



# C/Randomization

[FrontPage] [TitleIndex] [WordIndex]

**Note:** You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at <http://www.cs.yale.edu/homes/aspnes/#classes>.

## Contents

1. Generating random values in C
  1. The rand function from the standard library
  2. Better pseudorandom number generators
  3. Random numbers without the pseudo
  4. Issues with RAND\_MAX
2. Randomized algorithms
  1. Randomized search
  2. Quickselect and quicksort
3. Randomized data structures
  1. Randomized tree balancing
  2. Universal hash families

Randomization is a fundamental technique in algorithm design, that allows programs to run quickly when the average-case behavior of an algorithm is better than the worst-case behavior. It is also heavily used in games, both in entertainment and gambling. The latter application gives the only known example of a programmer being murdered for writing bad code 🌐 <http://www.zdnet.co.uk/news/security-management/1999/11/11/comdex-99-the-mysterious-death-of-larry-volk-2075068/>, which shows how serious good random-number generation is.

## 1. Generating random values in C

If you want random values in a C program, there are three typical ways of getting them, depending on how good (i.e. uniform, uncorrelated, and unpredictable) you want them to be.

### 1.1. The rand function from the standard library

E.g.

Toggle line numbers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int
5 main(int argc, char **argv)
6 {
7     printf("%d\n", rand());
8     return 0;
9 }
```

The `rand` function, declared in `stdlib.h`, returns a random integer in the range 0 to `RAND_MAX` (inclusive) every time you call it. On machines using the GNU C library `RAND_MAX` is equal to `INT_MAX` or  $2^{31}-1$ , but it may be as small as 32767. There are no particularly strong guarantees about the quality of random numbers that `rand` returns, but it should be good enough for casual use, and has the advantage that as part of the C standard you can assume it is present almost everywhere.

Note that `rand` is a **pseudorandom number generator**: the sequence of values it returns is predictable if you know its starting state (and is still predictable from past values in the sequence even if you don't know the starting state, if you are clever enough). It is also the case that the initial seed is fixed, so that the program above will print the same value every time you run it (this is a feature: it permits debugging randomized programs).

If you want to get different sequences, you need to **seed** the random number generator using `srand`. A typical use might be:

Toggle line numbers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int
6 main(int argc, char **argv)
7 {
8     srand(time(0));
9     printf("%d\n", rand());
10    return 0;
11 }
```

Here `time(0)` returns the number of seconds since the epoch (00:00:00 UTC, January 1, 1970, for POSIX systems, not counting leap seconds). Note that this still might give repeated values if you run it twice in the same second, and it's extremely dangerous if you

expect to distribute your code to a lot of people who want different results, since two of your users *are* likely to run it twice in the same second. See the discussion of `/dev/urandom` below for a better method.

## 1.2. Better pseudorandom number generators

There has been quite a bit of research on pseudorandom number generators over the years, and much better pseudorandom number generators than `rand` are available. The current champion for simulation work is the **Mersenne Twister**, which runs about 4 times faster than `rand` in its standard C implementation and passes a much wider battery of statistical tests. Its English-language home page is at <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>. As with `rand`, you still need to provide an initial seed value.

There are also **cryptographically secure pseudorandom number generators**, of which the most famous is [WikiPedia: Blum Blum Shub](#). These cannot be predicted based on their output if seeded with a true random value (under certain cryptographic assumptions: hardness of factoring for Blum Blum Shub). Unfortunately, cryptographic PRNGs are usually too slow for day-to-day use.

## 1.3. Random numbers without the pseudo

If you really need actual random numbers and are on a Linux or BSD-like operating system, you can use the special device files `/dev/random` and `/dev/urandom`. These can be opened for reading like ordinary files, but the values read from them are a random sequence of bytes (including null characters). A typical use might be:

Toggle line numbers

```
1 #include <stdio.h>
2
3 int
4 main(int argc, char **argv)
5 {
6     unsigned int randval;
7     FILE *f;
8
9     f = fopen("/dev/random", "r");
10    fread(&randval, sizeof(randval), 1, f);
11    fclose(f);
12
13    printf("%u\n", randval);
14
15    return 0;
16 }
```

(A similar construction can also be used to obtain a better initial seed for `srand` than `time(0)`.)

