# Определения, обозначения и сокращения

ГПСЧ - генератор псевдослучайных чисел.

НИСТ - Национальный институт стандартов и технологий.

# Введение

ГПСЧ находят широкое применение в криптографии (в системах защиты информации), в численных методах (метод Монте-Карло), моделировании и многих других областях. Для проверки их работы рекомендуется использовать статистические тесты. К примеру, Национальный Исследовательский институт Стандартов и Технологий (NIST) рекомендует 15 таких тестов [1]. В последние годы появились новые тесты, основанные на методах теории информации [2] (или сжатии данных).

**union**

В наше время существует проблема поиска стойкого ГПСЧ. Стойкость генератора определяется его способностью проходить статистические тесты, основанные на поиске закономерностей в последовательностях псевдослучайных чисел. Цель данной работы - адаптировать набор таких тестов для проверки ГПСЧ на случайность. Новизна состоит в том, что кроме тестов НИСТ будут использован новый тест: стопка книг. Весь этот набор тестов будет проверять не только стандартные генераторы, которые уже существуют в наше время, но и будет проверен новый ГПСЧ.

# Изучение тестов

Были изучены статистические тесты НИСТ, описание которых можно найти в [4]. Суть тестов в выявлении отклонений в последовательности чисел, которые выдаёт ГПСЧ, от случайной последовательности.

# Описание тестов (как нист, так и стопки книг…)

Здесь будет описание

(Определение уровня значимости)

**Frequency Test (Частотный тест)**

Цель теста выявить, действительно ли количество нулей и единиц во входной последовательности приближённо такое же, как у предполагаемой случайной последовательности. Тест оценивает близость доли единиц к 0.5.

В этом тесте единицы переводятся сами в себя, а нули – в -1. Считается сумма всех элементов, делится на корень из длины последовательности и вычисляется дополнительная функция ошибок [10] от полученного числа. Если полученное значение функции меньше 0.05 (уровень значимости), значит принимается гипотеза о не случайности последовательности. Иначе, последовательность считается случайной.

**Frequency Test within a Block (Частотный тест в пределах блока)**

Цель теста выявить, действительно ли количество нулей и единиц внутри блока размера *М* во входной последовательности приближённо (*М / 2)*, как у предполагаемой случайной последовательности.

Последовательность делиться на непересекающиеся блоки, длины *М*. Подсчитывается частота единиц в каждом блоке, и с помощью критерия согласия Пирсона измеряется соответствие выявленной пропорции единиц в последовательности, ожидаемой (1/2).

**Runs Test (**

The focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits. A run of length *k* consists of exactly *k* identical bits and is bounded before and after with a bit of the opposite value. The purpose of the runs test is to determine whether the number of runs of

------Можно написать про LinearComplexity, что из результатов следует, что этот тест не пригоден для последовательностей, меньше, чем 10^6. Да не, фигня, типа только для параметра 8 не работает

# Адаптация тестов

Была скачана программа статистических тестов НИСТ с ресурса [5]. Данная программа предоставляет консольный интерфейс. Разобравшись с интерфейсом, я пришёл к выводу, что он [интерфейс] очень не удобен:

1. **Неудобный ввод данных.** Каждый раз приходится заново вводить определённое количество входных параметров, которые можно было бы поставить по умолчанию или передавать их в качестве аргументов (пример на ОС Linux: “$ > ./a.out param1=val1 param2=val2 …”).
2. **Единственный способ получения результирующей информации.** Результаты работы программы заносятся в файл в определённом формате в свою поддиректорию. Формат файла читаем и воспринимаем человеком, но не удобен для разработчика, который хочет быстро достать из неё информацию и по-своему её обработать.
3. **Плохо написанная программа с точки зрения расширяемости.** Программа написана на языке “C”. Она трудночитаема и сложно-разбираема (*смотри в словаре*). Компоненты программы сильно связаны друг с другом. Например, код, отвечающий за вывод результирующих данных и тестирование последовательностей на случайность, находится в одной и той же функции. Наличие глобальных переменных. Отсутствие директив, необходимых для того, чтобы заголовочный файл не мог повторяться при включении его в больше, чем один, исходный файл.
4. **Проблема распараллеливания.** Проблема, вытекающая из третьего пункта, заключается в том, что нельзя распараллелить программу при том факте, что каждая последовательность тестируется независимо от других, вследствие чего можно тестирование последовательности может происходить на разных потоках.

