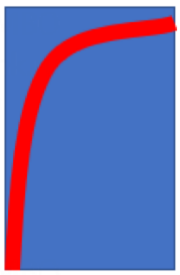
Лабораторная работа №1.

Описание:
Задание данной лабораторной работы заключалось в проверке наличия и нахождения элемента в двумерном массиве, отсортированном и по столбцам, и по строкам.

Задача:
Была поставлена задача проверить и сравнить время работы трех различных алгоритмов: линейного поиска (Linear search), бинарного поиска (Binary search) и экспоненциального поиска (Exponential search) на двух наборах входных данных.

- Первый набор (Ordinary table generation) – генерация данных, направленная на то, чтобы элементы, близкие по значению к тому, который нужно найти (target), находились на побочной диагонали матрицы.

- Второй набор (Hyperbolic table generation) – генерация данных, в которой элементы, близкие к target, находятся по индексам, примерно задающим гиперболу см. рисунок).



Ход работы:

Весь код был написан на языке C++, в среде разработки CLion. Тесты проводились на платформе Apple MacBook на процессоре M1. Все файлы с кодом находятся в репозитории на GitHub.

1. Были реализованы все функции поиска:

   На вход функции принимали текущую таблицу значений (двумерный вектор) и элемент, который требуется найти (value). Результатом работы является значение bool – найден элемент или не найден.

   - Linear search – поиск элемента, путем перехода к соседнему элементу. Начало в правом верхнем углу. Существует три случая:
     - текущий элемент равен target => мы нашли нужный элемент, заканчиваем алгоритм;
     - текущий элемент больше target: так как строки отсортированы по неубыванию, требуется проверить элемент, находящийся левее текущего => переходим на элемент, находящийся слева от текущего;
     - текущий элемент меньше target: заметим, что на прошлом шаге мы сдвигались влево до момента, когда текущий элемент не окажется меньше либо равен текущему => если target не найден, значит в этой строке он отсутствует.

Столбцы отсортированы по возрастанию, а значит элементы больше текущего находятся ниже него => переходим к следующей строке.

Сдвигаем текущий элемент до тех пор, пока его индексы не выйдут за границы (станут меньше нуля, либо станут больше или равны размера столбца / строки).

Реализация:

```cpp
bool LinearSearch(const std::vector<std::vector<int>>& data, const int& value) {

    size_t height = data.size(), width = data[0].size();
    size_t row = 0, column = width - 1;

    while (row < height) {
        if (data[row][column] == value) {
            return true;
        }

        if (data[row][column] < value) {
            ++row;
        } else {
            if (column == 0) {
                return false;
            }
            --column;
        }
    }

    return false;
}
```

- Binary search – поиск элемента похож на Linear search, однако сдвиг по строке влево теперь происходит путем перехода на ближайший элемент, найденный **бинарным** поиском. Существуют те же три случая, первый и третий аналогичны Linear search, отличается второй:
    - текущий элемент больше target: так как строки отсортированы по неубыванию, требуется проверить элементы, находящиеся левее текущего => запускаем бинарный поиск, где левая граница – нуль (минимальный возможны индекс), а правая – текущий столбец. Ищем элемент ближайший по значению к target и меньший чем он.

Сдвигаем текущий элемент до тех пор, пока его индексы не выйдут за границы (станут меньше нуля, либо станут больше или равны размера столбца / строки) или бинарный поиск не остановится на нулевом элементе, значение которого больше значения target (случай, когда по строке сдвигались влево до тех пор, пока не вышли за границу).

