**Алгоритм быстрого поиска**

**(k-ая порядковая статистика)**

**Постановка задачи**: дан массив элементов размера n и число k, необходимо найти k-ый минимум в массиве (элемент, который стоял бы на позиции k, если бы массив был отсортирован).

Будем считать, что n достигает значений порядка 10^5-10^6.

**I. Выберем три варианта решения данной задачи**

1. Реализуем алгоритм быстрого поиска наивным способом: отсортируем массив с использованием встроенной быстрой сортировки в C++ (как известно, асимптотика быстрой сортировки составляет, в худшем случае, Омега(n^2) при размере массива n, однако мат.ожидание времени работы этого подхода равно O(n \* log(n))).
2. В качестве альтернативы, посмотрим, как работает встроенная функция nth\_element в C++.
3. Сделаем оптимизацию быстрой сортировки и вручную напишем алгоритм, мат.ожидание времени работы которого будет O(n), а в худшем случае алгоритм отработает также за Омега(n^2).

\*тут описание алгоритма\*

Чтобы не хранить и не запускать все реализации отдельно, сделаем один общий файл main.cpp со всеми тремя реализациями. Соответствующие функции: kth\_naive, kth\_standard и kth\_hand\_made.

**II. Оценим количество затраченного времени и ресурсов на выполнение каждого алгоритма при помощи нагрузочных тестов**

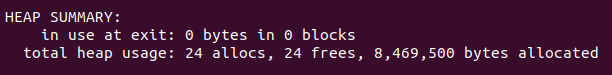
В файле gen.py будем генерировать очередной тест для наших алгоритмов в следующем формате:

n

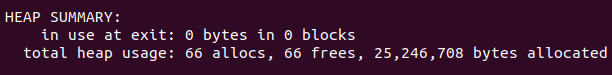
a[0] a[1] a[2] ... a[n - 1]

k

1. Для начала, запустим каждый алгоритм отдельно и с помощью утилиты valgrind посмотрим, сколько примерно затрачивается памяти на тесте с количеством элементов, близком к 10^6 (если точнее - 991352). Тест приведён в файле maxtest.txt.

Каждый алгоритм в отдельности потратил:

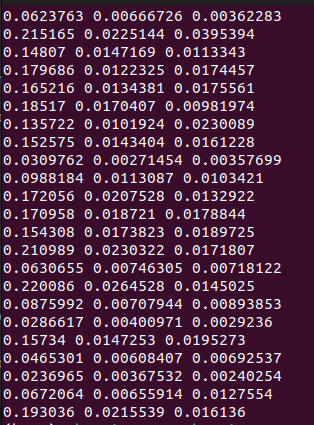
При запуске общего main файла получился следующий результат:



Собственно, это похоже на правду, так как внутри как раз работает 3 алгоритма с такими же затратами (число немного не круглое из-за переиспользования некоторых переменных).

2. Чтобы оценить время работы, будем внутри main.cpp считать время работы каждого варианта реализации с использованием библиотеки chrono. Перед выполнением будем запоминать стартовую точку через chrono::system\_clock::now(), а потом вычислять разницу между концом и началом работы конкретного подхода. На выходе main.cpp получим строчку с тремя вещественными числами: сколько отработала каждая реализация (в порядке, в котором подходы были разобраны).

В файле script.sh генерируются 100 случайных тестов, а после результат работы main.cpp на каждом из них выводится в файл time\_res.txt. Скрин части файла приведён ниже:



Как можно заметить, первый алгоритм отработал дольше всех, второй сильно быстрее первого, а третий зачастую чуть быстрее второго. Таким образом, рукописная реализация показала себя наилучшим образом. Однако, можно предположить, что библиотечная реализациия kth\_standard использует тот же алгоритм, что и в kth\_hand\_made с поправкой на выбор элемента в partition.

**III. Оценим время работы математически**

Докажем, что мат.ожидание времени работы рукописного алгоритма действительно O(n), что намного меньше асимптотики быстрой сортировки.

\*доказательство\*

**IV. Выводы**

Как и ожидалось, вариант оптимизации быстрой сортировки отработал быстрее, чем наивное решение. Это было подтверждено как математически, так и на практике.