Лабораторная работа 1

Генералов Даниил, НПИбд-01-21, 1032212280

18 марта 2023 г.

В данной лабораторной работе требуется выполнить симуляции методом Монте-Карло.

Этот отчет представлен как экспорт Jupyter-блокнота (https://jupyter.org), использующего ядро evcxr (https://github.com/evcxr/evcxr).

[2]:
```
// Эти команды проверяют функционирование ядра evcxr,
// а также включают информацию о времени выполнения ячейки
// и кеширование артифактов компиляции.
:timing 1
:sccache 1
```

[2]:
```
Timing: true
sccache: true
```

[3]:
```
:timing 1
:sccache 1

// Здесь мы описываем все зависимости для выполнения ячеек этого блокнота.
// Выполнение этой ячейки скачает и соберет все зависимости заранее.
:dep rand = "0.8.5"
:dep itertools = "0.10.5"
:dep rand_distr = "0.4.3"
:dep plotters = { version="0.3.4", default_features = false, features =␣
↪["evcxr", "all_series"] }
```

[3]:
```
Timing: false
sccache: true
```

[4]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"

/// Мы описываем интерфейс, который будут реализовывать случайные опыты.
/// Он состоит из одного типа и нескольких функций.
trait Experiment {
    /// Тип, описывающий исход выполнения одного опыта.
```

```rust
    type Outcome: Ord+Eq+std::hash::Hash+Sized;

    /// Метод, возвращающий список исходов, которые считаются желаемыми в опыте.
    fn desired_outcomes(&self) -> Vec<Self::Outcome>;

    /// Метод, который принимает экземпляр генератора случайных чисел
    /// и выдает результат выполнения одного опыта.
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome;

    /// Метод, который проводит много опытов и составляет
    /// гистограмму исходов с количеством того. сколько раз они встречались.
    fn collect_stats(&self, trials: usize) -> std::collections::HashMap<Self::
↪Outcome, usize> {
        let mut outcomes = std::collections::HashMap::new();
        let mut rng = rand::thread_rng();
        for _ in 0..trials {
            // Для каждого раза опыта, проводим один опыт, и увеличиваем его␣
↪счетчик в гистограмме.
            let outcome = <Self as Experiment>::try_it(&self, &mut rng);
            outcomes.entry(outcome).and_modify(|i| *i += 1).or_insert(1);
        }
        outcomes
    }

    /// Метод, который считает вероятность того, что исход опыта будет одним из␣
↪данных.
    fn probability_of_outcomes(&self, trials: usize, desired_outcomes: &[Self::
↪Outcome]) -> f64 {
        let outcomes = self.collect_stats(trials);
        let mut ok_outcomes = 0.0;
        for outcome in desired_outcomes {
            ok_outcomes += (*outcomes.get(&outcome).unwrap_or(&0) as f64);
        }
        ok_outcomes / (trials as f64)
    }

    /// Метод, который считает вероятность того, что исход опыта будет одним из␣
↪желаемых.
    fn probability_of_desired(&self, trials: usize) -> f64 {
        self.probability_of_outcomes(trials, &<Self as Experiment>::
↪desired_outcomes(&self))
    }

    /// Описание опыта в виде строки.
    fn description(&self) -> String;
}
```

```rust
/// Интерфейс для опытов, которые возвращают непрерывные значения,
/// которые нельзя сравнивать напрямую.
trait PartialExperiment {
    type Outcome: PartialOrd+PartialEq+Sized;

    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome;

    /// Метод, который проводит опыт много раз и возвращает список
    /// исходов, которые были получены.
    fn collect_stats(&self, trials: usize) -> Vec<Self::Outcome> {
        let mut outcomes = Vec::with_capacity(trials);
        let mut rng = rand::thread_rng();
        for _ in 0..trials {
            let outcome = <Self as PartialExperiment>::try_it(&self, &mut rng);
            outcomes.push(outcome)
        }
        outcomes
    }
    fn description(&self) -> String;
}

