Алгоритмы и структуры данных

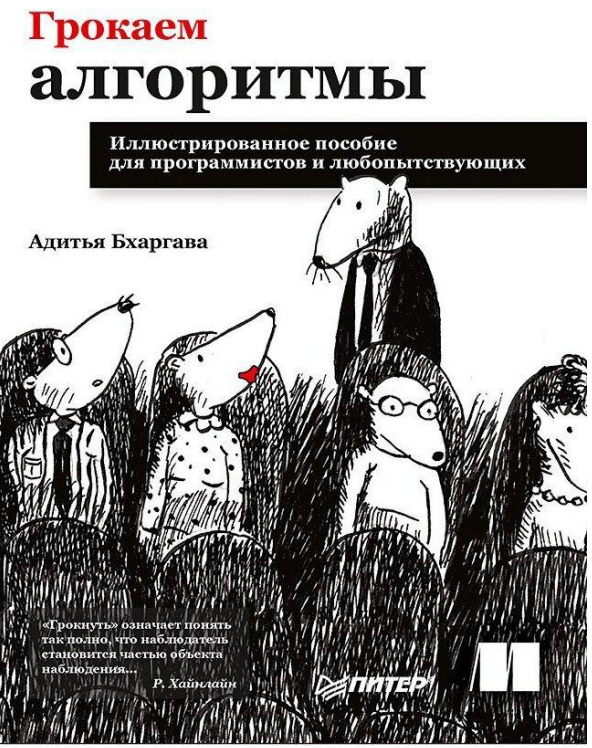
Доцент, к.т.н. Дёмин Антон Юрьевич

Учебный план

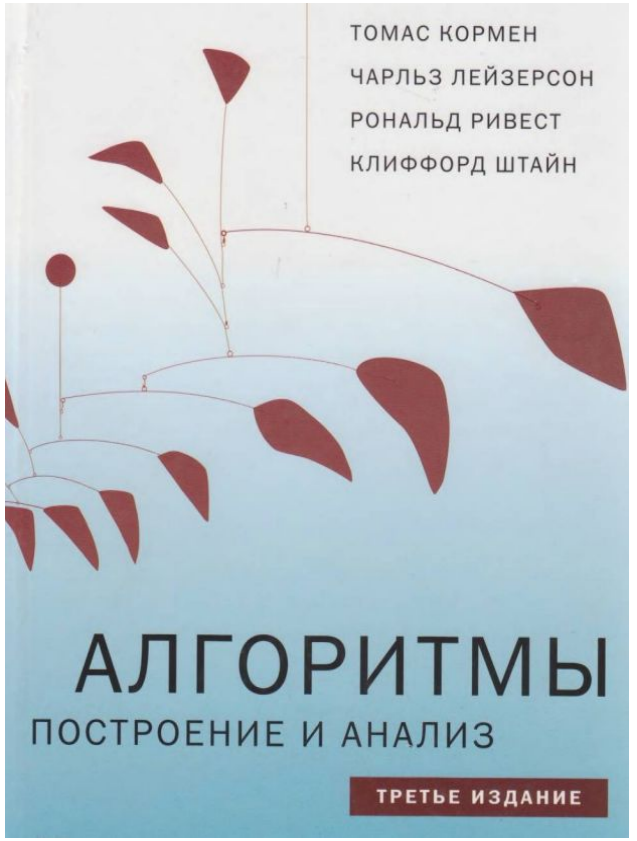
- Лекции – 32 часа
- Лабораторные работы – 32 часа (8 лабораторных работ)
  - Оценка сложности алгоритмов
  - Алгоритмы поиска
  - Сортировка
  - Простые типы данных
  - Стек, Двух связанные списки, Очереди
  - Имитационное моделирование на основе очередей
  - Инвертированные индексы
  - Двоичные деревья
- Зачет/Экзамен

Литература


РОБЕРТСЕДЖВИК Принстонский Университет
Фундаментальные алгоритмы
третья редакция
на C++
Части1-4


Грокаем
алгоритмы
Иллюстрированное пособие для программистов и любопытствующих
Адитья Бхаргава
«Грокнуть» означает понять так полно, что наблюдатель становится частью объекта наблюдения...
Р. Хайнлайн
ПИТЕР


АЛГОРИТМЫ
Вводный курс
Томас Х. Кормен


ТОМАС КОРМЕН
ЧАРЛЬЗ ЛЕЙЗЕРСОН
РОНАЛЬД РИВЕСТ
КЛИФФОРД ШТАЙН
АЛГОРИТМЫ
ПОСТРОЕНИЕ И АНАЛИЗ
ТРЕТЬЕ ИЗДАНИЕ


Н. Вирт
Алгоритмы и структуры данных

Алгоритм

- **Алгоритм** – это формально описанная вычислительная процедура, получающая **входные данные** (input), и выдающая **результат** на выход (output)
- **Алгоритм –** точное предписание, которое задает **вычислительный процесс**, начинающийся с произвольного **исходного данного** и направленный на получение полностью определенного этим исходным данным **результата**.
- **Алгоритм** – это четкое описание по выполнению некоторого **процесса обработки данных**, который через разумное конечное число шагов приводит к **решению задачи** данного типа для любых допустимых **вариантов исходных данных**.

Свойства алгоритмов

- **Дискретность (прерывность)** - алгоритм должен представлять процесс решения задачи как последовательное выполнение простых шагов.
- **Детерминированность. Определенность** - каждое правило алгоритма должно быть четким, однозначным и не оставлять места для вариаций.
- **Результативность (конечность)** - алгоритм должен приводить к решению задачи за конечное число шагов.
- **Массовость** - алгоритм решения задачи разрабатывается в общем виде и должен быть применим для некоторого класса задач, различающихся только входными данными. При этом входные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма

Структура данных

- **Структура данных** – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.
- **Типичные операции:**
  - добавление данных;
  - изменение данных;
  - удаление данных;
  - поиск данных.

