Необходимо выполнить профилирование программы с помощью профилировщика *gprof* из набора компилятора *G++*, выявить «узкие места» и ускорить их. После этого требуется продиагностировать работу полученного кода.

Рекомендуется (но не обязательно) выполнять задание на Linux. На Windows лучше воспользоваться профилировщиком через WSL (Windows Subsystem for Linux). Также вместо *gprof* разрешается использовать любой другой профилировщик (Callgrind, gperftools и др.) с учётом разницы в принципе их работы и обзора результатов.

В прикреплённом архиве *task2.zip* находятся файлы *problem.txt* и *solution??.cpp*. Первый файл содержит условие задачи, с которой вам приходилось сталкиваться на прошлом курсе. В остальных файлах содержатся верные решения этой задачи, по одному на каждого. Варианты:
- Залевский Александр: *solution01.cpp*
- Латышев Артём: *solution02.cpp*
- Юхимчук Александр: *solution03.cpp*

**1.** Подготовьте входные данные для прогона решения. Сгенерируйте файл входных данных согласно формату из условия задачи. Размер массива *n* и количества запросов *q* должны быть **не менее 10^7** (10 миллионов). Типы запросов выбирайте случайно равновероятно.

**2.** Соберите программу с отладочными символами (ключ -g) и со включённым профилированием (ключ -pg). Команда для компиляции для gprof будет выглядеть примерно так: **`g++ solutionXX.cpp -g -pg -O0 --std=c++20 -o prog`**

**3.** Запустите исполняемый файл на подготовленных входных данных. Убедитесь, что программа выдаёт адекватные результаты.

**4.** После завершения прогона в папке появится файл *gmon.out*. Это профиль программы. Просмотрите его с помощью утилиты gprof. Сохраните результаты и продемонстрируйте их. Какие функции расходуют больше всего времени суммарно? Какие функции вызываются чаще других?

**5.** Просмотрите исходный код функций, вызываемых наиболее часто. Попробуйте их оптимизировать, например, за счёт сокращения избыточных действий (лишние арифметические действия, обращения к памяти и пр.), не теряя корректности программы. Также разрешаются незначительные изменения архитектуры программы. *Подсказка:* если нет идей, может помочь *построчное* профилирование (gprof -l), которое показывает количество вызовов каждой строки.

**6.** Повторите пункты 2-4. Как изменились простой профиль (flat profile) программы и граф вызовов? Насколько велик прирост производительности?

**7.** Проверьте оптимизированную программу на отсутствие утечек памяти с помощью Valgrind и санитайзера (по отдельности, не сразу вместе). Valgrind нужно доустановить, санитайзер идёт в комплекте с компилятором. Так как время работы может быть очень долгим, подготовьте набор входных данных поменьше: *n* и *q* порядка 10^5 (100 тысяч).

Диагностику с Valgrind запускайте так:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
         --verbose ./prog
```

Для прогона программы с адрес-санитайзером её нужно перекомпилировать: добавьте ключ
`-fsanitize=address`
и пересоберите, а затем запустите как в п. 3.

Приведите логи сообщений Valgrind и адрес-санитайзера. Были ли какие-то проблемы? Если да, то где? Сравните время работы программы с Valgrind, с санитайзером и без обоих инструментов.