Релизация:

```cpp
bool BinarySearch(const std::vector<std::vector<int>>& data, const int& value) {

    size_t height = data.size(), width = data[0].size();
    size_t row = 0, column = width - 1;

    while (row < height) {
        if (data[row][column] == value) {
            return true;
        }

        if (data[row][column] < value) {
            ++row;
        } else {
            size_t left = 0, right = column;

            while (left < right) {
                size_t mid = (left + right + 1) / 2;
                if (data[row][mid] <= value) {
                    left = mid;
                } else {
                    right = mid - 1;
                }
            }

            if (data[row][left] == value) {
                return true;
            }
            if (left == 0 && data[row][left] > value) {
                break;
            }
            column = left;
        }
    }
    return false;
}
```

- Exponential search – поиск элемента похож на Linear search, однако сдвиг по строке влево теперь происходит путем перехода на ближайший элемент, найденный **экспоненциальным** поиском. Существуют те же три случая, первый и третий аналогичны Linear search, отличается второй:
    - текущий элемент больше target: так как строки отсортированы по неубыванию, требуется проверить элементы, находящиеся левее текущего => запускаем **экспоненциальный** поиск с текущего столбца и сдвигаемся на текущую степень двойки до тех пор, пока индекс не окажется меньше нуля или элемент по этому индексу не станет меньше target. Затем запускаем бинарный поиск, где найденный индекс – левая граница, а правая – найденный индекс + текущая степень двойки (самая обычная реализация экспоненциального поиска).

  Сдвигаем текущий элемент до тех пор, пока его индексы не выйдут за границы (станут меньше нуля, либо станут больше или равны размера столбца / строки) или экспоненциальный поиск не остановится на нулевом элементе, значение которого больше значения target (случай, когда по строке сдвигались влево до тех пор, пока не вышли за границу).

Релизация:

```cpp
bool ExponentialSearch(const std::vector<std::vector<int>> &data, const int &value) {

    size_t height = data.size(), width = data[0].size();
    size_t row = 0, column = width - 1;

    while (row < height) {
        if (data[row][column] == value) {
            return true;
        }

        if (data[row][column] < value) {
            ++row;
        } else {
            int step = 1;
            int current = static_cast<int>(column);

            while (current >= 0) {
                if (data[row][current] < value) {
                    break;
                }

                current -= step;
                step *= 2;
            }

            size_t left = std::max(current, 0), right =
                    std::min(static_cast<size_t> (current + step), data[row].size() - 1);

            while (left < right) {
                size_t mid = (left + right + 1) / 2;
                if (data[row][mid] <= value) {
                    left = mid;
                } else {
                    right = mid - 1;
                }
            }

            if (data[row][left] == value) {
                return true;
            }
            if (left == 0 && data[row][left] > value) {
                break;
            }
            column = left;
        }
    }

    return false;
}
```

2. Были реализованы функции генерации данных.

   На вход функция принимает значение текущей ширины таблицы, а высота задана нам по условию (она вынесена в константу WIDTH = 2^13). Результатом работы является двумерный вектор, в котором лежит сформированная таблица.

   - Ordinary table generation – генерация, в которой элементы близкие к target находятся на побочной диагонали.

Реализация:

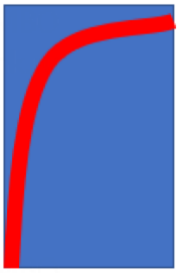
```cpp
std::vector<std::vector<int>> OrdinaryTableGeneration(const int& height) {
    int width = WIDTH;

    std::vector<std::vector<int>> result(height, std::vector<int>(width));

    for (int row = 0; row < height; ++row) {
        for (int column = 0; column < width; ++column) {
            result[row][column] = (width / height * row + column) * 2;
        }
    }

    return result;
}
```

- Hyperbolic table generation – генерация данных, в которой элементы, близкие к target, находятся по индексам, примерно задающим гиперболу (см. рисунок).



Реализация:

```cpp
std::vector<std::vector<int>> HyperbolicTableGeneration(const int& height) {
    int width = WIDTH;

    std::vector<std::vector<int>> result(height, std::vector<int>(width));

    for (int row = 0; row < height; ++row) {
        for (int column = 0; column < width; ++column) {
            result[row][column] = (width / height * row * column) * 2;
        }
    }

    return result;
}
```

3. Результаты запусков (все значения представлены в наносекундах).

Для каждой генерации данных были запущены каждый из трех алгоритмов и было замерено время.

