**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**

(**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**)» (**МАИ)**

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра 304 Вычислительные машины, системы и сети

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ **№ 1**

**“Формирование массивов**

**экспериментальных данных”**

Группа: *М30-209Б-19*

Выполнил:

*Кузнецов И.И*

Проверил:

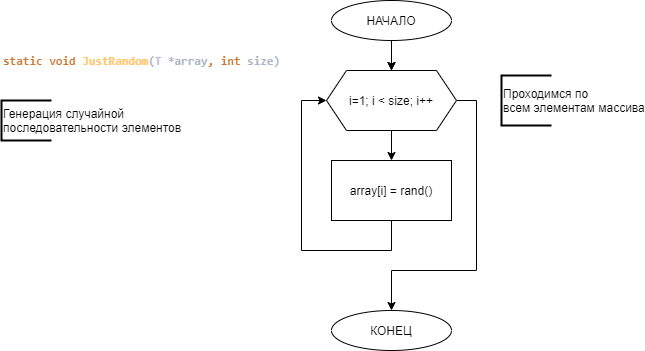
*Ким Роман Валерьевич*

**Москва 2020**

**Описание функции для генерации случайной последовательности**

Проходимся по всем элементам массива и случайно генерируем элемент.

**Структурная схема алгоритма**

****

**Код алгоритма**

**static void JustRandom(T \*array, int size) {  
 for (int i = 0; i < size; ++i) {  
 array[i] = rand();  
 }  
}**

**График времени формирования последовательности**

**График последовательности**

**Описание функции для генерации возрастающей последовательности**

Так как нам требуется получить возрастающую последовательность нам нужно чтобы каждый следующий элемент был больше чем предыдущий.

Это достигается путем создания возрастающей прямой, “вокруг” которой генерируются элементы.

Возрастающая прямая получается таким образом:

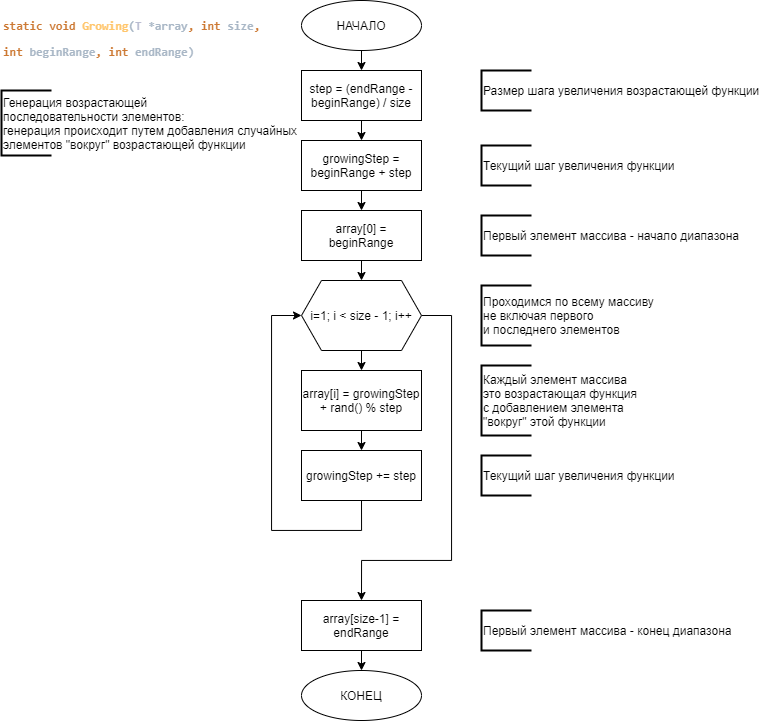
1) Вычисляем шаг(step) возрастания функции, по которой строится прямая, т.е. на сколько следующий элемент должен был быть больше предыдущего, (не учитывая случайные числа) и учитывая то, что у нас существует минимальное и максимальное значение в интервале.

2) GrowingStep отвечает за текущее положение точки в интервале, то есть мы запоминаем какое значение прямой (без случайных чисел) было на прошлой итерации и добавляем к нему еще один шаг возрастания функции, так как следующее значение в интервале должно быть больше предыдущего, для того чтобы функция имела правильный вид.

3) Значение текущего элемента мы получаем как сумма “чистой” прямой, т.е. growingStep и rand()%step, т.к. функция rand будет генерировать числа в диапазоне от 0 до step-1 и мы можем гарантировать что следующий элемент будет больше предыдущего.

