

## Лаба по сортам

Цель данной лабораторной работы – посмотреть на алгоритмы с различной асимптотикой, научиться анализировать время работы алгоритмов и включать разные степени оптимизации.

Заодно, это – практика написания кода по описанному алгоритму. Рекомендуем писать код самостоятельно. Если напишете быстро, то много времени не потратите, а если закопаетесь – ну, значит, практика вам явно пойдет на пользу.

По возможности, постарайтесь сделать свою жизнь легче заранее. Планируйте код таким образом, чтобы с ним было удобно работать – меньше переписывать между пунктами или при замене размера массива. Используйте свои наработки и готовые куски кода. Можете заранее прочитать весь текст лабы, чтобы спланировать работу над кодом, это тоже помогает.

Можете использовать стартер-коды, которые идут дальше. Это небольшие работающие кусочки кода, которые реализуют нужный вам функционал. Можете пока использовать их как магические формулы, не пытаясь разбираться в деталях, как они устроены. Полностью мы их разберем к середине следующего семестра.

Стартер-код для вывода в файл:

```
#include <fstream>
using namespace std;
int main()
{
    ofstream f("1.csv", ios::out);

    // csv - стандартный формат для хранения данных прямым текстом.

    f << "uno uno uno dos quatro" << endl; // работаете как с привычным cout

    return 0;
}
```

Графики стройте с помощью python (matplotlib). Напоминаем про примеры кода с буткемпа:

<https://github.com/n-popov/pre-python/blob/main/second-day.ipynb>

<https://github.com/ivikiwi-1111/Bootcamp---livesaver>

Стартер-код для измерения времени:

```
#include <iostream>
#include <chrono>
int main() {
    auto start = std::chrono::high_resolution_clock::now();
    for (long i = 1; i < 1000000000; i += 1); // здесь то что вы хотите измерить
    auto end = std::chrono::high_resolution_clock::now();
    auto nsec = end - start;
    std::cout << nsec.count() << " нсек." << std::endl;
    return 0;
}
```

Функция для измерения времени (возвращает время в секундах, можно использовать ее вместо первого примера):

```
#include <chrono>
double get_time() {
    return std::chrono::duration_cast<std::chrono::microseconds>
        (std::chrono::steady_clock::now().time_since_epoch()).count() / 1e6;
}
```

Функция для произвольных чисел (возвращает произвольное целое число в диапазоне от min до max, ее тоже можно использовать, не влезая в ее внутренний мир):

```
#include <chrono>
#include <random>
int rand_uns(int min, int max) {
    unsigned seed = std::chrono::steady_clock::now().time_since_epoch().count();
    static std::default_random_engine e(seed);
    std::uniform_int_distribution<int> d(min, max);
    return d(e);
}
```

## 0. Пузырек и его товарищи

Начнем с простого. Напишите три разных сортировки с временной сложностью  $O(N^2)$ . Это может быть пузырек, вставка, шейкер, простой выбор, непростой выбор или что угодно, что вы еще науглите/найдете в лекциях.

Постройте график зависимости времени работы от размера массива с тремя кривыми – по одной для каждой сортировки.

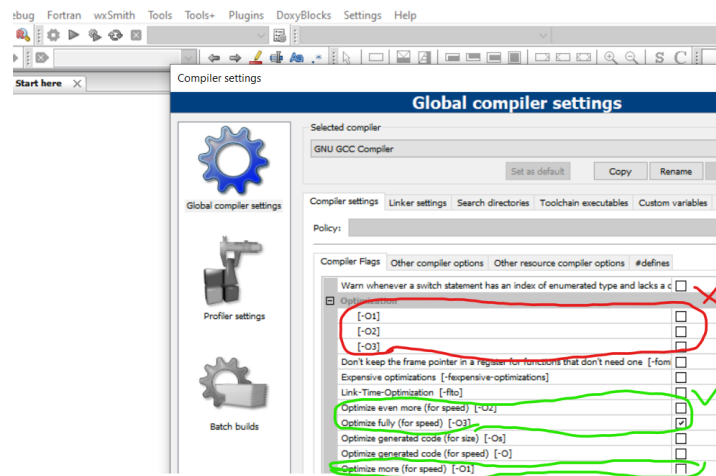
Докажите, что сложность этих алгоритмов действительно  $O(N^2)$ .

Далее подсказки в режиме спойлера, светло-серым по белому (для новичков в компах и интерфейсах – выделяйте текст мышкой, и он или сразу станет читаемым, или копируйте в любой текстовый редактор).

Доказать сложность в данном случае проще всего в двойном логарифмическом масштабе. Если  $t = O(N^2)$ , то тогда  $t \sim C * N^2 \Rightarrow \ln t = \ln C + 2 * \ln N$ . То есть параболы в координатах  $t$  и  $N$  превратятся в прямые в координатах  $\ln t$  и  $\ln N$ . Наклон прямых будет говорить о максимальной степени  $N$ , а высота над началом координат – о коэффициенте при этой максимальной степени.