Анализ алгоритмов

- Эффективность алгоритмов определяется различными характеристиками, зависящими от исходных данных (размерность обозначим как ***n***):
  - Время работы ***T***(***n***);
  - Количество выполняемых операций (кол-во арифметических операций, операций сравнений, операций обращения к диску и т.п.);
  - Объемом используемой памяти ***M***(***n***);
  - Адаптируемость алгоритма к различным компьютерам;
  - Простота, изящество.

Анализ трудоемкости алгоритма

- Целью анализа трудоёмкости алгоритмов является нахождение оптимального алгоритма для решения данной задачи.

Вычислительная сложность алгоритма

- **Вычислительная сложность алгоритма** — *это функция, определяющая зависимость объёма работы, выполняемой некоторым алгоритмом, от свойств входных данных. Объём работы обычно измеряется абстрактными понятиями времени и пространства, называемыми* ***вычислительными ресурсами****.*
- **Время** определяется количеством элементарных шагов, необходимых для решения проблемы, тогда как **пространство** определяется объёмом памяти или места на носителе данных.
- Центральный вопрос разработки алгоритмов: «как изменится время исполнения и объём занятой памяти в зависимости от размера входа и выхода?»

Асимптотический анализ

«По сути, задача их сводилась к анализу кривой относительного познания в области ее асимптотического приближения к абсолютной истине.» **А. и Б. Стругацкие. Понедельник начинается в субботу.**

Асимптотический анализ — метод описания предельного поведения функций.

Например, в функции $f(n)=n^2+3n$ при стремлении n к бесконечности слагаемое $3n$ становится пренебрежительно малым, поэтому про функцию $f(n)$ говорят, что она асимптотически эквивалентна n^2, при $n \to \infty$ или записывают как $f(n) \sim n^2$

Асимптотическая сложность алгоритмов

- **Асимптотическая сложность** (асимптотическое описание временной сложности) - оценка скорости роста времени работы алгоритмов, предназначенных для решения одной и той же задачи, при больших объемах входных данных.

- **Количество элементарных операций**, затраченных алгоритмом для решения конкретного экземпляра задачи, зависит не только ***от размера входных данных***, но и ***от самих данных***.

- **Асимптотическая сложность алгоритмов** обычно рассматривается для ***худшего случая*** входных данных, т.е. сами данные приводят к наибольшему числу элементарных операций. Например, при сортировке пузырьком, худший случай представляют исходные данные отсортированные в обратном порядке.

- Асимптотическая сложность алгоритмов записывается через нотацию большого ***O***, или ***Big O Notation***

Хороший, плохой, средний

- **Худший случай (worst case)** — это когда входные данные требуют максимальных затрат времени и памяти.

- **Лучший случай (best case)** — полная противоположность worst case, самые удачные входные данные. Пример: В случае поиска — когда алгоритм находит нужный элемент с первого раза.

- **Средний случай (average case) —** это алгоритм, который выполняет среднее количество шагов над входными данными из ***n*** элементов. Пример: В случае поиска элемент находится в середине массива, либо проводится ряд вычислительных экспериментов со случайными данными.

