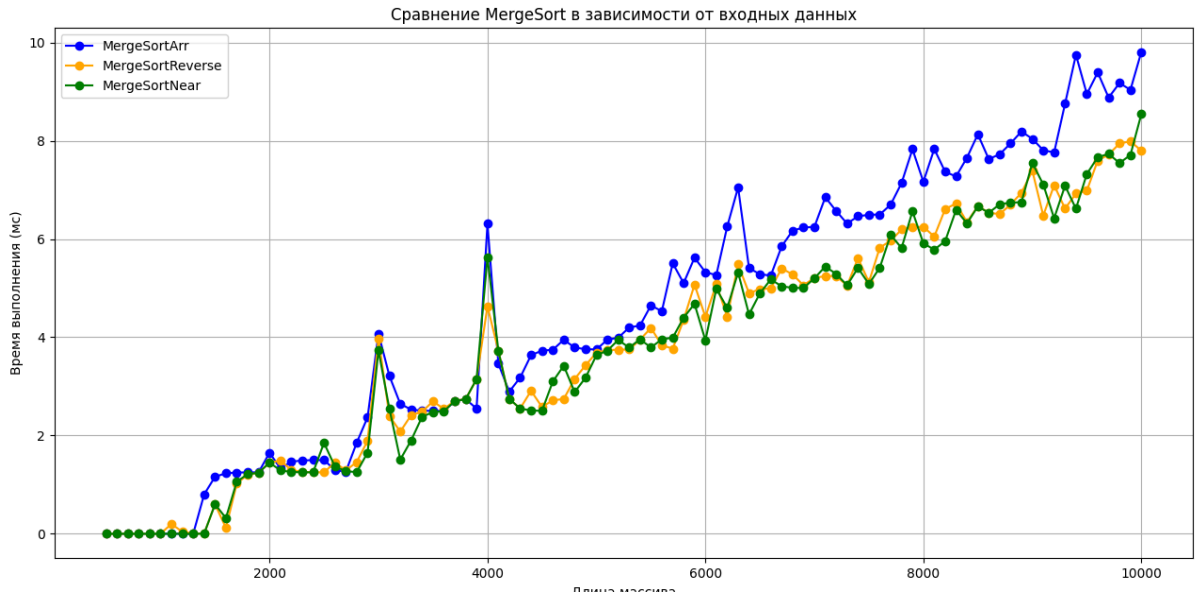
Графики времени выполнения (в мс) MergeSort и MergeInsertionSort для разных границ перехода к InsertionSort для частично сортированного массива



Вывод на частично-сортированном

Из графиков видно, что на частично-сортированном массиве комбинированный алгоритм с переходом 20 (вычислено из предыдущего пункта) показывает себя значительно лучше. чем обычный MergeSort. Причем нетрудно заметить, что то, что InsertionSort быстро работает на частично отсортированных массивах хорошо видно, т.к. время выполнение комбинированного алгоритма растет не так интенсивно, как MergeSort. Выводы о влиянии массива между одним и тем же алгоритмом будут сделаны на последующих графиках.

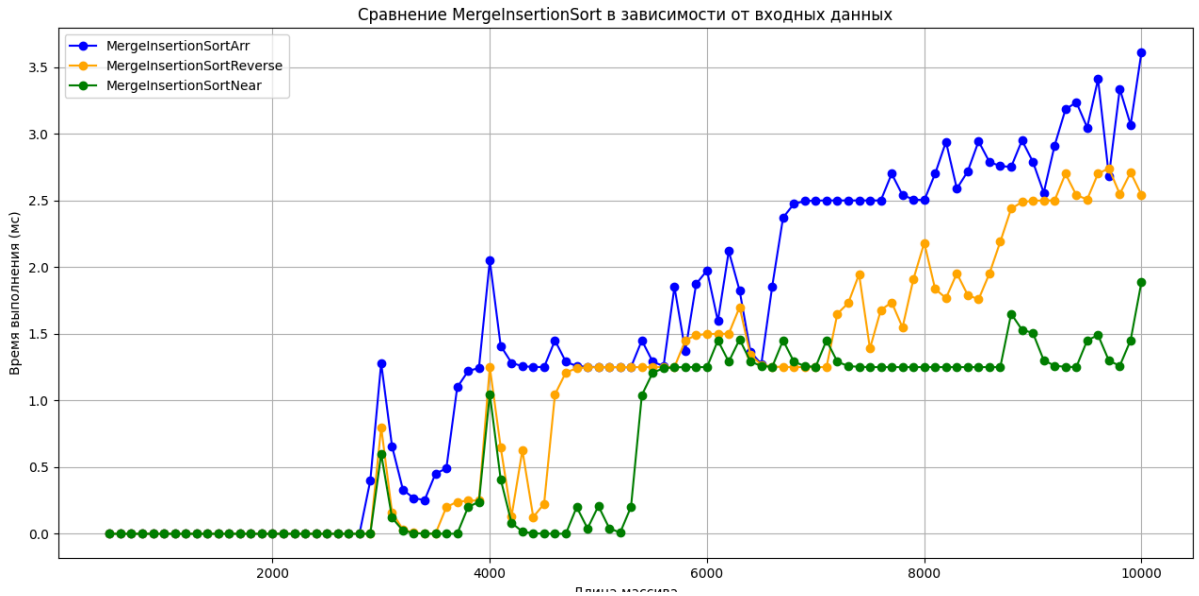
График времени выполнения MergeSort в зависимости от разных входных данных



