МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и струтуры данных»

Тема: Принцип «Разделяй и властвуй»

Студент гр. 1301 _____ Устинов Г.А.

Преподаватель _____ Родионова Е.А.

Санкт-Петербург

2022

Цель работы

Изучить и реализовать алгоритм поиска пары наиболее близлежащих точек на плоскости.

Задание

Решить задачу с использованием принципа «Разделяй и властвуй».

Определить теоретическую асимптотическую сложность решения.

Эмпирически оценить временную сложность решения для «average-case».

Отчет по лабораторной работе должен содержать:

1. Титульный лист с указанием варианта.
2. Постановка задачи. Описание реализуемого класса и методов.
3. Теоретическую асимптотическую сложность реализованного алгоритма
4. Оценка временной сложности эмпирическим способом.
5. Пример работы.
6. Листинг.

Задачи:

1. О влюбленных улитках. На квадрате земли 1 км 2 располагаются улитки-гермафродиты. В момент времени каждая из улиток с постоянной скоростью 1 см/с ползет к улитке, являющейся ближайшей к ней в момент времени t = 0, выбрав её в качестве спутника жизни. Определить время, через которое первая пара улиток достигнет друг друга или наличие ситуации, приводящей улиток в замешательство, наиболее эффективным способом.

Выполнение работы

Задача фактически состоит в том, чтобы на заданном множестве точек найти пару наиболее близлежащих точек.

Для того, чтобы решение по принципу «Разделяй и властвуй» имело осмысленную реализацию, нужно разбивать на каждом шаге набор точек согласно их расположению на плоскости, например, по оси X. Для этого сначала отсортируем массив точек. Для этого используется стандартная функция *sort*, гарантированно сортирующая массив за nlogn.

Теперь на каждом шаге будем разбивать массив пополам и рассматривать его половины до тех пор, пока количество точек на рассматриваемом отрезке не позволит решить задачу тривиально — пока на отрезке не останется две точки. Каждая точка на отрезке обрабатывается ровно один раз — когда для неё считается расстояние до её пары по базовому случаю. На обратном ходе рекурсии сравним полученные результаты на каждом из отрезков, и выберем наименьший.

Однако, на рассмотренном отрезке такое решение не является полным — оно не учитывает точки, которые находятся рядом с линией, по которой проходит разбиение. Для того, чтобы рассмотреть такие пары, будем снова проходить по точкам, но теперь по оси Y. Для этого на обратном ходе рекурсии будем сортировать отрезок теперь по Y-координате. Для сортировки используется встроенная функция *inplace_merge*, которая имеет две оценки быстродействия, в зависимости от доступности системных ресурсов — nlogn, когда невозможно выделить память для буфера, и n, когда памяти достаточно. На отсортированном отрезке теперь можно рассмотреть точки, которые находятся не дальше от линии разбиения, чем найденное решение, и для каждой посчитать расстояния до её соседей по Y-координате, каждый раз улучшая решение, если такое находится, причём для каждой подобранной

точки количество соседей будет достаточно мало. Всего количество слияний равно высоте получающегося двоичного дерева разбиений, то есть logn. Таким образом, обратный ход рекурсии может быть оценён как nlogn.

По выходу из рекурсии, получается кратчайшее расстояние на множестве точек, которое учитывает все варианты парообразования.

Для выполнения задачи был ипользован стандартный тип *vector*, позволяющий использовать итераторы случайного доступа, и выбирать отрезок массива за минимальное время, а такжу были реализованы три впомогательных типа — *Solution*, *Snail* и *Slice*. *Solution* содержит информацию об успешности решения и результат. *Snail* представляет собой пару чисел с плавающей точкой — координаты точки. *Slice* — пара итераторов над массивом точек и длина отрезка, который они ограничивают, также *Slice* инкапсулирует логику разбиения массива.

Входной точкой решения является функция *findClosestDistance*, которая проводит изначальную подготовку входных данных, сортировку и передачу подотовленных данных основной логике — функции *findClosestDistanceOnSlice*, которая в свою очередь реализует прямой и обратный ход рекурсивного алгоритма, а также передачу управления функции, решающей элементарную задачу в данном случае — *baseCase*.