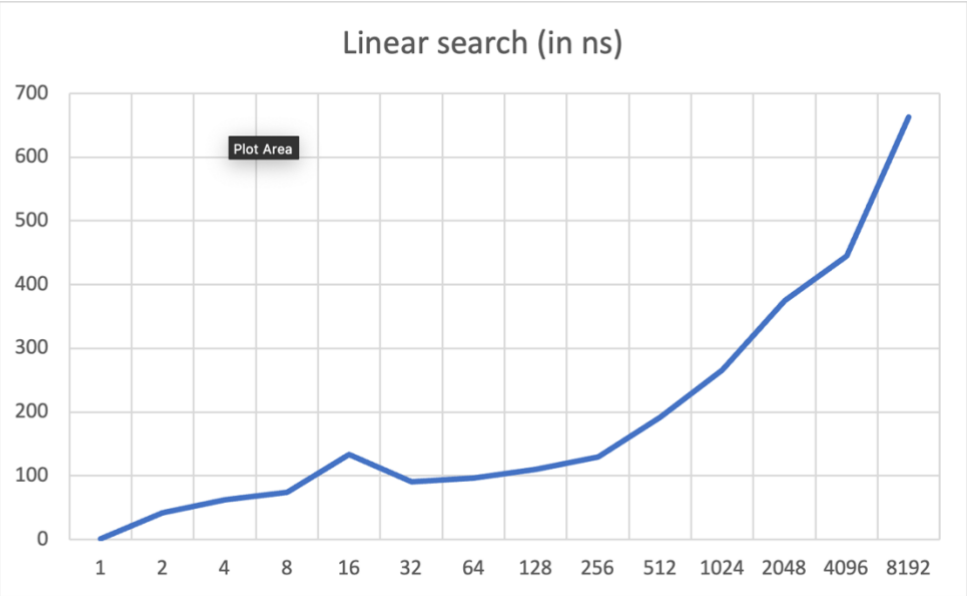
Ordinary table generation.

- Linear search for Ordinary table generation

  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  Можно заметить, что график практически стабильно неубывает (существуют неточности, вызванные погрешностями в измерениях). Однако при больших значениях время начинает резко возрастать в геометрической прогрессии.

WIDTH	Linear search
1	1,30233
2	41,93176
4	62,58538
8	74,32955
16	132,92122
32	90,12423
64	96,656
128	109,93639
256	129,22281
512	191,43233
1024	265,35182
2048	374,40905
4096	445,08097
8192	662,86406

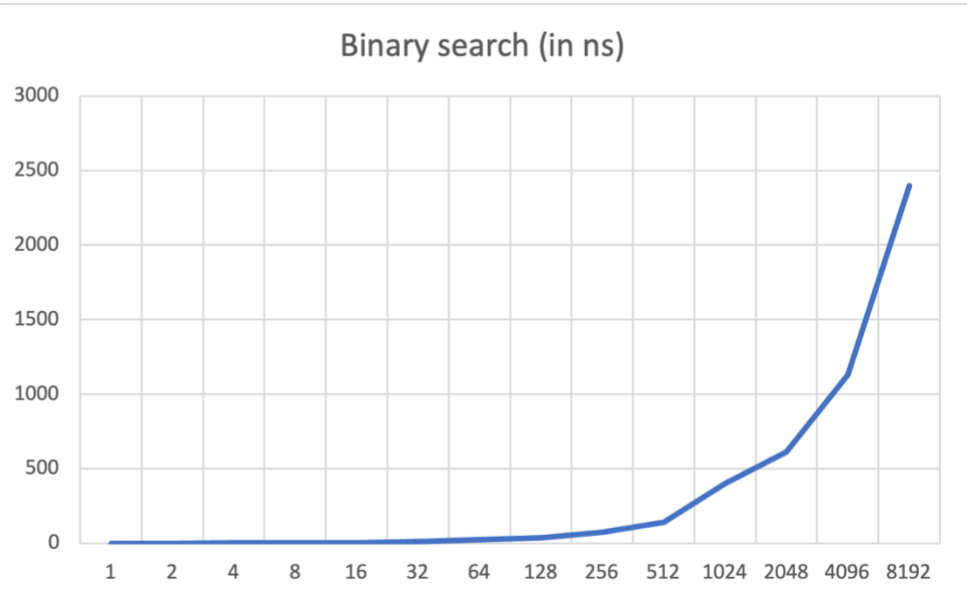


- Binary search for Ordinary table generation

  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  График лежит ниже, чем график Linear search, для маленьких значений, однако заметен резкий рост для больших значений.