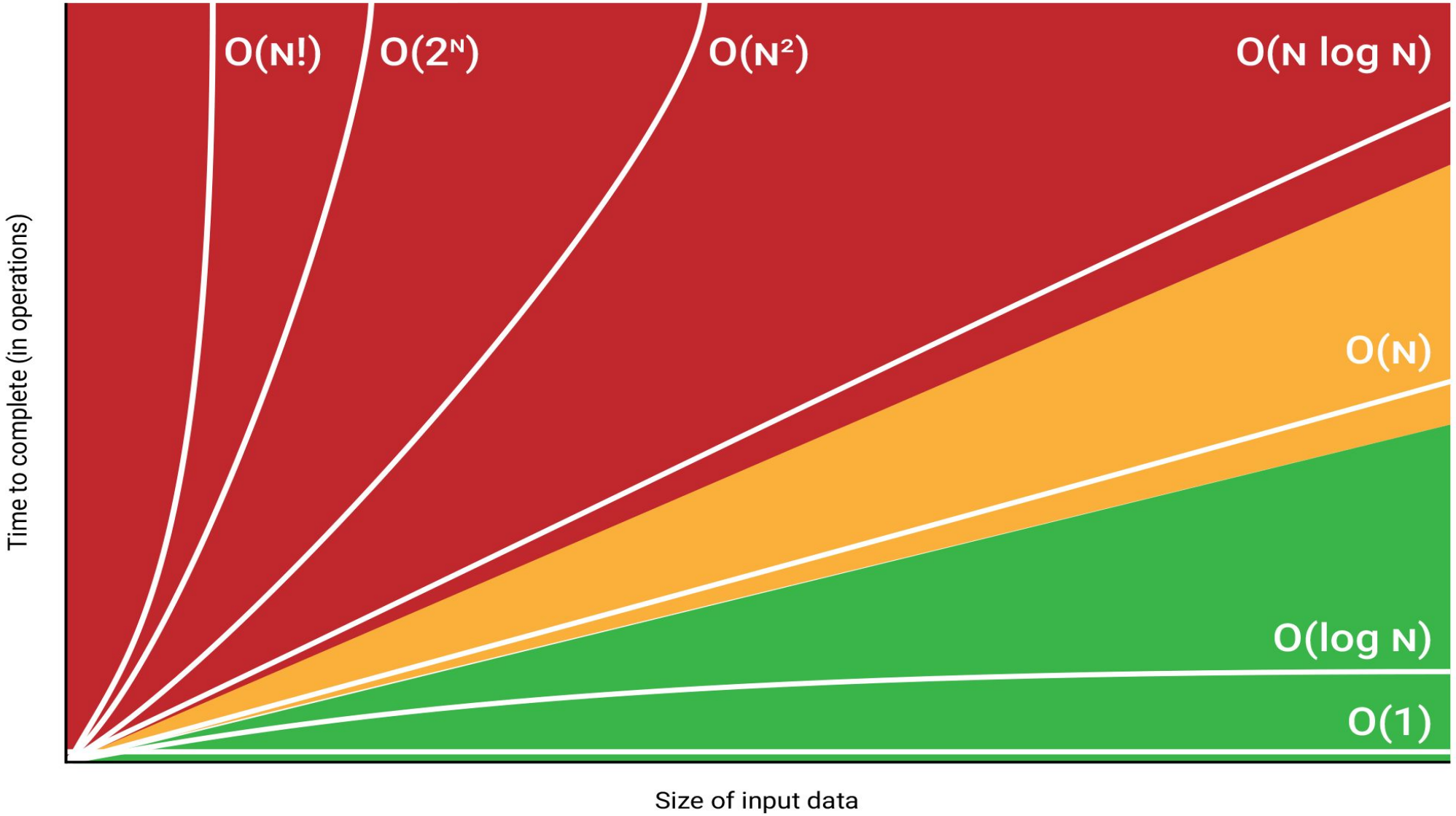
Big O Notation

- **Big O** обозначает верхнюю границу сложности алгоритма.
  Это идеальный инструмент для поиска ***worst case***.

- **Big Omega** (**Ω**) обозначает нижнюю границу сложности,
  и её правильнее использовать для поиска ***best case***.

- **Big Theta** (**Θ**) располагается между ***O*** и омегой
  и показывает точную функцию сложности алгоритма.
  С её помощью правильнее искать ***average case***.

Big O complexity Chart



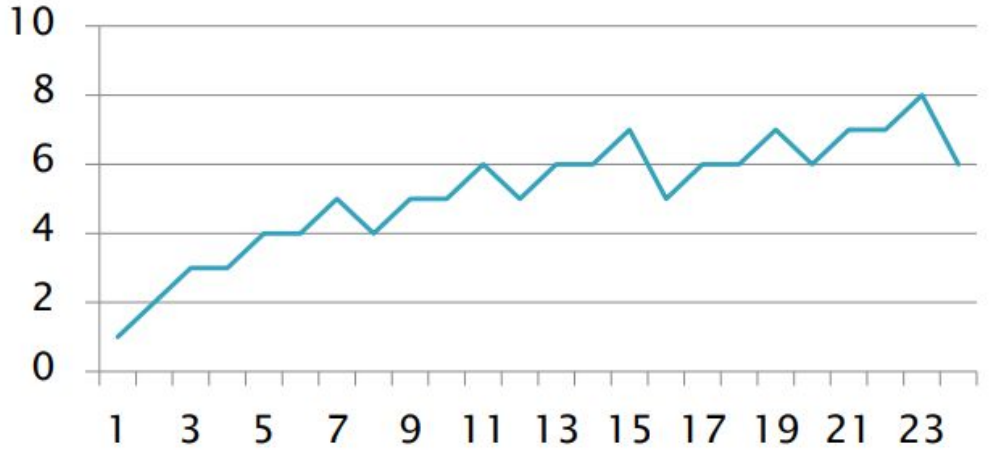
Градации сложности алгоритмов (1)

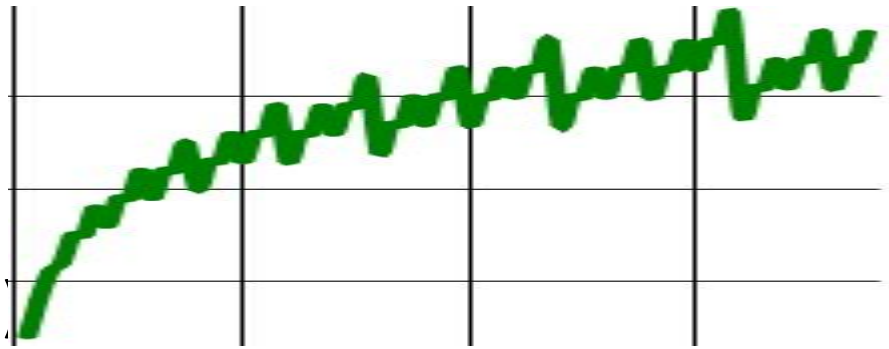
- $f(n) = O(1)$ – константная
  - Примеры:
    - алгоритм, обращения к элементу массива
    - оператор присвоения с арифметическими вычислениями
    - Алгоритм расчета суммы первых 10 элементов массива
- $f(n) = O(log\ n)$ – логарифмическая
  - Примеры:
    - бинарный поиск в отсортированном массиве
    - алгоритмы быстрого возведения в степень

Алгоритм быстрого возведения в степень

- Алгоритм предназначен для возведения числа ***x*** в натуральную степень ***n***. При этом не обязательно перемножать число ***x n*** раз. Используется свойство $x^{2n} = (x^2)^n$

```
long mypow(long x, long n)
    {
    long b = 1;
    while (n != 0)
        {
        if (n % 2 == 0) { n /= 2; x *= x; }
          else { n--; b *= x; }
        }
    return b;
    }
```





- Сложность выполнения алгоритма $O\ log_2(n)$

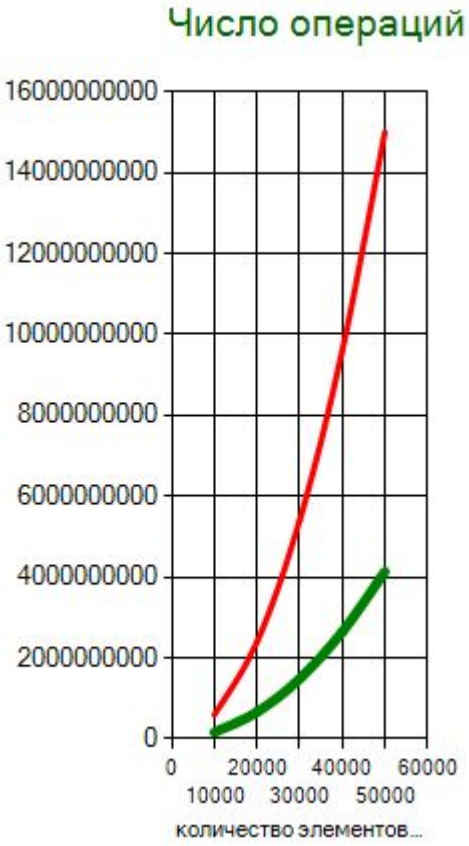
Градации сложности алгоритмов (2)