/// Наконец, мы объявляем структуру, которая будет содержать ответы на задания.
#[derive(Default, Debug)]
struct Answers {
    task_1: Option<f64>,
    task_2: Option<f64>,
    task_3: Option<f64>,
    task_4: Option<f64>,
    task_5: Option<f64>,
    task_6: Option<f64>,
    task_7: Option<u64>,
    task_8: Option<(f64, f64, f64)>,
    task_9: Option<f64>,
    task_10: Option<f64>,
}
let mut answers = Answers::default();
```

[4]:
```
Timing: true
sccache: true
```

[5]:
```rust
:timing 1
:sccache 1
:dep rand = "0.8.5"
:dep itertools = "0.10.5"
```

```rust
use itertools::Itertools;

struct Task1;

/// В контексте данной задачи, вектор -- это пара двух действительных чисел.
type Vec2 = (f64, f64);

/// Вспомогательный метод считает длину вектора.
fn magnitude(v: &Vec2) -> f64 {
    let (a, b) = v;
    (a*a + b*b).sqrt()
}

impl Experiment for Task1 {
    type Outcome = bool;
    fn description(&self) -> String {
        "Чему равна вероятность того, что случайный треугольник, нарисованный␣
 ↪внутри
квадрата со стороной 1, является тупоугольным?".to_string()
    }
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome {
        // Генерируем три пары точек, каждая координата -- от 0 до 1.
        let pa: (f64, f64) = (rng.gen(), rng.gen());
        let pb: (f64, f64) = (rng.gen(), rng.gen());
        let pc: (f64, f64) = (rng.gen(), rng.gen());
        // Считаем векторы между точками
        let va = (pa.0 - pb.0, pa.1 - pb.1);
        let vb = (pa.0 - pc.0, pa.1 - pc.1);
        let vc = (pc.0 - pb.0, pc.1 - pb.1);
        // Считаем длины этих трех векторов
        let magnitudes: Vec<f64> = [va, vb, vc].iter().map(|x| magnitude(x)).
 ↪collect();
        let ma = magnitudes[0];
        let mb = magnitudes[1];
        let mc = magnitudes[2];

        // Перебирая все перестановки этих трех векторов
        for m in [ma, mb, mc].iter().permutations(3) {
            // проверяем, квадрат двух катетов меньше ли квадрата гипотенузы
            if (m[0] * m[0]) + (m[1] * m[1]) < (m[2] * m[2]) {
                // если да, то такой треугольник тупоугольный.
                return true;
            }
        }
        // Если же такой перестановки сторон не существует,
        // то треугольник не тупоугольный.
        false
```

```
    }

    fn desired_outcomes(&self) -> Vec<Self::Outcome> {vec![true]}
}

let a = Task1{};
let prob = a.probability_of_desired(1_000_000);
println!("{} -- {}", a.description(), prob);
answers.task_1 = Some(prob);
```

```
Чему равна вероятность того, что случайный треугольник, нарисованный внутри
квадрата со стороной 1, является тупоугольным? -- 0.725906
```

[5]:
```
Timing: false
sccache: true
```

[6]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"

struct Task2;

impl Experiment for Task2 {
    type Outcome = bool;
    fn description(&self) -> String {
        "Чему равна вероятность того, что случайный треугольник, нарисованный␣
 ↪внутри
прямоугольника, у которого одна сторона в 2 раза длиннее другой, является␣
 ↪тупоугольным?".to_string()
    }
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome {
        // Эта задача очень похожа на предыдущую, поэтому большинство кода␣
 ↪идентично.

        // Генерируем три пары точек, где первая координата -- от 0 до 1, а␣
 ↪вторая -- от 0 до 2.
        let pa: (f64, f64) = (rng.gen(), rng.gen::<f64>()*2.0);
        let pb: (f64, f64) = (rng.gen(), rng.gen::<f64>()*2.0);
        let pc: (f64, f64) = (rng.gen(), rng.gen::<f64>()*2.0);
        // Считаем векторы между точками
        let va = (pa.0 - pb.0, pa.1 - pb.1);
        let vb = (pa.0 - pc.0, pa.1 - pc.1);
        let vc = (pc.0 - pb.0, pc.1 - pb.1);
        // Считаем длины этих трех векторов
        let magnitudes: Vec<f64> = [va, vb, vc].iter().map(|x| magnitude(x)).
 ↪collect();
```

```
        let ma = magnitudes[0];
        let mb = magnitudes[1];
        let mc = magnitudes[2];

        // Перебирая все перестановки этих трех векторов
        for m in [ma, mb, mc].iter().permutations(3) {
            // проверяем, квадрат двух катетов меньше ли квадрата гипотенузы
            if (m[0] * m[0]) + (m[1] * m[1]) < (m[2] * m[2]) {
                // если да, то такой треугольник тупоугольный.
                return true;
            }
        }
        // Если же такой перестановки сторон не существует,
        // то треугольник не тупоугольный.
        false
    }

    fn desired_outcomes(&self) -> Vec<Self::Outcome> {vec![true]}
}

let a = Task2{};
let prob = a.probability_of_desired(1_000_000);
println!("{} -- {}", a.description(), prob);
answers.task_2 = Some(prob);
```

