



И. Ю. Травкин,

Сахалинский государственный университет, г. Южно-Сахалинск

ЭЛЕМЕНТЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVASCRIPT: ДВЕ ЗАДАЧИ ИЗ ТЕОРИИ ВЕРОЯТНОСТЕЙ

Аннотация

В статье подробно рассмотрены два примера составления программ в функциональном стиле, знакомство с которым является частью подготовки к решению современных задач обработки больших данных. Выбранные задачи, приведенный исходный код и объяснения автора могут служить материалом для одного или двух элективных занятий на соответствующую тему.

Ключевые слова: функциональное программирование, функциональный стиль, JavaScript, теория вероятностей, занимательные задачи.

DOI: 10.32517/2221-1993-2018-17-7-40-45

Контактная информация

Травкин Иван Юрьевич, ст. преподаватель кафедры математики Сахалинского государственного университета, г. Южно-Сахалинск; *адрес:* 693008, Сахалинская область, г. Южно-Сахалинск, Коммунистический пр-т, д. 33; *телефон:* (4242) 45-23-73; *e-mail:* iwannt@gmail.com

I. Yu. Travkin,
Sakhalin State University, Yuzhno-Sakhalinsk

FUNCTIONAL STYLE PROGRAMMING IN JAVASCRIPT: TWO PROBABILITY PROBLEMS

Abstract

The article contains detailed explanations for two examples of solving simple probability problems with simulations written in JavaScript using functional programming paradigm. Provided material is designed to be used in teaching introductory level of functional programming in upper secondary school.

Keywords: functional programming, functional style, JavaScript, probability theory, entertaining tasks.

Введение

Функциональное программирование предлагает непривычный для большинства программистов подход к пониманию и формулированию вычисления: вместо списка инструкций, означающих строго заданную последовательность действий исполнителя, приходится иметь дело с «декларативными» выражениями — программа перестает быть пошаговой инструкцией («рецептом») и становится совокупностью определений, содержащих описание требуемого результата вычисления (часто такое описание дается в рекурсивной форме).

Знакомство с функциональным программированием еще в школе может вооружить будущего студента инструментарием, актуальность которого возрастает в связи с задачами обработки так называемых *больших данных* (Big Data). Часто приводимый пример — модель MapReduce, разработанная инженерами компании Google с целью упростить создание приложений для распределенной обработки колоссальных объемов хранимой на серверах информации и основанная на принципах, заимствованных из языков функционального программирования [8]. В задачах машинного обучения (еще один пример), также связанных с обработкой больших данных и требующих от программиста более серьезной математической подготовки, трудно переоценить роль инструмента, позволяющего формулировать программу в терминах математических преобразований, не заботясь, например, о непосредственном управлении памятью вычислительной машины.

Язык программирования JavaScript, не являясь функциональным в том же смысле, в котором эту характеристику обычно применяют к языкам Haskell, ML или LISP, перенимает некоторые общие принципы, что позволяет составлять программы в так называемом *функциональном стиле*. Кроме того, JavaScript является самым распространенным на планете языком программирования [7] — любой современный интернет-браузер имеет встроенные инструменты, позволяющие экспериментировать с небольшими программами на JavaScript и таким образом изучать язык. Цель данной статьи: используя «подручное средство» — интернет-браузер со встроенной средой отладки, дать конкретный и готовый к использованию учителем пример составления программ в функциональном стиле.

Важные замечания

Статья адресована главным образом учителям информатики, при этом автор постарался сделать изложение настолько доступным, чтобы его мог усвоить обычный старшеклассник. Предполагается наличие у читателя опыта составления программ на уровне школьных задач и начальных представлений из теории вероятностей. При чтении важно последовательно

знакомиться с предложенными задачами: идея вычислительного эксперимента и исходный код, рассмотренные в первой задаче, используются и далее.