Причины такой сложной реализации — это то, что мы должны учитывать минимальное и максимальное значение в диапазоне, если бы данного условия не было бы, мы могли бы брать генерировать элементы как сумма предыдущего элемента и какого-то случайного числа (>0).

**Структурная схема алгоритма**

****

**Код алгоритма**

**static void Growing(T \*array, int size, int beginRange, int endRange) {  
 T step = (endRange - beginRange) / size;  
 int growingStep = beginRange + step;  
 array[0] = beginRange;  
 for (int i = 1; i < size - 1; ++i) {  
 array[i] = growingStep + rand() % (int) step;  
 growingStep += step;  
 }  
 array[size - 1] = endRange;  
}**

**График времени формирования последовательности**

**График последовательности**

**Описание функции для генерации убывающей последовательности**

Так как нам требуется получить убывающую последовательность нам нужно чтобы каждый следующий элемент был меньше, чем предыдущий.

Это достигается путем создания убывающей прямой, “вокруг” которой генерируются элементы.

Убывающая прямая получается таким образом:

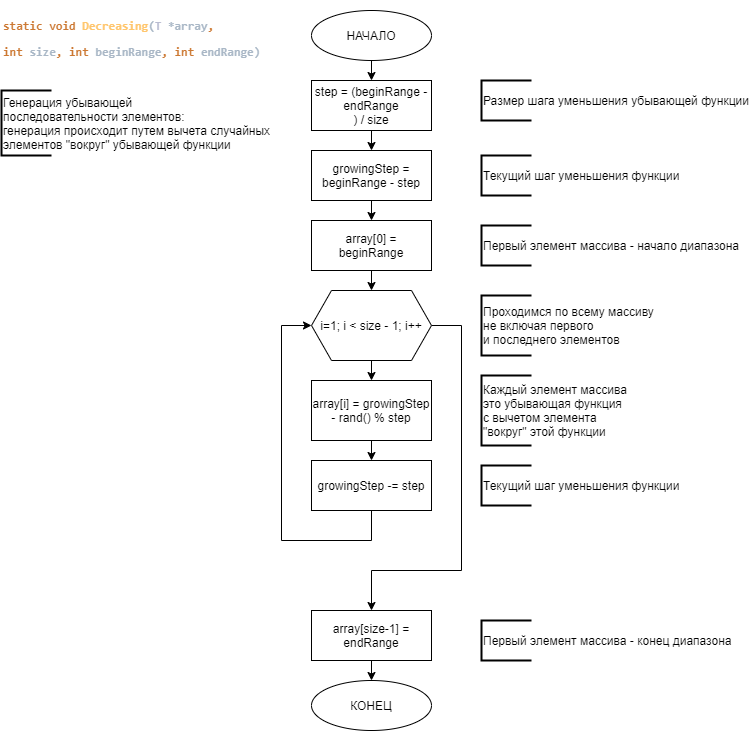
1) Вычисляем шаг(step) убывания функции, по которой строится прямая, т.е. на сколько следующий элемент должен был быть меньше предыдущего, (не учитывая случайные числа) и учитывая то, что у нас существует минимальное и максимальное значение в интервале.

2) GrowingStep отвечает за текущее положение точки в интервале, то есть мы запоминаем какое значение прямой (без случайных чисел) было на прошлой итерации и вычитываем из него еще один шаг убивания функции, так как следующее значение в интервале должно быть меньше предыдущего, для того чтобы функция имела правильный вид.

3) Значение текущего элемента мы получаем как разность “чистой” прямой, т.е. growingStep и rand()%step, т.к. функция rand будет генерировать числа в диапазоне от 0 до step-1 и мы можем гарантировать что следующий элемент будет меньше предыдущего.

Причины такой сложной реализации — это то, что мы должны учитывать минимальное и максимальное значение в диапазоне, если бы данного условия не было бы, мы могли бы брать генерировать элементы как разность предыдущего элемента и какого-то случайного числа (>0).