В виду перечисленных недостатков интерфейса, я принял решение, чтобы разработать собственную программу (уже на языке C++), в которую включу только код с тестами. Как известно, язык программирования С++ является надмножеством языка С (не считая некоторых исключений, под которые данная программа не проходит [***подумать, стоит ли это писать, либо нужно расписывать и объяснять почему так***]). Так как функции, которые реализуют тесты, занимаются ещё и выводом результирующей информации в файл, пришлось удалить эту часть кода из функций.

В связи с тем, что код функций был изменён (хоть и незначительно), поведение тестов могло измениться (не считаю отсутствие вывода в файл). Соответственно, чтобы не допустить возможных ошибок, был написан ряд модульных тестов для функций, реализующих статистические тесты ([***что представляют собой эти тесты***]).

----кек----

Также был использован новый статистический тест, основанный на алгоритме стопка книг, который можно загрузить с ресурса [1]. Его в том числе пришлось адаптировать к написанной мной программе, которая в результате стала являть собой адаптацию НИСТ тестов и стопки книг.

# Описание программы

---------------------------------Нужно где-то описать, что имеем дело с **битовыми** последовательностями---------------------------

Программа запускается через консоль. При запуске ей передаются аргументы:

1. статистические тесты, которые нужно запустить (всего 17 тестов, нужно ввести последовательность из 17 нулей и единиц, где единица – тест используется, 0 – тест пропускается);
2. количество последовательностей, которые будут тестироваться;
3. название генератора (**список генераторов перечислен во главе такой-то**) или название файла (формат: “file=type reading=’filename’”, где type reading – {2, 8}, что означает, файл считывается по биту или по байту символа ASCII [‘0’ и ‘1’]);
4. количество длин последовательностей
5. перечисление длин (длина последовательности указывается в битах, например «1000» - значит длина последовательности 1000 бит).

Во время работы программа выдаёт разную отладочную информацию, связанную с запуском как отдельных тестов, так и в совокупности (например, время работы всей программы).

В конце работы программа выдаёт текстовый файл, в котором содержатся результаты тестирования последовательностей. Первой строкой указываются названия тестов (и с параметром, которым они запускались). Каждая следующая строка выглядит следующим образом: размер последовательности и результат её тестирования по каждому тесту (количество последовательностей, которые не прошли данный тест). Столбцы разделяются табуляцией. Это сделано для того, чтобы можно было легко копировать и вставлять данный файл в табличный документ, например, в программу Microsoft Excel или LibreOffice Calc.

# Отличие от НИСТ реализации

В виду перечисленных недостатков (в таком-то **пункте**)в моей реализации есть хороший API (Application Programming Interface). Каждый тест обёрнут в класс, который наследован от абстрактного класса *IStatisticalTest*. Эти классы перегружают чистую виртуальную функцию *test*. Сделано это для того, чтобы иметь возможность распространять на них общее поведение. Например, хочется замерить, сколько процессорного времени потратил каждый тест. Вместо того, чтобы замерять время над каждым тестом (писать 17 замеров времени, т.е. 17 дублирований кода), можно с помощью виртуального полиморфизма положить каждый объект класса, обёртывающего тест, в контейнер, и в цикле, запуская тест, написать один замер времени возле вызова виртуального метода *test*.

Данный API удобен тем, что если появится потребность написать GUI для данной программы, не нужно перенаправлять ввод/вывод программы, не нужно разбирать файл с результатами программно, чтобы вывести их в пользовательском интерфейсе. Достаточно вызвать функцию *runStatisticalTests*, передав ей в качестве параметров:

1. итераторы на начало и конец последовательности (**\***);
2. ключи, в которых содержится информация о том, какие тесты запускать, а какие пропустить (\*\*);
3. параметры, с которыми нужно запускать тесты (в дополнение можно отметить, что параметры можно передать по умолчанию, вызвав конструктор класса *TestParameters* без аргументов).

