Random Number Generators

pocket computer science vol. 001

printed by printer_scanner

Random Number Generators

pocket computer science vol. 001
printed by printer_scanner

0.33772900904173040.415407740605423960.
71986156906773790.69343559610643650.405
767253681004860.70422880452792880.4903
47943354341840.81401396993256950.183864
296860780430.45178920629139150.19992952
7748199830.377259030972352470.8179911984
4332570.30730736225128250.7728348197790
760.30323502739289610.36364446088514570
.441991136208107040.44743387341902840.52
167564916063270.68715206949632530.783016
07169501160.287467739569825740.307162387
341870740.84001280134927250.972311184392
39350.88323328549804090.721503333316927
0.70713185503217680.158321159523183930.178
379140872751090.19128241662495670.248928
205525657420.125347874754736520.74135859
42243550.35935470747338850.429694715987
44360.75008117686156740.8265258658544581
0.72798748771135790.78465397706429570.04
9652873633502240.167940846919580.519624
786127830.302899227686001640.4422131449
5446240.354799914366766830.29039363643
07540.217858571221839710.44864834066544
510.50152263300857160.55334883884900470
.47670584533431780.0080569934532248590.
062392171417200440.58717544177816270.699
74245622109850.76737283217137660.6543090
1237443820.63420531231024450.2097854839
7953390.88354700332217060.2899046678575
7210.22511715518704190.092577504062142160
.81434655115932550.222691416172928980.238
557914463137120.43960004514121010.350164
542090664940.57516488157096420.2002998
56189074530.98576681136135870.4714575669
14932350.027748288558589130.58538519332
069260.201772526558406270.4168145418090
61860.73646079264946350.86250758680726
450.36722949067016720.26175852211756003

0.091454729978884820.21595338391381570.5
6968911828538290.64614451116077440.02922
9903486519770.210776518984888030.814209
32849678530.88392949467413520.139708884
982374440.52160707302874990.39016649390
8254530.0081565252021524870.44366706180
7691670.83916397640481550.5807887557486
4660.35224415004162780.8511386026627943
0.98689538213870650.205198773079121160.6
297070277339150.0278646436381331240.788
19375476651120.339244436585300460.318013
88341180760.219753414363618840.626397465
09962110.86188891751018030.4078104591920
88140.87996033836641940.1863113225282218
0.33175725960132940.0501586755474161540.2
51121413167168850.59933588930375350.68149
918427743580.400954958104460470.8818481
159740430.93788402876338320.01847685379
29489440.0210434132035248740.0361739437
126771260.97788059267585960.46443492450
930.30112168269509090.03263404390931868
0.74156677378693340.21310526272951670.587
21090715415450.194085665721419160.5192247
7925714830.38626990823601860.1243140668
84885220.49802076924227910.212978045530
313850.20879870491626650.130378143203599
530.231911681267269460.3546043936979794
70.85855635736450210.41046594460391760.1
71048950912498830.68365406145724950.348
80314651176360.202595802762408850.82885
834511809290.36293527251926780.73880877
62438580.92700017081429410.268591055965
27620.77059226333287410.4387129479015916
50.24148387129262040.55463907140514610.8
0183011172963140.37424291645252450.06298
8699724091380.58543859151734150.89184789
437999720.138181009709151370.53056302706
938680.395196885639760960.6076310246510
3150.58157544316137550.409880567113741770
.0453723629802162960.39378762605905540.
73276338948306810.96653022254722340.221
807772382172970.055221170893651640.81027
650531315420.99406543337645270.23309588
9316936370.32748169731811760.97300500489
48740.370125407359983560.164117402319960
750.95498222146600820.93712807402118850
.96027707332736090.492170620618694430.2
0901760349848320.50970158447876250.6979
8398266481750.0175339228769337830.65823
747139089890.90073076375351890.513436413
08764630.43047655179793610.1050497254615

"Anyone who considers arithmetic methods of producing random digits is of course in a state of sin."

-John von Neumann

Random numbers are an essential part of modern computing. We rely on random numbers for our video games, banking, and art. Our illusion of our privacy online is held up by random numbers. But it's hard to approximate random numbers with computers.

When we work with random numbers on computers, we are constantly working with approximations of randomness. There are two main ways of generating approximately random numbers on computers: pseudorandom number generators or PRNG's and Hardware random number generators or HRNG's.

John von Neumann developed the first PRNG. His algorithm is called the middle square method. It takes a number and squares it, creating our "random number." The algorithm repeats itself several times to ensure the random number is sufficiently random.

This algorithm is a good representation of those that followed, but with one major problem. All sequences with this algorithm will eventually repeat themselves, some very quickly.

The sequence "0000" will always return "0000". We call this the period length. A longer period length makes an algorithm a more desirable approximation of randomness.

