Алгоритмы и структуры данных-1

SET 3. Задача A3.

Осень 2024. Клычков М. Д.

- ID посылки по задаче `A1i`: 292810314
- Ссылка на репозиторий `GitHub`

Внутренняя инфраструктура для экспериментального анализа

Внутренняя инфраструктура для экспериментального анализа практически полностью была скопирована из предыдущей задачи. Из изменений добавился только общий генератор случайных чисел, распространяющийся на весь проект (это позволительно, так как мы запускаем программу в одном потоке).

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

 public:
  ArrayGenerator(int min_val, int max_val, size_t max_size, double nearly_factor);
  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"

#include "random_utils.h"

std::vector<int> ArrayGenerator::GenerateRandomVec() {
  auto random_engine = GetRandomEngine();
  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());

  auto random_engine = GetRandomEngine();
  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)
    : min_val_{mn}, max_val_{mx}, max_size_{max_size}, nearly_factor_{nearly_factor} {
  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"

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)
      : generator_{min_val, max_val, max_size, nearly_sorted_factor} {
  }

  SortTesterResult TestQuick(size_t size);
  SortTesterResult TestIntro(size_t size);
};

#endif
```

Рис. 3: SortTester Header File

```cpp
#include "tester.h"

#include <vector>

#include "sort.h"

SortTesterResult SortTester::TestQuick(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();
  QuickSort(random, 0, size - 1);
  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();
  QuickSort(reversed, 0, size - 1);
  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();
  QuickSort(nearly_sorted, 0, size - 1);
  end = std::chrono::high_resolution_clock::now();
  result.nearly_sorted_time = std::chrono::duration_cast<Ms>(end - start).count();

  return result;
}

SortTesterResult SortTester::TestIntro(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 param = static_cast<long long>(2 * std::log2(size));

  auto start = std::chrono::high_resolution_clock::now();
  IntroSort(random, 0, size - 1, param);
  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();
  IntroSort(reversed, 0, size - 1, param);
  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();
  IntroSort(nearly_sorted, 0, size - 1, param);
  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

```cpp
#ifndef RANDOM_UTILS_H
#define RANDOM_UTILS_H

#include <random>

inline std::mt19937& GetRandomEngine() {
  static thread_local std::mt19937 engine{std::random_device{}()};
  return engine;
}

#endif
```

Рис. 5: `Random Utils Header File`

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

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`.

Сначала проанализируем каждый алгоритм сортировки по отдельности. Начнем с `QuickSort`.

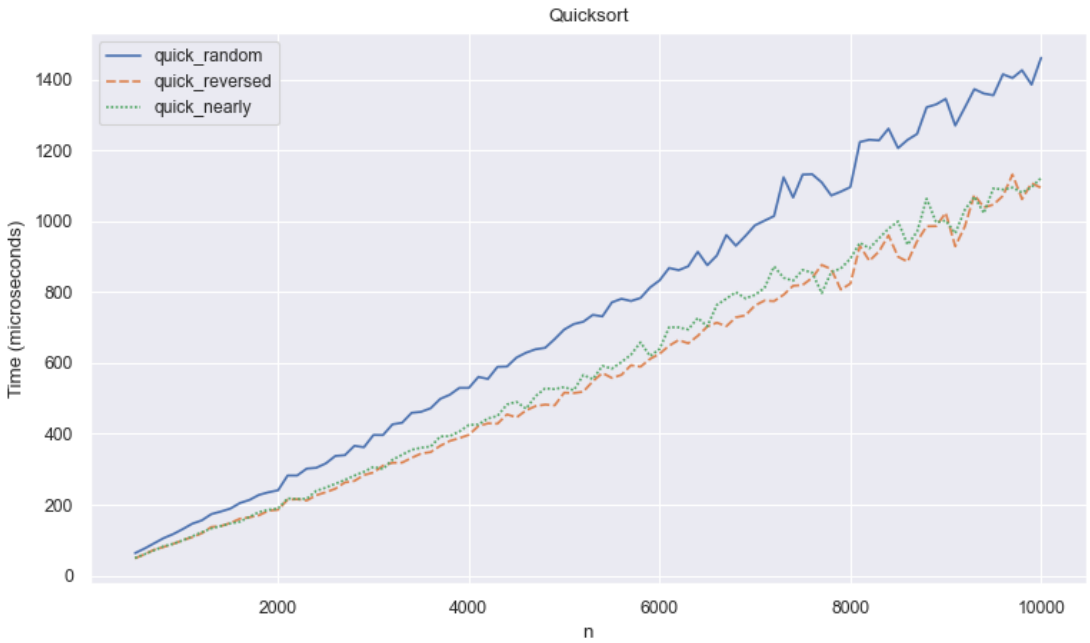

Рис. 6: `QuickSort`

Можем заметить, что сортировка случайно заполненного неупорядоченного массива ≈ 1.33 раза больше, чем сортировка почти упорядоченного по неубыванию и упорядоченного по невозрастанию массивов, время сортировки которых при равном n приблизительно равны.

Казалось бы, сортировка почти упорядоченного массива должна лидировать по времени исполнения, однако при случайно выбранном элементе `pivot` это не так. Действительно, при выборе крайних элементов в качестве `pivot` приходится совершать большое количество перестановок элементов, которые сильно влияют на производительность. Во избежании такого поведения иногда

применяют алгоритм `MedianOfThree` для выбора не случайного значения `pivot`, а заданного по определенному правилу.