По поводу строгости изложения математических вопросов необходимо заметить, что автором выбран неформальный стиль, опирающийся на то интуитивное представление о вероятности, которое должно формироваться в старших классах школы. Понятия случайной величины и математического ожидания как «среднего» изложены в сильно упрощенной форме, дающей лишь самое необходимое представление.

В качестве среды для исполнения исходного кода удобно использовать любой веб-браузер с функцией «инструменты разработчика». Автор рекомендует Google Chrome версии 65 или более новый (можно загрузить «переносную» версию для установки на USB-диск [9]).

Чтобы начать работу в Google Chrome, откройте новую вкладку (сочетание клавиш Ctrl+T), активируйте инструменты разработчика (клавиша F12) и перейдите в открывшейся панели на вкладку консоли JavaScript (*Console*). Исходный код, приводимый в тексте статьи на отдельных строках, вводится в консоль JavaScript в соответствующей изложению последовательности.

Автор рекомендует предварительно познакомиться с документацией к «стрелочным» функциям [2], а также к методам *map()*, *reduce()*, *filter()* и *slice()* объекта *Array* [3–6] и иметь ее под рукой. В ряде случаев совместного применения *map* и *reduce* можно обойтись одним *reduce*, но такой подход кажется автору менее наглядным с учетом рассматриваемых задач. Аналогичный аргумент можно привести в пользу выбранного способа работы с массивами — предпочтение отдано наглядности, т. е. возможности без дополнительных усилий сопоставить структуру данных и выполняемые с ними преобразования с общей идеей излагаемого решения.

Практика составления программ в функциональном стиле — именно стиле, так как речь идет об отдельных приемах, умело сочетаемых с более традиционным для JavaScript подходом, — стала популярной относительно недавно, в том числе благодаря внедрению языкового стандарта ECMAScript 2015 (ES6). У читателя, знакомого с «традиционным» JavaScript, может возникнуть представление, что все сводится к использованию новых форм синтаксиса и надстроек, делающих язык неузнаваемым для тех, кто не интересовался его развитием в последние годы [10]. На самом деле, функциональный стиль выражается в стремлении к декомпозиции алгоритмов на элементарные функции, сочетание которых дает требуемый результат [11]. Именно эту идею желает передать автор на материале выбранных им задач.

Парадокс дней рождения

Представим следующую ситуацию. В некотором месте собрались совершенно незнакомые и никак не связанные друг с другом люди, всего n человек. Не исключено, что среди них есть родившиеся в один день года (т. е. одного числа, возраст может не совпадать). Каким должно быть минимальное число незнакомцев, чтобы вероятность таких совпадений была не меньше 50 %?

Вероятность того, что два случайно выбранных незнакомца родились в один день, равна $1/365 \approx 0,003$, т. е. всего 0,3 % (для простоты считаем, что в году всегда 365 дней, и все дни равновероятны). Если $n > 365$, т. е. когда незнакомцев больше, чем дней в году, то совпаде-

ния становятся неизбежными, и вероятность достигает 100 %. Можно предположить, что число незнакомцев, соответствующее вероятности 50 %, должно быть около половины от общего числа дней в году, т. е. примерно 182, но вычисления (см., например, [1]) дают неожиданный ответ: при $n = 22$ имеем вероятность 47,6 %, а уже при $n = 23$ вероятность становится 50,7 %, т. е. достаточно всего 23 человек!

Результат кажется удивительным даже после того, как лично выполнишь соответствующие расчеты (отсюда название «парадокс»). Непосредственным способом принять контринтуитивный факт служит эксперимент — проверка опытом. Если бы мы могли организовать описанную встречу для 23 совершенно незнакомых людей, причем повторили этот эксперимент достаточно большое число раз, то примерно в половине случаев среди незнакомцев нашлись бы двое, родившиеся в один день года. Такой эксперимент, основанный на свойстве относительной частоты случайного события приближаться к его вероятности при достаточно большом числе испытаний, можно смоделировать на компьютере.