Вывод для MergeSort

Из графиков видно, что полностью несортированный массив требует больше времени для выполнения сортировки, в то время как отсортированный в обратном порядке и почти-сортированный массив занимают примерно одинаковое количество времени, однако данная разница незначительна на небольших массивах. Это можно объяснить тем, что при сдвиге указателей в merge указатели в отсортированных массивах двигаются более “плавно” (логично) нежели при полностью несортированном массиве.

График времени выполнения для MergeInsertionSort на разных входных данных



Вывод для MergeInsertionSort

Из графика видно, что во времени работы есть видимые изменения. **Самое долгое время работы на несортированном массиве, быстрее работает на массиве, сортированном в обратном порядке и самое быстрое выполнение на частично сортированном массиве.** Это объясняется тем, что InsertionSort медленнее всего работает на массиве, сортированном в обратном порядке, т.к. надо сделать максимальное количество сравнений и перемещений, однако это компенсируется тем, что MergeSort работает быстрее на таких массивах, чем на обычных (прошлый график), а при больших входных данных основное время работы алгоритма выполняется MergeSort, кроме того на обычных входных данных MergeSort работает дольше, чем на сортированных в обратном порядке, InsertionSort работает не сильно быстрее, чем на сортированном в обратном порядке массиве, и на больших данных основное время работы - MergeSort. Поэтому на несортированных входных данных комбинированная сортировка работает дольше, чем на данных. сортированных в обратном порядке. На частично сортированных данных MergeSort работает примерно за такое же время как и на сортированных в обратном порядке данных, однако InsertionSort работает ощутимо быстрее, т.к. количество сравнений и перемещений при сортировке минимальное, поэтому комбинированный алгоритм в общем работает быстрее на частично сортированных входных данных, нежели на остальных случаях.

Классы ArrayGenerator и SortTest

ArrayGenerator

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>

class ArrayGenerator {
 public:
  static std::vector<int> generateRandomArray() {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(0, 6000);
    std::vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
      arr[i] = distr(generator);
    }
    return arr;
  }

  static std::vector<int> generateReverseSortedArray() {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(0, 6000);
    std::vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
      arr[i] = distr(generator);
    }
    std::sort(arr.begin(), arr.end());
    std::reverse(arr.begin(), arr.end());
    return arr;
  }

  static std::vector<int> generateNearlySortedArray() {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(0, 6000);
    std::vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
      arr[i] = distr(generator);
```