Общая сложность решения складывается из двух слагаемых — сортировки и рукурсивного алгоритма. Сортировка имеет гарантированную асимптотику nlogn. Сложность алгритма, в свою очередь, раскладывается на слагаемые следующим образом: n (последовательный обход отсортированного массива) + n (слияние) * logn (высота двоичного дерева разбиения), или, в случае нехватки ресурсов, n + n * logn (слияние) * logn.

В первом случае итоговая сложность алгоритма равна

$$n * logn,$$

во втором же случае

$$n * \log^2 n.$$

Эмпирическая оценка временной сложности

Для оценки временной сложности полученного алгоритма были сгенерированы 10 наборов по 1000000 случайных точек так, чтобы точки в пределах одного набора не повторялись. Затем, для каждого набора точек были проведены замеры быстродействия на первых n значениях, последовательно увеличивая n с шагов в 10000. Результаты замеров на разных наборах усреднены, и на основе этих усреднённых значений построен график, наглядно иллюстрирующий зависимость времени выполнения в зависимости от коичества входных данных.

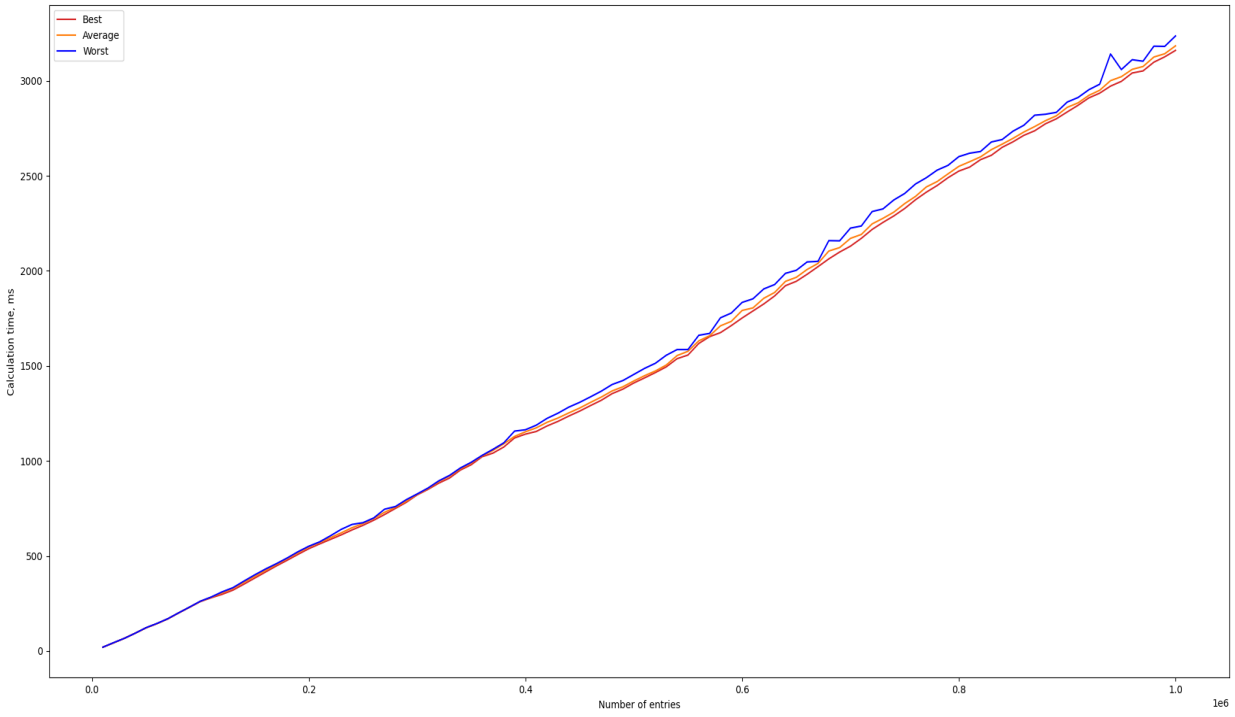

Рисунок 1 - График, построенный на основе полученных значений

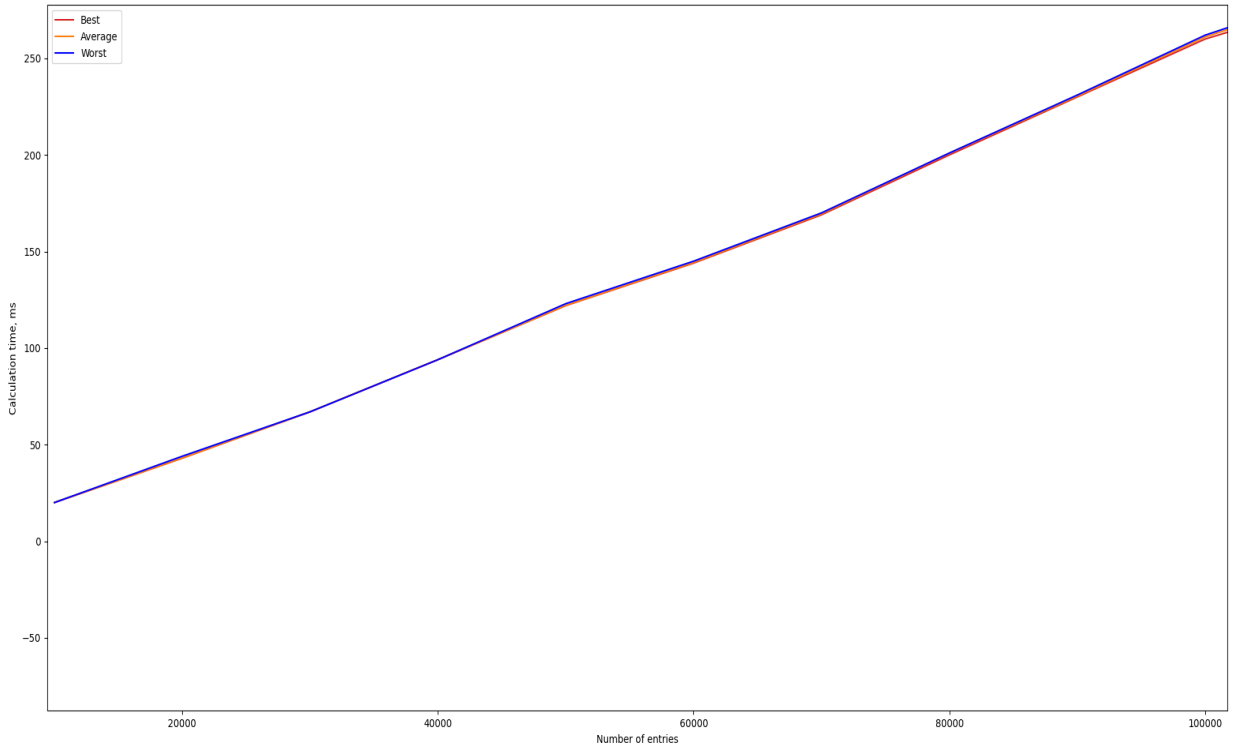

Рисунок 2 - Увеличенное изображение графика для количества записей в наборе до 100 тысяч