(**\***) Стоит заметить, что в моей реализации контейнер и итераторы спрятаны в *typedef*, в котором указан конкретный контейнер *vector<bool>* и соответствующий ему итератор. Плюс данного подхода, что если хочется изменить тип контейнер, достаточно поменять тип в одном месте. Минус такого подхода: всю программу придётся перекомпилировать, что занимает в районе нескольких минут, так как почти все компоненты программы зависят от типа контейнера.

Функция возвращает массив элементов типа перечисления, который имеет три значения: SUCCESS – тест завершился успешно, последовательность, признана случайной; FAILURE – тест завершился, последовательность признана неслучайной, CANCELLED – тест прервался, в связи с определённым обстоятельствами (неправильный переданный аргумент, последовательность не удовлетворяет критерию, поэтому тест не может дать ответ, случайна она или нет и т.д.). Элементы в массиве располагаются в порядке вызова тестов. Чтобы узнать, какой элемент принадлежит тесты, нужно вызвать функцию *getStatisticalTestNames*, с переданным ему ключом (\*\*) и размером последовательности. Функция возвращает массив строк, в котором хранятся названия статистических тестов. Индекс в массиве строк соответствует индексу в массиве результата запуска тестов. Соответственно, получаем всю необходимую информацию о результатах тестирования одной последовательности: название теста и результат (последовательность случайна, не случайна или тест отменён).

Также чтобы не писать свою реализацию считывания из файла битовых последовательностей в различных форматах: через биты или ASCII символы - можно воспользоваться готовыми функциями *readSequenceByBitFromFile* и *readSequenceByByteFromFile*, соответственно.

# Проверка теста стопки книг

Тест стопка книг –

# Проверка стандартных генераторов

Были проверены ГПСЧ в стандартной библиотеке C++ std.

# Оптимизация программы

Тестирование ГПСЧ с помощью предыдущей версии программы занимало много времени для длин последовательностей больше, чем 100 Кбит. Например, время, затрачиваемое на тестирование 100 последовательностей с длиной в 100 Кбит, занимало примерно 10 минут.

Для ускорения работы программы была использована библиотека OpenMP [2]. Так как в реализованном алгоритме последовательности псевдослучайных чисел тестируются независимо друг от друга, с помощью директив, предоставляемых библиотекой, работа программы в этой части стала параллельной. Т.е. в каждый момент времени работы программы N последовательностей тестируются в N различных потоках, и, если мы имеем N исполнительных устройств, то тестируются параллельно (параллелизм в пространстве). Время работы уменьшается пропорционально количеству задействованных исполнительных устройств.

# Реализация нового статистического теста

Тест «Order Test», описанный в публикации [5] (раздел 3.2 «Order Test»), является модификацией теста «стопки книг» ([5] раздел 3.1 «The Book Stack Test»). Тест был выбран в соответствие с быстротой его работы.

**Описание теста**

На вход тесту подаётся последовательность из символов алфавита *А*. Есть отсортированный массив («стопка»), который хранит частоту (счётчик) встречаемости символа в последовательности до момента *t* (позиция, до которой включительно последовательность была просмотрена). Элементы в массиве отсортированы таким образом, чтобы в начале массива (в вершине стопки) находились те символы, которые встречались в последовательности чаще других. Каждая итерация алгоритма описывается следующим образом: достаётся следующий символ из последовательности, потом ищется в стопке. После нахождения данного элемента, счётчик текущего символа увеличивается на единицу. Далее стопка сортируется (в соответствие со свойством).

**Реализация**

Класс, который реализует тест, выделяет память под: контейнер, который реализует «стопку» (хранит частоту встречаемости символа в последовательности), соответствие в виде массива (ключ: символ алфавита *А*; значение: индекс элемента в контейнере – для быстрого нахождения элемента в стопке), массив границ. Массив границ представляет собой множество элементов, каждый из которых является границей конкретного *ряда* элементов массива частот (определение ряда даётся дальше).

