Министерство Образования Российской Федерации
Российский Государственный Гуманитарный Университет
Институт Информационных Наук и Информационной Безопасности
Факультет Защиты Информации

Кафедра математической и программной
защиты информации

КИЖВАТОВ ИЛЬЯ СЕРГЕЕВИЧ

ПРЕДСКАЗАНИЕ СЛЕДУЮЩЕГО БИТА ПСЕВДОСЛУЧАЙНОЙ ПОСЛЕДОВАТЕЛЬНОСТИ ПРИ ПОМОЩИ АЛГОРИТМА МАШИННОГО ОБУЧЕНИЯ C4.5

Курсовая работа
студента 3 курса дневного отделения

Студент _____

Научный руководитель
д. т. н., профессор В. С. Анашин

Отметка

Москва 2002

Оглавление

Введение

В данной работе исследуется эффективность предсказания следующего бита псевдослучайных последовательностей при помощи обучаемого машинного алгоритма C4.5. Исследуемые последовательности вырабатывались усечёнными конгруэнтными генераторами первой, второй и третьей степени

Проблема предсказания следующего бита заключается в следующем [2]: мы обладаем некоторой битовой последовательностью $b_0, b_1, \ldots b_n$, которая выработана каким-либо детерминированным алгоритмом – псевдослучайным битовым генератором (pseudorandom bit generator, PRBG). Параметры генератора неизвестны. Алгоритм, который может вычислить по известным битам следующий бит последовательности b_{n+1} с вероятностью, большей ½ (насколько большей, обсуждается в дальнейшем), называется предсказателем следующего бита (next bit predictor, NBP). NBP, который может обнаружить любой предсказуемый PRBG без априорных предположений о свойствах этого генератора, определён в [1] как общий предсказатель следующего бита (general next bit predictor, GNBP).

На роль GNBP в [2] выдвигается алгоритм C4.5, являющийся стандартом de facto среди алгоритмов машинного обучения. В качестве обоснования приводятся замечательные результаты предсказания линейного конгруэнтного генератора по простому модулю, усечённого до младшего бита, и регистра сдвига.

Независимая проверка эффективности алгоритма C4.5 для предсказания экстремально усечённых (до одного бита) конгруэнтных генераторов является главной целью данного исследования. Другая цель – оценка предсказуемости самих генераторов.

При построении генераторов автор пользовался материалом из работ Кнута [5] и Эммериха (Emmerich) [1], а также руководством [3] (программная реализация). Алгоритм C4.5 взят с сайта его разработчика Дж. Куинлана (J. R. Quinlan) [4] и применялся, как предложено в [2].

1. Алгоритм C4.5

Данный алгоритм предназначен в общем случае для решения проблемы классификации и использует принципы машинного обучения (machine learning, ML). Главной стадией работы любого алгоритма ML является тренировочная стадия, или обучение (training) на учебном наборе данных. Учебные данные представляют собой множество наборов атрибутов (образцов, instances) с соответствующим каждому набору-образцу классом (категорией). Задачей алгоритма на стадии обучения является выработка правил для корректной классификации образцов. На рабочей стадии выработанные правила используются для классификации неизвестных образцов.

В [2] среди других алгоритмов ML предпочтение было отдано C4.5 по следующим причинам:

- свободно распространяется в виде исходного кода,
- вырабатывает как деревья решений (decision trees), так и наборы правил для принятия решений в форме if… then (production rules). Обе формы более читабельны по сравнению с другими подходами (например, нейросетями),
- эффективно реализован для работы с большими наборами данных,
- адекватно справляется с шумом (noise) в наборе данных (образцами, выходящими за рамки характерных для всего набора зависимостей)
- хорошо известен в области ML и является стандартом de facto.

C4.5 является реализацией алгоритма ID3 (Information Dichotomizer 3) и сохраняет его основную идею. ID3 работает на стадии обучения следующим образом [2].

Входные данные представлены множеством образцов, сопоставленных с классами. Каждый образец – это набор пар атрибут-значение и пара класс-значение. В результате обработки входных данных алгоритм строит дерево решений, которое по значениям атрибутов может корректно классифицировать образец.