**Структурная схема алгоритма**

****

**Код алгоритма**

static void Decreasing(T \*array, int size, int beginRange, int endRange) {  
 T step = (beginRange - endRange) / size;  
 int decreasingStep = beginRange - step;  
 array[0] = beginRange;  
 for (int i = 1; i < size - 1; ++i) {  
 array[i] = decreasingStep - rand() % (int) step;  
 decreasingStep -= step;  
 }

array [size - 1] = endRange;  
}

**График времени формирования последовательности**

**График последовательности**

**Описание функции для генерации синусоидальной последовательности**

Последовательность получения синусоидальной последовательности:

1. Высчитываем шаг увеличения синуса, так как у нас есть несколько интервалов (пр.: 5 повторяющихся синусов) то мы должны учитывать их. Играет роль итератора по синусоидальной функции, которая является ведущей функцией (т.е. учитываем, что функция должна то плавно возрастать, то плавно убывать).
2. Высчитываем шаг центра синусоиды для того, чтобы учитывать макс. и мин. значения в диапазоне.
3. Высчитываем диапазон, в котором будут генерироваться случайные числа.
4. После мы получаем значение элемента на текущей итерации, логика такова:

4.1). У нас есть макс. и мин. значения диапазона, поделим их разность пополам (step) (\*).

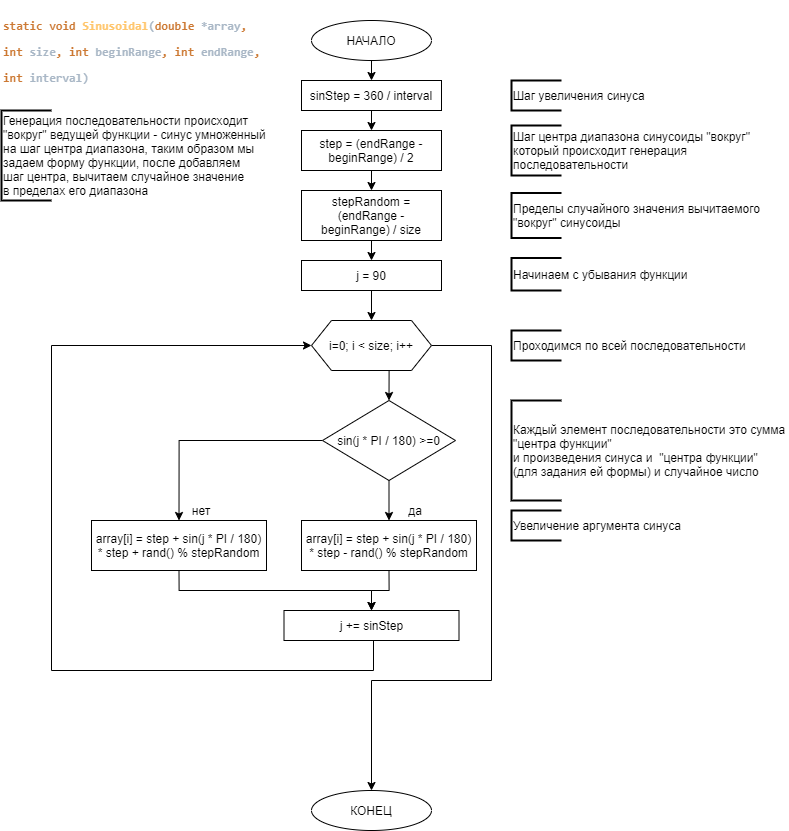
4.2). Если мы добавим к текущему значению элемента (\*), то у нас будет центральное значение диапазона.

4.3). Так как функция синус изменяется от [-1;1] то умножая синус функции на (\*) мы получаем ожидаемую форму графика последовательности и гарантируем что не зайдем за границу диапазона.

4.4). Сложив 4.2) и 4.3) мы получим график функции синус в нужном нам диапазоне.

4.5). Складываем или вычитываем случайную величину в зависимости от текущего знака синуса, для того чтобы не вылезти за границы диапазона.

**Структурная схема алгоритма**



**Код алгоритма**

**static void Sinusoidal(double \*array, int size, int beginRange, int endRange, int interval) {  
 double sinStep = 360.0 / interval;  
 int step = (endRange - beginRange) / 2;  
 int stepRandom = (endRange - beginRange) / size;  
 double j = 90.0;  
 for (int i = 0; i < size; ++i) {**