- $f(n) \sim O(n^{1/2})$ – сублинейная
  - Примеры:
    - Поиск Гровера
- $f(n) \sim O(n)$ – линейная
  - Примеры:
    - перебор всех элементов массива, матрицы, в том числе линейный поиск
    - «оптимизированный» алгоритм вычисления числа Фибоначчи (без рекурсии)
    - простая рекурсия для вычисления суммы:

```
int sum(int n)
  {
  if (n == 1) return 1;
  return n + sum(n - 1);
  }
```

- $f(n) = O(n \log n)$ – линейно-логарифмическая
  - Примеры:
    - сортировка с помощью двоичного дерева
    - Алгоритм QuickSort
- $f(n) \sim O(n^k)$ – полиномиальная (задачи класса ***P***)
  - Примеры:
    - Сортировка пузырьком (n^2), сортировка вставками (n^2) (субквадратичная сложность)
    - Обычное умножение матриц nxn (n^3)

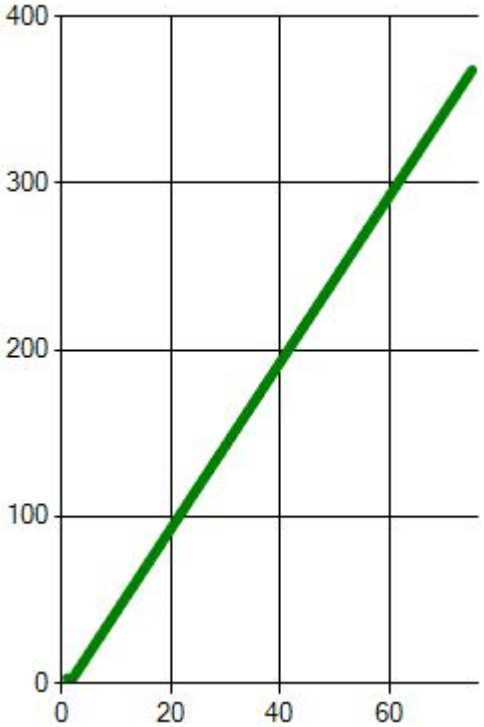
- Buble sort
- Insert sort



Алгоритм вычисления числа Фибоначчи

- $f3 = f1 + f2 \rightarrow f4 = f2 + f3 \rightarrow \cdots fn = fn-2 + fn-1$

```
double Fibonacci(int IndexNumber)
    {
      double f1, f2, f3; //Объявляем переменные
      f1 = f2 = 1.0; //Задаем известные значения для 1го и 2го числа
      f3 = 0.0;
      int i = 3; //Объявляем и задаем значение переменной шага
      while (i <= IndexNumber)//Пока не достигнем необходимого номера числа Фибоначчи
       {
        f3 = f2 + f1; // Определяем I ое число по формуле
                  //Меняем для следующего шага переменные
        f1 = f2; //f2 становится f1
        f2 = f3; //f3 становится f2
        i++; //Увеличиваем шаг (номер числа) на единицу
       }
      return f3; //Возвращаем последнее посчитанное значение
    }
```



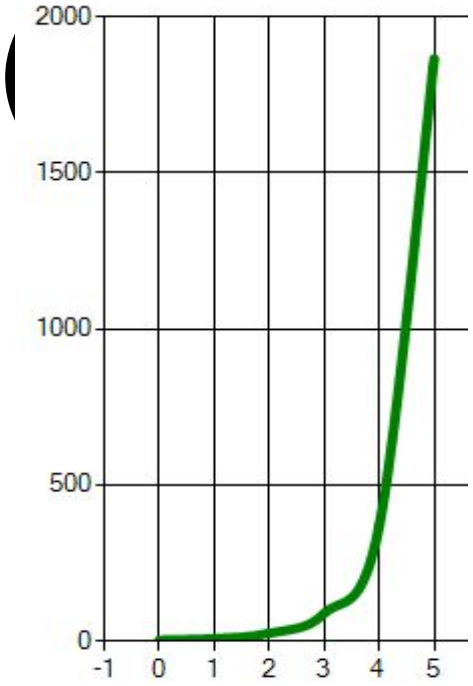
- Сложность выполнения алгоритма $O(n)$

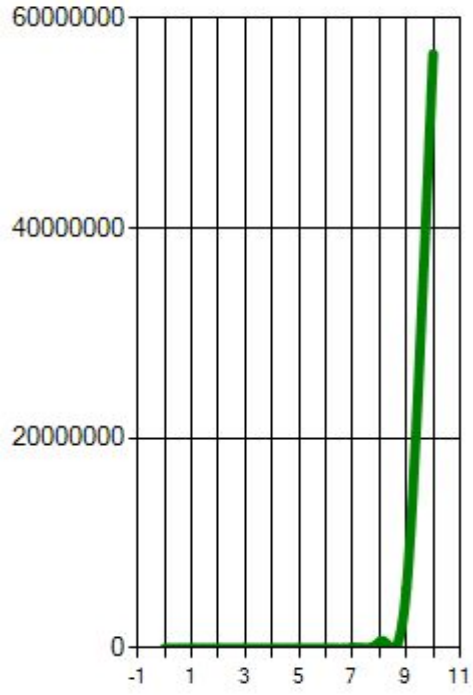
Градации сложности алгоритмов (

- $f(n) = O(\boldsymbol{c^n})$ – экспоненциальная
  - Примеры:
    - Нативный рекурсивный алгоритм вычисления чисел Фибоначчи
    - В криптографии атака методом "грубой силы"