Both `/dev/random` and `/dev/urandom` derive their random bits from physically random properties of the computer, like time between keystrokes or small variations in hard disk rotation speeds. The difference between the two is that `/dev/urandom` will always give you some random-looking bits, even if it has to generate extra ones using a cryptographic pseudo-random number generator, while `/dev/random` will only give you bits that it is confident are in fact random. Since your computer only generates a small number of genuinely random bits per second, this may mean that `/dev/random` will exhaust its pool if read too often. In this case, a read on `/dev/random` will block (just like reading a terminal with no input on it) until the pool has filled up again.

Neither `/dev/random` nor `/dev/urandom` is known to be secure against a determined attacker, but they are about the best you can do without resorting to specialized hardware.

## 1.4. Issues with RAND\_MAX

The problem with `rand` is that getting a uniform value between 0 and  $2^{31}-1$  may not be what you want. It could be that `RAND_MAX` is be too small; in this case, you may have to call `rand` more than once and paste together the results. But there can be problems with `RAND_MAX` even if it is bigger than the values you want.

For example, suppose you want to simulate a die roll for your video craps machine, but you don't want to get whacked by Johnny "The Debugger" when the Nevada State Gaming Commission notices that 6-6 is coming up slightly less often than it's supposed to. A natural thing to try would be to take the output of `rand` mod 6:

Toggle line numbers

```
1 int d6(void) {
2     return rand() % 6 + 1;
3 }
```

The problem here is that there are  $2^{31}$  outputs from `rand`, and 6 doesn't divide  $2^{31}$ . So 1 and 2 are slightly more likely to come up than 3, 4, 5, or 6. This can be particularly noticeable if we want a uniform variable from a larger range, e.g.  $0..[(2/3)\cdot 2^{31}]$ .

We can avoid this with a technique called **rejection sampling**, where we reject excess parts of the output range of `rand`. For rolling a die, the trick is to reject anything in the last extra bit of the range that is left over after the largest multiple of the die size. Here's a routine that does this, returning a uniform value in the range 0 to `n-1` for any positive `n`, together with a program that demonstrates its use for rolling dice:

Toggle line numbers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <time.h>
```

```

5
6 /* return a uniform random value in the range 0..n-1 inclusive */
7 int
8 randRange(int n)
9 {
10     int limit;
11     int r;
12
13     limit = RAND_MAX - (RAND_MAX % n);
14
15     while((r = rand()) >= limit);
16
17     return r % n;
18 }
19
20 int
21 main(int argc, char **argv)
22 {
23     int i;
24
25     srand(time(0));
26
27     for(i = 0; i < 40; i++) {
28         printf("%d ", randRange(6)+1);
29     }
30
31     putchar('\n');
32
33     return 0;
34 }

```

### randRange.c

More generally, rejection sampling can be used to get random values with particular properties, where it's hard to generate a value with that property directly. Here's a program that generates random primes:

Toggle line numbers

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <time.h>
5
6 /* return 1 if n is prime */
7 int
8 isprime(int n)
9 {

```

```
10     int i;
11
12     if(n % 2 == 0 || n == 1) { return 0; }
13
14     for(i = 3; i*i <= n; i += 2) {
15         if(n % i == 0) { return 0; }
16     }
17
18     return 1;
19 }
20
21 /* return a uniform random value in the range 0..n-1 inclusive */
22 int
23 randPrime(void)
24 {
25     int r;
26
27     /* extra parens avoid warnings */
28     while(!isprime((r = rand())));
29
30     return r;
31 }
32
33 int
34 main(int argc, char **argv)
35 {
36     int i;
37
38     srand(time(0));
39
40     for(i = 0; i < 10; i++) {
41         printf("%d\n", randPrime());
42     }
43
44     return 0;
45 }
```

 randPrime.c

One temptation to avoid is to re-use your random values. If, for example, you try to find a random prime by picking a random  $x$  and trying  $x$ ,  $x+1$ ,  $x+2$ , etc., until you hit a prime, some primes are more likely to come up than others.