Today there are many kinds of random number algorithms. We will break down the two random number generation algorithms most frequently used on our computers.

xorshift128+

xorshift128+ is the algorithm used by most popular web browsers. When you run a random number generator in the browser, like Math.random() in JavaScript.

Math.random() is a function that when called, returns a floating point number between zero and less than one. Here is an example of what Math.random() outputs:

0.2727425239831447
0.0463378865050111

These languages elect to have the browser choose the algorithm. We call this implementation-defined. xorshift128+ is the preferred algorithm of the browser because it is incredibly lightweight and the fastest algorithm within their class. Let's look at xorshift128+ in detail:

xorshift128+

```
uint64_t state0 = 1;
uint64_t state1 = 2;

uint64_t xorshift128plus() {
    uint64_t s1 = state0;
    uint64_t s0 = state1;
    state0 = s0;
    s1 ^= s1 << 23;
    s1 ^= s1 >> 17;
    s1 ^= s0;
    s1 ^= s0 >> 26;
    state1 = s1;
    return state0 + state1;
}
```

The algorithm begins by defining two seed values. The essential function of the algorithm is to take two seed numbers and manipulate them enough times through arithmetic to become unrecognizable. We do this using bitwise operators. We represent bitwise operators in the code with the symbols `>>` and `<<`.

Programming languages are written on top of each other to reduce complexity, something we call the layers of abstraction. The programmer does not have to understand everything about how a computer works to create a program. Bitwise operators behave a bit differently and reach underneath the program to manipulate the code at the level of the bit, or the 1's and 0's.

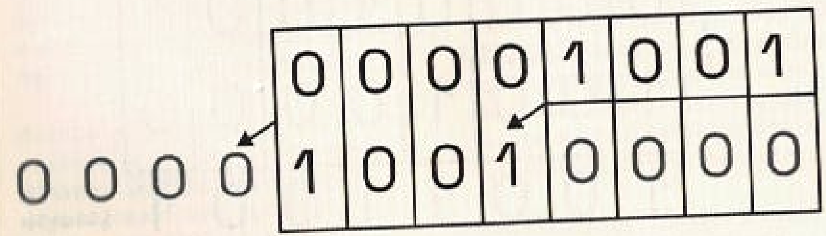
The operators `<<` and `>>` designate left-shifts and right-shifts. For example, if we wrote `8 << 3`, the left shift operator shifts the first operand – 8 – a designated number of bits to the left, in this case, 3. Excess bits shifted off to the left are discarded, and zero bits shifted from the right. You can see this in action in Figure 1. This particular algorithm starts by left-shifting our seed value by 23 bits.

This is a very clever algorithmic approach. By performing shifts at the bit level, we can ensure that, in appearance, the new number does not arithmetically relate to the one preceding it. Then, we apply our second operator, the exclusive or operator, or XOR. We write this programmatically as `=^`. XOR means one or the other, but not both. It compares the binary representations of two numbers and outputs a 0 where the corresponding bits are the same and a one where they are different. In this case, we compare the binary representations of our `seed 1` value and our left-shifted `seed 1`. The algorithm then goes through several more cycles of left and right-shifting and finally outputs our "random" number.

Two questions may arise from looking at this algorithm. First, why did they shift the bits by the numbers 23, 17, and 26? Researchers carefully selected these numbers because they create the largest period in this random number generator. A period is the number of times a program can run before it starts to repeat itself. xorshift128+ has a period of length of 2^128 - 1 (hence the name).

The second question you may be asking is how a seed gets selected. Now you're thinking like a cryptographer. xorshift128+ is not cryptographically secure.

`Bit Shifting 8<< 3`



XOR Shifting Bits

0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	0	1	1	0	0	1

(For any Math.random() users reading this, use window.crypto.getRandomValues for cryptographically secure numbers).

Seeds often act as a backdoor preventing something from being truly random. It doesn't matter how many times you shake up a number, anyone can reverse-engineer the value as long as a seed is deterministically selected. It would be like a hurricane proofing a house and then leaving the windows open. Especially because these algorithms are already available for dissecting on the internet, and if you know a generator's internal state, you see the future.

Brendan Eich developed JavaScript in 10 days in the early nineties. Math.random() isn't cryptographically secure because until 1992, due to The Cold War, there was a ban on the export of cryptography in the United States. That law gradually became relaxed, but as Eich developed JavaScript, releasing secure algorithms to the internet would be the equivalent of exporting munitions to enemies of the state.

Mersenne Twister

The second algorithm is standardly implemented by programming languages such as Python, Ruby, R, IDL, and PHP. Developed in 1997, Makoto Matsumoto and Takuji Nishimura named The Mersenne Twister after Mersenne primes, a prime number one less than the power of two, reflecting the period lengths of these two algorithms.