- $f(n) = O(\mathrm{n!})$ – факториальная
  - Примеры:
    - задача коммивояжёра
    - Поиск всех перестановок или размещений
    - Вычисление факториала через рекурсию:

```
int Factorial (int n)
{   int num = n;
    if (n == 0) return 1;
    for (int i = 0; i < n; i++)
        num = n * Factorial(n - 1);
    return num;}
```



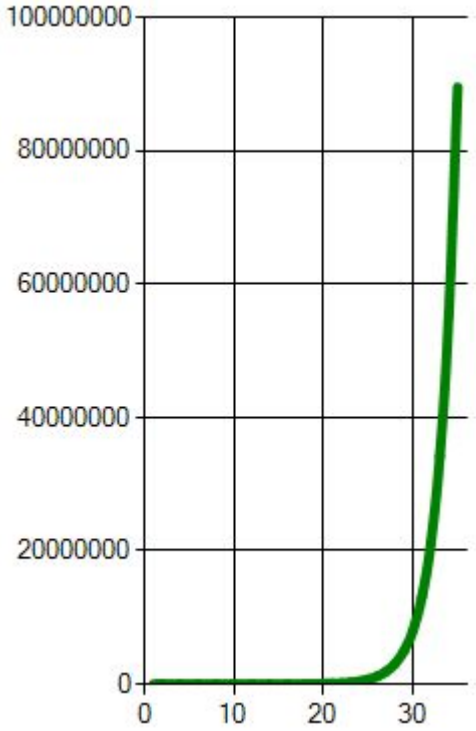


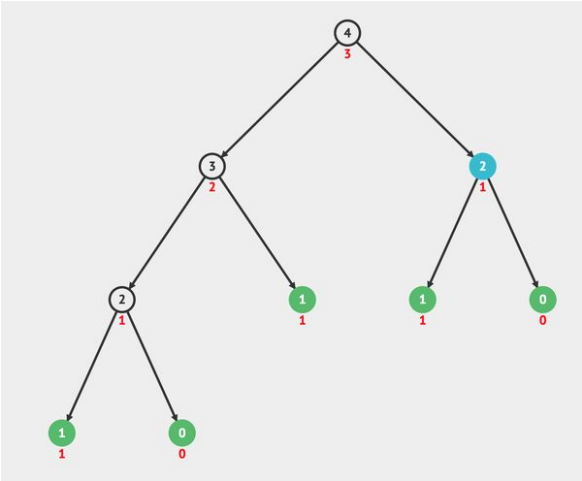
Фибоначчи на собеседовании https://habr.com/ru/articles/4496

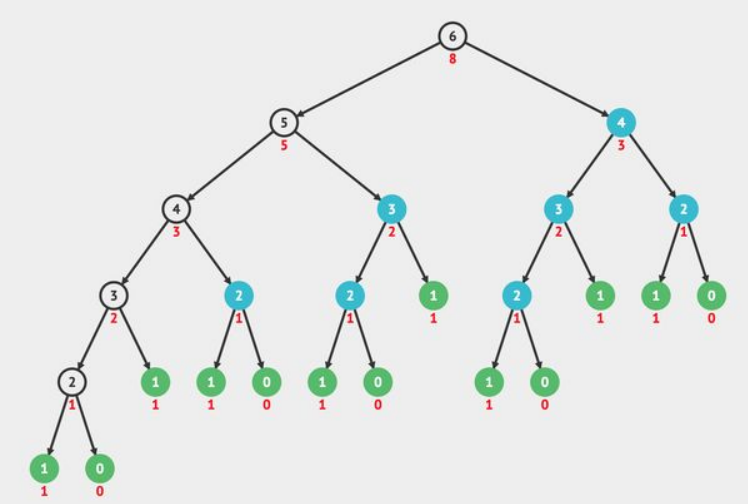
Рекурсивный алгоритм вычисления числа Фибоначчи

```
int RFibonacci(int n)
    {
    if (n <= 1) return n;
    else return (RFibonacci(n - 2) + RFibonacci(n - 1);
    }
```