WIDTH	Binary
1	0,06469
2	0,3128
4	0,87132
8	2,04537
16	4,34915
32	10,47072
64	24,25712
128	36,45489
256	71,66826
512	141,79509
1024	399,04825
2048	609,71632
4096	1127,37727
8192	2398,24684

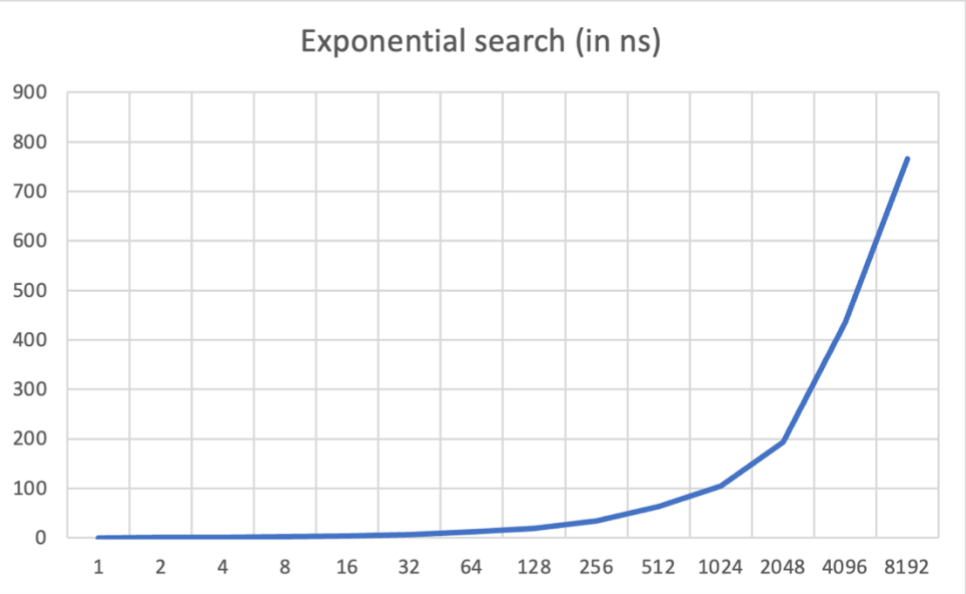


- Exponential search for Ordinary table generation

  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  Поведение графика похоже на Binary search. Однако для больших значений время растет не так быстро.

WIDTH	Exponential
1	0,05699
2	0,50855
4	1,44019
8	2,46564
16	3,79951
32	6,06339
64	12,06235
128	19,11332
256	34,27549
512	62,90034
1024	104,47523
2048	193,63345
4096	434,73959
8192	765,56962



- Linear vs Binary vs Exponential search for Ordinary table generation

  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  Можно заметить, что для маленьких данных оптимальным вариантом является Exponential search, однако на больших данных он уступает Linear search.

  Binary search – самый медленный.

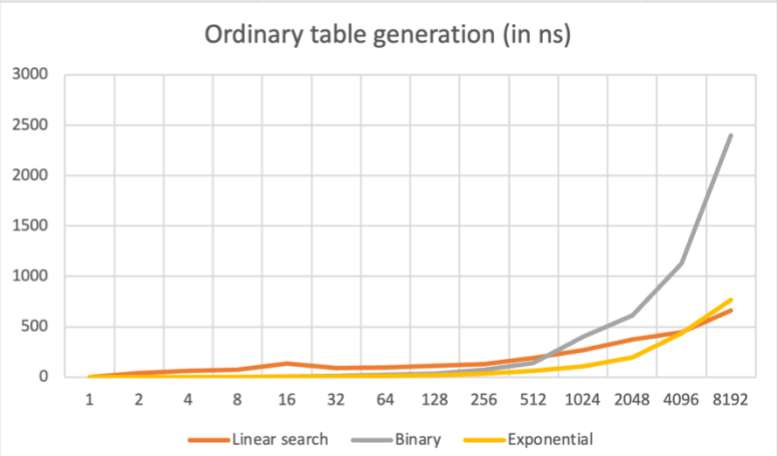
  Причина таких результатов заключается в том, что при такой генерации (элементы близкие по значению target лежат на побочной диагонали) мы чередуем шаги влево и вниз. Соответственно, вниз каждый алгоритм переходит за O(1), а влево – все по разному.