Рассмотрим `IntroSort`.

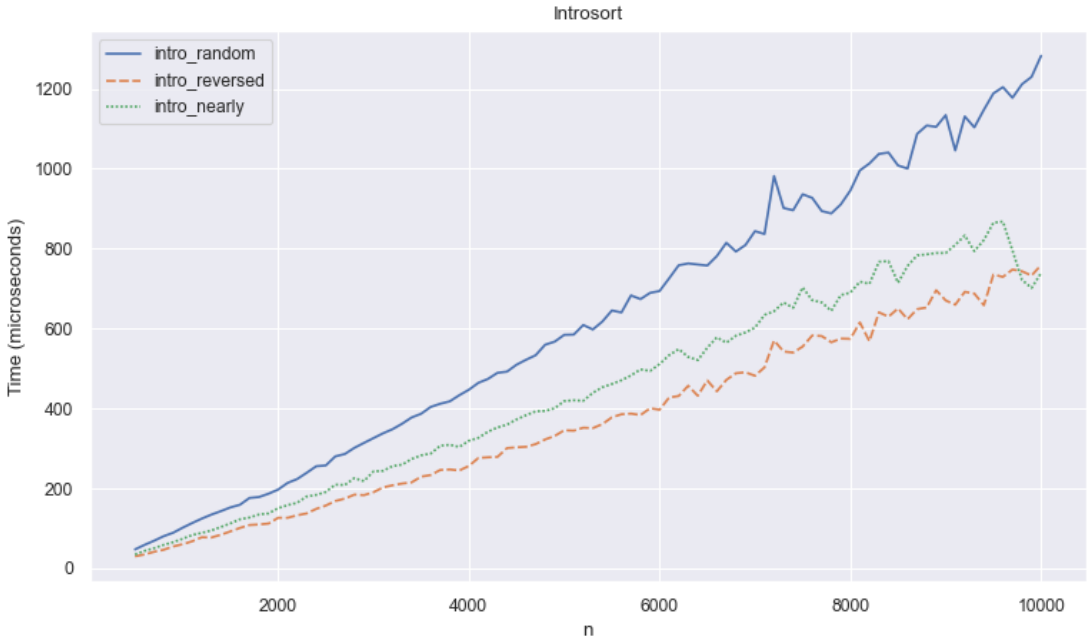

Рис. 7: `IntroSort`

На этом графике наблюдается та же ситуация — случайно заполненный массив «проигрывает» по времени двум другим видам массивов. Также здесь наблюдается более сильное расхождение по времени между обратно отсортированными и почти отсортированными массивами.

Лучший результат отсортированных по невозрастанию массивов может быть обеспечен тем, что при переходе на `HeapSort BuildMaxHeap` отрабатывает быстрее на такого рода массивах, но только в пределах скрытых констант, на асимптотику это никак не влияет.

Теперь же перейдем к сравнению двух алгоритмов: `IntroSort` и `QuickSort`.

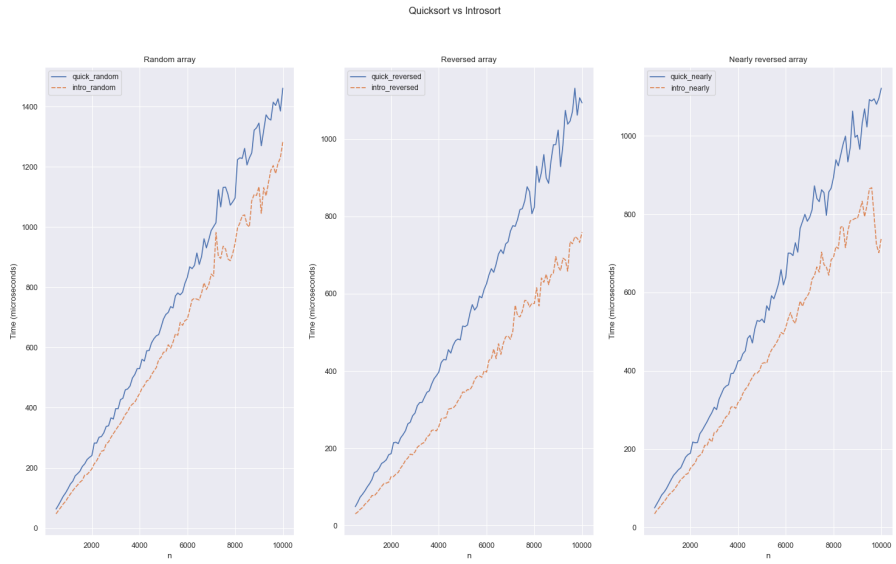

Рис. 8: `IntroSort` *vs.* `QuickSort`

Результаты *поражающие*! В каждом из случаев `IntroSort` показывает результат лучше другого алгоритма. Все это благодаря тому, что гибридная сортировка сочетает в себе достоинства трех различных алгоритмов.

Выводы

Основные выводы по графикам были сделаны выше, тут же только отметим, что при выборе между этими двумя сортировками стоит использовать вариант `IntroSort`.