Чтобы разыграть встречу незнакомцев на компьютере, достаточно сгенерировать n чисел — случайных дней года, как если бы это были дни рождения наших незнакомцев, и проверить на наличие совпадений.

В JavaScript случайные числа получают с помощью функции *Math.random()*, которая возвращает действительные значения в интервале (0; 1). Если умножить такое число на 365 и взять целую часть от результата с помощью функции *Math.floor()*, то мы получим случайное целое число от 0 до 364, т. е. ровно 365 возможных значений (пусть счет дней начинается с нуля).

Чтобы получить n случайных значений, подготовим массив на n «мест» и заполним их случайными значениями:

1) создадим массив из n ячеек, заполненных нулями (необходимо, чтобы в ячейках что-то было до того, как мы начнем преобразовывать значения):

```
Array(n).fill(0)
```

2) преобразуем с помощью *map* значения в ячейках, заменив их случайными номерами от 0 до 364:

```
...map(_ => Math.floor(Math.random()*365))
```

Полное определение функции для получения n случайных дней рождения имеет вид:

```
random_days = (n) => Array(n).fill(0).map(_ =>
  Math.floor(Math.random()*365));
```

Например, при $n = 10$ мы можем получить следующий результат:

```
> random_days(10);
[25, 215, 237, 340, 102, 179, 282, 68, 308, 137]
```

Теперь необходимо реализовать проверку на совпадения. Сначала сформулируем критерий: в массиве есть совпадения, если массив содержит больше одного элемента и либо первый элемент равен одному из последующих, либо совпадения есть среди последующих. Переводя с естественного языка на язык JavaScript, мы получим рекурсивное определение функции, вычисляющей логическое значение ИСТИНА или ЛОЖЬ:

```
has_same = (days) => days.length > 1 &&
  (days.slice(1).includes(days[0]) ||
  has_same(days.slice(1)));
```

Здесь:

`days.length` возвращает число элементов в массиве сравниваемых дней рождения;

`days.slice(1)` возвращает массив без первого элемента («последующие»);

метод `includes()` проверяет наличие в массиве конкретного значения (совпадение «первого» с одним из «последующих»).

Проверим на примерах:

```
> has_same([10, 12, 31, 14]);
false
> has_same([10, 12, 31, 12]);
true
```

Эксперимент предполагает многократное (N раз) разыгрывание встречи незнакомцев, и если представить результат отдельного розыгрыша одним числом: 1 — в случае «успеха» (есть незнакомцы, родившиеся в один день) и 0 — в случае «неудачи» (все незнакомцы родились в разные дни), то общее число «успехов» будет равно сумме чисел — результатов всех розыгрышей:

$$M = X_1 + X_2 + \dots + X_N = \sum_{i=1}^N X_i,$$

где

$$X_i = \begin{cases} 1, & \text{если «успех» в } i\text{-м розыгрыше,} \\ 0, & \text{если «неудача» в } i\text{-м розыгрыше.} \end{cases}$$

Преобразовать результат розыгрыша в число, т. е. получить значение X_i (для всех $i = 1, 2, \dots, N$), можно с помощью тернарной условной операции:

$$\begin{aligned} & \text{условие} ? \text{результат1} : \text{результат2} = \\ & = \begin{cases} \text{результат1, если условие} = \text{ИСТИНА,} \\ \text{результат2, если условие} = \text{ЛОЖЬ.} \end{cases} \end{aligned}$$

Тогда

$$X_i = \text{has_same}(\text{random_days}(n)) ? 1 : 0.$$

Поступим следующим образом: разместим результаты N розыгрышей в ячейках массива (каждая ячейка — как бы отдельный розыгрыш), а затем найдем общую сумму:

1) подготовим массив из N ячеек, заполненных нулями:

```
Array(N).fill(0)
```

2) преобразуем с помощью `map` значения в ячейках, заменив их результатами розыгрышей:

```
...map(_ => has_same(random_days(n))
? 1 : 0)
```

3) найдем с помощью `reduce` сумму значений из всех ячеек, т. е. общее число «успехов»:

```
...reduce((x, y) => x + y)
```

4) получим окончательный результат как отношение M/N .

