**Лабораторная работа 1. Алгоритмы поиска в матрице.**

## **1. Программное обеспечение**

* Запускалось на ноутбуке macbook pro 2017 15
* Mac os ventura
* node.js v19.0.0
* npm v8.19.2

**2. Код**

# **2.1 Реализация трех алгоритмов**

| const bin\_search\_on\_rows = (A, target) => {  for (const row of A) {  let l = 0;  let r = row.length - 1;    while (r - l > 0) {  let m = div(r + l,2);  if (row[m] < target) {  l = m + 1;  } else {  r = m;  }  }   if (row[r] === target) {  return true  }  }  return false; };  const ladder\_solve = (A, target) => {  let i = 0;  let j = A[0].length - 1;   while(i < A.length && j >= 0) {  if (A[i][j] == target) {  return true;  } else if (A[i][j] < target) {  i++;  } else {  j--;  }  }  return false; }  const ladder\_exp\_solve = (A, target) => {  const N = A.length;  const M = A[0].length;    let i = 0;  let j = M - 1;   while (i < N && j >= 0) {  if (A[i][j] == target) {  return true;  }  if (A[i][j] < target) {  i++;  } else {  bound = 1;  while (j - bound >= 0 && A[i][j - bound] >= target) {  bound \*= 2;  }   l = Math.max(0, j - bound - 1);  r = j - div(bound,2);   while (r - l > 1) {  m = div(l + r, 2);  if (A[i][m] >= target) {  r = m;  } else {  l = m;  }  }   if (A[i][r] == target) {  return true;  } else {  j = r;  i++;  }  }  }  return false; } |
| --- |

# **2.2 Реализация двух генераций**

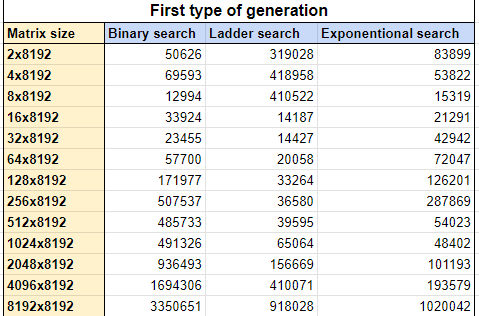
| const generate\_data\_set = (M, N) => {  const DATA\_SET\_NUM = process.env['DATASET'];   let a = new Array(M);  for (let i = 0; i < M; i++) {  a[i] = new Array(N);  }   let target = 0;   if (DATA\_SET\_NUM === '1') {  for (let i = 0; i < M; i++) {  for (let j = 0; j < N; j++) {  a[i][j] = (Math.floor(N/M) \* i + j) \* 2;  }  }  target = 2\*N + 1;  } else if (DATA\_SET\_NUM === '2') {  for (let i = 0; i < M; i++) {  for (let j = 0; j < N; j++) {  a[i][j] = (Math.floor(N/M) \* i \* j) \* 2;  }  }  target = 16\*N + 1;  } else {  throw new Error("DATASET env variable required. Try DATASET=1 npm run start")  }  return [target, a] } |
| --- |

# **2.3 Кусок кода запуска**

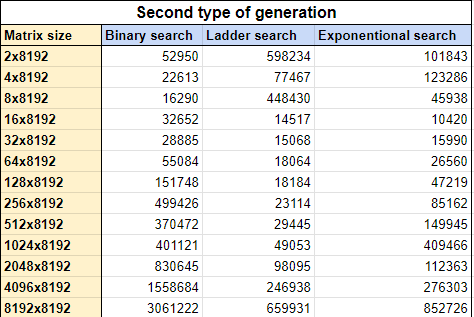
| const calculate\_results = (x) => {  const N = Math.pow(2, 13);  const M = Math.pow(2, x);  const NUMBER\_OF\_LAUNCHES = 95;   process.stdout.write(M + " ");   const [target,A] = generate\_data\_set(M, N);    let start = 0;  let measure\_time = 0;  let average\_time = 0;  for (let i = 0; i < NUMBER\_OF\_LAUNCHES; i++) {  start = process.hrtime();  bin\_search\_on\_rows(A, target);  measure\_time = process.hrtime(start)[1];  average\_time += measure\_time;  }  average\_time /= NUMBER\_OF\_LAUNCHES;   process.stdout.write(average\_time + " ");   average\_time = 0;  for (let i = 0; i < NUMBER\_OF\_LAUNCHES; i++) {  start = process.hrtime();  ladder\_solve(A, target);  measure\_time = process.hrtime(start)[1];  average\_time += measure\_time;  }  average\_time /= NUMBER\_OF\_LAUNCHES;   process.stdout.write(average\_time + " ");   average\_time = 0;  for (let i = 0; i < NUMBER\_OF\_LAUNCHES; i++) {  start = process.hrtime();  ladder\_exp\_solve(A, target);  measure\_time = process.hrtime(start)[1];  average\_time += measure\_time;  }  average\_time /= NUMBER\_OF\_LAUNCHES;   process.stdout.write(average\_time + "\n"); };  const main = () => {  for (let x = 1; x < 14; x++) {  calculate\_results(x);  } };  main(); |
| --- |