А теперь лезем в настройки компилятора.

**Внимание!** Если у вас именно Code::Blocks, то там есть коварство в интерфейсе. Верхние тычки не работают, если вы их ткнете, то ничего не изменится. Вам нужны те, которые ниже и подписаны подробно.



Постройте график с четырьмя кривыми для одной из сортировок за  $O(N^2)$  – по одной кривой на каждую степень оптимизации.

## 2. А теперь настоящие быстрые сортировки

Напишите три разных сортировки с временной сложностью  $O(N \log N)$ . Это может быть куча, Хоар, расческа, слияние или что угодно, что вы еще науглите/найдете в лекциях.

Постройте график зависимости времени работы от размера массива с тремя кривыми – по одной для каждой сортировки.

Докажите, что сложность этих алгоритмов действительно  $O(N \log N)$ .

Для того, чтобы это доказать, вам нужно свести график к какому-нибудь линейному виду. Предлагаю построить зависимость  $t/N \log N$  от  $N$ . Как вы догадываетесь, должна получиться константа. Если придумаете что-то более элегантное, как двойной логарифмический масштаб для полинома – пишите и рассказывайте.

### 3. $O(N^2)$ vs $O(N \log N)$

Весьма напрашивающийся пункт. Нарисуйте кривые из пунктов 0 и 2 на одном графике. Возможно, вам для этого придется какие-то из них пересчитать, чтобы масштабы массивов соответствовали. И не запутайтесь в том, какую оптимизацию используете – она должна быть одинаковой для всех расчетов в этом пункте.

#### 4. Зависимость от начальных данных

Сравните время работы алгоритмов на отсортированном массиве, массиве произвольных чисел и массиве, отсортированном в обратную сторону. То есть лучший, средний и худший случай.


Для каждого из шести реализованных алгоритмов постройте по три кривых – лучший, средний и худший случай. Подумайте, как их сгруппировать, чтобы наиболее наглядно продемонстрировать разницу – в этом пункте может получиться несколько картинок.

Опять же – не запутайтесь во флагах и галочках, оптимизация должна быть одинаковой.

## 5. \*Небольшие массивы

В этом пункте вам нужно посмотреть на такую неочевидную штуку, как время работы шести предыдущих сортировок на небольших массивах (примерно 1 – 1000 элементов). Суть в том, что асимптотика начинает работать при очень больших  $N$  – неспроста во всех выкладках и оценках мы так за просто отбрасывали все слагаемые, кроме тех, что указаны в самой асимптотике. А при малых  $N$  эти слагаемые и коэффициенты при них вылезают, что может сильно повлиять на итоговое время работы.

На этом графике тоже должно быть шесть кривых, по одной для каждого из реализованных алгоритмов. Про оптимизацию не забывайте.

Если у вас получается стопка нулей, облако точек или ступенчатое чудовище, см ниже 

Время работы сортировки на небольших массивах (особенно до 10 элементов) может быть меньше чувствительности таймера. Не верьте `chrono`, что его `nanoseconds` это честные наносекунды – он вполне может считать в десятках тысяч или миллионах наносекунд. Таймер штука системная, и на разных машинах/ОС его точность может отличаться.

В этом случае повторяйте сортировку и усредняйте по количеству повторений – 1000, 10000, 1000000 – сколько понадобится, чтобы графики стали гладкими. Да, это одна из причин автоматизировать процесс отрисовки графиков. Я предупреждала.

Не забывайте, что время работы алгоритма зависит от входных данных – если вы раз за разом сортируете уже отсортированный массив, то это будет не совсем то время, которое нам нужно. Или совсем не то, тут уж какой алгоритм. Если вы повторяете сортировку – подумайте, как каждый раз сортировать новый произвольный массив.

Особенно учитывая, что `random` – штука медленная. Если вы вызываете его в рамках таймера, он перекроет все другие эффекты, которые вы пытались посмотреть. Выносите его за пределы таймера и не стесняйтесь жрать дополнительную память, в этом пункте у нас мелкие массивы.



#### 6\*. Зависимость от типа данных

Если вы любитель пунктов со звездочкой, то вот еще один. Скорее всего, в предыдущих пунктах вы сортировали инты. А попробуйте отсортировать массивы других типов данных и сравнить время работы.

Будет наиболее наглядно, если на одном графике изобразить кривые для времени сортировки разных типов – char, short int, int, long int, long long int, а также нескольких структур. Особенно интересны «тяжелые» структуры – с большим количеством полей и статическими массивами.

А еще оптимизация, ага.