Все вычисление результата численного эксперимента примет вид функции:

```
simulate = (n, N) => Array(N).fill(0).map(_ =>
has_same(random_days(n)) ? 1 : 0).
reduce((x, y) => x + y) / N;
```

Например, при $n = 100$, $N = 10\,000$ мы можем увидеть следующий результат (повторив действие, читатель по-

лучит близкое, но отличающееся значение частоты, так как эксперимент случайный):

```
> simulate(100, 10000);
0.9999
```

Полученное значение превышает 0,5 (50 %), что позволяет заключить: ста незнакомцев достаточно для того, чтобы вероятность совпадения дней рождения хотя бы у двоих была не меньше 50 %. Нас же интересует «пороговое» значение, т. е. минимальное n , при котором выполняется условие. Мы не станем искать n полным перебором, а повторим эксперимент для значений, выбранных из заданного промежутка с постоянным шагом (например, для $n = 10, 20, \dots, 100$).

Определим функцию, которая позволит получить выборку результатов экспериментов с различными значениями n :

```
experimental_data = (a, b, s, N) => {
let K = Math.floor((b - a) / s) + 1;
return Array(K).fill(0).map((_, i) =>
[a + s * i, simulate(a + s * i, N)])
};
```

Значения n выбираются из промежутка от a (включая) до b с шагом s ; всего таких значений $K = \lfloor (b - a) / s \rfloor + 1$. Результатом вычисления является массив из K элементов, в каждой ячейке которого записана пара значений вида: $[n, M/N]$, где $n = a + s \cdot i$ (число незнакомцев), $i = 0, 1, \dots, K - 1$ (порядковый номер ячейки в массиве), а M/N — относительная частота, т. е. результат эксперимента при заданном n .

Произведем вычисление для n от 10 до 100 с шагом 10 и $N = 1000$:

```
> data = experimental_data(10, 100, 10, 1000);
[[10, 0.118], [20, 0.408], [30, 0.692],
[40, 0.901], [50, 0.971], [60, 0.997],
[70, 0.999], [80, 1], [90, 1], [100, 1]]
```

Последний штрих: преобразуем полученные данные (массив пар) в формат текста-таблицы, который легко воспринимается при непосредственном изучении.

«Сцепим» значения внутри пар символом табуляции (`join(«\t»)`), а между парами вставим переход на новую строку (`join(«\n»)`):

```
print_data = (pairs) => pairs.map(pair => pair.
join("\t")).join("\n").replace(/\.\/g, ",");
```

Последнее преобразование `replace(/\.\/g, «,»)` нужно для того, чтобы заменить точку, разделяющую целую и дробную части числа в JavaScript, на привычную запятую.

Теперь имеем:

```
> print_data(data);
"10 0,118
20 0,408
30 0,692
40 0,901
50 0,971
60 0,997
70 0,999
80 1
90 1
100 1"
```

Анализ таблицы позволяет заключить, что «пороговое» значение n , т. е. минимальное число незнакомцев, при котором вероятность совпадения дней рождения хотя бы у двоих становится не меньше 50 %, действительно находится между $n = 20$ и $n = 30$. А уже

при $n = 50$ вероятность почти достигает 100 %. Кажется невероятным, но когда мы едем в автобусе в час пик, среди пассажиров с большой вероятностью есть двое, родившиеся в один день!