Дерево решений – это древовидный граф, каждый узел (node) которого представляет собой атрибут, каждая дуга (arc) – возможное значение этого атрибута, а каждый лист (leaf, тупиковый узел) – ожидаемое значение категории для образца, который описывается путём от корня дерева к этому листу.

Основная идя ID3 при построении дерева заключается в том, что в каждом узле среди ещё не использованных атрибутов определяется тот, который наиболее информативен для классификации образцов, соответствующих пути от корня дерева к до этого узла. Наиболее информативный атрибут сопоставляется с этим узлом, и далее такой подход рекурсивно повторяется в каждом узле. Если в какой-то момент все атрибуты оказываются использованными, то создаётся лист, которому присваивается значение наиболее часто встречающейся среди оставшихся образцов категории.

Информативность атрибута оценивается в ID3 функцией прироста (gain function), которая использует энтропию по Шеннону. Эта функция, по сути дела, вычисляет разницу в информации, необходимой для отнесения элемента к классу **C**, до и после использования атрибута $\mathbf{a_i}$ в качестве узла или, что тоже самое, разницу в энтропии набора образцов до и после разделения этого набора в соответствии со значением атрибута $\mathbf{a_i}$. Чем больше значение функции прироста, тем большую роль играет атрибут для классификации.

Псевдокод ID3 выглядит так:

```
Функция ID3(R, C, S)    // R-набор атрибутов, C-категория, S-набор данных
{
 если S пусто,
      то возвратить единственный узел со значением «Ошибка»

если S содержит записи с одинаковой категорией,
      то возвратить единственный узел со значением этой категории
```

если R пусто,
      то возвратить единственный узел со значением наиболее часто встречающейся категории в S

пусть D – атрибут с максимальным значением прироста Gain(D, S) среди атрибутов в R

пусть $\{d_j \mid j = 1,2,\ldots m\}$ – значения атрибута D

пусть $\{S_j \mid j = 1,2,\ldots m\}$ – подмножества S, состоящие соответственно из образцов со значениями d_j атрибута D

возвратить дерево с корнем D и дугами d_1, d_2,… d_m, ведущими соответственно в деревья

      ID3(R-{D}, C, S_1), ID3(R-{D}, C, S_2),… ID3(R-{D}, C, S_m)   // рекурсия
}

Главными усовершенствованиями C4.5 по сравнению с ID3 являются усечение (pruning) результирующего дерева (устранение избыточности, иногда положительно сказывающееся на классификации) и выработка правил классификации в форме if… then по созданному дереву решений.

В нашем случае имеется всего две категории 0 и 1, соответствующие значению предсказываемого бита. Каждый атрибут также принимает значения {0, 1}, соответствующие предыдущим известным битам. Вопрос заключается в том, как подготовить битовую последовательность для того, чтобы можно было обучить C4.5 предсказывать следующий бит. Идея здесь такова: в качестве образцов берутся участки исходной последовательности определённой длины **fl+1**, называемой длиной окна (framelength). Первые **fl** битов образца являются значениями атрибутов, последний бит, который и требуется предсказать, определяет класс, к которому относится образец. Из последовательности $\mathbf{b_0}$, $\mathbf{b_1}$, … $\mathbf{b_n}$ получится **n-fl** образцов:

$$\mathbf{s_0: b_0, b_1, \ldots b_{fl-1}, b_{fl}}$$
$$\mathbf{s_1: b_1, b_2, \ldots b_{fl}, b_{fl+1}}$$
$$\ldots$$
$$\mathbf{s_{n-fl}: b_{n-fl}, b_{n-fl+1}, \ldots b_{n-1}, b_n}$$

Длина окна является ключевым параметром, влияющим на точность предсказания. Алгоритм будет пытаться найти зависимости на отрезках длины fl между предыдущими битами (атрибутами) и следующим битом (классом). Проверка того, насколько точно научился алгоритм предсказывать следующий бит, проводится на другом участке последовательности – тестовых данных, которые формируются так же, как и учебные. Соотношение правильных предсказаний и общего размера массива тестовых данных является точностью предсказания (accuracy).