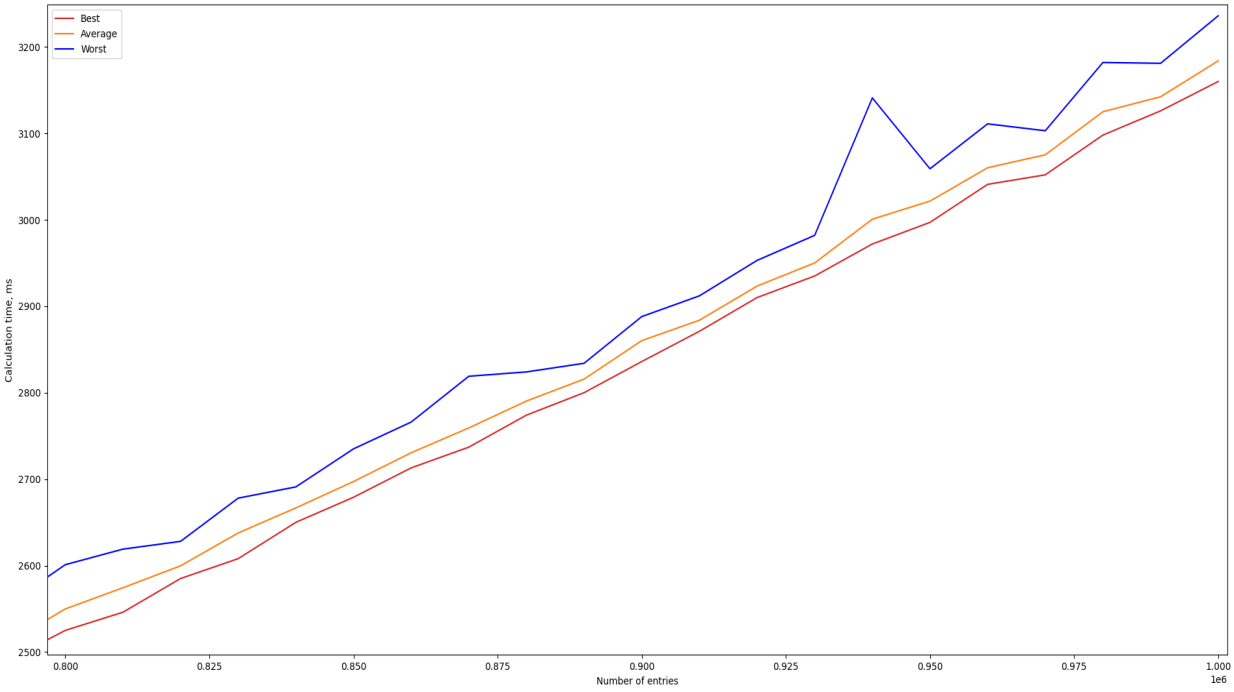

Рисунок 3 — Увеличенное изображения графика для количества записей в наборе от 800 тысяч до 1 миллиона

При данных размерах входных данных и быстродейсвтии алгоритма сложно хоть сколько-нибудь точно оценить соотношение прироста времени выполнения к приросту количества данных на входе. При столь малых значениях времени слишком большое значение оказывают такие факторы, как состояние системы, на которой выполняются измерения, и работа планировщика задач. Генерация же наборов большего размера не выглядит состоятельной в силу непропорционального увеличения времени генерации по сравнению с увеличением размера наборов из-за необходимости поддержания уникальности каждой записи, учитывая допустимые пределы значений, требующие разрешения значений подходящий к разрешению значений, выдаваемых генератором случайных чисел. К примеру, генерация наборов размером в 1 миллион, на которых производились измерения, заняла более шести часов для каждого.

Пример работы

Следующий код служит примером работы с разработанным алгоритмом.

```
vector<Snail> snails;
// инициализация входных значений
Solution res = findClosestDistance(snails);
if (res.status == kOk)
    cout << res.answer;
```

Разработанный программный код см. в приложении А.

Выводы

Был изучен и реализован алгоритм поиска ближайший точек на плоскости, а также способы оценки быстродействия алгоритмов, средства автоматизации замеров и обработки их результатов.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл **src/main.cpp**

```
#include <algorithm>
#include <chrono>
#include <exception>
#include <fstream>
#include <iostream>
#include <vector>

#include "generate.hh"
#include "snail.hh"
#include "solve.hh"

using namespace std;

vector<Snail> readSnails()
{
   ifstream in("snails.txt");
   if (!in)
       throw runtime_error("No snails file found.");
   size_t n;
   in >> n;
   vector<Snail> snails;
   try {
       while (n--) {
           double x, y;
           in >> x >> y;
           snails.push_back({ x, y });
       }
   } catch (exception &) {
       throw runtime_error("File format error.");
   }
   in.close();
   return snails;
}

void measure(vector<Snail> &snails, size_t n)
{
   vector<Snail> slice;
   copy(
       snails.cbegin(),
```

```
        snails.cbegin() + n,
        back_inserter(slice)
    );
    chrono::steady_clock timer;
    chrono::time_point<chrono::steady_clock> start = timer.now();
    Solution s = findClosestDistance(slice);
    chrono::time_point<chrono::steady_clock> finish = timer.now();
    cout << "Given " << n << " snails, calculated " << s.answer;
    cout << " in " << (finish - start) / chrono::milliseconds(1) <<
"ms" << endl;
}

int main()
{
        vector<Snail> snails;
    try {
        snails = readSnails();
        cout << "Read " << snails.size() << " snails." << endl;
    } catch (runtime_error &) {
        cout << "Snails file format error." << endl;
        size_t n = 0;
        cout << "How many to generate: ";
        do {
            cin >> n;
            if (n == 0)
                cout << "Try again: ";
        } while (n == 0);
        snails = generate(n);
        cout << "Generated " << n << " snails." << endl;
    }
    size_t step = 10000;
    for (size_t size = step; size <= snails.size(); size += step) {
        measure(snails, size);
    }
        return 0;
}
```