Задача про перестановки

Пусть дана перестановка из n чисел, т. е. взятые в некотором порядке числа от 1 до n . Например: (1, 5, 3, 4, 2) — перестановка длины $n = 5$. Числа, входящие в перестановку, будем называть *элементами данной перестановки*. *Максимум* — это такой элемент перестановки, который больше двух соседних с ним элементов. Например, в перестановке (1, 5, 3, 4, 2) есть два максимума: 5 (так как $5 > 1$ и $5 > 3$) и 4 (так как $4 > 3$ и $4 > 2$). Заметим, что первый и последний элементы не могут быть максимумами (максимум определен для «внутренних» элементов, имеющих два соседних с ними).

Вопрос: сколько в среднем максимумов содержит перестановка длины n ?

Например, существует всего шесть различных перестановок длины $n = 3$:

(1, 2, 3), (2, 1, 3), (1, 3, 2), (3, 2, 1), (3, 1, 2), (2, 3, 1).

Указанным перестановкам соответствуют следующие числа максимумов:

0, 0, 1, 0, 0, 1.

Получаем, что среднее число максимумов в перестановке длины $n = 3$ равно:

$$(0 + 0 + 1 + 0 + 0 + 1) / 6 = 2 / 6 = 1 / 3.$$

Легко догадаться, что наименьшее возможное число максимумов в перестановке любой длины равно нулю: если расположить элементы строго по возрастанию, то максимумов не будет. Наибольшее возможное число максимумов не может превышать половины длины перестановки, так как максимумы должны быть отделены друг от друга хотя бы одним элементом.

Таким образом, *среднее число максимумов k в перестановке длины n удовлетворяет условию $0 \leq k \leq n/2$* .

Найдем зависимость k от n с помощью случайного эксперимента.

В отличие от эксперимента к первой задаче, где разыгрывалось *случайное событие*, здесь мы будем разыгрывать *случайную величину*. Случайная величина — это переменная, которая в отдельном испытании принимает одно, и только одно из своих возможных значений, причем каждому значению соответствует определенная вероятность. Важнейшая характеристика случайной величины — ее *математическое ожидание*, равное сумме произведений возможных значений на соответствующие им вероятности. *Число максимумов в случайно выбранной перестановке заданной длины есть случайная величина, математическое ожидание которой равно среднему числу максимумов*.

Например, число максимумов в перестановке длины $n = 3$ равно 0 или 1 в зависимости от того, какая перестановка выбрана: в двух из шести случаев, т. е. с вероятностью $2/6$, число максимумов равно 1, в остальных четырех случаях, т. е. с вероятностью $4/6$, число максимумов равно 0 (см. выше). Тогда математическое ожидание числа максимумов в случайной перестановке длины $n = 3$ равно:

$$1 \cdot 2/6 + 0 \cdot 4/6 = 1/3,$$

т. е. найденному ранее среднему числу максимумов.

Эксперимент, основанный на статистическом смысле математического ожидания, позволит нам оценить среднее число максимумов без перебора всех возможных перестановок. Для этого произведем N испытаний, в каждом из которых выберем случайную перестановку длины n и определим число максимумов в ней, получив тем самым N значений случайной величины, а затем найдем среднее арифметическое этих значений, которое и будет статистической оценкой математического ожидания.

Начнем с определения числа максимумов в данной перестановке.

Пусть дан массив xs , содержащий n элементов (т. е. перестановка). Ни первый, ни последний элементы массива не могут быть максимумами, требуется проверить остальные $n - 2$ элементов:

1) возьмем «усеченный» массив, т. е. элементы со второго по $(n - 1)$ -й:

```
xs.slice(1, xs.length - 1)
```

Помним, что для элементов массива счет начинается с нуля;

2) преобразуем выбранные элементы следующим образом: запишем 1, если данный элемент — максимум, и 0 — в противном случае. Учтем, что нумерация элементов в исходном и «усеченном» массивах отличается на единицу!

```
...map((x,i) => x > xs[i] && x > xs[i+2] ? 1 : 0)
```

3) сложим записанные значения:

```
...reduce((x, y) => x + y)
```

Полное определение функции для нахождения числа максимумов в перестановке имеет вид:

```
count_maxes = (xs) => xs.slice(1, xs.length - 1).
  map((x,i) => x > xs[i] && x > xs[i+2] ? 1 : 0).
  reduce((x, y) => x + y);
```

Проверим:

```
> count_maxes([1, 2, 3, 4, 5]);
0
> count_maxes([1, 2, 3, 5, 4]);
1
> count_maxes([1, 3, 2, 5, 4]);
2
```

Теперь займемся получением случайной перестановки.