```
    }
    std::sort(arr.begin(), arr.end());
    for (int i = 0; i < 15; ++i) {
      std::uniform_int_distribution<> distrs(0, 500);
      int idx1 = distrs(generator);
      int idx2 = distrs(generator);
      std::swap(arr[idx1], arr[idx2]);
    }
    for (int i = 500; i <= 9900; i += 100) {
      std::uniform_int_distribution<> distrs(i, i + 100);
      for (int j = 0; j < 3; ++j) {
        int idx1 = distrs(generator);
        int idx2 = distrs(generator);
        std::swap(arr[idx1], arr[idx2]);
      }
    }
    return arr;
  }
};
```

SortTester

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>

class SortTester {
 private:
  static void insertionSort(std::vector<int>& vect, int start, int end)
{
    for (int i = start + 1; i <= end; ++i) {
      int key = vect[i];
      int j = i - 1;
      while (j >= start && vect[j] > key) {
        vect[j + 1] = vect[j];
        j--;
      }
      vect[j + 1] = key;
    }
  }
```

```
  static void merge(std::vector<int>& vect, int start, int middle, int
end) {
    int left_half_size = middle - start + 1;
    int right_half_size = end - middle;
    std::vector<int> left_half(left_half_size);
    std::vector<int> right_half(right_half_size);
    int i = 0;
    int j = 0;
    for (i = 0; i < left_half_size; ++i) {
      left_half[i] = vect[start + i];
    }
    for (i = 0; i < right_half_size; ++i) {
      right_half[i] = vect[middle + 1 + i];
    }
    i = 0;
    int curr = start;
    while (i < left_half_size && j < right_half_size) {
      if (left_half[i] <= right_half[j]) {
        vect[curr] = left_half[i];
        i++;
        curr++;
      } else {
        vect[curr] = right_half[j];
        j++;
        curr++;
      }
    }
    for (i; i < left_half_size; ++i) {
      vect[curr] = left_half[i];
      curr++;
    }
    for (j; j < right_half_size; ++j) {
      vect[curr] = right_half[j];
      curr++;
    }
  }

  static void mergeInsertionSort(std::vector<int>& vect, int start, int
end, int threshold) {
    if (end - start + 1 <= threshold) {
      insertionSort(vect, start, end);
      return;
    }
```

```
    if (start >= end) {
      return;
    }
    int middle = start + (end - start) / 2;
    mergeInsertionSort(vect, start, middle, threshold);
    mergeInsertionSort(vect, middle + 1, end, threshold);
    merge(vect, start, middle, end);
  }

  static void mergeSort(std::vector<int>& vect, int start, int end) {
    if (start >= end) {
      return;
    }
    int middle = start + (end - start) / 2;
    mergeSort(vect, start, middle);
    mergeSort(vect, middle + 1, end);
    merge(vect, start, middle, end);
  }

 public:
  static long long testMergeSort(std::vector<int>& arr) {
    auto start = std::chrono::high_resolution_clock::now();
    mergeSort(arr, 0, arr.size() - 1);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    return
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
  }

  static long long testMergeInsertionSort(std::vector<int>& arr, int
threshold) {
    auto start = std::chrono::high_resolution_clock::now();
    mergeInsertionSort(arr, 0, arr.size() - 1, threshold);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    return
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
  }
};
```

Общий код для получения результатов

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
#include <random>
#include <chrono>

class ArrayGenerator {
 public:
  static std::vector<int> generateRandomArray() {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(0, 6000);
    std::vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
      arr[i] = distr(generator);
    }
    return arr;
  }

  static std::vector<int> generateReverseSortedArray() {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(0, 6000);
    std::vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
      arr[i] = distr(generator);
    }
    std::sort(arr.begin(), arr.end());
    std::reverse(arr.begin(), arr.end());
    return arr;
  }

  static std::vector<int> generateNearlySortedArray() {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(0, 6000);
    std::vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
      arr[i] = distr(generator);
    }
    std::sort(arr.begin(), arr.end());
    for (int i = 0; i < 15; ++i) {
      std::uniform_int_distribution<> distrs(0, 500);
      int idx1 = distrs(generator);
      int idx2 = distrs(generator);
```

```
      std::swap(arr[idx1], arr[idx2]);
    }
    for (int i = 500; i <= 9900; i += 100) {
      std::uniform_int_distribution<> distrs(i, i + 100);
      for (int j = 0; j < 3; ++j) {
        int idx1 = distrs(generator);
        int idx2 = distrs(generator);
        std::swap(arr[idx1], arr[idx2]);
      }
    }
    return arr;
  }
};

class SortTester {
 private:
  static void insertionSort(std::vector<int>& vect, int start, int end)
{
    for (int i = start + 1; i <= end; ++i) {
      int key = vect[i];
      int j = i - 1;
      while (j >= start && vect[j] > key) {
        vect[j + 1] = vect[j];
        j--;
      }
      vect[j + 1] = key;
    }
  }

  static void merge(std::vector<int>& vect, int start, int middle, int
end) {
    int left_half_size = middle - start + 1;
    int right_half_size = end - middle;
    std::vector<int> left_half(left_half_size);
    std::vector<int> right_half(right_half_size);
    int i = 0;
    int j = 0;
    for (i = 0; i < left_half_size; ++i) {
      left_half[i] = vect[start + i];
    }
    for (i = 0; i < right_half_size; ++i) {
      right_half[i] = vect[middle + 1 + i];
    }
```