**if (sin (j \* PI / 180) >= 0)  
 array[i] = step + sin(****j \* PI / 180) \* step - rand() % stepRandom;**

**else**

**array[i] = step + sin(j \* PI / 180) \* step + rand() %stepRandom;  
 j += sinStep;  
 }**

**График времени формирования последовательности**

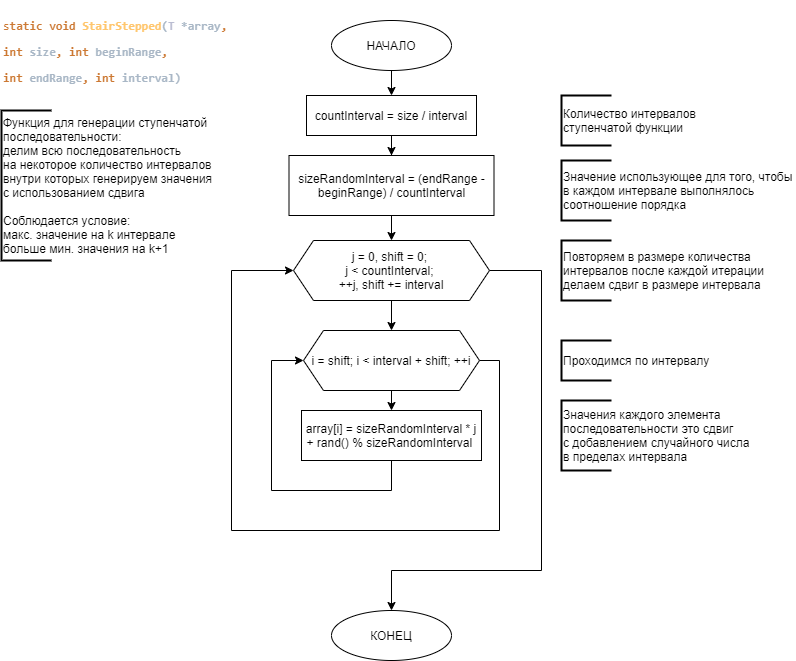
**График последовательности**

**Описание функции для генерации ступенчатой последовательности**

Последовательность генерации ступенчатой последовательности:

1. Высчитываем количество интегралов в последовательности, в которых будет выполняться соотношение: минимальное значение k+1 интервала больше максимального значения k-го интервала.
2. Высчитываем sizeRandomInterval, данный параметр как раз выполняет роль ограничителя, для выполнения соотношения
3. Далее мы идем проходимся по всем интервалам, параметр shift во внешнем цикле отвечает за передвижение границ внутреннего цикла
4. Генерируем значение: если у нас первый интервал, то значение будет 0 + rand [0; sizeRandomInterval - 1], в следующем интервале минимальное значение должно быть больше, чем максимальное значение в предыдущем. Поэтому в следующем интервале значения будет sizeOfInterval + rand [0; sizeRandomInterval - 1], что как раз гарантирует выполнение соотношения, так как максимальное значение в пред. интервале будет sizeRandomInterval-1, а минимальное значение на текущем интервале sizeRandomInterval ч.т.д.

**Структурная схема алгоритма**



**Код алгоритма**

**static void StairStepped(T \*array, int size, int beginRange, int endRange, int interval) {  
 int countInterval = size / interval;  
 int sizeRandomInterval = (endRange - beginRange) / countInterval;  
 for (int j = 0, shift = 0; j < countInterval; ++j,  
 shift += interval) { //сдвиг вправо для создания нескольких интервалов {  
 for (int i = shift; i < interval + shift; ++i) {  
 array[i] = sizeRandomInterval \* j + rand() % sizeRandomInterval;  
 }  
 }  
}**