Возьмем массив, состоящий из расположенных по порядку чисел от 1 до n , и «перетасуем» его. Основное здесь — это «тасовка», т. е. случайное перемешивание, гарантирующее одинаковые шансы получить любую из возможных перестановок. Реализуем это следующим образом: будем брать по одному элементу из упорядоченного массива, выбирая очередной элемент в случайном порядке, и помещать в новый массив.

Сначала определим вспомогательные функции, решающие следующие задачи:

- выбор случайного элемента из массива — функция *take_random*;
- исключение из массива элементов, уже помещенных в новый массив, — функция *exclude*;
- помещение элементов исходного массива, взятых по одному в случайном порядке, в новый массив — функция *take_all* (использует *take_random* и *exclude*).

Рассмотрим перечисленные функции в соответствующем порядке:

```
take_random = (xs) => xs[Math.floor(Math.
random() * xs.length)];
```

Здесь `Math.floor(Math.random()*xs.length)` позволяет получить случайный порядковый номер элемента в массиве `xs`. С `Math.random()` мы уже знакомы: умножая случайное значение от 0 до 1 на `xs.length`, мы получаем случайное значение от 0 до `xs.length`, а `Math.floor()` переводит его в целое (порядковый номер — целое число).

```
exclude = (xs, ys) =>
  xs.filter(x => !ys.includes(x));
```

С помощью `filter` мы отбираем лишь те элементы массива `xs`, которых еще нет в массиве `ys`. Помним, что восклицательный знак означает отрицание: если `ys.includes(x)` истинно для любого значения `x`, содержащегося в массиве `ys`, то `!ys.includes(x)` истинно только для таких `x`, которые *не* содержатся в массиве `ys`.

```
take_all = (xs, ys) => ys.length == xs.length ?
  ys : take_all(xs, ys.concat(take_random
(exclude(xs, ys))));
```

Массив `ys` сначала пуст, а в процессе вычисления наполняется элементами из `xs`. Если число элементов в `ys` становится равно числу элементов в `xs`, т. е. все элементы из `xs` уже помещены в `ys`, вычисление завершается ответом — `ys`. Иначе продолжаем наполнение `ys`, вычисляя `take_all` (рекурсия) с тем же `xs`, но *обновленным* `ys`: добавляем с помощью `concat` очередной элемент, взятый с помощью `take_random` из числа таких элементов `xs`, которых нет в `ys` (используем `exclude`).

Проверим:

```
> take_all([1, 2, 3, 4, 5], []);
[1, 5, 4, 3, 2]
> take_all([1, 2, 3, 4, 5], []);
[2, 4, 1, 5, 3]
```

Теперь мы готовы определить функцию, вычисляющую случайную перестановку:

```
shuffle = (xs) => take_all(xs, []);
permutation = (n) => shuffle(Array(n).fill(0).
  map((_, i) => i + 1));
```

Здесь `Array(n).fill(0).map((_, i) => i + 1)` позволяет получить массив с числами от 1 до `n` (т. е. массив, который «тасуется»).