Алгоритм C4.5 запускался автором на ПК под управлением ОС Linux. Реализация алгоритма требует на входе наличия трёх файлов .names, .data и .test, содержащих соответственно описание атрибутов и классов, учебные образцы и тестовые данные. На выходе регистрировался процент ошибок предсказания. Исходная последовательность генерировалась в ASCII–файл, из разных частей которого при помощи вспомогательной «нарезающей» программы формировались файлы .data и .test .

2. Исследуемые генераторы

В ходе работы изучению были подвергнуты два линейных, два квадратичных и два кубических конгруэнтных генератора по модулю 2^{32} (кроме одного линейного по модулю $2^{31}-1$), усечённые до одного требуемого бита.

Первый линейный генератор имел вид

$$\mathbf{lcg1(x) = 1664525*x + 1013904223 \ (mod \ 2^{32})}$$

Он был заимствован в [3] и удовлетворяет требования максимального периода [5].

Второй линейный генератор

$$\mathbf{lcg2(x) = 16807*x \ (mod \ 2^{31}-1)}$$

также заимствован в [3] (генератор «минимального стандарта» Парка и Миллера, Park and Miller "Minimal Standard" generator). Он представляет особый интерес, так как в работе [2] использовался похожий мультипликативный генератор

$$\mathbf{f(x) = 1078318381*x \ (mod \ 2^{31}-1)}$$

и было заявлено, что такой генератор обладает хорошими спектральными характеристиками и принят за образец очень хорошего линейного генератора с периодом $\mathbf{2^{31}-2}$. Его младший бит оказался замечательно предсказуемым, чего нельзя сказать о генераторе **lcg2**.
При программной реализации **lcg2** для корректного умножения двух 32-битных чисел по модулю 32-битного числа без переполнения 32-битной переменной использовался алгоритм Шраге (Schrage) [3]. Он основан на приближённой факторизации модуля

$$\mathbf{m = aq + r, \ q = [m/a], \ r = m \bmod a}.$$

Если $\mathbf{r < q}$ и $\mathbf{0 < x < m-1}$, то $\mathbf{a(x \bmod q)}$ и $\mathbf{r[z/q]}$ лежат в пределах $\mathbf{0, \ 1, \ \ldots \ m-1}$. Тогда

$$\mathbf{a*x \bmod m} = \begin{cases} \mathbf{a(x \bmod q) - r[z/q]}, \text{ если это значение больше 0} \\ \\ \mathbf{a(x \bmod q) - r[z/q] + m} \text{ в другом случае} \end{cases}$$

В **lcg2** использовались константы $\mathbf{q = 127773}$ и $\mathbf{r = 2836}$.

Квадратичный генератор

$$\mathbf{qcg1(x) = 56*x^2 + 9*x + 9 \ (mod \ 2^{32})}$$

построен на основании утверждения о том, что функция

$\mathbf{g(x) = 1 + x + 8*h(x)}$, где $\mathbf{h(x)}$ – любая совместимая функция,

транзитивна по модуля $\mathbf{2^k}$ при любом целом неотрицательном k. В данном случае в качестве $\mathbf{h(x)}$ был произвольно выбран полином $\mathbf{7*x^2 + x + 1}$. Дополнительно проведена проверка, что **qcg1** удовлетворяет требованиям максимального периода [1, 5], а именно

$$f(x) = ax^2 + bx + c \pmod{m=p_1^{w1}p_2^{w2}\ldots p_r^{wr}},$$
$$a, b, c \quad Z_m$$

транзитивна, если

$a \equiv 0 \pmod{p_i}$ ¿
$b \equiv 1 \pmod{p_i}$ для $1 \quad i \quad r$
$c \equiv 0 \pmod{p_i}$
$a \equiv b-1 \pmod 4$, если $4|m$
$ac \equiv 3 \pmod 9$, если $9|m$

Второй квадратичный генератор

$$qcg2(x) = 56*x^2 + 9*x + 9 \pmod{2^{32}}$$

построен при помощи подбора параметров, удовлетворяющих требованиям максимального периода.