## 2. Randomized algorithms

## 2.1. Randomized search

This is essentially rejection sampling in disguise. Suppose that you want to find one of many needles in a large haystack. One approach is to methodically go through the straws/needles one at a time until you find a needle. But you may find that your good friend the adversary has put all the needles at the end of your list. Picking candidate at random is likely to hit a needle faster if there are many of them.

Here is a (silly) routine that quickly finds a number whose high-order bits match a particular pattern:

Toggle line numbers

```
1 int
2 matchBits(int pattern)
3 {
4     int r;
5
6     while(((r = rand()) & 0x70000000) != (pattern & 0x70000000));
7
8     return r;
9 }
```

This will find a winning value in 8 tries on average. In contrast, this deterministic version will take a lot longer for nonzero patterns:

Toggle line numbers

```
1 int
2 matchBitsDeterministic(int pattern)
3 {
4     int i;
5
6     for(i = 0; (i & 0x70000000) != (pattern & 0x70000000); i++);
7
8     return i;
9 }
```

The downside of the randomized approach is that it's hard to tell when to quit if there are no matches; if we stop after some fixed number of trials, we get a Wikipedia: Monte Carlo algorithm that may give the wrong answer with small probability. The usual solution is to either accept a small probability of failure, or interleave a deterministic backup algorithm that always works. The latter approach gives a Wikipedia: Las Vegas algorithm whose running time is variable but whose correctness is not.

## 2.2. Quickselect and quicksort

QuickSelect, or Hoare's FIND, is an algorithm for quickly finding the  $k$ -th largest element in an unsorted array of  $n$  elements. It runs in  $O(n)$  time on average, which is the best one can hope for (we have to look at every element of the array to be sure we didn't miss a small one that changes our answer) and better than the  $O(n \log n)$  time we get if we sort the array first using a comparison-based sorting algorithm.

The idea is to pick a random pivot and divide the input into two piles, each of which is likely to be roughly a constant fraction of the size of the original input.<sup>1</sup> It takes  $O(n)$  time to split the input up (we have to compare each element to the pivot once), and in the recursive calls this gives a geometric series. We can even do the splitting up in place if we are willing to reorder the elements of our original array.

If we recurse into both piles instead of just one, we get QuickSort, a very fast and simple comparison-based sorting algorithm. Here is an implementation of both algorithms:

Toggle line numbers

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4
5  /* reorder an array to put elements <= pivot
6   * before elements > pivot.
7   * Returns number of elements <= pivot */
8  static int
9  splitByPivot(int n, int *a, int pivot)
10 {
11     int lo;
12     int hi;
13     int temp; /* for swapping */
14
15     assert(n >= 0);
16
17     /* Dutch Flag algorithm */
18     /* swap everything <= pivot to bottom of array */
19     /* invariant is i < lo implies a[i] <= pivot */
20     /* and i > hi implies a[i] > pivot */
21     lo = 0;
22     hi = n-1;
23
24     while(lo <= hi) {
25         if(a[lo] <= pivot) {
26             lo++;
27         } else {
28             temp = a[hi];
29             a[hi--] = a[lo];
30             a[lo] = temp;
31         }


```



```
32     }
33
34     return lo;
35 }
36
37 /* find the k-th smallest element of an n-element array */
38 /* may reorder elements of the original array */
39 int
40 quickselectDestructive(int k, int n, int *a)
41 {
42     int pivot;
43     int lo;
44
45     assert(0 <= k);
46     assert(k < n);
47
48     if(n == 1) {
49         return a[0];
50     }
51
52     /* else */
53     pivot = a[rand() % n];    /* we will tolerate non-uniformity */
54
55     lo = splitByPivot(n, a, pivot);
56
57     /* lo is now number of values <= pivot */
58     if(k < lo) {
59         return quickselectDestructive(k, lo, a);
60     } else {
61         return quickselectDestructive(k - lo, n - lo, a + lo);
62     }
63 }
64
65 /* sort an array in place */
66 void
67 quickSort(int n, int *a)
68 {
69     int pivot;
70     int lo;
71
72     if(n <= 1) {
73         return;
74     }
75
76     /* else */
77     pivot = a[rand() % n];    /* we will tolerate non-uniformity */
78
79     lo = splitByPivot(n, a, pivot);
80
```