Наконец, используя функции `permutation` и `count_maxes`, мы можем сгенерировать `N` случайных перестановок заданной длины и определить среднее число максимумов в них, а затем, изменяя длину перестановки, выявить зависимость среднего числа максимумов от длины перестановки. Здесь можно снова воспользоваться функциями `simulate`, `experimental_data` и `print_data` (см. исходный код к первой задаче), слегка изменив одну из них:

```
simulate = (n, N) => Array(N).fill(0).
  map(_ => count_maxes(permutation(n))).
  reduce((x, y) => x + y) / N;
```

Произведем окончательное вычисление и получим данные для анализа:

```
> data = experimental_data(10, 100, 10, 1000);
[[10, 2.66], [20, 5.954], [30, 9.306],
 [40, 12.658], [50, 15.979], [60, 19.357],
 [70, 22.559], [80, 26.032], [90, 29.246],
 [100, 32.628]]
> print_data(data);
"10 2,66
20 5,954
30 9,306
40 12,658
50 15,979
60 19,357
70 22,559
80 26,032
90 29,246
100 32,628"
```

Для наглядности скопируем выведенную в консоль таблицу на новый лист Microsoft Excel и отобразим данные с помощью графика *Точечная диаграмма* (см. рис.).

Приведенный на рисунке график позволяет предположить, что среднее число максимумов в перестановке прямо пропорционально ее длине. И хотя точки графика не лежат в точности на одной прямой, такую прямую можно получить примерно, используя метод наименьших квадратов (подробности метода опускаем):

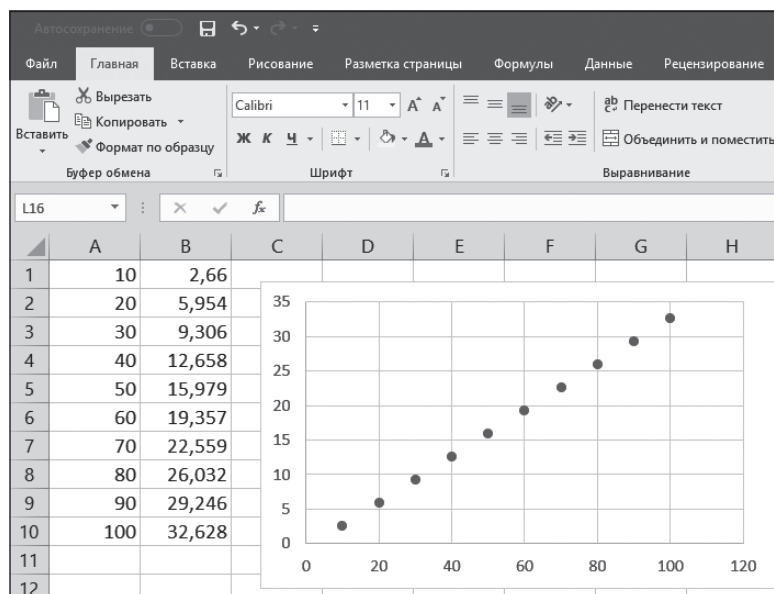


Рис. График экспериментального распределения данных.
По горизонтальной оси отложены значения n (длина перестановки),
по вертикальной — «экспериментальное» среднее число максимумов k

$$k = 0,33301n - 0,67767 \approx 1/3n - 2/3.$$

Найденная экспериментально зависимость совпадает с точным решением: среднее число максимумов перестановки линейно зависит от ее длины и выражается указанным выше соотношением.

Подробно о точном решении можно узнать из курса лекций по теории вероятностей Дж. Блитцштейна [12], у которого автор позаимствовал эту задачу в слегка измененной постановке.

Случайный эксперимент, как было сказано выше, не требует перебора всех возможных перестановок: чтобы получить достаточно точный результат, нам понадобилось всего по 1000 случайных перестановок каждой длины. Заметим, что если различных перестановок длины $n = 10$ существует $10! = 3\,628\,800$, то различных перестановок длины $n = 20$ будет уже $20! \approx 2,4 \cdot 10^{18}$, а при $n = 100$ имеем $100! \approx 9,3 \cdot 10^{157}$.