- Сложность выполнения алгоритма $O(2^n)$

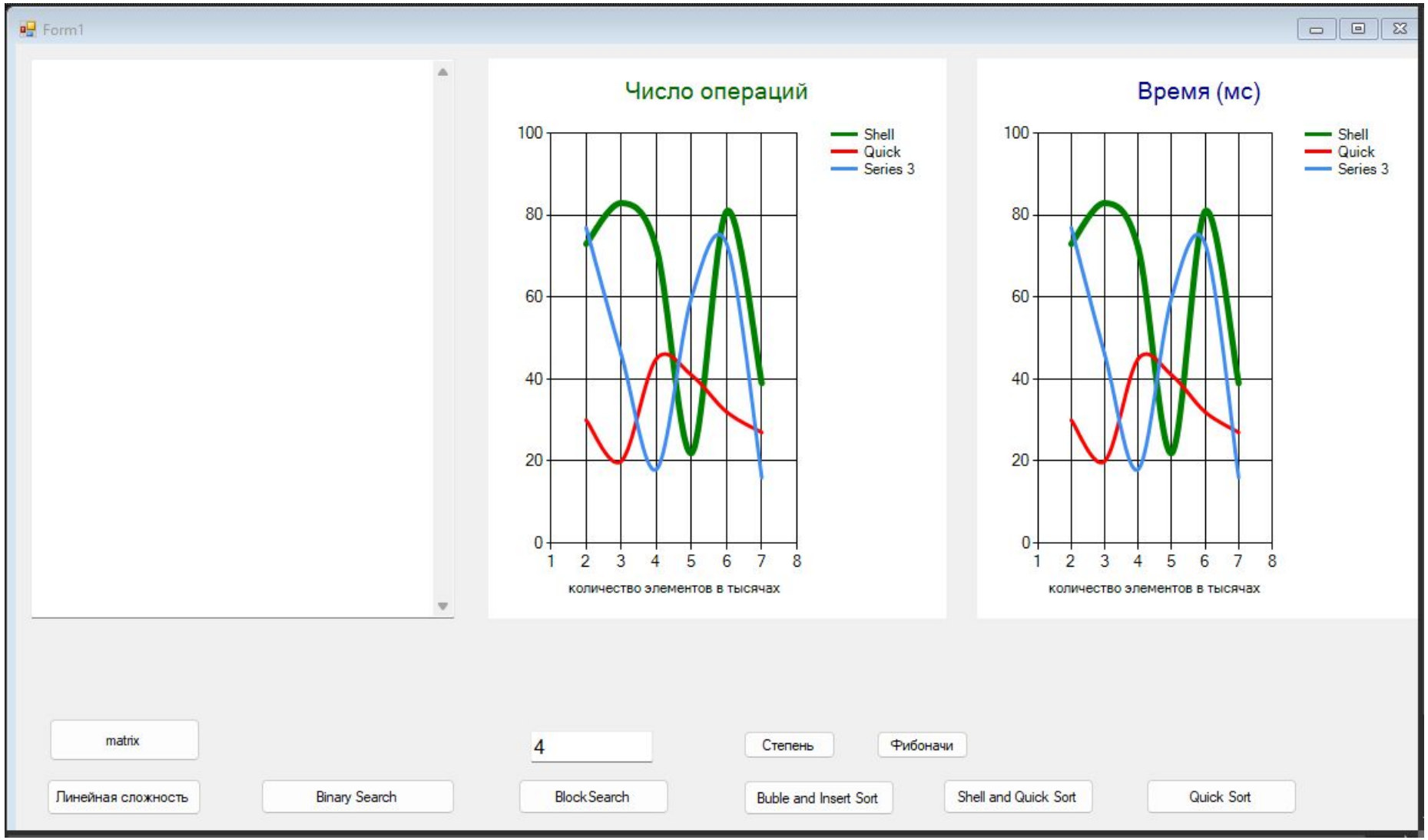






Выполнение лабораторных работ

Создание интерфейса C#

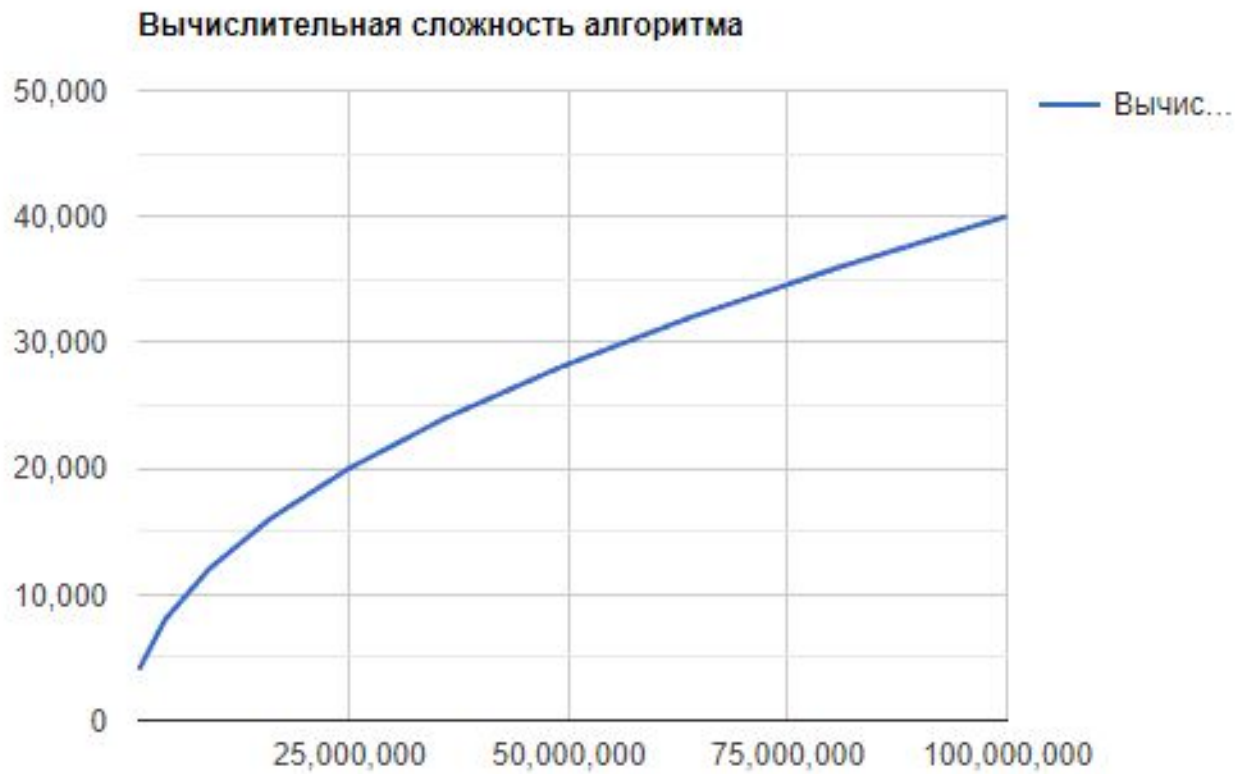


Создание интерфейса на Web странице

Add point 700 18955

Поиск минимума в матрице

Поиск минимума на главной диагонали



Создание интерфейса на html

```
<table>
<tr>
   <td>
     <p>
     <button onclick="AllElements();"> Поиск минимума в матрице </button>
     </p>
     <p>
     <button onclick="MainDiagonal();"> Поиск минимума на главной диагонали </button>
     </p>
   </td>
   <td>
     <div id = "container" style = "width: 550px; height: 400px; margin: 0 auto"> </div>
   </td>
 </tr>
</table>
```