```
    i = 0;
    int curr = start;
    while (i < left_half_size && j < right_half_size) {
      if (left_half[i] <= right_half[j]) {
        vect[curr] = left_half[i];
        i++;
        curr++;
      } else {
        vect[curr] = right_half[j];
        j++;
        curr++;
      }
    }
    for (i; i < left_half_size; ++i) {
      vect[curr] = left_half[i];
      curr++;
    }
    for (j; j < right_half_size; ++j) {
      vect[curr] = right_half[j];
      curr++;
    }
  }

  static void mergeInsertionSort(std::vector<int>& vect, int start, int
end, int threshold) {
    if (end - start + 1 <= threshold) {
      insertionSort(vect, start, end);
      return;
    }
    if (start >= end) {
      return;
    }
    int middle = start + (end - start) / 2;
    mergeInsertionSort(vect, start, middle, threshold);
    mergeInsertionSort(vect, middle + 1, end, threshold);
    merge(vect, start, middle, end);
  }

  static void mergeSort(std::vector<int>& vect, int start, int end) {
    if (start >= end) {
      return;
    }
    int middle = start + (end - start) / 2;
```

```
    mergeSort(vect, start, middle);
    mergeSort(vect, middle + 1, end);
    merge(vect, start, middle, end);
  }

 public:
  static long long testMergeSort(std::vector<int>& arr) {
    auto start = std::chrono::high_resolution_clock::now();
    mergeSort(arr, 0, arr.size() - 1);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    return
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
  }

  static long long testMergeInsertionSort(std::vector<int>& arr, int
threshold) {
    auto start = std::chrono::high_resolution_clock::now();
    mergeInsertionSort(arr, 0, arr.size() - 1, threshold);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    return
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
  }
};

template <typename T>
void printVector(const std::vector<T>& vec) {
  for (int i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << ", ";
  }
  std::cout << std::endl << std::endl;
}

int main() {
  std::vector<int> randomArrParent =
ArrayGenerator::generateRandomArray();
  std::vector<int> reverseArrParent =
ArrayGenerator::generateReverseSortedArray();
  std::vector<int> nearlySortedArrParent =
ArrayGenerator::generateNearlySortedArray();
  double timeMergeArr;
  double timeMergeReverse;
  double timeInsertionMergeReverse;
  double timeMergeNear;
```

```
  double timeInsertionMergeNear;
  double timek10;
  double timek20;
  double timek30;
  double timek40;
  double timek50;
  std::vector<double> vecTimeMergeArr;
  std::vector<double> vecTimeMergeReverse;
  std::vector<double> vecTimeInsertionMergeReverse;
  std::vector<double> vecTimeMergeNear;
  std::vector<double> vecTimeInsertionMergeNear;
  std::vector<double> veck10;
  std::vector<double> veck20;
  std::vector<double> veck30;
  std::vector<double> veck40;
  std::vector<double> veck50;
  std::vector<int> is;
  for (int i = 500; i <= 10000; i += 100) {
    is.push_back(i);
    for (int j = 0; j < 5; ++j) {
      std::vector<int> randomArr =
std::vector<int>(randomArrParent.begin(), randomArrParent.begin() + i);
      timeMergeArr += SortTester::testMergeSort(randomArr);
      randomArr = std::vector<int>(randomArrParent.begin(),
randomArrParent.begin() + i);
      timek10 += SortTester::testMergeInsertionSort(randomArr, 20);
      randomArr = std::vector<int>(randomArrParent.begin(),
randomArrParent.begin() + i);
      timek20 += SortTester::testMergeInsertionSort(randomArr, 50);
      randomArr = std::vector<int>(randomArrParent.begin(),
randomArrParent.begin() + i);
      timek30 += SortTester::testMergeInsertionSort(randomArr, 100);
      randomArr = std::vector<int>(randomArrParent.begin(),
randomArrParent.begin() + i);
      timek40 += SortTester::testMergeInsertionSort(randomArr, 300);
      randomArr = std::vector<int>(randomArrParent.begin(),
randomArrParent.begin() + i);
      timek50 += SortTester::testMergeInsertionSort(randomArr, 500);

      std::vector<int> reverseArr =
std::vector<int>(reverseArrParent.begin(), reverseArrParent.begin() +
i);
      timeMergeReverse += SortTester::testMergeSort(reverseArr);
```