Кубические генераторы

$$ccg2(x) = 40*x^3 + 8*x^2 + 25*x + 89 \pmod{2^{32}}$$

$$ccg2(x) = 8*x^3 + 9*x + 9 \pmod{2^{32}}$$

получены на основании упомянутого способа построения транзитивной функции.

Каждый генератор оформлен в виде отдельной программы. Листинги программ приведены в приложении. При вызове указывается начальное значение переменой (seed), длина генерируемой последовательности и номер бита, с которого производится съём. Последовательность выводится в стандартный поток вывода в виде нулей и единиц в коде ASCII.

3. Результаты предсказания

Для каждого генератора проверялись последовательности, снятые с 16 бита, кроме lcg2, у которого брался младший разряд. На каждой последовательности обучение проводилось с разными длинами окон и иногда с разными длинами учебного набора данных, то есть части периода, на которой алгоритм обучался. Ниже в таблицах 1 – 6 приведены точности предсказания на тестовых наборах, выраженные в процентах. В первом столбце каждой таблицы указаны длина участка последовательности, на котором проводилось обучение (в битах и в частях периода) и длина участка для тестирования.

Таблица 1. Точность предсказания **`lcg1`**

Framelength	100	200	400	2000
32K training / 32K test (1/2 period)	**53,9**	**57,7**	**65,8**	
4K / 8K (1/16)				**57,4**

Таблица 2. Точность предсказания **`lcg2`** (младший бит)

Framelength	10	20	40	80	160	320	640	1000	1500
32K / 32K	50,0	50,0	50,0	50,1	50,1	49,9	49,7	49,8	50,1

Таблица 3. Точность предсказания **`qcg1`**

Framelength	100	200	400	2000	3000
32K / 32K (1/2)	**64,9**	**79,1**	**27,9**		
4K/10K (1/16)				**90,0**	**82,6**

Таблица 4. Точность предсказания **`qcg2`**

Framelength	100	200	400	1000	2000	3000	4000
32K / 32K (1/2)	50,1	50,1	49,7	**52,3**			
4K/10K (1/16)					50,1	49,5	
10K/10K (~1/6)							50,6

Таблица 5. Точность предсказания **`ccg1`**

Framelength	1000	1500
32K / 32K (1/2)	**51,8**	**53,5**
5K/32K (~1/13)		49,9

Таблица 6. Точность предсказания **`ccg2`**

Framelength	100	200	400	800	1600
32K / 32K (1/2)	**51,5**	**51,3**	**58,1**	**55,7**	**53,2**
17 bit 32K/32K (1/4)	50,1	50,2	**50,5**	**50,6**	**50,8**

4. Критерий оценки результатов

При анализе результатов необходимо определить, при какой точности предсказания мы считаем, что C4.5 в роли GNBP предсказывает действительно лучше, чем брошенная монета. В качестве критерия оценки авторы [2] предлагают статистику

$$S = \overline{x} \updownarrow 0{,}5 \,?\, \frac{0{,}5}{\sqrt{N}}$$

где $\overline{x}$ - число ошибок предсказания по отношению ко всему числу предсказаний N.

В соответствии с центральной предельной теоремой, с ростом N **S ~ N(0;1)**. Выдвигается исходная гипотеза (null-hypothesis): GNBP не предсказывает следующий бит лучше, чем монета, и альтернативная гипотеза: GNBP всё же предсказывает лучше монеты. Тогда при уровне значимости α альтернативная гипотеза принимается в случае, если $S > F(\alpha)$. При α=0,05 F=1,96 (двусторонняя проверка, two-side test). В таблице 7 приведены критические значения точности для уровня значимости 0,05 в зависимости от размера тестового набора N.

Таблица 7. Критические значения точности предсказания A.

N, бит	50K	32K	10K	8K
A,%	50,44	50,5	50,98	51,09

В таблицах 1 – 6 значения, для которых альтернативная гипотеза подтвердилась при уровне значимости 0,05, выделены жирным шрифтом и подчёркиванием.