Чему равна вероятность того, что случайный треугольник, нарисованный внутри прямоугольника, у которого одна сторона в 2 раза длиннее другой, является тупоугольным? -- 0.7984

[6]:
```
Timing: true
sccache: true
```

[7]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"

struct Task3;

impl Experiment for Task3 {
    type Outcome = bool;
    fn description(&self) -> String {
        "Пусть a, b и c - независимые случайные величины, распределенные␣
 ↪равномерно на [0,1].
Рассмотрим квадратное уравнение a*x^2+b*x+c=0. Чему равна вероятность того, что␣
 ↪его решения -
действительные числа?".to_string()
```

```
    }
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome {
        // Генерируем три числа от 0 до 1
        let a: f64 = rng.gen();
        let b: f64 = rng.gen();
        let c: f64 = rng.gen();
        // Смотрим на значение дискриминанта квадратного уравнения -- (b^2) -␣
 ↪(4ac).
        let discriminant_sq = (b * b) - (4.0 * a * c);
        // Если оно больше или равно нулю, то у уравнения есть действительные␣
 ↪решения.
        discriminant_sq >= 0.0
    }

    fn desired_outcomes(&self) -> Vec<Self::Outcome> {vec![true]}
}

let a = Task3{};
let prob = a.probability_of_desired(1_000_000);
println!("{} -- {}", a.description(), prob);
answers.task_3 = Some(prob);
```

Пусть a, b и c - независимые случайные величины, распределенные равномерно на [0,1].
Рассмотрим квадратное уравнение a*x^2+b*x+c=0. Чему равна вероятность того, что его решения -
действительные числа? -- 0.254331

[7]:
```
Timing: false
sccache: true
```

[8]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"

use rand::prelude::{SliceRandom, IteratorRandom};
struct Task4(usize);

impl Experiment for Task4 {
    type Outcome = bool;
    fn description(&self) -> String {
        format!("В самолете {} мест, и все билеты проданы пассажирам. Первым в␣
 ↪самолет заходит
рассеянный учёный и, не посмотрев на билет, занимает первое попавшееся место.␣
 ↪Далее пассажиры
```

```
входят по одному. Если вошедший видит, что его место свободно, он занимает свое␣
 ↪место. Если же
место занято, то вошедший занимает первое попавшееся свободное место. Найдите␣
 ↪вероятность того,
что пассажир, вошедший последним, займет место согласно своему билету.", self.0)
    }
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome {
        // Решение существует, только если мест не меньше 2.
        assert!(self.0 >= 2);
        // Сначала создаем список мест, список пустых мест и список пассажиров.
        // Пассажиры идут в случайном порядке.
        let mut seats: Vec<Option<usize>> = vec![None; self.0];
        let mut empty_seats: Vec<usize> = (0..self.0).collect();
        let mut passengers: Vec<usize> = (0..self.0).collect();
        passengers.shuffle(rng);

        // Сначала первый пассажир занимает свое место случайно.
        let first_passenger = passengers.pop();
        let first_seat_idx = *empty_seats.iter().choose(rng).unwrap();
        seats[first_seat_idx] = first_passenger;
        empty_seats.retain(|i| *i != first_seat_idx);

        // Затем, все пассажиры до последнего занимают место.
        while empty_seats.len() != 1 {
            let next_passenger = passengers.pop().unwrap();
            // Если место этого пассажира не занято, то пассажир занимает свое␣
 ↪место.
            if seats[next_passenger].is_none() {
                seats[next_passenger] = Some(next_passenger);
                empty_seats.retain(|i| *i != next_passenger);
            } else { // иначе он занимает случайное место
                // Он выбирает одно из мест, которые остались пустыми.
                let random_empty_seat = *empty_seats.iter().choose(rng).unwrap();
                seats[random_empty_seat] = Some(next_passenger);
                empty_seats.retain(|i| *i != random_empty_seat);
            }
        }

        // Наконец, мы смотрим -- место последнего пассажира занято ли?
        let last_passenger = passengers.pop().unwrap();
        seats[last_passenger].is_none()
    }

    fn desired_outcomes(&self) -> Vec<Self::Outcome> {vec![true]}
}

let a = Task4(220);
```

```
let prob = a.probability_of_desired(1_000_000);
println!("{} -- {}", a.description(), prob);
answers.task_4 = Some(prob);
```

```
В самолете 220 мест, и все билеты проданы пассажирам. Первым в самолет заходит
рассеянный учёный и, не посмотрев на билет, занимает первое попавшееся место.
Далее пассажиры
входят по одному. Если вошедший видит, что его место свободно, он занимает свое
место. Если же
место занято, то вошедший занимает первое попавшееся свободное место. Найдите
вероятность того,
что пассажир, вошедший последним, займет место согласно своему билету. --
0.49964
```