  Linear search сдвигается на 1 элемент влево за O(1).

  Binary search за $O(\log_2(current\_width))$, где current_width – это количество элементов левее текущего.

  Exponential search тоже за O(1), однако засчет большого количества проверок и операций константа тут больше, что становится заметно на больших тестах.

WIDTH	Linear search	Binary	Exponential
1	1,30233	0,06469	0,05699
2	41,93176	0,3128	0,50855
4	62,58538	0,87132	1,44019
8	74,32955	2,04537	2,46564
16	132,92122	4,34915	3,79951
32	90,12423	10,47072	6,06339
64	96,656	24,25712	12,06235
128	109,93639	36,45489	19,11332
256	129,22281	71,66826	34,27549
512	191,43233	141,79509	62,90034
1024	265,35182	399,04825	104,47523
2048	374,40905	609,71632	193,63345
4096	445,08097	1127,3773	434,73959
8192	662,86406	2398,2468	765,56962



Hyperbolic table generation.

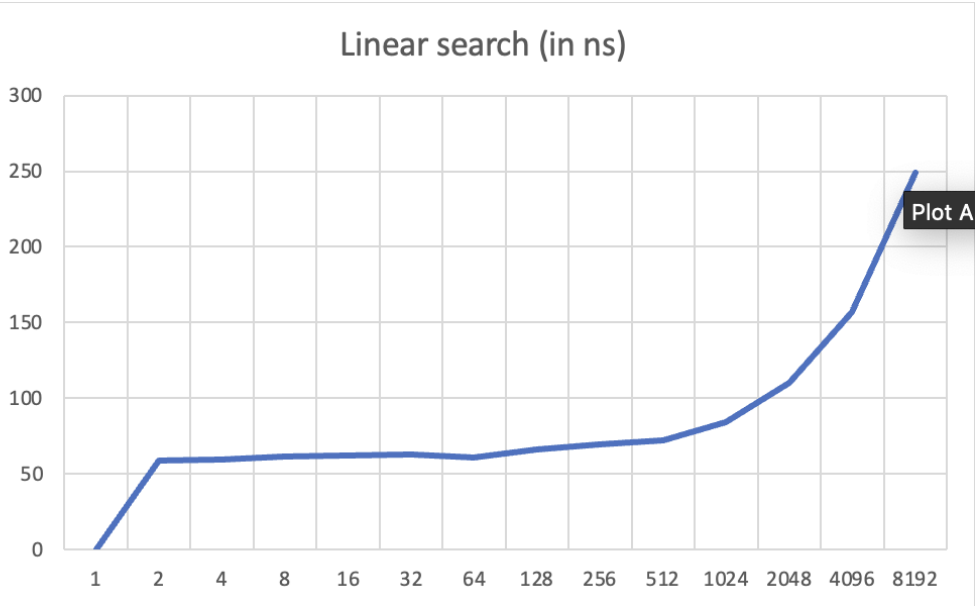
- Linear search for Hyperbolic table generation

  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  Заметен резкий скачок во времени в самом начале, происходящий из-за количества операций сдвига влево (необходимо пройти практически всю таблицу). Заметим, что и сложность этого алгоритма составляет O(height + width).

  Затем график возрастает.