```
      reverseArr = std::vector<int>(reverseArrParent.begin(),
reverseArrParent.begin() + i);
      timeInsertionMergeReverse +=
SortTester::testMergeInsertionSort(reverseArr, 20);

      std::vector<int> nearlySortedArr =
          std::vector<int>(nearlySortedArrParent.begin(),
nearlySortedArrParent.begin() + i);
      timeMergeNear += SortTester::testMergeSort(nearlySortedArr);
      nearlySortedArr = std::vector<int>(nearlySortedArrParent.begin(),
nearlySortedArrParent.begin() + i);
      timeInsertionMergeNear +=
SortTester::testMergeInsertionSort(nearlySortedArr, 20);
    }
    timeMergeArr /= 5.;
    timeMergeReverse /= 5.;
    timeInsertionMergeReverse /= 5.;
    timeMergeNear /= 5.;
    timeInsertionMergeNear /= 5.;
    timek10/=5.;
    timek20/=5.;
    timek30/=5.;
    timek40/=5.;
    timek50/=5.;


    vecTimeMergeArr.push_back(timeMergeArr);
    vecTimeMergeReverse.push_back(timeMergeReverse);
    vecTimeInsertionMergeReverse.push_back(timeInsertionMergeReverse);
    vecTimeMergeNear.push_back(timeMergeNear);
    vecTimeInsertionMergeNear.push_back(timeInsertionMergeNear);
    veck10.push_back(timek10);
    veck20.push_back(timek20);
    veck30.push_back(timek30);
    veck40.push_back(timek40);
    veck50.push_back(timek50);
  }
  printVector(is);
  printVector(vecTimeMergeArr);
  printVector(veck10);
  printVector(veck20);
  printVector(veck30);
  printVector(veck40);
```