[8]:
```
Timing: true
sccache: true
```

[9]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"

struct Task5(f64);

impl PartialExperiment for Task5 {
    type Outcome = f64;
    fn description(&self) -> String {
        format!("Диаметр круга имеет равномерное распределение на [0,{}]. Чему
 ↪равна средняя площадь
круга?", self.0)
    }
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome {
        // Мы выбираем диаметр, умножая случайное число от 0 до 1 на
 ↪максимальный диаметр.
        // Это дает случайное число от 0 до максимума.
        let diameter = rng.gen::<f64>() * self.0;
        let radius = diameter / 2.0;
        // Площадь круга: \pi R^2
        let area = std::f64::consts::PI * radius * radius;
        area
    }
}

let a = Task5(5.0);
let output = {
    // Собираем результаты опыта
    let stats = a.collect_stats(100_000);
```

```
    // и считаем среднее значение
    let avg: f64 = stats.iter().sum::<f64>() / (stats.len() as f64);
    avg
};
println!("{} -- {}", a.description(), output);
answers.task_5 = Some(output);
```

Диаметр круга имеет равномерное распределение на [0,5]. Чему равна средняя площадь
круга? -- 6.537402833236494

[9]:
```
Timing: false
sccache: true
```

[10]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"

/// Для этой задачи нужно хранить количество шагов,
/// а также нижнюю границу интервала случайных чисел.
struct Task6 {
    pub remaining_n: usize,
    pub last_answer: f64,
}

impl Task6 {
    /// Сначала нижняя граница равна нулю.
    pub fn new(n: usize) -> Self {
        Task6 {
            remaining_n: n,
            last_answer: 0.0f64,
        }
    }

    /// На каждом шаге мы меняем границу диапазона
    /// и возвращаем эту границу в результате этой функции.
    pub fn run_single_step<T: rand::Rng>(&mut self, rng: &mut T) -> Option<f64> {
        if self.remaining_n == 0 {return None;}
        // Мы создаем случайное число
        let random: f64 = rng.gen();
        // в диапазоне от нижней границы до 1
        let range_size = 1.0 - self.last_answer;
        // где минимальное число -- это нижняя граница
        let random = self.last_answer + (random * range_size);
        self.last_answer = random;
        self.remaining_n -= 1;
```

```
        Some(random)
    }

    /// Чтобы взять все шаги, мы перемножаем все нижние границы.
    pub fn run_until_completion<T: rand::Rng>(&mut self, rng: &mut T) -> f64 {
        // Начинаем с единицы -- нулевого элемента относительно умножения
        let mut multiplication = 1.0f64;
        // На каждом шаге, умножаем это значение на нижнюю границу
        while let Some(value) = self.run_single_step(rng) {
            multiplication *= value;
        }
        multiplication
    }
}

/// Считаем среднее значение ответа
let avg = {
    let mut rng = rand::thread_rng();
    let mut sum_of_answers = 0.0;
    let mut count = 0;
    for _ in 0..10_000 {
        let mut experiment = Task6::new(100_000);
        sum_of_answers += experiment.run_until_completion(&mut rng);
        count += 1;
    }
    sum_of_answers / (count as f64)
};
println!("Пусть x(1) выбирается наугад из интервала (0,1). Далее x(2) выбирается␣
 ↪наугад из интервала
(x(1),1). Далее x(3) выбирается наугад из интервала (x(2),1) и так далее, т.е.␣
 ↪x(n+1) выбирается наугад
из интервала (x(n),1). Чему равно среднее значение произведения x(1)x(2)...x(n)␣
 ↪если n велико? -- {}", avg);
answers.task_6 = Some(avg);
```

```
Пусть x(1) выбирается наугад из интервала (0,1). Далее x(2) выбирается наугад из
интервала
(x(1),1). Далее x(3) выбирается наугад из интервала (x(2),1) и так далее, т.е.
x(n+1) выбирается наугад
из интервала (x(n),1). Чему равно среднее значение произведения x(1)x(2)...x(n)
если n велико? -- 0.3688154219891535
```