Итак, оказалось, что лидером по непредсказуемости оказался линейный конгруэнтный генератор. При любой длине окна, которую позволили

установить вычислительные ресурсы, C4.5 не смог предсказать следующий бит. Для сравнения приведём результаты предсказания генератора из [2]:

Таблица 8. Точность предсказания генератора из работы [2] (младший бит)

Framelength	10	20	40	80	160	320	640
50K/50K	78,4	99,9	100,0	100,0	100,0	100,0	100,0

В данной работе длины окон меньше 100 не давали точности более 65% даже при обучении на полупериоде, что свидетельствует о том, что исследуемые здесь генераторы намного сильнее в отношении предсказания.

Худшие результаты по предсказуемости показал **qcg1**. Кубический генератор **ccg2**
прошёл тест при обучении на полупериоде уже при длине окна 100.
Кубический генератор **ccg1** при обучении на полупериоде прошёл тест при длине окна 1000, и похожие результаты продемонстрировал квадратичный **qcg2**.
Линейный генератор **lcg1**оказался предсказуемым с точностью 57,4% при обучении 1/16 части периода при размере окна 2000 и занимает второе место по предсказуемости после **qcg1**.

На примере результатов предсказания **ccg2** обнаружилась интересная зависимость: с увеличением окна точность предсказания сначала возрастает, а затем снова уменьшается, достигая максимума 58,1% при длине окна 400. Для проверки закономерности была получена последовательность с 17 разряда генератора, а обучение велось на ¼ периода. В этом случае наблюдается только прирост точности предсказания (таблица 6). Скорее всего, такие эффекты связаны с перетренировкой (overtraining) алгоритма предсказания, которая чаще всего наступает при избыточности учебных данных. В этом

случае алгоритм вырабатывает слишком жёсткие критерии классификации, что приводит к увеличению числа ошибок на стадии классификации с незнакомых образцов.

Заключение

В ходе исследования показано, что C4.5 действительно обнаруживает слабости PRGB. Это значит, что если какой-либо PRBG предсказывается C4.5, то он имеет изъяны. Остаётся нерешённой обратная проблема: если наш алгоритм оказывается не в состоянии предсказать какую-либо последовательность, является ли генератор сильным, то есть может ли C4.5 обнаруживать все слабости произвольного PRBG?

Два применения C4.5 в качестве GNBP, которые действительно обоснованы существующими результатами, указаны в [2]. Это, во-первых, быстрое обнаружение плохих (flawed) генераторов, во-вторых, классификация PRBG по их стойкости к предсказанию. Необходимо, правда, определить критерии стойкости. В [2] в их качестве предлагаются наряду с длиной окна размер дерева решений и количество обучающих данных, требуемое для предсказания с достаточной точностью.

При использовании в качестве критериев длины окна и количества учебных образцов, необходимых для подтверждения гипотезы о наличии предсказания, генераторы, рассмотренные в данной работе, по непредсказуемости классифицируются так:

`lcg2` (самый стойкий)
`ccg1`
`qcg2`
`ccg2`
`lcg1`
`qcg1` (самый предсказуемый)

Список литературы

Emmerich F. Equidistributional properties of quadratic congruential pseudorandom numbers. - Journal of Computable and Applied Mathematics. - N 79 (1997).

Hernandez, J. C., Sierra, J. M., Mex-Perera et al. Using the general next bit predictor like an evaluation criteria.

Numerical Recipes in C: The Art of Scientific Computing. – Chapter 7. Random Numbers. - (from www.nr.com)

Кнут Д. Искусство програмирования . – Том 2. Получисленные алгоритмы. – Москва: Вильямс. – 2000г.