Файл **src/snail.hh**

```
#pragma once
#ifndef SNAIL_HH
#define SNAIL_HH
```

```
struct Snail {
       Snail(double x, double y) : x(x), y(y) {}
   Snail &operator=(const Snail &other) = default;

   double distanceTo(const Snail &other) const;

   bool operator==(const Snail &other) const;

       double x;
       double y;
};

#endif  // SNAIL_HH
```

Файл **src/snail.cpp**

```
#include "snail.hh"

#include <cmath>

using namespace std;

double Snail::distanceTo(const Snail &other) const
{
   return abs(sqrt(pow(x - other.x, 2) + pow(y - other.y, 2)));
}

bool Snail::operator==(const Snail &other) const
{
   return x == other.x && y == other.y;
}
```

Файл **src/solve.hh**

```
#pragma once
#ifndef SOLVE_HH
#define SOLVE_HH

#include <vector>

#include "snail.hh"
```

```
enum SolutionStatus {
   kOk,
   kNotEnough,
};

struct Solution {
   Solution() : answer(-1), status(kNotEnough) {}
   Solution(SolutionStatus s, double ans)
       : answer(ans), status(s)
   {}
   double answer;
   SolutionStatus status;
};

Solution findClosestDistance(std::vector<Snail> snails);

#endif  // SOLVE_HH
```

Файл **src/solve.cpp**

```
#include "solve.hh"

#include <algorithm>
#include <vector>

using namespace std;

struct Slice {
   Slice(vector<const Snail *>::iterator l, vector<const Snail
*>::iterator r)
       : left(l), right(r), len(r - l)
   {}

   vector<Slice> split()
   {
       return {
           {left, right - len / 2},
           {right - len / 2, right}
       };
   }
```

```
    vector<const Snail *>::iterator left;
    vector<const Snail *>::iterator right;
    ptrdiff_t len;
};


Solution baseCase(Slice &slice)
{
    vector<const Snail *>::iterator right = slice.left + 1;
    Solution res = {
        kOk,
        (*slice.left)->distanceTo(**right)
    };
    if ((*slice.left)->y > (*right)->y) {
        const Snail *t = *right;
        *right = *slice.left;
        *slice.left = t;
    }
    return res;
}

Solution findClosestDistanceInSquare(Slice &slice)
{
    if (slice.len < 2)
        return Solution();
    if (slice.len < 3)
        return baseCase(slice);
    vector<Slice> split = slice.split();
    double mid_x = (*split[1].left)->x;

    Solution res = findClosestDistanceInSquare(split[0]);
    Solution tmp_res = findClosestDistanceInSquare(split[1]);
    if (tmp_res.status == kOk
        && (res.status != kOk
        || tmp_res.answer < res.answer))
        res = tmp_res;
    inplace_merge(
        slice.left,
        split[1].left,
        slice.right,
        [](const Snail *a, const Snail *b)
        {
            return a->y < b->y;
        }
```

```
    );
    for (auto i = slice.left; i != slice.right; ++i) {
        if (abs((*i)->x - mid_x) <= res.answer) {
            for (vector<const Snail *>::iterator j = i + 1;
                j != slice.right
                && (*j)->y - (*i)->y <= res.answer;
                ++j) {

                double ans = (*i)->distanceTo(**j);
                if (ans < res.answer)
                    res.answer = ans;
            }
        }
    }
    return res;
}

Solution findClosestDistance(vector<Snail> snails)
{
    if (snails.size() < 2)
        return Solution();
    vector<const Snail *> sorted(snails.size(), nullptr);
    transform(
        snails.cbegin(),
        snails.cend(),
        sorted.begin(),
        [](const Snail &s) { return &s; }
    );
    sort(
        sorted.begin(),
        sorted.end(),
        [](const Snail *first, const Snail *second)
        {
            return first->x < second->x
                   || first->x == second->x && first->y < second->y;
        }
    );
    Slice slice = {sorted.begin(), sorted.end()};
    Solution res = findClosestDistanceInSquare(slice);
    return res;
}
```