Алгоритмы и структуры данных-1

SET 3. Задача A2.

Осень 2024. Клычков М. Д.

- ID посылки по задаче `A1i`: 292469913
- Ссылка на репозиторий `GitHub`

Внутренняя инфраструктура для экспериментального анализа

Были реализованы классы `ArrayGenerator` и `SortTester` для организации удобного замера работы алгоритмов.

```
#ifndef GENERATOR_H
#define GENERATOR_H

#include <algorithm>
#include <random>
#include <vector>

class ArrayGenerator {
 private:
  int min_val_{};
  int max_val_{};
  size_t max_size_{};
  double nearly_factor_{};  // 0 < x < 1, where 1 is n swaps
  std::vector<int> random_{};
  std::vector<int> reversed_{};
  std::vector<int> nearly_sorted_{};
  std::mt19937 random_engine_{};

  std::vector<int> GenerateRandomVec();
  void InitRandom();
  void InitReversed();
  void InitNearlySorted();

 public:
  ArrayGenerator(int min_val, int max_val, size_t max_size, double nearly_factor, int seed);
  std::vector<int> Random(size_t size);
  std::vector<int> Reversed(size_t size);
  std::vector<int> NearlySorted(size_t size);
};

#endif
```

Рис. 1: `ArrayGenerator Header File`

```cpp
#include "generator.h"

std::vector<int> ArrayGenerator::GenerateRandomVec() {
  std::uniform_int_distribution distr{min_val_, max_val_};
  std::vector<int> result{};
  result.reserve(max_size_);

  for (size_t i = 0; i < max_size_; ++i) {
    result.push_back(distr(random_engine_));
  }

  return result;
}

void ArrayGenerator::InitRandom() {
  random_ = GenerateRandomVec();
}
void ArrayGenerator::InitReversed() {
  reversed_ = GenerateRandomVec();
  std::sort(reversed_.begin(), reversed_.end(), std::greater<>());
}
void ArrayGenerator::InitNearlySorted() {
  nearly_sorted_ = GenerateRandomVec();
  std::sort(nearly_sorted_.begin(), nearly_sorted_.end());

  size_t swaps_count = static_cast<size_t>(nearly_factor_ * static_cast<double>(max_size_));

  while (swaps_count > 0) {
    size_t i = random_engine_() % max_size_;
    size_t j = random_engine_() % max_size_;
    if (i != j) {
      std::swap(nearly_sorted_[i], nearly_sorted_[j]);
      --swaps_count;
    }
  }
}

ArrayGenerator::ArrayGenerator(int mn, int mx, size_t max_size, double nearly_factor, int seed)
    : min_val_{mn}, max_val_{mx}, max_size_{max_size}, nearly_factor_{nearly_factor} {
  random_engine_.seed(seed);
  InitRandom();
  InitReversed();
  InitNearlySorted();
}

std::vector<int> ArrayGenerator::Random(size_t size) {
  std::vector<int> result(size);
  std::copy_n(random_.begin(), size, result.begin());
  return result;
}
std::vector<int> ArrayGenerator::Reversed(size_t size) {
  std::vector<int> result(size);
  std::copy_n(reversed_.begin(), size, result.begin());
  return result;
}
std::vector<int> ArrayGenerator::NearlySorted(size_t size) {
  std::vector<int> result(size);
  std::copy_n(nearly_sorted_.begin(), size, result.begin());
  return result;
}
```

Рис. 2: `ArrayGenerator Cpp File`

```cpp
#ifndef TESTER_H
#define TESTER_H

#include "generator.h"
#include "sort.h"

struct SortTesterResult {
  long long random_time;
  long long reversed_time;
  long long nearly_sorted_time;
};

class SortTester {
 private:
  using Ms = std::chrono::microseconds;

  ArrayGenerator generator_;

 public:
  SortTester(int min_val, int max_val, size_t max_size, double nearly_sorted_factor, int seed)
      : generator_{min_val, max_val, max_size, nearly_sorted_factor, seed} {
  }

  SortTesterResult TestMerge(size_t size);
  SortTesterResult TestMergeInsertion(size_t threshold, size_t size);
};

#endif
```