[10]:
```
Timing: true
sccache: true
```

[11]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"
:dep rand_distr = "0.4.3"
:dep plotters = { version="0.3.4", default_features = false, features =␣
↪["evcxr", "all_series"] }

use rand_distr::Distribution;

/// Для этого задания мы должны хранить:
struct Task7 {
    /// количество людей в городе
    pub citizen_count: u64,
    /// вероятность того, что один человек будет на поезде
    pub citizen_train_probability: f64,
    /// текущий размер поезда
    pub current_train_capacity: u64,
}

impl Task7 {
        pub fn new() -> Self {
        Task7 {
            citizen_count: 2500,
            citizen_train_probability: 6.0 / 30.0,
            // Сначала пусть размер поезда равен нулю.
            current_train_capacity: 0,
        }
    }

    /// Для определенного размера мы считаем вероятность переполнения поезда
    pub fn overfill_probability<T: rand::Rng>(&self, rng: &mut T, samples:␣
↪usize) -> f64 {
        // Количество людей на поезде в один день следует биномиальному␣
↪распределению
        let train_users_distribution = rand_distr::Binomial::new(
            self.citizen_count, self.citizen_train_probability
        ).unwrap();
        // Считаем количество раз, когда поезд переполнен
        let mut overfilled = 0;
        for _ in 0..samples {
            // Если измерение из распределения оказывается больше размера поезда,
            // считаем это как переполнение поезда
            let train_users = train_users_distribution.sample(rng);
            if train_users > self.current_train_capacity {
                overfilled += 1
            }
        }
```

```
        (overfilled as f64) / (samples as f64)
    }

    /// На каждом шаге мы оцениваем разницу между вероятностями
    /// переполнения, которая наблюдалась и которая должна быть на самом деле,
    /// и изменяем размер поезда, чтобы совместить их.
    pub fn adjust_train_capacity(&mut self, desired: f64, got: f64) {
        // Погрешность -- это разница между двумя вероятностями
        let error = got - desired;
        // Если погрешность отрицательная, значит поезд слишком большой
        if error < 0.0 {
            self.current_train_capacity -= 1;
        } else { // Иначе поезд слишком маленький
            self.current_train_capacity += 1;
        }

        // (Для простоты я здесь использую шаг в одну единицу размера --
        // для оптимальности этот шаг должен быть пропорциональным погрешности.)
    }
}