```
  printVector(veck50);
  printVector(vecTimeMergeReverse);
  printVector(vecTimeInsertionMergeReverse);
  printVector(vecTimeMergeNear);
  printVector(vecTimeInsertionMergeNear);
}
```

Результаты, на основе которых строились графики

size: 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400, 2500, 2600, 2700, 2800, 2900, 3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000, 4100, 4200, 4300, 4400, 4500, 4600, 4700, 4800, 4900, 5000, 5100, 5200, 5300, 5400, 5500, 5600, 5700, 5800, 5900, 6000, 6100, 6200, 6300, 6400, 6500, 6600, 6700, 6800, 6900, 7000, 7100, 7200, 7300, 7400, 7500, 7600, 7700, 7800, 7900, 8000, 8100, 8200, 8300, 8400, 8500, 8600, 8700, 8800, 8900, 9000, 9100, 9200, 9300, 9400, 9500, 9600, 9700, 9800, 9900, 10000

merge_arr: 1.24813e-308, 2.49625e-309, 4.99251e-310, 9.98502e-311, 1.997e-311, 3.99401e-312, 7.98802e-313, 1.5976e-313, 3.19521e-314, 0.8, 1.16, 1.232, 1.2464, 1.24928, 1.24986, 1.64997, 1.32999, 1.466, 1.4932, 1.49864, 1.49973, 1.29995, 1.25999, 1.852, 2.3704, 4.07408, 3.21482, 2.64296, 2.52859, 2.50572, 2.50114, 2.50023, 2.70005, 2.74001, 2.548, 6.3096, 3.46192, 2.89238, 3.17848, 3.6357, 3.72714, 3.74543, 3.94909, 3.78982, 3.75796, 3.75159, 3.95032, 3.99006, 4.19801, 4.2396, 4.64792, 4.52958, 5.50592, 5.10118, 5.62024, 5.32405, 5.26481, 6.25296, 7.05059, 5.41012, 5.28202, 5.2564, 5.85128, 6.17026, 6.23405, 6.24681, 6.84936, 6.56987, 6.31397, 6.46279, 6.49256, 6.49851, 6.6997, 7.13994, 7.82799, 7.1656, 7.83312, 7.36662, 7.27332, 7.65466, 8.13093, 7.62619, 7.72524, 7.94505, 8.18901, 8.0378, 7.80756, 7.76151, 8.7523, 9.75046, 8.95009, 9.39002, 8.878, 9.1756, 9.03512, 9.80702

merge_ins_arr_20: 8.89397e-308, 1.77879e-308, 3.55759e-309, 7.11518e-310, 1.42304e-310, 2.84607e-311, 5.69214e-312, 1.13843e-312, 2.27686e-313, 4.55371e-314, 9.10743e-315, 1.82149e-315, 3.64297e-316, 7.28594e-317, 1.45719e-317, 2.91438e-318, 5.82874e-319, 1.16575e-319, 2.3315e-320, 4.66398e-321, 9.33784e-322, 1.87745e-322, 3.95253e-323, 9.88131e-324, 0.4, 1.28, 0.656, 0.3312, 0.26624, 0.253248, 0.45065, 0.49013, 1.09803, 1.21961, 1.24392, 2.04878, 1.40976, 1.28195, 1.25639, 1.25128, 1.25026, 1.45005, 1.29001, 1.258, 1.2516, 1.25032, 1.25006, 1.25001, 1.25, 1.45, 1.29, 1.258, 1.8516, 1.37032, 1.87406, 1.97481, 1.59496, 2.11899, 1.8238, 1.36476, 1.27295, 1.85459, 2.37092, 2.47418, 2.49484, 2.49897, 2.49979, 2.49996, 2.49999, 2.5, 2.5, 2.5, 2.7, 2.54, 2.508, 2.5016, 2.70032, 2.94006, 2.58801, 2.7176, 2.94352, 2.7887, 2.75774, 2.75155, 2.95031, 2.79006, 2.55801, 2.9116, 3.18232, 3.23646, 3.04729, 3.40946, 2.68189, 3.33638, 3.06728, 3.61346

merge_ins_arr_50: 8.89389e-308, 1.77878e-308, 3.55755e-309, 7.11511e-310, 1.42302e-310, 2.84604e-311, 5.69209e-312, 1.13842e-312, 2.27683e-313, 4.55367e-314,

9.10734e-315, 1.82147e-315, 3.64294e-316, 7.28587e-317, 1.45717e-317, 2.91435e-318, 5.82869e-319, 1.16575e-319, 2.3315e-320, 4.66398e-321, 9.33784e-322, 1.87745e-322, 3.95253e-323, 9.88131e-324, 0, 1.2, 0.64, 0.328, 0.0656, 0.21312, 0.442624, 0.0885248, 0.217705, 1.04354, 1.20871, 1.84174, 1.36835, 1.27367, 1.25473, 1.25095, 1.25019, 1.25004, 1.25001, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25, 1.45, 1.69, 1.338, 1.6676, 1.53352, 1.7067, 1.94134, 1.58827, 1.51765, 1.30353, 1.26071, 1.65214, 1.53043, 1.70609, 1.54122, 2.30824, 2.46165, 2.49233, 2.49847, 2.49969, 2.49994, 2.49999, 2.5, 2.5, 2.5, 2.7, 2.74, 2.748, 2.7496, 2.94992, 2.78998, 2.758, 2.7516, 2.75032, 2.95006, 2.59001, 3.118, 3.0236, 3.40472, 2.88094, 3.37619, 3.47524, 3.49505, 3.09901, 3.8198