**График времени формирования последовательности**

**График последовательности**

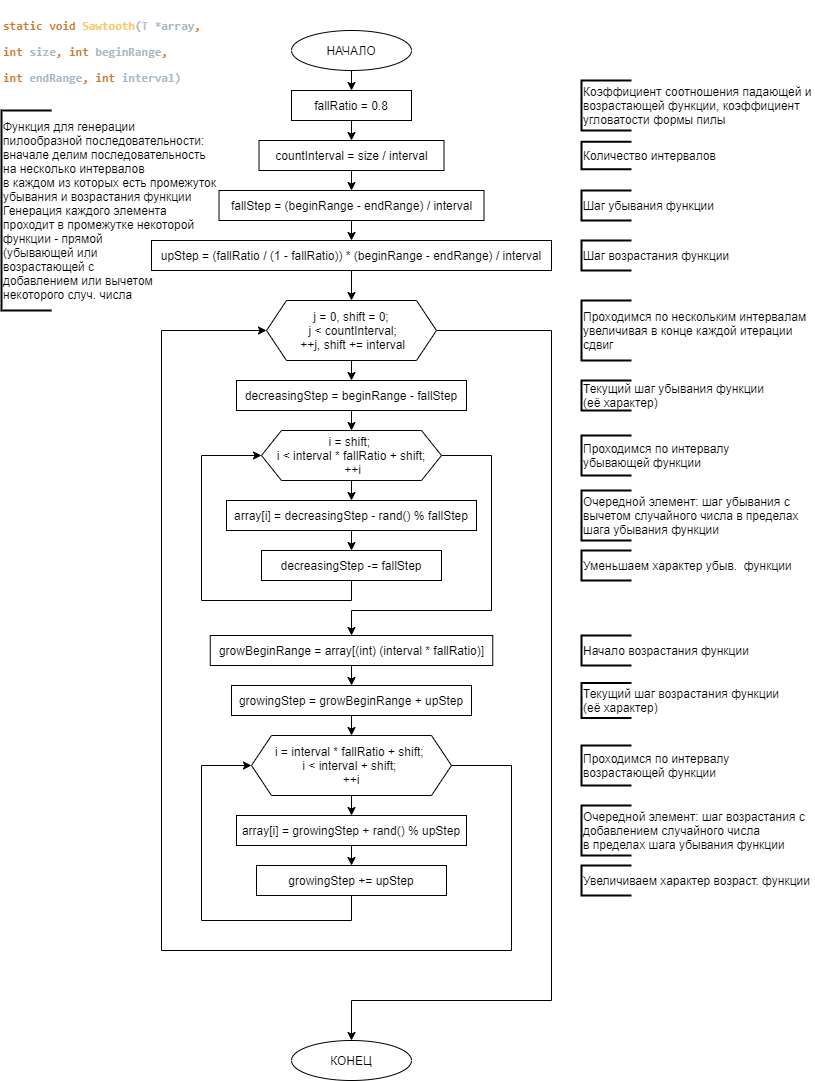
**Описание функции для генерации пилообразной последовательности**

Логика генерации последовательности аналогична логики генерации возрастающей и убывающей последовательности. Но также мы учитываем то, что у нас есть несколько интервалов, на которых функция должна вначале убывать, а далее обратно возрастать.

Логика такова:

1. Мы самостоятельно задаем коэффициент, который отвечает за соотношение убывающей части графика и возрастающей. Т.е. если у нас в одном интервале 1000 элементов, а коэффициент равен 0.8, то 800 элементов будут убывать, а 200 возрастать.
2. Далее мы высчитываем шаг возрастания и убывания функции с учетом коэффициента, так как у возрастающей и убывающей части графиков, разная скорость падения/возрастания. Аналогичен по функциональности с возрастающей/убывающей последовательностью.
3. Далее мы делим всю последовательность на несколько интервалов, в каждом из которых есть две части интервала (размер каждого из интервалов определяется коэффициентом): часть возрастания и убывания соответственно. Генерируем значения по такой же логике, как и у возрастающей/убывающей последовательности.

**Структурная схема алгоритма**



**Код алгоритма**

**static void Sawtooth(T \*array, int size, int beginRange, int endRange, int interval) {  
 double fallRatio = 0.8;  
 int countInterval = size / interval;  
  
 int fallStep = (beginRange - endRange) / interval;  
 int upStep = (fallRatio / (1 - fallRatio)) \* (beginRange - endRange) / interval;  
  
 for (int j = 0, shift = 0; j <= countInterval; ++j, shift += interval) {  
 int decreasingStep = beginRange - fallStep;  
  
 for (int i = shift; i < interval \* fallRatio + shift; ++i) {  
 array[i] = decreasingStep - rand() % fallStep;  
 decreasingStep -= fallStep;  
 }  
  
 int growBeginRange = array[(int) (interval \* fallRatio)];  
 int growingStep = growBeginRange + upStep;  
  
 for (int i = interval \* fallRatio + shift; i < interval + shift; ++i) {  
 array[i] = growingStep + rand() % upStep;  
 growingStep += upStep;  
 }  
 }  
}**