Приложение

lcg1.cpp

```
/*
      Truncated "quick'n'dirty" linear congruential generator modulo 2^32,
      multiplier 1664525 and sumandus 1013904223.
      Output contains the specified bit of the generated number
*/

#include <iostream.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *errv[])
{
 unsigned long i;   // the counter
 unsigned long length;    // length of sequence to generate
 unsigned long x;   // the pseudorandom number
 unsigned long truncx;    // truncated ps-random number
 unsigned int bitn;       // bit for output

 if (argc < 4)
      {
       cout << endl << "  Too few parameters. Usage: ";
       cout << "  lcg seed length bitn" << endl;
       cout << "          seed - initial number" << endl;
       cout << "          length - sequence length" << endl;
       cout << "          bitn - number of bit for output (0 - 31)" << endl <<
endl;
       return 1;
      }
 bitn = atoi(argv[3]);
 if (bitn > 31)
      {
       cout << endl << "  Bit number value too large. Must be 0 - 31" << endl <<
endl;
       return 1;
      }

 length = atol(argv[2]);
 x = atol(argv[1]);

 truncx = (x << (31 - bitn)) >> 31;
 cout << truncx;
 for (i = 2; i <= length; i++)
      {
       x = 1664525L * x + 1013904223L;
       truncx = (x << (31 - bitn)) >> 31;
       cout << truncx;
      }
 cout << endl;
 return 0;
}
```

lcg2.cpp

```
/*
   Truncated multiplicative congruential generator modulo 2147483647 = 2^31 - 1
   with multiplier 16807 = 7^5 (Park & Miller "Minimal Standard" generator).
   Using Shrage method for modular multiplication with constants q = 127773, r =
2836.
   Output contains the last bit of the generated number
*/

#include <iostream.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *errv[])
{
 long i;      // the counter
 long length;    // length of sequence to generate
 long x;      // the pseudorandom number
 long truncx;       // truncated ps-random number
 int bitn;    // bit for output
 long k;      // extra integer for modular multiplication

 if (argc < 4)
      {
       cout << endl << "  Too few parameters. Usage: ";
       cout << "  tqcg seed length bitn" << endl;
       cout << "          seed - initial number" << endl;
       cout << "          length - sequence length" << endl;
       cout << "          bitn - number of bit for output (0 - 31)" << endl <<
endl;
       return 1;
      }
 bitn = atoi(argv[3]);
 if (bitn > 31)
      {
       cout << endl << "  Bit number value too large. Must be 0 - 31" << endl <<
endl;
       return 1;
      }

 length = atol(argv[2]);
 x = atol(argv[1]);
 if (x == 0)            // human-proof check
      {
       cout << endl << "Seed 0 is prohibited: generator is multiplicative" <<
endl << endl;
       return 1;
      }

 truncx = x & 0x0001;
 cout << truncx;
 for (i = 2; i <= length; i++)
      {
       k = x / 127773;              // here is Shrage method
       x = 16807*(x - k*127773) - 2836*k;
       if (x < 0) x += 2147483647;
       truncx = x & 0x0001;
       cout << truncx;
      }
 cout << endl;
 return 0;
}
```

qcg1.cpp

```
/*
      Truncated quadratic congruential generator modulo 2^32,
      Output contains the specified bit of the generated number
*/

#include <iostream.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *errv[])
{
 unsigned long i;   // the counter
 unsigned long length;    // length of sequence to generate
 unsigned long x;   // the pseudorandom number
 unsigned long truncx;    // truncated ps-random number
 unsigned int bitn;       // bit for output

 if (argc < 4)
      {
       cout << endl << "  Too few parameters. Usage: ";
       cout << "  tqcg seed length bitn" << endl;
       cout << "          seed - initial number" << endl;
       cout << "          length - sequence length" << endl;
       cout << "          bitn - number of bit for output (0 - 31)" << endl <<
endl;
       return 1;
      }
 bitn = atoi(argv[3]);
 if (bitn > 31)
      {
       cout << endl << "  Bit number value too large. Must be 0 - 31" << endl <<
endl;
       return 1;
      }

 length = atol(argv[2]);
 x = atol(argv[1]);

 truncx = (x << (31 - bitn)) >> 31;
 cout << truncx;
 for (i = 2; i <= length; i++)
      {
       x = 56*x*x + 9*x + 9;
       truncx = (x << (31 - bitn)) >> 31;
       cout << truncx;
      }
 cout << endl;
 return 0;
}
```