Заключение

Рассмотренные в статье задачи, исходный код и предложенное объяснение могут служить материалом для одного или двух элективных занятий, имеющих целью познакомить старшеклассников с элементами функционального программирования, быстро проникающими в популярные языки. Автор считает, что использованного им «подручного средства» — браузерной консоли JavaScript — достаточно и для более глубокого погружения в функциональное программирование, однако требуется дополнительная методическая работа

по переложению примеров из «классических» функциональных языков и отбору конкретных задач.

Список использованных источников

1. Аль-Халили Дж. Парадокс. Девять великих загадок физики. СПб.: Питер, 2017.
2. Стрелочные функции. https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions
3. Array.prototype.filter(). https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
4. Array.prototype.map(). https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/map
5. Array.prototype.reduce(). https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce
6. Array.prototype.slice(). https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/slice
7. Crockford D. The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language. <https://www.crockford.com/javascript/popular.html>
8. Dean J., Ghemawat S. MapReduce: simplified data processing on large clusters // Communications of the ACM. 2008. N 51(1). https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/dean/dean_html/
9. Google Chrome Portable. https://portableapps.com/apps/internet/google_chrome_portable
10. Jang P. Modern JavaScript Explained For Dinosaurs. <https://medium.com/the-node-js-collection/modern-javascript-explained-for-dinosaurs-f695e9747b70>
11. Nakoryakov V. Two Years of Functional Programming in JavaScript: Lessons Learned. <https://hackernoon.com/two-years-of-functional-programming-in-javascript-lessons-learned-1851667c726>
12. Stat 110 on YouTube. <https://projects.iq.harvard.edu/stat110/youtube>

НОВОСТИ

Мировой рынок платформ «умного города» через пять лет вырастет до \$755 миллионов

Мировой рынок платформ «умного города» к 2027 году достигнет \$755 млн. Такие данные приводятся в последнем отчете Navigant Research. По мнению аналитиков, концепция создания таких платформ становится все более понятной. Крупнейшие мировые вендоры, такие как Nokia, Microsoft и Cisco, разрабатывают проекты, в состав которых входит множество решений, покрывающих самые разные области — от видеонаблюдения и создания сетей до парковки и мониторинга окружающей среды в «умных городах».

Интеграция сервисов «умного города» неизбежна и необходима, также как и их централизованная координация, говорят аналитики Navigant Research. Заказчики все чаще стремятся выработать единую стратегию подключения различных городских служб к интернету вещей. Таким образом, появление единой платформы позволит реализовать концепцию «город как услуга», а также будет способствовать организации эффективного взаимодействия между различными городами, выстраиванию партнерских отношений, необходимых для успешного развития городской среды в будущем.

«Платформенный подход к созданию «умного города» способствует внедрению инноваций и обеспечивает непрерывность предоставления городских сервисов. Он позволяет городам стать полноценными участниками современного мира, основанного на бизнес-платформ-

мах», — говорит Эрик Вудс, директор по исследованиям Navigant Research.

В качестве образцов управления ресурсами и предоставления услуг «умные города» могли бы использовать модели Uber, Airbnb, Amazon, Google и Facebook, считают аналитики.

Платформенный подход к реализации концепции «умного города» позволит сделать его более удобным для рядового пользователя, будет способствовать созданию интуитивно понятного интерфейса. А значит, предлагаемые технологии станут востребованы. Тем не менее вопросы обеспечения информационной безопасности по-прежнему остаются одними из ключевых в этой сфере. Особенно в случае, если созданием городских сервисов будут активно заниматься коммерческие компании.

Сегодня на рынок «умных городов» стремится выйти множество разработчиков платформ. По мнению Navigant Research, это может привести к тому, что сама идея создания «умных городов» может быть скомпрометирована чрезмерными PR-активностями, в то время как граждане ждут от нее настоящих инновационных преобразований. В таких условиях, по мнению аналитиков, правительствам городов необходимо разрабатывать платформенную стратегию, основанную на принципах открытости, использования стандартов и согласованности с другими технологиями и бизнес-программами.

(По материалам CNews)