```
81     quickSort(lo, a);
82     quickSort(n - lo, a + lo);
83 }
84
85
86 /* shuffle an array */
87 void
88 shuffle(int n, int *a)
89 {
90     int i;
91     int r;
92     int temp;
93
94     for(i = n - 1; i > 0; i--) {
95         r = rand() % i;
96         temp = a[r];
97         a[r] = a[i];
98         a[i] = temp;
99     }
100 }
101
102 #define N (1024)
103
104 int
105 main(int argc, char **argv)
106 {
107     int a[N];
108     int i;
109
110     srand(0); /* use fixed value for debugging */
111
112     for(i = 0; i < N; i++) {
113         a[i] = i;
114     }
115
116     shuffle(N, a);
117
118     for(i = 0; i < N; i++) {
119         assert(quickselectDestructive(i, N, a) == i);
120     }
121
122     shuffle(N, a);
123
124     quickSort(N, a);
125
126     for(i = 0; i < N; i++) {
127         assert(a[i] == i);
128     }
129 }
```

```
130     return 0;  
131 }
```


 quick.c

## 3. Randomized data structures

### 3.1. Randomized tree balancing

Suppose we insert  $n$  elements into an initially-empty binary search tree in random order with no rebalancing. Then each element is equally likely to be the root, and all the elements less than the root end up in the left subtree, while all the elements greater than the root end up in the right subtree, where they are further partitioned recursively. This is exactly what happens in quicksort, so the structure of the tree will exactly mirror the structure of an execution of quicksort. In particular, the average depth of a node will be  $O(\log n)$ , giving us the same expected search cost as in a balanced binary tree.

The problem with this approach is that we don't have any guarantees that the input will be supplied in random order, and in the worst case we end up with a linked list. The solution is to put the randomization into the algorithm itself, making the structure of the tree depend on random choices made by the program itself.

A **skip list** ( Pugh, 1990) is a randomized tree-like data structure based on linked lists. It consists of a level 0 list that is an ordinary sorted linked list, together with higher-level lists that contain a random sampling of the elements at lower levels. When inserted into the level  $i$  list, an element flips a coin that tells it with probability  $p$  to insert itself in the level  $i+1$  list as well.

Searches in a skip list are done by starting in the highest-level list and searching forward for the last element whose key is smaller than the target; the search then continues in the same way on the next level down. The idea is that the higher-level lists act as express lanes to get us to our target value faster. To bound the expected running time of a search, it helps to look at this process backwards; the reversed search path starts at level 0 and continues going backwards until it reaches the first element that is also in a higher level; it then jumps to the next level up and repeats the process. On average, we hit  $1+1/p$  nodes at each level before jumping back up; for constant  $p$  (e.g.  $1/2$ ), this gives us  $O(\log n)$  steps for the search.

The space per element of a skip list also depends on  $p$ . Every element has at least one outgoing pointer (on level 0), and on average has exactly  $1/(1-p)$  expected pointers. So the space cost can also be adjusted by adjusting  $p$ . For example, if space is at a premium, setting  $p = 1/10$  produces  $10/9$  pointers per node on average—not much more than in a linked list—but still gives  $O(\log n)$  search times.

Below is an implementation of a skip list. To avoid having to allocate a separate array of pointers for each element, we put a length-1 array at the end of `struct skiplist` and rely on C's lack of bounds checking to make the array longer if necessary. A dummy head element stores pointers to all the initial elements in each level of the skip list; it is given the dummy key `INT_MIN` so that searches for values less than any in the list will report this value. Aside from these nasty tricks, the code for search and insertion is pretty

straightforward. Code for deletion is a little more involved, because we have to make sure that we delete the leftmost copy of a key if there are duplicates (an alternative would be to modify `skiplistInsert` to ignore duplicates).

Toggle line numbers