qcg2.cpp

```
/*
      Truncated quadratic congruential generator modulo 2^32,
      Output contains the specified bit of the generated number
*/

#include <iostream.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *errv[])
{
 unsigned long i;   // the counter
 unsigned long length;    // length of sequence to generate
 unsigned long x;   // the pseudorandom number
 unsigned long truncx;    // truncated ps-random number
 unsigned int bitn;       // bit for output

 if (argc < 4)
      {
       cout << endl << "  Too few parameters. Usage: ";
       cout << "  tqcg seed length bitn" << endl;
       cout << "          seed - initial number" << endl;
       cout << "          length - sequence length" << endl;
       cout << "          bitn - number of bit for output (0 - 31)" << endl <<
endl;
       return 1;
      }
 bitn = atoi(argv[3]);
 if (bitn > 31)
      {
       cout << endl << "  Bit number value too large. Must be 0 - 31" << endl <<
endl;
       return 1;
      }

 length = atol(argv[2]);
 x = atol(argv[1]);

 truncx = (x << (31 - bitn)) >> 31;
 cout << truncx;
 for (i = 2; i <= length; i++)
      {
       x = 568*x*x + 13*x + 149;
       truncx = (x << (31 - bitn)) >> 31;
       cout << truncx;
      }
 cout << endl;
 return 0;
}
```

ccg1.cpp

```
/*
      Truncated cubic congruential generator modulo 2^32,
      Output contains the specified bit of the generated number
*/

#include <iostream.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *errv[])
{
 unsigned long i;   // the counter
 unsigned long length;    // length of sequence to generate
 unsigned long x;   // the pseudorandom number
 unsigned long truncx;    // truncated ps-random number
 unsigned int bitn;       // bit for output

 if (argc < 4)
      {
       cout << endl << "  Too few parameters. Usage: ";
       cout << "  tccg seed length bitn" << endl;
       cout << "          seed - initial number" << endl;
       cout << "          length - sequence length" << endl;
       cout << "          bitn - number of bit for output (0 - 31)" << endl <<
endl;
       return 1;
      }
 bitn = atoi(argv[3]);
 if (bitn > 31)
      {
       cout << endl << "  Bit number value too large. Must be 0 - 31" << endl <<
endl;
       return 1;
      }

 length = atol(argv[2]);
 x = atol(argv[1]);

 truncx = (x << (31 - bitn)) >> 31;
 cout << truncx;
 for (i = 2; i <= length; i++)
      {
       x = 40*x*x*x + 8*x*x + 25*x + 89;
       truncx = (x << (31 - bitn)) >> 31;
       cout << truncx;
      }
 cout << endl;
 return 0;
}
```

ccg2.cpp

```
/*
      Truncated cubic congruential generator modulo 2^32,
      Output contains the specified bit of the generated number
*/

#include <iostream.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *errv[])
{
 unsigned long i;   // the counter
 unsigned long length;    // length of sequence to generate
 unsigned long x;   // the pseudorandom number
 unsigned long truncx;    // truncated ps-random number
 unsigned int bitn;       // bit for output

 if (argc < 4)
      {
       cout << endl << "  Too few parameters. Usage: ";
       cout << "  tccg seed length bitn" << endl;
       cout << "           seed - initial number" << endl;
       cout << "           length - sequence length" << endl;
       cout << "           bitn - number of bit for output (0 - 31)" << endl <<
endl;
       return 1;
      }
 bitn = atoi(argv[3]);
 if (bitn > 31)
      {
       cout << endl << "  Bit number value too large. Must be 0 - 31" << endl <<
endl;
       return 1;
      }

 length = atol(argv[2]);
 x = atol(argv[1]);

 truncx = (x << (31 - bitn)) >> 31;
 cout << truncx;
 for (i = 2; i <= length; i++)
      {
       x = 8*x*x*x + 9*x + 9;
       truncx = (x << (31 - bitn)) >> 31;
       cout << truncx;
      }
 cout << endl;
 return 0;
}
```