// Чтобы проверить, что ответ сошелся, мы рисуем график размера поезда от шагов.
extern crate plotters;
use plotters::prelude::*;

let figure = {
    // Требуется достичь одной неудачи из 100 дней.
    let one_fail_per_days = 100.0f64;

    let mut task = Task7::new();
    let mut rng = rand::thread_rng();
    let figure = evcxr_figure((640, 480), |root|{
        // Инициализируем область для рисования графика
        root.fill(&WHITE)?;
        let root = root.margin(10, 10, 10, 10);

        // График имеет декартовы координаты (0--2000)x(0--2500)
        let mut chart = ChartBuilder::on(&root)
        .x_label_area_size(20)
        .y_label_area_size(40)
        .build_cartesian_2d(0f32..2000f32, 0f32..2500f32)?;

        // У графика есть сетка с 10 делениями по Y-оси
         chart.configure_mesh()
        .y_labels(10)
        .light_line_style(&TRANSPARENT)
        .disable_x_mesh()
```

```
        .draw()?;

        let mut points = vec![];

        for i in 0..2000 {
            // Записываем текущий размер поезда
            points.push( (i as f32, task.current_train_capacity as f32) );
            // Считаем вероятность переполнения поезда
            let overfill_prob = task.overfill_probability(&mut rng, 1000);
            // Изменяем объем поезда относительно вероятности неудачи
            task.adjust_train_capacity(1.0 / one_fail_per_days, overfill_prob);
        }
        // Рисуем точки как красную линию
        chart.draw_series(LineSeries::new(
            points,
            &RED,
        ))?;

        // Выводим текстовый ответ
        println!("В поселке {} жителей. Каждый из них примерно {} раз в месяц␣
 ↪ездит на поезде в город (т.е.
с вероятностью {}), выбирая дни поездок по случайным мотивам независимо от␣
 ↪остальных. Поезд
ходит один раз в сутки. Какой наименьшей вместимостью должен обладать поезд,␣
 ↪чтобы он
переполнялся в среднем не чаще одного раза в {} дней (т.е. с вероятностью {})?␣
 ↪-- {}",
        task.citizen_count,
            task.citizen_train_probability/ (1.0 / 30.0),
            task.citizen_train_probability,
            one_fail_per_days,
            1.0 / one_fail_per_days,
            task.current_train_capacity
        );
        answers.task_7 = Some(task.current_train_capacity);

        Ok(())
    });
    figure
};
figure
```

```
В поселке 2500 жителей. Каждый из них примерно 6 раз в месяц ездит на поезде в
город (т.е.
с вероятностью 0.2), выбирая дни поездок по случайным мотивам независимо от
остальных. Поезд
ходит один раз в сутки. Какой наименьшей вместимостью должен обладать поезд,
```

```
чтобы он
переполнялся в среднем не чаще одного раза в 100 дней (т.е. с вероятностью
0.01)? -- 546
```

[11]:
```
Timing: false
sccache: true
```

[12]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"

use rand::prelude::{SliceRandom, IteratorRandom};
struct Task8(usize);

impl Experiment for Task8 {
    type Outcome = bool;
    fn description(&self) -> String {
        format!("В зале кинотеатра {} мест и все {} билетов были распроданы.␣
 ↪Когда посетители пришли в
зал, свет в зале не работал (номера мест не видны) и каждому пришлось выбирать␣
 ↪себе место наугад.
Чему равна вероятность того, что все {} посетителей сели мимо своих мест␣
 ↪(указанных в билете)?", self.0, self.0, self.0)
    }
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome {
        // Составляем список посетителей и перемешиваем его
        let mut visitors: Vec<usize> = (0..self.0).collect();
        visitors.shuffle(rng);
        for (i, p) in visitors.iter().enumerate() {
            if i == *p {
                // Если чей-то номер совпал с их номером билета, то опыт не␣
 ↪удался
                return false;
            }
        }
        // Иначе, все сели мимо своих мест, и опыт удался
        true
    }

    fn desired_outcomes(&self) -> Vec<Self::Outcome> {vec![true]}
}

let a = Task8(5);
let aans = a.probability_of_desired(1_000_000);
println!("{} -- {}", a.description(), aans);
let b = Task8(50);
```

```
let bans = b.probability_of_desired(1_000_000);
println!("{} -- {}", b.description(), bans);
let c = Task8(100);
let cans = c.probability_of_desired(1_000_000);
println!("{} -- {}", c.description(), cans);
answers.task_8 = Some((aans, bans, cans));
```

```
В зале кинотеатра 5 мест и все 5 билетов были распроданы. Когда посетители
пришли в
зал, свет в зале не работал (номера мест не видны) и каждому пришлось выбирать
себе место наугад.
Чему равна вероятность того, что все 5 посетителей сели мимо своих мест
(указанных в билете)? -- 0.366745
В зале кинотеатра 50 мест и все 50 билетов были распроданы. Когда посетители
пришли в
зал, свет в зале не работал (номера мест не видны) и каждому пришлось выбирать
себе место наугад.
Чему равна вероятность того, что все 50 посетителей сели мимо своих мест
(указанных в билете)? -- 0.367519
В зале кинотеатра 100 мест и все 100 билетов были распроданы. Когда посетители
пришли в
зал, свет в зале не работал (номера мест не видны) и каждому пришлось выбирать
себе место наугад.
Чему равна вероятность того, что все 100 посетителей сели мимо своих мест
(указанных в билете)? -- 0.367601
```