Организация серии вычислительных экспериментов

C#

```
for (int n = 10000; n <= 2000000; n += 50000)
   {
   int[] myarray = new int[n];
   //Генерация массива
   Random rand = new Random();
   for (int i = 0; i < n; i++)
      myarray[i] = rand.Next();
   //Средний случай для бинарного поиска:
   int item = myarray[0];
   //Сортировка массива
   Array.Sort(myarray);
   k = 0;
   //Вызов метода для n-го эксперимента
   }
```

Python

```
for i in range(10, 1000, 1):
    myList = []
    r = 100000
    for j in range(i):
        myList.append(randint(-r, r))
    myList.sort()
    #Худший случай для бинарного поиска:
    n = 1000002 #Элемент для поиска
    k = 0
    #Вызов метода для n-го эксперимента
```

Организация серии вычислительных экспериментов JavaScript

```
function matrix(m, n)
{ var result = []
   for(var i = 0; i < n; i++)
   { result.push(new Array(m).fill(0)) } return
result}

function getRandom(min, max)
{return Math.random() * (max - min) + min }

function setRandonMatrix(matrix)
{let row = matrix.length;
 let column = matrix[0].length;
 for (var i = 0; i < row; i++)
     for (var j = 0; j < column; j++)
     matrix[i][j] = getRandom(-100,100);
return matrix;}
```

```
//Цикл для организации численных
экспериментов
for (var i = 1000; i <= 10000; i += 1000)
{
//Создаем матрицу NxN
myarray = matrix(i,i);
//Заполняем матрицу случайными числами
myarray = setRandonMatrix(myarray);
//Проводим i вычислительный эксперимент
k = 0;
min = getMinmatrix(myarray);
}
```

Оценки времени эксперимента

C# внутри цикла вычислительных экспериментов

```
Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();
//Вызов метода для n-го эксперимента
stopwatch.Stop();
```

Результат в
stopwatch.ElapsedTicks
stopwatch.ElapsedMilliseconds

При описании объекта stopwatch вне цикла:

```
stopwatch.Reset();
stopwatch.Start();
//Вызов метода для n-го эксперимента
stopwatch.Stop();
```

Python внутри цикла вычислительных экспериментов

```
import time
start_time = time.monotonic_ns()
```

вызов n-го эксперимента

Результат в

```
time.monotonic_ns() - start_time
```

аналогично JavaScript:

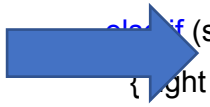
```
const {performance} = require('perf_hooks');
var startTime = performance.now()
//Вызов метода для n-го эксперимента
var endTime = performance.now()
```

Результат в

```
endTime - startTime
```

Оценка количества операций C#

```
private int BinarySearch(int[] array,
        int searchedValue, int left, int right)
    {
    while (left <= right)
       { var middle = (left + right) / 2;
        if (searchedValue == array[middle])
           return middle;
        else if (searchedValue < array[middle])
           right = middle - 1;
         else
           left = middle + 1;
       }
    return -1;
    }
```



```
private int BinarySearch(int[] array, int searchedValue, int left, int right)
    {
    while (left <= right)
       { k++; //Проверка условия в while
         var middle = (left + right) / 2;
         k++; //вычисление middle
         k += 2; //проверка условия и обращение к элементу массива
          if (searchedValue == array[middle])
             {return middle;}
          else if (searchedValue < array[middle])
             { right = middle - 1;
              k += 3; //Проверка условия и обращение к элементу массива и вычисление
             }
          else
             { left = middle + 1;
              k++; //вычисление left
             }
       }
    return -1;
    }
```

Оценка количества операций Python

```
def BinarySearch(myList, SearchedValue):
    left = 0
    right = len(myList)-1
    index = -1
    global k
    k += 4
    while (left<=right):
        middle = (left+right)//2
        k += 2
        if SearchedValue == myList[middle]:
            index = middle
            k += 3
            break
        else:
            if SearchedValue < myList[middle]:
                right = middle -1
                k += 3
            else:
                left = middle + 1
                k += 1
    return index
```

Оценка количества операций JavaScript

```
function getMinmatrix(matrix)
  { let row = matrix.length; k++;
    let column = matrix[0].length; k++;
    result = matrix[0][0]; k =+2;
    for (var i = 0; i < row; i++)
      { k += 2;
      for (var j = 0; j < column; j++)
        { k += 2;
          k += 2;
          if (matrix[i][j] < result)
             { result = matrix[i][j];
               k += 2; }
        }
      }
  return result;
  }
```

Построение графиков C#

- Приложение **WinForms (.NetFrameWork)**
  - компонент **Chart**
    - Свойство **Series** (2-3 серии для графиков)
      - Свойство **ChartType – Spline**
      - Свойство **BorderWidth –** толщина линии графика
      - Свойство **Color** – цвет графика
      - Свойство **LegendText** –текст легенды для графика
    - Свойство **Titles** (один заголовок для графика)
      - Свойство **Text**
    - Доступ из кода программы к свойству **Points**
      - Метод **Clear** – очистка всех точек графика
      - Метод **AddXY** – добавление точки на график

```
this.chart1.Series[0].Points.Clear();
this.chart2.Series[0].Points.Clear();
this.chart1.Series[1].Points.Clear();
this.chart2.Series[1].Points.Clear();
```

В цикле вычислительных экспериментов:

```
this.chart1.Series[0].Points.AddXY(n, k);
this.chart2.Series[0].Points.AddXY(n, stopwatch.ElapsedTicks);
```

Построение графиков Python

```
import matplotlib.pyplot as plt
x = []
y = []

for i in range(10, 1000, 1):
    k = 0
    result = BinarySearch(myList, n)
    x.append(i) #добавление размерности данных
    y.append(k) #добавление числа операций
plt.plot(x,y)
```