**График времени формирования последовательности**

**График последовательности**

**Описание функции для генерации квази-последовательности**

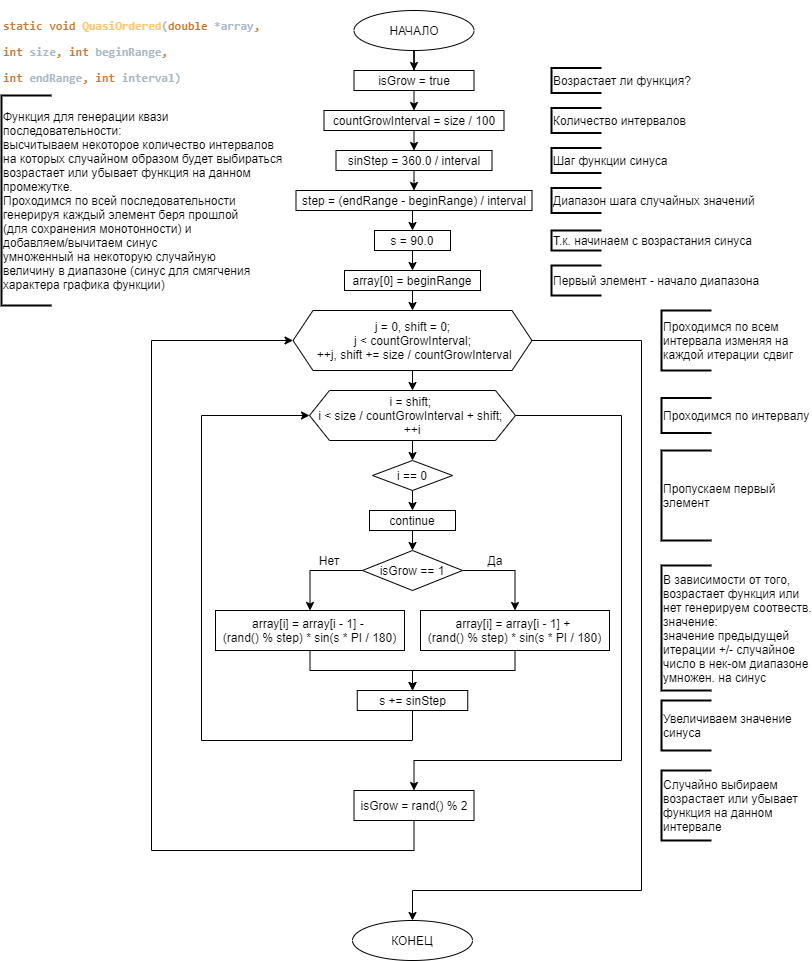
Логика генерация такова:

1. Вначале высчитываем количество интервалов, на которых случайным образом будет определяться будет возрастать или убывать функция.
2. Проходимся по всем интервалам, внешний цикл отвечает за правильный сдвиг итератора во внутреннем цикле, то есть с какого по какой элемент будет работать очередной цикл.
3. В каждом интервале случайным образом определяется будет возрастать функция или убывать
4. После мы генерируем значения в последовательности в зависимости от ее характера (убывает/возрастает):

4.1). Если функция возрастает на данном интервале, то берем предыдущее значение и добавляем к нему случайное, умножая его на синус, для того чтобы график функции был более” скругленным”.

4.2.) Если функция убывает, то действуем по такой же логике, как и в 4.1.), но теперь мы вычитаем значение случайного значение из предыдущего.

**Структурная схема алгоритма**



**Код алгоритма**

**static void QuasiOrdered(double \*array, int size, int beginRange, int endRange, int interval) {  
 bool isGrow = true;  
 int countGrowInterval = size / 100;  
 double sinStep = 360.0 / interval;  
 int step = (endRange - beginRange) / interval;  
 double s = 90.0;  
 array[0] = beginRange;  
 for (int j = 0, shift = 0; j <= countGrowInterval; ++j, shift += size / countGrowInterval) {  
 for (int i = shift; i < size / countGrowInterval + shift; ++i) {  
 if (i == 0) continue;  
 if (isGrow == 1) {  
 array[i] = array[i - 1] + (rand() % step) \* sin(s \* PI / 180);  
 } else {  
 array[i] = array[i - 1] - (rand() % step) \* sin(s \* PI / 180);  
 }  
 s += sinStep;  
 }  
 isGrow = rand() % 2;  
 }  
}**

**График времени формирования последовательности**

**График последовательности**

**Вывод:**

Мы разработали программу, которая генерирует различные типы последовательностей, построили графики данных последовательностей и замерили время формирования этих последовательностей.