[12]:
```
Timing: true
sccache: true
```

[13]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"
:dep itertools = "0.10.5"

use rand::prelude::{SliceRandom, IteratorRandom};

/// В этом задании есть 4 группы, из которых набираются члены в группу␣
 ↪определенного размера;
/// здесь мы храним размеры этих групп.
struct Task9{
    pub group_size: usize,
    pub cat1_size: usize,
    pub cat2_size: usize,
    pub cat3_size: usize,
    pub cat4_size: usize,
}
```

```rust
impl Task9 {
    pub fn new() -> Self {
        Task9 {
            group_size: 6,
            cat1_size: 6,
            cat2_size: 6,
            cat3_size: 10,
            cat4_size: 12,
        }
    }
}

use core::iter::repeat;
use itertools::Itertools;
impl Experiment for Task9 {
    type Outcome = u8;
    fn description(&self) -> String {
        format!("На факультете работает {} профессоров, {} доцентов, {} старших␣
 ↪преподавателей и {}
ассистентов. Из работников факультета случайным образом формируется комитет из␣
 ↪{} участников.
Чему равна вероятность того, что в комитет войдет по крайней мере один человек␣
 ↪каждой должности?",
        self.cat1_size,
        self.cat2_size,
        self.cat3_size,
        self.cat4_size,
        self.group_size,
        )
    }
    fn try_it<T: rand::Rng>(&self, rng: &mut T) -> Self::Outcome {
        // Группу можно набрать, только если членов всех остальных групп␣
 ↪достаточно
        assert!(self.cat1_size + self.cat2_size + self.cat3_size + self.
 ↪cat4_size >= self.group_size);
        // Составляем список, состоящий из значений 1, 2, 3 и 4 для каждой из␣
 ↪исходных групп.
        let mut members: Vec<u8> = Vec::with_capacity(self.cat1_size + self.
 ↪cat2_size + self.cat3_size + self.cat4_size);
        members.extend(repeat(1).take(self.cat1_size));
        members.extend(repeat(2).take(self.cat2_size));
        members.extend(repeat(3).take(self.cat3_size));
        members.extend(repeat(4).take(self.cat4_size));
        // Перемешиваем этот список
        members.shuffle(rng);
```

```
        // Берем первые несколько элементов, из них выбираем уникальные и␣
↪считаем их
        members.iter().take(self.group_size).unique().count() as u8
    }

    /// Нам нужно, чтобы уникальных элементов было 4.
    fn desired_outcomes(&self) -> Vec<Self::Outcome> {vec![4]}
}

let a = Task9::new();
let prob = a.probability_of_desired(1_000_000);
println!("{} -- {}", a.description(), prob);
answers.task_9 = Some(prob);
```

На факультете работает 6 профессоров, 6 доцентов, 10 старших преподавателей и 12 ассистентов. Из работников факультета случайным образом формируется комитет из 6 участников.
Чему равна вероятность того, что в комитет войдет по крайней мере один человек каждой должности? -- 0.378413

[13]:
```
Timing: false
sccache: true
```

[14]:
```
:timing 1
:sccache 1
:dep rand = "0.8.5"
:dep rand_distr = "0.4.3"