# **3. Результаты запусков**

## **3.1. Три алгоритма на первых данных**



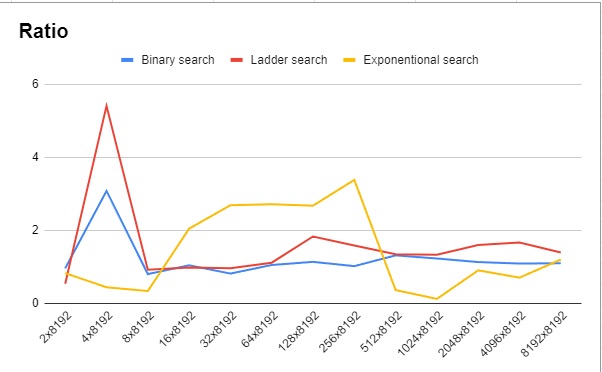
**3.2. Три алгоритма на вторых данных**

****

# **4. Визуализация**

# 

## **Отношение времени работы на первом типе генерации матриц ко времени работы на втором типе генерации матриц**



# **5. Выводы**

## **5.1 Первые два графика.**

Решение бинарным поиском отстает от двух других, что и понятно из функций асимптотики данных алгоритмов.

Разберемся с решениями лесенкой и экспоненциальным на первых данных. На самом деле, при маленьких M, они не сильно отличаются друг от друга, так как самый большой элемент матрицы (A[M-1][N-1] = (N/M\*(M-1) + N - 1)\*2 = N\*2 - N/M\*2 - 2) будет меньше таргета 2N + 1, а значит наши алгоритмы пройдутся из правого верхнего угла до правого нижнего угла за линейное время. Отличие в графиках объясняется сборкой мусора на движке v8 для Javascript.

На вторых данных лесенка и экспоненциальный показали разные результаты при маленьких M. Разберемся почему. Самый большой элемент в матрице для вторых данных - это A[M-1][N-1] = N/M\*(M-1)(N-1)\*2 = (N - N/M)(N - 1)\*2 = (N^2 - N - N^2/M + N/M) \* 2. Заметим, что target = 16N + 1 будет находится внутри матрицы начиная с некоторого M. Путем несложных вычислений, получаем, что с x > 0 наш таргет будет меньше максимального элемента матрицы, а для наших данных (x лежит в отрезке [1, 13]) это выполняется всегда. Таким образом, наш экспоненциальный поиск на при маленьких M (на графике от 2 до 1000) будет превосходить над линейным по строке. Это и видно на самом графике. А после преобладает линейный, так как экспоненциальный поиск хорош только в случае, когда данные лежат в начале поиска, что в нашем случае не валидно, так как данные при некоторых M лежат далеко внутри матрицы, что и дает фору для линейного поиска.

## **5.2. Разберем третий график.**

На первых данных таргета просто нет в матрице (таргет больше самого максимального элемента) при маленьких M, тогда он просто идёт по правому краю (от левого верхнего угла до правого нижнего), а во втором дата сете он появляется за счёт того, что данные очень быстро растут. Так, он начинает запускать экспоненциальный поиск по строке, что замедляет его по сравнению с первыми данными, который просто идет по столбцу линейным поиском. Далее лидерство переходит экспоненциальному по вторым данным, так как в первом таргет уже находится внутри матрицы. Также во вторых данных матрица симметрична (A[i][j] == A[j][i]), поэтому, таргет будет искаться в правой половине матрицы, что значительно ускоряет его по сравнению с первым набором данных.