merge_ins_arr_100: 8.89391e-308, 1.77878e-308, 3.55756e-309, 7.11513e-310, 1.42303e-310, 2.84605e-311, 5.6921e-312, 1.13842e-312, 2.27684e-313, 4.55368e-314, 9.10737e-315, 1.82147e-315, 3.64295e-316, 7.28589e-317, 1.45718e-317, 2.91435e-318, 5.82869e-319, 1.16575e-319, 2.3315e-320, 4.66398e-321, 9.33784e-322, 1.87745e-322, 3.95253e-323, 0.4, 0.88, 1.776, 1.3552, 1.27104, 0.254208, 0.250842, 0.450168, 1.09003, 1.21801, 1.4436, 1.28872, 2.25774, 1.45155, 1.29031, 1.25806, 1.25161, 1.25032, 1.45006, 1.49001, 1.298, 1.2596, 1.25192, 1.45038, 1.49008, 1.49802, 1.4996, 1.89992, 1.97998, 2.196, 2.4392, 2.48784, 2.49757, 3.09951, 3.0199, 2.60398, 2.5208, 1.70416, 1.54083, 2.30817, 2.46163, 2.49233, 2.49847, 2.69969, 2.53994, 2.50799, 2.5016, 2.50032, 2.70006, 3.14001, 2.828, 2.7656, 2.55312, 2.91062, 2.58212, 2.71642, 2.74328, 2.94866, 2.98973, 2.99795, 2.79959, 3.15992, 3.83198, 3.7664, 4.15328, 3.83066, 4.36613, 4.07323, 4.01465, 3.80293, 4.16059, 4.23212, 4.04642

merge_ins_arr_300: 8.89405e-308, 1.77881e-308, 3.55762e-309, 7.11524e-310, 1.42305e-310, 2.8461e-311, 5.69219e-312, 1.13844e-312, 2.27688e-313, 4.55375e-314, 9.10751e-315, 1.8215e-315, 3.643e-316, 0.4, 1.08, 1.616, 1.5232, 1.30464, 1.26093, 1.45219, 1.29044, 1.45809, 1.29162, 1.25832, 1.45166, 3.29033, 2.25807, 1.45161, 1.49032, 1.49806, 1.69961, 2.33992, 2.66798, 2.9336, 2.98672, 6.39734, 3.67947, 2.73589, 3.34718, 3.06944, 3.61389, 3.72278, 3.74456, 3.74891, 2.74978, 2.54996, 2.50999, 2.502, 2.9004, 2.78008, 2.95602, 2.5912, 2.71824, 3.14365, 3.82873, 3.76575, 4.75315, 4.55063, 4.71013, 3.94203, 4.18841, 4.03768, 4.80754, 4.16151, 4.4323, 4.88646, 5.57729, 5.31546, 5.06309, 5.61262, 5.12252, 5.2245, 5.8449, 6.16898, 6.2338, 6.24676, 6.24935, 6.44987, 6.48997, 6.49799, 6.6996, 7.13992, 7.62798, 7.5256, 7.90512, 7.98102, 8.3962, 8.27924, 8.25585, 9.05117, 9.01023, 9.00205, 5.80041, 5.76008, 5.55202, 6.3104

merge_ins_arr_500: 8.48798e-315, 1.6976e-315, 3.39519e-316, 6.79039e-317, 1.35808e-317, 1, 0.2, 0.04, 0.208, 1.0416, 1.20832, 1.24166, 1.24833, 1.24967, 1.64993, 3.12999, 1.826, 1.3652, 1.27304, 1.25461, 1.25092, 1.65018, 2.33004, 2.66601, 2.7332, 5.74664, 3.54933, 2.70987, 3.74197, 3.74839, 3.74968, 3.74994, 4.74999, 5.35, 5.87, 10.174, 4.6348, 2.92696, 3.58539, 2.91708, 3.58342, 3.91668, 3.78334, 3.75667, 3.95133, 3.79027, 3.75805, 4.55161, 4.91032, 5.18206, 5.43641, 5.08728, 5.21746, 5.44349, 6.2887, 6.25774, 7.45155, 8.09031, 8.41806, 7.08361, 7.41672, 7.68334, 8.53667, 7.70733, 7.54147, 8.30829, 9.86166, 9.37233, 8.87447, 9.77489, 9.75498, 9.951, 10.7902, 11.158, 10.6316, 10.7263, 7.14526, 7.42905, 6.68581, 6.53716, 6.50743, 7.10149, 7.4203, 7.68406, 7.73681, 7.94736, 7.78947, 8.55789, 7.91158, 8.78232, 8.95646, 8.99129, 8.79826, 9.55965, 9.51193, 10.1024