/// Для этой задачи требуется хранить экземпляр распределения,
/// членами которого являются пары действительных чисел.
struct Task10<T: rand_distr::Distribution<[f64; 2]>> {
    distribution: T,
}

// Эти функции взяты из данного ответа на StackOverflow:
// https://stackoverflow.com/a/2049593/5936187
// Они связаны с вопросом того, находится ли точка внутри треугольника.
fn sign(a: [f64; 2], b: [f64; 2], c: [f64; 2]) -> f64 {
    (a[0]-c[0]) * (b[1]-c[1]) - (b[0]-c[0]) * (a[1]-c[1])
}

fn point_in_triangle(p: [f64;2], a: [f64; 2], b: [f64; 2], c: [f64; 2]) -> bool {
    let d1 = sign(p, a, b);
    let d2 = sign(p, b, c);
    let d3 = sign(p, c, a);
```

```
    let has_neg = (d1<0.0) || (d2<0.0) || (d3<0.0);
    let has_pos = (d1>0.0) || (d2>0.0) || (d3>0.0);
    !(has_neg && has_pos)
}

impl<T> Experiment for Task10<T>
where T: rand_distr::Distribution<[f64; 2]>
{
    type Outcome = bool;
    fn description(&self) -> String {
        format!("Чему равна вероятность того, что 4 случайно выбранные точки в␣
 ↪данном выпуклом
множестве являются вершинами вогнутого четырехугольника?")
    }
    fn try_it<R: rand::Rng>(&self, rng: &mut R) -> Self::Outcome {
        // Сначала мы выбираем 4 точки из данного распределения
        let (a,b,c,d) = (
            self.distribution.sample(rng),
            self.distribution.sample(rng),
            self.distribution.sample(rng),
            self.distribution.sample(rng),
        );
        // Чтобы определить, является ли четырехугольник вогнутым, мы проверяем:
        // Можно ли выбрать три точки так, что четвертая точка находится посреди␣
 ↪них?
        // Если да, то этот четырехугольник вогнутый,
        // а иначе он равен своей выпуклой оболочке и поэтому выпуклый.
        for ordering in [a,b,c,d].iter().permutations(4) {
            if point_in_triangle(*ordering[0], *ordering[1], *ordering[2],␣
 ↪*ordering[3]) {
                return true;
            }
        }
        false
    }

    fn desired_outcomes(&self) -> Vec<Self::Outcome> {vec![true]}
}

let a = Task10 {
  distribution: rand_distr::UnitDisc,
};

let prob = a.probability_of_desired(1_000_000);
println!("{} -- {}", a.description(), prob);
answers.task_10 = Some(prob);
```

Чему равна вероятность того, что 4 случайно выбранные точки в данном выпуклом множестве являются вершинами вогнутого четырехугольника? -- 0.295522

[14]:
```
Timing: true
sccache: true
```

[17]:
```
// Посмотреть состояние ответов на задания
println!("Задание 1: {:?}", answers.task_1.unwrap());
println!("Задание 2: {:?}", answers.task_2.unwrap());
println!("Задание 3: {:?}", answers.task_3.unwrap());
println!("Задание 4: {:?}", answers.task_4.unwrap());
println!("Задание 5: {:?}", answers.task_5.unwrap());
println!("Задание 6: {:?}", answers.task_6.unwrap());
println!("Задание 7: {:?}", answers.task_7.unwrap());
println!("Задание 8: {:?}", answers.task_8.unwrap());
println!("Задание 9: {:?}", answers.task_9.unwrap());
println!("Задание 10: {:?}", answers.task_10.unwrap());
```

```
Задание 1: 0.725906
Задание 2: 0.7984

Задание 3: 0.254331
Задание 4: 0.49964
Задание 5: 6.537402833236494
Задание 6: 0.3688154219891535
Задание 7: 546
Задание 8: (0.366745, 0.367519, 0.367601)
Задание 9: 0.378413
Задание 10: 0.295522
```

[ ]: