**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++.

Вспомним алгоритм работы. На очередной итерации часть массива с left по right разбивается на две части: [left, mid] и [mid + 1, right]. Элемент под номером mid называется “опорным”. Слева от него находятся все элементы участка массива, которые меньше a[mid], а справа – все, которые больше. Далее в результате рекурсивной сортировки соответствующих частей и их соединения получается отсортированный участок массива. Для улучшения времени работы есть различные подходы к выбору элемента, который станет опорным. Мы стремимся сделать так, чтобы на каждой итерации отрезок массива делился примерно на пополам.

1. В качестве альтернативы, посмотрим, как работает встроенная функция nth\_element в C++. Попытаемся оценить, какой алгоритм используется под капотом.
2. Сделаем оптимизацию быстрой сортировки и вручную напишем алгоритм.

Идея оптимизации следующая. За основу возьмём метод разбиения массива элементов из быстрой сортировки (будем бить по рандомному). Пусть нам надо найти k-ую порядковую статистику, а после рассечения опорный элемент встал на позицию mid. Тогда рассмотрим 3 варианта развития событий:

* если k == mid, то ответ найден – это элемент a[mid];
* если k < mid, то нам нет смысла идти в правую часть массива, поэтому отправляемся рекурсивно искать k-ую статистику в левую часть;
* наконец, если k > mid, то нам нет смысла искать ответ в левой части массива, поэтому рекурсивно переходим в правую и решаем на ней задачу поиска (k - 1 - mid)-ой статистики.

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**

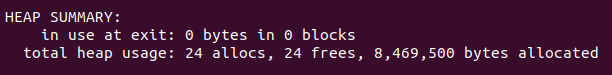
В файле gen.py будем генерировать очередной тест для наших алгоритмов в следующем формате:

n

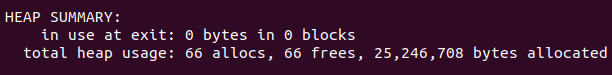
a[0] a[1] a[2] ... a[n - 1]

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 8469500 байт:

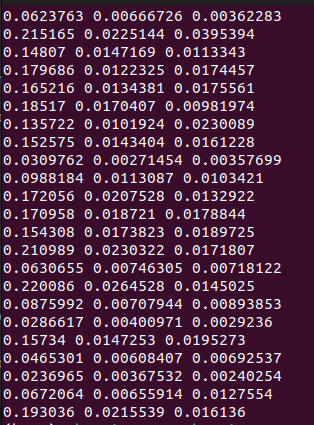
При запуске общего main.cpp файла получился следующий результат:



Собственно, это похоже на правду, так как внутри как раз работает 3 алгоритма с такими же затратами (число немного не круглое и меньше из-за переиспользования некоторых переменных). Поскольку ни один алгоритм не использует дополнительную память и видоизменяет исходный массив, делаем вывод, что все три варианта занимают O(n) памяти.

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 методе partition.

**III. Оценим время работы математически**

Более подробные доказательства можно посмотреть на Викиконспектах: [быстрая сортировка](https://neerc.ifmo.ru/wiki/index.php?title=%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) и [k-ая порядковая статистика](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_k-%D0%BE%D0%B9_%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%BE%D0%B2%D0%BE%D0%B9_%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B8).

Как мы можем убедиться, в худшем случае, асимптотика обоих алгоритмов составляет Θ(n^2) – когда все элементы в массиве равны.

При условии хорошего выбора опорного элемента на каждой итерации быстрой сортировки, как уже было сказано выше, отрезок массива будет распадаться примерно напополам. Это даёт нам мат.ожидание времени работы, равное O(n \* log(n)).

Из-за того, что в оптимизации kth\_hand\_made мы рассматриваем всегда только одну часть массива после разбиения, а во вторую никогда не переходим, мат.ожидание времени работы получается оценить как O(n) с большой константой при условии, что функция разбиения совершает внутри не более (n - 1) сравнений.

В файле calc.py приведён код, который считает, во сколько раз скорость работы kth\_standard и kth\_hand\_made меньше скорости наивного алгоритма. Ответы были усреднены для всех тестов.



Проверим, насколько это соответствует математическим расчётам. Если рассматривать n как число порядка 10^5-10^6, то логарифм n будет принимать значения от 16 до 20. Сделав поправку на то, что во время работы kth\_hand\_made может возникать большая константа (согласно доказательству с Викиконспектов, это число достигает 4-х), получается, что в нашем случае константа, в среднем, не превышает значение 2, что является достаточно хорошим показателем.

Также заметим, что рукописный вариант действительно вышел чуть быстрее встроенной k-ой порядковой статистики.

**IV. Выводы**

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

Также оказалось, что библиотечная реализация не настолько оптимальна, как рукописный вариант, хоть она и показала себя достаточно хорошо.

Итого, все поставленные задачи данной лабораторной работы были выполнены.