merge_reverse: 8.8934e-308, 1.77868e-308, 3.55736e-309, 7.11472e-310, 1.42294e-310, 2.84589e-311, 0.2, 0.04, 0.008, 0.0016, 0.60032, 0.120064, 1.02401, 1.2048, 1.24096, 1.44819, 1.48964, 1.29793, 1.25959, 1.25192, 1.25038, 1.45008, 1.29002, 1.458, 1.8916, 3.97832, 2.39566, 2.07913, 2.41583, 2.48317, 2.69663, 2.53933, 2.70787, 2.74157, 3.14831, 4.62966, 3.72593, 2.74519, 2.54904, 2.90981, 2.58196, 2.71639, 2.74328, 3.14866, 3.42973, 3.68595, 3.73719, 3.74744, 3.74949, 3.9499, 4.18998, 3.838, 3.7676, 4.35352, 5.0707, 4.41414, 5.08283, 4.41657, 5.48331, 4.89666, 4.97933, 4.99587, 5.39917, 5.27983, 5.05597, 5.21119, 5.24224, 5.24845, 5.04969, 5.60994, 5.12199, 5.8244, 5.96488, 6.19298, 6.2386, 6.24772, 6.04954, 6.60991, 6.72198, 6.3444, 6.66888, 6.53378, 6.50676, 6.70135, 6.94027, 7.38805, 6.47761, 7.09552, 6.6191, 6.92382, 6.98476, 7.59695, 7.71939, 7.94388, 7.98878, 7.79776

merge_ins_reverse: 8.89408e-308, 1.77882e-308, 3.55763e-309, 7.11526e-310, 1.42305e-310, 2.8461e-311, 5.69221e-312, 1.13844e-312, 2.27688e-313, 4.55377e-314, 9.10753e-315, 1.82151e-315, 3.64301e-316, 7.28603e-317, 1.45721e-317, 2.91441e-318, 5.82884e-319, 1.16575e-319, 2.3315e-320, 4.66398e-321, 9.33784e-322, 1.87745e-322, 3.95253e-323, 9.88131e-324, 0, 0.8, 0.16, 0.032, 0.0064, 0.00128, 0.000256, 0.200051, 0.24001, 0.248002, 0.2496, 1.24992, 0.649984, 0.129997, 0.625999, 0.1252, 0.22504, 1.04501, 1.209, 1.2418, 1.24836, 1.24967, 1.24993, 1.24999, 1.25, 1.25, 1.25, 1.25, 1.25, 1.45, 1.49, 1.498, 1.4996, 1.49992, 1.69998, 1.34, 1.268, 1.2536, 1.25072, 1.25014, 1.25003, 1.25001, 1.25, 1.65, 1.73, 1.946, 1.3892, 1.67784, 1.73557, 1.54711, 1.90942, 2.18188, 1.83638, 1.76728, 1.95346, 1.79069, 1.75814, 1.95163, 2.19033, 2.43807, 2.48761, 2.49752, 2.4995, 2.4999, 2.69998, 2.54, 2.508, 2.7016, 2.74032, 2.54806, 2.70961, 2.54192

merge_near: 8.89475e-308, 1.77895e-308, 3.5579e-309, 7.1158e-310, 1.42316e-310, 2.84632e-311, 5.69264e-312, 1.13853e-312, 2.27706e-313, 4.55411e-314, 0.6, 0.32, 1.064, 1.2128, 1.24256, 1.44851, 1.2897, 1.25794, 1.25159, 1.25032, 1.85006, 1.37001, 1.274, 1.2548, 1.65096, 3.73019, 2.54604, 1.50921, 1.90184, 2.38037, 2.47607, 2.49521, 2.69904, 2.73981, 3.14796, 5.62959, 3.72592, 2.74518, 2.54904, 2.50981, 2.50196, 3.10039, 3.42008, 2.88402, 3.1768, 3.63536, 3.72707, 3.94541, 3.78908, 3.95782, 3.79156, 3.95831, 3.99166, 4.39833, 4.67967, 3.93593, 4.98719, 4.59744, 5.31949, 4.4639, 4.89278, 5.17856, 5.03571, 5.00714, 5.00143, 5.20029, 5.44006, 5.28801, 5.0576, 5.41152, 5.0823, 5.41646, 6.08329, 5.81666, 6.56333, 5.91267, 5.78253, 5.95651, 6.5913, 6.31826, 6.66365, 6.53273, 6.70655, 6.74131, 6.74826, 7.54965, 7.10993, 6.42199, 7.0844, 6.61688, 7.32338, 7.66468, 7.73294, 7.54659, 7.70932, 8.54186

merge_ins_arr: 6.49473e-57, 1.29895e-57, 2.59789e-58, 5.19579e-59, 1.03916e-59, 2.07831e-60, 4.15663e-61, 8.31326e-62, 1.66265e-62, 3.3253e-63, 6.65061e-64, 1.33012e-64, 2.66024e-65, 5.32048e-66, 1.0641e-66, 2.12819e-67, 4.25639e-68, 8.51277e-69, 1.70255e-69, 3.40511e-70, 6.81022e-71, 1.36204e-71, 2.72409e-72, 5.44818e-73, 1.08964e-73, 0.6, 0.12, 0.024, 0.0048, 0.00096, 0.000192, 3.84e-05, 7.68e-06, 0.200002, 0.24, 1.048, 0.4096, 0.08192, 0.016384, 0.0032768, 0.00065536, 0.000131072, 2.62144e-05, 0.200005, 0.040001, 0.208, 0.0416, 0.00832001, 0.201664, 1.04033, 1.20807, 1.24161, 1.24832, 1.24966, 1.24993, 1.24999, 1.45, 1.29, 1.458, 1.2916, 1.25832, 1.25166, 1.45033, 1.29007, 1.25801, 1.2516, 1.45032, 1.29006, 1.25801, 1.2516, 1.25032, 1.25006, 1.25001, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25, 1.25, 1.65,

1.53, 1.506, 1.3012, 1.26024, 1.25205, 1.25041, 1.45008, 1.49002, 1.298, 1.2596, 1.45192, 1.89038