Рис. 3: SortTester Header File

```cpp
#include "tester.h"

#include <vector>

SortTesterResult SortTester::TestMerge(size_t size) {
  SortTesterResult result{};

  std::vector<int> random = generator_.Random(size);
  std::vector<int> reversed = generator_.Reversed(size);
  std::vector<int> nearly_sorted = generator_.NearlySorted(size);

  auto start = std::chrono::high_resolution_clock::now();
  MergeSort(random);
  auto end = std::chrono::high_resolution_clock::now();
  result.random_time = std::chrono::duration_cast<Ms>(end - start).count();

  start = std::chrono::high_resolution_clock::now();
  MergeSort(reversed);
  end = std::chrono::high_resolution_clock::now();
  result.reversed_time = std::chrono::duration_cast<Ms>(end - start).count();

  start = std::chrono::high_resolution_clock::now();
  MergeSort(nearly_sorted);
  end = std::chrono::high_resolution_clock::now();
  result.nearly_sorted_time = std::chrono::duration_cast<Ms>(end - start).count();

  return result;
}

SortTesterResult SortTester::TestMergeInsertion(size_t threshold, size_t size) {
  SortTesterResult result{};
  MergeInsertionSorter sorter;
  sorter.threshold = threshold;

  std::vector<int> random = generator_.Random(size);
  std::vector<int> reversed = generator_.Reversed(size);
  std::vector<int> nearly_sorted = generator_.NearlySorted(size);

  auto start = std::chrono::high_resolution_clock::now();
  sorter.MergeInsertionSort(random);
  auto end = std::chrono::high_resolution_clock::now();
  result.random_time = std::chrono::duration_cast<Ms>(end - start).count();

  start = std::chrono::high_resolution_clock::now();
  sorter.MergeInsertionSort(reversed);
  end = std::chrono::high_resolution_clock::now();
  result.reversed_time = std::chrono::duration_cast<Ms>(end - start).count();

  start = std::chrono::high_resolution_clock::now();
  sorter.MergeInsertionSort(nearly_sorted);
  end = std::chrono::high_resolution_clock::now();
  result.nearly_sorted_time = std::chrono::duration_cast<Ms>(end - start).count();

  return result;
}
```

Рис. 4: `SortTester Cpp File`

В функции `main()` происходит создание сразу трех объектов `SortTester` с разными `seed`. Каждый запуск происходит по три раза на каждом из тестеров, затем результаты усредняются.

В качестве значений `threshold` были выбраны числа $\{5, 10, 20, 50, 100, 200, 400\}$.

Некоторые замечания: эта реализация не является оптимальной по крайней мере потому, что один и тот же код вызова сортировок повторятся для каждого алгоритма, однако было принято решение оставить такую реализацию, так как она уж точно не содержит никаких подводных камней, которые могут повлиять на временную оценку.

Анализ

Для построения графиков использовался язык `Python`.

Для начала проанализируем маленькие значения `threshold`, а именно $\{5, 10, 20, 50\}$:

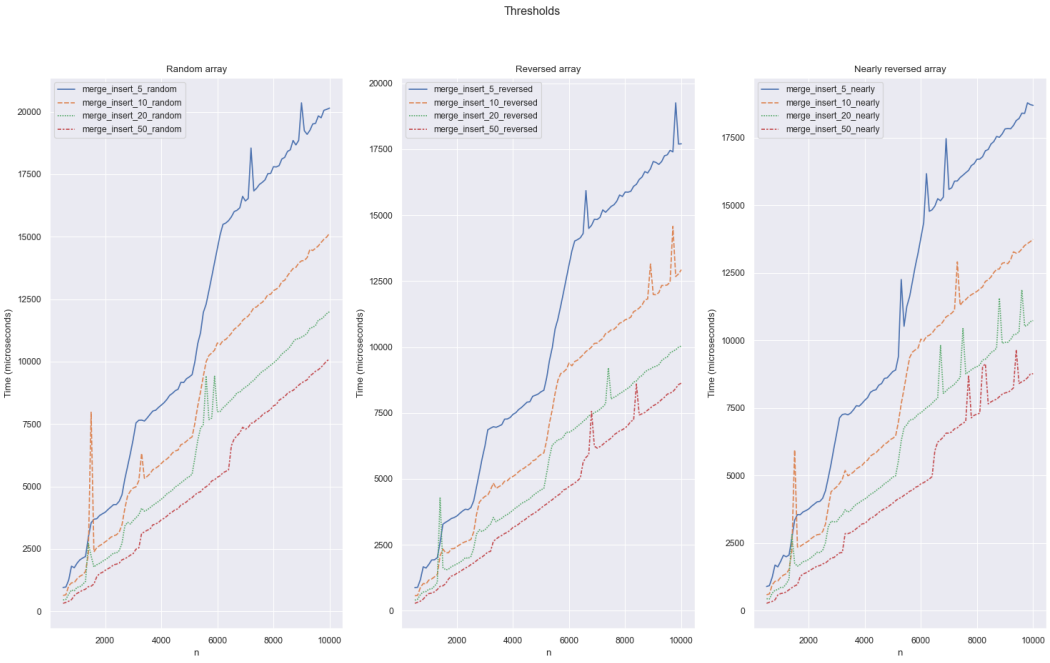

Рис. 5: Малые значения `threshold`

Замечаем, что в любом случае (при любом типе сгенерированного массива) «побеждает» наибольшее здесь значение — 50.