WIDTH	Linear search
1	0,07875
2	59,01025
4	59,24283
8	61,33593
16	62,17048
32	62,54971
64	60,74445
128	66,37693
256	69,63512
512	72,13407
1024	84,47463
2048	110,42672
4096	157,22132
8192	249,19805

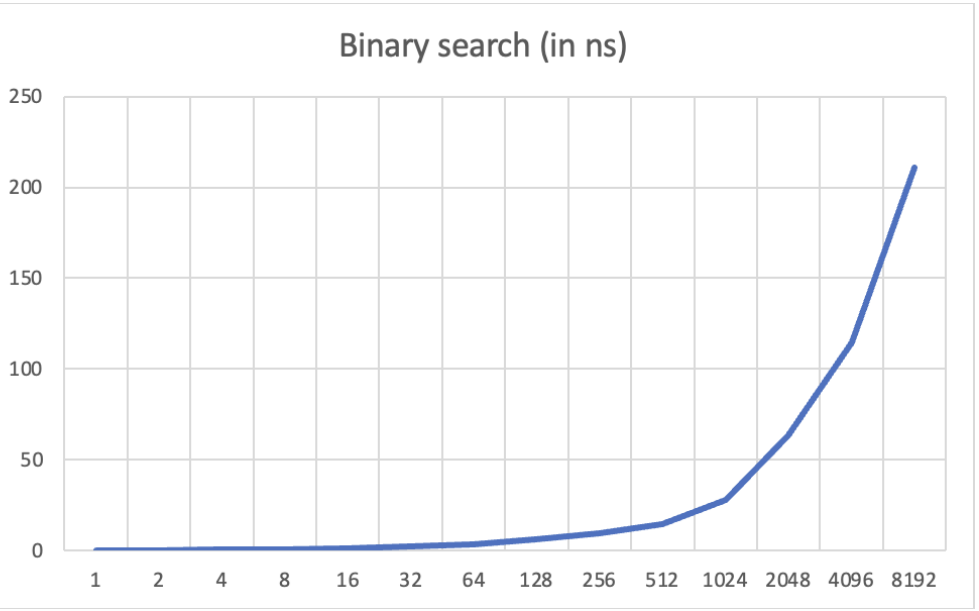


- Binary search for Hyperbolic table generation

  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  В случае Binary search, все идет стабильно – для маленьких значений алгоритм работает быстро, для больших – дольше.

WIDTH	Binary
1	0,04554
2	0,16779
4	0,33812
8	0,63506
16	1,29519
32	2,35592
64	3,50363
128	5,86087
256	9,62305
512	14,69305
1024	27,95988
2048	63,33043
4096	114,49868
8192	210,80451

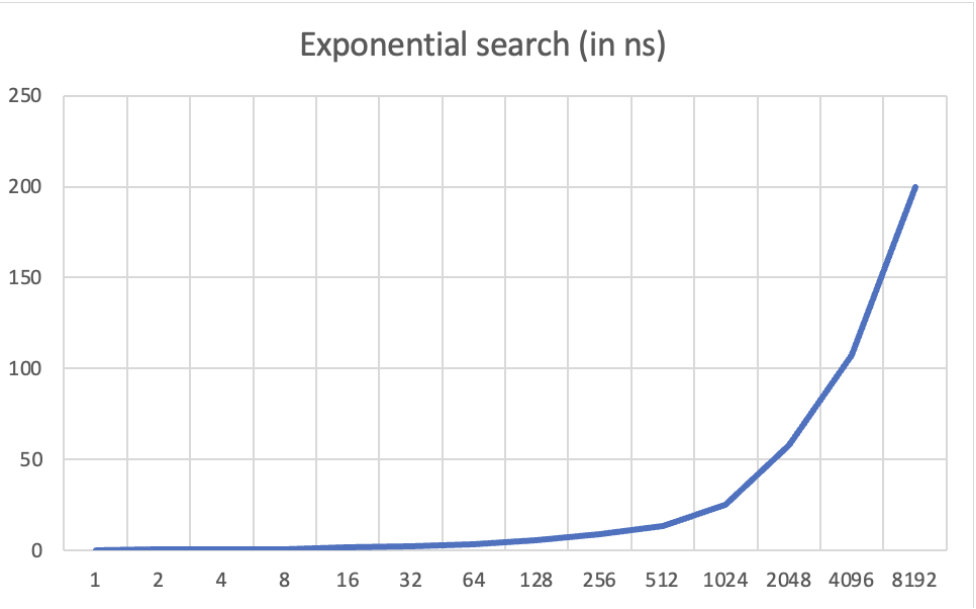


- Exponential search for Hyperbolic table generation

  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  График похож на график Binary search, однако он возрастает чуть медленнее.