On the next page is an implementation by Matsumoto and Nishimura of the Mersenne Twister written in C. We won't go through it line by line like the previous one, but it follows the same basic principles of shifting and rotating. The Twister is a much larger algorithm than xorshift128+. Because of this, this algorithm is much more complex, uses more memory, and is slower than other algorithms. But, it has a few advantages.

The Mersenne Twister's main advantage is an enormous period length. 2^19937-1. This number, for reference, is larger than the number of atoms in the observable universe. While a long period is not a guarantee of quality in a random number generator, short periods, such as the 232 standards in older software, can be problematic. Attackers can observe and record sequences with too short a period can. Sequences with long periods force potential attackers to select alternate methods.

```c
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL
#define UPPER_MASK 0x80000000UL
#define LOWER_MASK 0x7fffffffUL

static unsigned long mt[N];
static int mti=N+1;
void init_genrand(unsigned long s)
{
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
            (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
            mt[mti] &= 0xffffffffUL;
        }
}

void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
    init_genrand(19650218UL);
    i=1; j=0;
    k = (N>key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
          + init_key[j] + j;
        mt[i] &= 0xffffffffUL;
        i++; j++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=N-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
          - i;
        mt[i] &= 0xffffffffUL;
        i++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
    }

    mt[0] = 0x80000000UL;
}

unsigned long genrand_int32(void)
{
    unsigned long y;
    static unsigned long mag01[2]={0x0UL, MATRIX_A};


    if (mti >= N) {
        int kk;

        if (mti == N+1)
            init_genrand(5489UL);

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }

    y = mt[mti++];

    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y;
}
```

```c
/* generates a random number on [0,1]-real-interval */
double genrand_real1(void)
{
    return genrand_int32()*(1.0/4294967295.0);
    /* divided by 2^32-1 */
}

/* generates a random number on [0,1)-real-interval */
double genrand_real2(void)
{
    return genrand_int32()*(1.0/4294967296.0);
    /* divided by 2^32 */
}

/* generates a random number on (0,1)-real-interval */
double genrand_real3(void)
{
    return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
    /* divided by 2^32 */
}

/* generates a random number on [0,1) with 53-bit resolution*/
double genrand_res53(void)
{
    unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
    return(a*67108864.0+b)*(1.0/9007199254740992.0);
}
/* These real versions are due to Isaku Wada, 2002/01/09 added */

int main(void)
{
    int i;
    unsigned long init[4]={0x123, 0x234, 0x345, 0x456}, length=4;
    init_by_array(init, length);
    printf("1000 outputs of genrand_int32()\n");
    for (i=0; i<1000; i++) {
      printf("%10lu ", genrand_int32());
      if (i%5==4) printf("\n");
    }
    printf("\n1000 outputs of genrand_real2()\n");
    for (i=0; i<1000; i++) {
      printf("%10.8f ", genrand_real2());
      if (i%5==4) printf("\n");
    }
    return 0;
}

/*
   A C-program for MT19937, with initialization improved 2002/1/26.
   Coded by Takuji Nishimura and Makoto Matsumoto.

   Before using, initialize the state by using init_genrand(seed)
   or init_by_array(init_key, key_length).

   Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
   All rights reserved.


   Any feedback is very welcome.
   http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
   email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
*/W
```

Another advantage of the Mersenne Twister is more random seed inputs. This algorithm uses real-time clock values as seeds that return to the millisecond. Real-time numbers are a popular and more secure way to generate random numbers. This is because a millisecond is impossibly small to detect with the human eye, giving the illusion of randomness.

The Mersenne Twister passes all Diehard tests: tests written to assure the quality of random number generators. This algorithm provides enough randomness for specific tasks like video games but is unsuitable for requiring high-quality randomnesses, such as cryptography applications, statistics, or numerical analysis.

There are a few reasons for this. First, the seed isn't random enough. Though taking millisecond clock values are random enough to the human eye to be considered entirely random, they are still too deterministic for reverse engineering purposes.

If we wanted to create a secure random algorithm for use such as banking, we would need a level of unpredictability with our seed values that the Twister does not provide. When producing seed values, unpredictably is impossible to achieve arithmetically.

Hardware random number generators or HRNG's create a closer approximation of randomness. They take raw input data from high entropy real-world situations such as thermal noise, white electrical noise, the decay of a radioactive particle, avalanche noise in semiconducting materials. Tracking mouse movements can also be considered random enough.

When we use many cryptographic algorithms, your computer stores random values generated through hardware input (such as mouse movements) and store the values for use as seed values.

However, no matter what we do to try to facilitate randomness, human intervention always creates deterministic conditions.

Thank you for reading, and be on the lookout for article #2 where we will talk about the Sieve of Eratosthenes.