Сравним победителя с бОльшими значениями, а именно $\{100, 200, 400\}$:

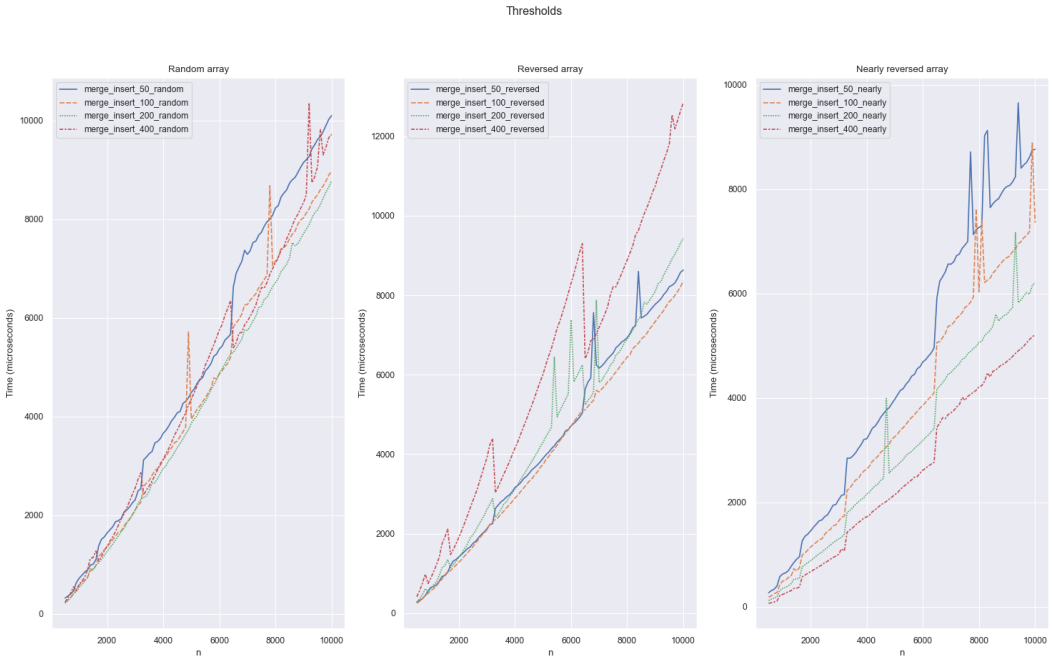

Рис. 6: Большие значения `threshold`

Тут уже не такая однозначная ситуация. Наибольшее значение `threshold` имеет абсолютное преимущество только в одном случае — практически отсортированный массив, и в то же время проигрывает всем другим значениям в отсортированных по убыванию массивах. Это объясняется тем, что сортировке вставками приходится работать на значительном количестве элементов, а, как известно, при отсортированном массиве `Insertion sort` линейна по времени, при обратно отсортированном — квадрат.

Интересно увидеть это на другом графике:

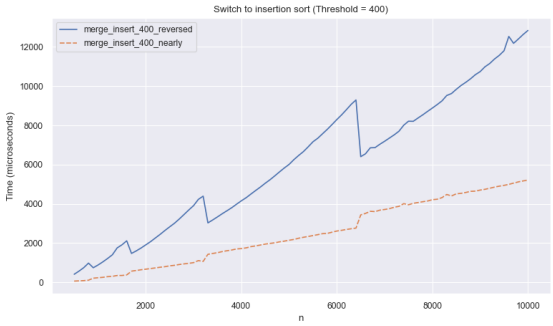

Рис. 7: Переход на `Insertion Sort`

Четко виден момент, когда `Insertion Sort` работает на максимальном количестве элементов — 400 (*напомню* шаг 100), затем сразу резкий спуск — на долю `Insertion Sort` выпадает меньшее количество элементов.

Вернемся к основной линии повествования. Лучшие временные показатели на всех видах массивов показали значения `threshold` 100 и 200, поэтому далее будем рассматривать их.