Элементы в стопке упорядочены по частоте встречаемости, поэтому наша стопка представляет собой совокупность *рядов*, подряд идущих элементов с одинаковой частотой. Достаточно хранить «ссылку» (индекс) на последний элемент в этом ряду, чтобы быстро получить все элементы этого *ряда*. Назовём эту «ссылку» *границей* – индекс в стопке, указывающий на конец этого *ряда*. Набор таких границ будем хранить в хэш-таблице. Хэш-таблица нужна для получения границы по ключу (частота ряда) за константное время.

Допустим, вытаскиваем очередной символ из последовательности. С помощью соответствия достаём индекс элемента в стопке. Увеличиваем счётчик (частоту) данного символа на единицу. Теперь данный элемент стопки должен переместиться в новый или уже существующий ряд с новой частотой (которая на единицу больше, чем предыдущая частота текущего элемента). С помощью хэш-таблицы достаём по ключу (частоте) индекс, указывающий на следующий после последнего элемента ряда элемент. Меняем эти два элемента местами, увеличивая индекс границы на единицу (т.е. сдвигая нашу границу, тем самым расширяя ряд новой частоты). Если же данной границы нет, то создаём новую границу, добавив её в хэш-таблицу. Текущий элемент меняем с верхним элементом стопки. Новую границу ставим на соответствующее место.

Во время работы теста могут появляться границы, совпадающие с другими границами по индексу, на который они ссылаются. Т.е. появляться пустые ряды. Чтобы не засорять хэш-таблицу, их нужно удалять. Проверка на удаление происходит каждый раз, как вытащенный символ попадает в новый ряд с новой частотой, тем самым опустошая свой предыдущий ряд. С помощью std::map (хэш-таблицы из стандартной библиотеки С++) удаление можно сделать за константное время [4].

**Сложность работы**

Сложность алгоритма: O(N \* log(N)) – худший случай. Среднее время: O(N). Каждая итерация цикла выполняется в среднем за константное время.

**Принятие решения**

На вход тесту подаётся размер верхней стопки. Программа считает, сколько каждый символ побывал в этой верхней части стопки. И считает статистику хи-квадрат для этих данных. В соответствие с таблицей распределения хи-квадрат по статистике достаётся вероятность, которая сравнивается с 0,95. Если вероятность больше данного числа, то принимается гипотеза о не случайности последовательности. Иначе последовательность случайна.

**Результаты**

Тест проверялся на числе Пи, которое было признано случайным набором статистических тестов, имеющихся в данной программе. При стопки для двух элементов (0 и 1), на длинах последовательности до 1000 тест бракуют их: т.е. выводит, что последовательности неслучайны. Отладив программу последовательности из тысячи элементов, были получены такие результаты: количество единиц и нулей встречалось 511 и 489 раз, соответственно; количество раз, которое единица побывала в верхней части стопки, равно 822 (нуль – 178). Т.е. расхождение больше, чем в 4 раза. При увеличении длины последовательности расхождение только возрастает. Следовательно, таким конкретно таким тестом нельзя мерить случайность последовательностей.

# Список литературы

[1] <https://github.com/sashasasha-1987/book-stack>

[2] <https://www.nist.gov/programs-projects/nist-randomness-beacon>

[3] <https://github.com/grempe/nist-randomness-beacon>

[4] <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf>

[5] https://sourceforge.net/projects/randomanalysis/

[6] <http://boris.ryabko.net/published.pdf>

[7] http://ssd.sscc.ru/sites/default/files/content/attach/343/lecture\_openmp\_2015.pdf

[8] http://pi2e.ch/blog/2017/03/10/pi-digits-download/

[9] <http://en.cppreference.com/w/cpp/container/map/erase>

[10] https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F\_%D0%BE%D1%88%D0%B8%D0%B1%D0%BE%D0%BA