Построение графиков JavaScript. Подготовка HTML

Код HTML

```
<head> <title>Google Charts Complexity Graphic</title>
<script type="text/javascript"src="https://www.gstatic.com/charts/loader.js">
</script>
</head>
…
<script language='JavaScript'>
google.charts.load('current',{packages:['corechart']});
google.charts.setOnLoadCallback(drawChart); </script>
```

Построение простого графика JavaScript

Код Javascript

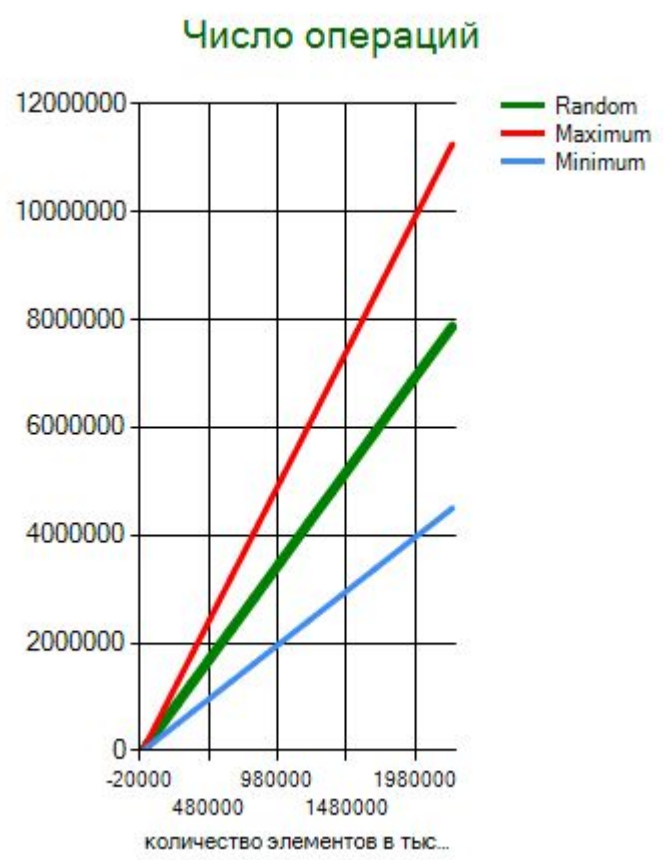
```
var data; var options; var chart;


function drawChart()
{ // Задаем chart и первоначальный график
data = new google.visualization.DataTable();
data.addColumn('number', 'Размерность данных’);
data.addColumn('number', 'Вычислительная сложность’);
data.addRows([ [100, 154], [200, 987], [300, 1376] ]); // задаем 3 точки
options = {'title':'Вычислительная сложность алгоритма', 'width':550,
'height':400}; // Задаем options
chart = new google.visualization.LineChart(document.getElementById
('container')); chart.draw(data, options);
}
```

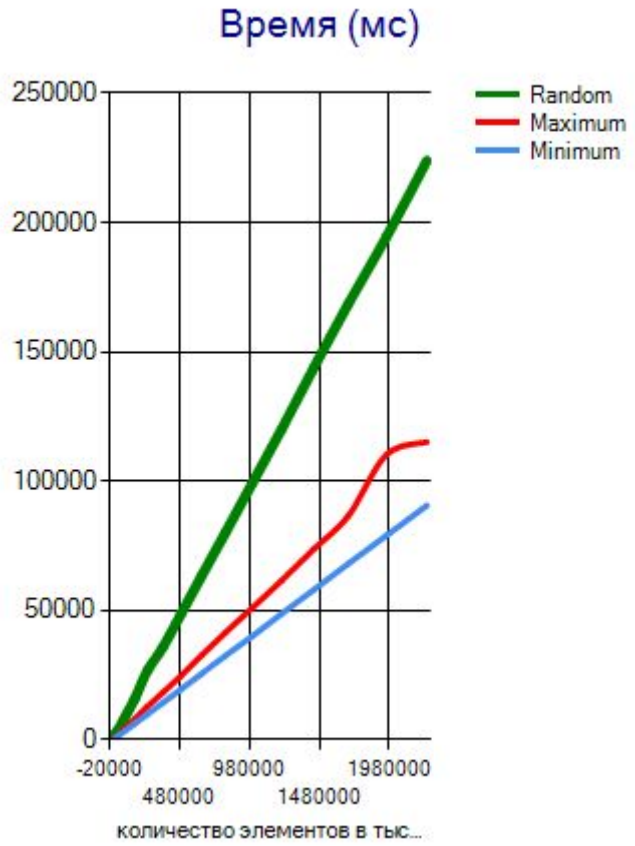
Построение графика на JavaScript по результатам численных экспериментов

```
function AllElements()
{ //Очищаем data для графика
var n = data.getNumberOfRows();
data.removeRows(0, n)
//Цикл для организации численных экспериментов
for (var i = 1000; i <= 10000; i += 1000)
{ //пропущена подготовка матриц
  //Проводим i вычислительный эксперимент
  k = 0;
  min = getMinmatrix(myarray);
  let x = i*i;
  let y = k;
//Добавляем в график точку
data.addRows ([ [x, y], ]); }
//Вызываем прорисовку графика
chart.draw(data, options); }
```

Коллизия при определении временной и вычислительной сложности

```
//подсчет отрицательных элементов в матрице
private int doMatrix(int[,] array, int n)
    {   int m = 0;
        int count = 0; ;
        for (int i = 0; i < n; i++)
            { k += 2;
            for (int j = 0; j < n; j++)
                { k += 2;
                if (array[i, j] < 0)
                {  count++;
                    k += 3;
                }
            }
        }
        return count;
    }
```





Случайные числа

Все элементы положительные

Все элементы отрицательные