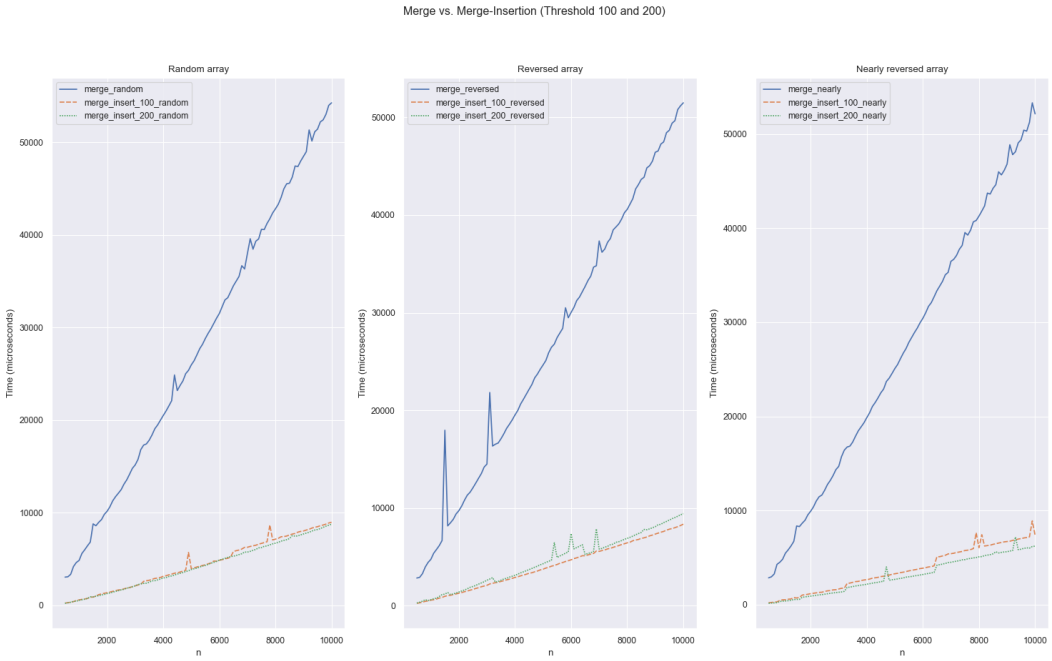

Рис. 8: `Merge Sort` *vs.* `Merge-Insertion Sort`

Невооруженным глазом видно, что лидирует сортировка `Merge-Insertion Sort`, преимущество ≈ 5 раз.

Единственный вопрос, оставшийся нерассмотренным, — сравнение сортировок по типам массивов.

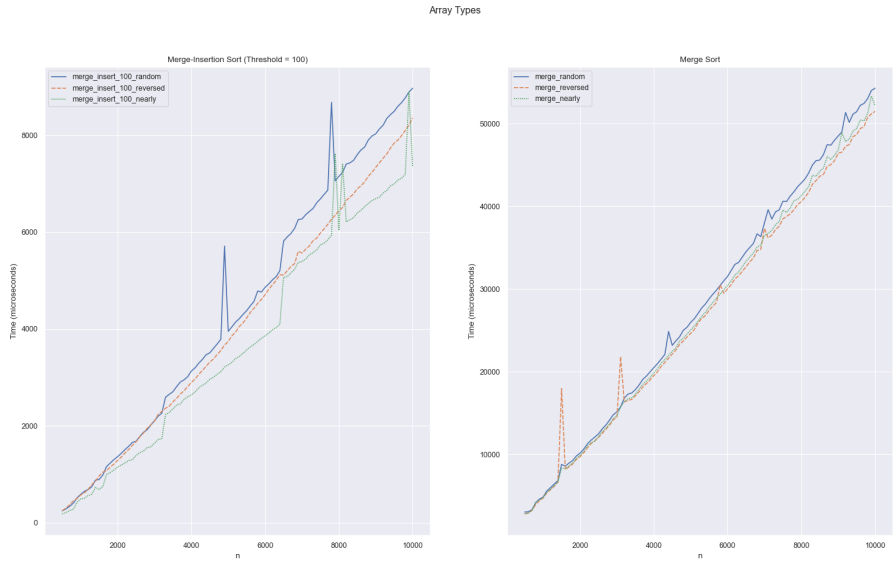

Рис. 9: Сравнение типов массивов

Как мы можем увидеть для `Merge Sort` совсем неважно, какой массив подается на вход — время сортировки не изменится. Что же касается `Merge-Insertion Sort` для `threshold` равном 100, тут уже наблюдается разброс, но главным образом это связано с описанным выше поведением `Insertion Sort` на разных типах массивов.

Выводы

Основные выводы по графикам были сделаны выше, тут же только отметим, что в любом случае при выборе между этими двумя сортировками стоит использовать вариант `Merge-Insertion Sort`. Также отмечу, что у этой *комбинированной* сортировки сохраняется свойство *устойчивости*, так как им обладают оба *родителя* — `Merge Sort` и `Insertion Sort`.