WIDTH	Exponential
1	0,04481
2	0,34407
4	0,53926
8	0,84799
16	1,41341
32	2,3653
64	3,56946
128	5,84897
256	8,76985
512	13,48707
1024	24,91987
2048	58,14458
4096	107,30595
8192	199,87687

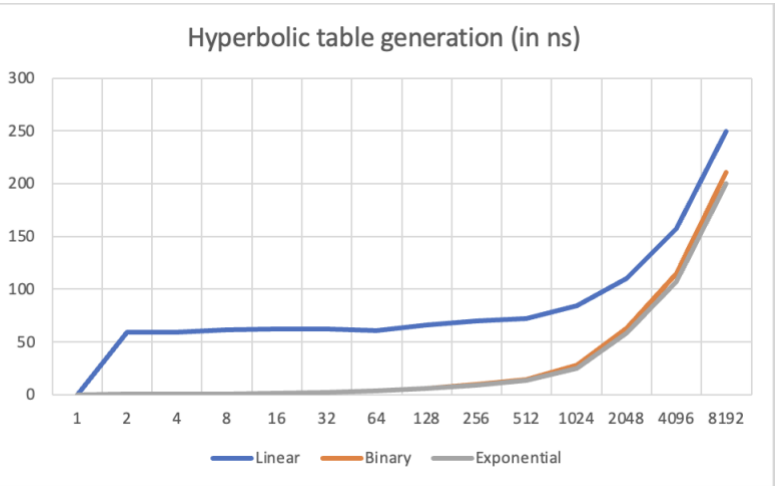


- Linear vs Binary vs Exponential search for Hyperbolic table generation
  В таблице слева представлены значения ширины таблицы и время в наносекундах, соответствующее данной ширине. Справа – график, показывающий зависимость времени от ширины таблицы.

  Заметим, что ситуация в случае Hyperbolic table generation показывает обычное соотношение времени для Linear, Binary and Exponential searches, основанное на вычислительной сложности.
  Linear search проходит всю таблицу слева направо по одному элементу => O(width).
  Binary search проходит всю таблицу слева направо за $O(\log_2(width))$.
  Exponential search проходит всю таблицу слева направо чуть меньше $O(\log_2(width))$ – примерно $O(\log_2(width / height))$.

  $O(width) > O(\log_2(width)) > O(\log_2(width / height))$
  График является подтверждением этому.

WIDTH	Linear	Binary	Exponential
1	0,07875	0,04554	0,04481
2	59,01025	0,16779	0,34407
4	59,24283	0,33812	0,53926
8	61,33593	0,63506	0,84799
16	62,17048	1,29519	1,41341
32	62,54971	2,35592	2,3653
64	60,74445	3,50363	3,56946
128	66,37693	5,86087	5,84897
256	69,63512	9,62305	8,76985
512	72,13407	14,69305	13,48707
1024	84,47463	27,95988	24,91987
2048	110,42672	63,33043	58,14458
4096	157,22132	114,49868	107,30595
8192	249,19805	210,80451	199,87687



4. **Выводы.**

В данной лабораторной работе мы протестировали и сравнили по времени три алгоритма поиска элемента в таблице (Linear, Binary and Exponential search), отсортированной и по строкам, и по столбцам. Данные генерировались двумя способами: Ordinary and Hyperbolic generation.

Были получены следующие результаты:

- На маленьких тестах Linear search проигрывает Binary and Exponential search.
- Linear search, несмотря на большую complexity = O(n + m), показывает неплохие результаты и может обогнать и Binary, и Exponential searches, например, на тестах, когда элементы близкие к target находятся на побочной диагонали.
- Время работы Linear search может расти как линейно, так и квадратично.
- Время работы Binary search растет пропорционально увеличению входных данных (width увеличивается в 2 раза => time увеличивается в 2 раза).
- Время работы Exponential search при увеличении width в 2 раза растет в среднем в 1,75 раз.

В дополнение, мы попрактиковались в написании проекта на C++, а также пользовании Гитом. Ссылка еще раз: https://github.com/Kaparya/Algorithms-and-Data-structures.git