```
1 #include <stdlib.h>
2 #include <assert.h>
3 #include <limits.h>
4
5 #include "skiplist.h"
6
7 #define MAX_HEIGHT (32)
8
9 struct skiplist {
10     int key;
11     int height;           /* number of next pointers */
12     struct skiplist *next[1]; /* first of many */
13 };
14
15 /* choose a height according to a geometric distribution */
16 static int
17 chooseHeight(void)
18 {
19     int i;
20
21     for(i = 1; i < MAX_HEIGHT && rand() % 2 == 0; i++);
22
23     return i;
24 }
25
26 /* create a skiplist node with the given key and height */
27 /* does not fill in next pointers */
28 static Skiplist
29 skiplistCreateNode(int key, int height)
30 {
31     Skiplist s;
32
33     assert(height > 0);
34     assert(height <= MAX_HEIGHT);
35
36     s = malloc(sizeof(struct skiplist) + sizeof(struct skiplist *) * (height - 1));
37
38     assert(s);
39
40     s->key = key;
41     s->height = height;
42
43     return s;
```

```
44 }
45
46 /* create an empty skiplist */
47 Skiplist
48 skiplistCreate(void)
49 {
50     Skiplist s;
51     int i;
52
53     /* s is a dummy head element */
54     s = skiplistCreateNode(INT_MIN, MAX_HEIGHT);
55
56     /* this tracks the maximum height of any node */
57     s->height = 1;
58
59     for(i = 0; i < MAX_HEIGHT; i++) {
60         s->next[i] = 0;
61     }
62
63     return s;
64 }
65
66 /* free a skiplist */
67 void
68 skiplistDestroy(Skiplist s)
69 {
70     Skiplist next;
71
72     while(s) {
73         next = s->next[0];
74         free(s);
75         s = next;
76     }
77 }
78
79 /* return maximum key less than or equal to key */
80 /* or INT_MIN if there is none */
81 int
82 skiplistSearch(Skiplist s, int key)
83 {
84     int level;
85
86     for(level = s->height - 1; level >= 0; level--) {
87         while(s->next[level] && s->next[level]->key <= key) {
88             s = s->next[level];
89         }
90     }
91
92     return s->key;
```

```
93 }
94
95 /* insert a new key into s */
96 void
97 skiplistInsert(Skiplist s, int key)
98 {
99     int level;
100     Skiplist elt;
101
102     elt = skiplistCreateNode(key, chooseHeight());
103
104     assert(elt);
105
106     if(elt->height > s->height) {
107         s->height = elt->height;
108     }
109
110     /* search through levels taller than elt */
111     for(level = s->height - 1; level >= elt->height; level--) {
112         while(s->next[level] && s->next[level]->key < key) {
113             s = s->next[level];
114         }
115     }
116
117     /* now level is elt->height - 1, we can start inserting */
118     for(; level >= 0; level--) {
119         while(s->next[level] && s->next[level]->key < key) {
120             s = s->next[level];
121         }
122
123         /* s is last entry on this level < new element */
124         /* do list insert */
125         elt->next[level] = s->next[level];
126         s->next[level] = elt;
127     }
128 }
129
130 /* delete a key from s */
131 void
132 skiplistDelete(Skiplist s, int key)
133 {
134     int level;
135     Skiplist target;
136
137     /* first we have to find leftmost instance of key */
138     target = s;
139
140     for(level = s->height - 1; level >= 0; level--) {
141         while(target->next[level] && target->next[level]->key < key) {
```

```


142         target = target->next[level];
143     }
144 }
145
146 /* take one extra step at bottom */
147 target = target->next[0];
148
149 if(target == 0 || target->key != key) {
150     return;
151 }
152
153 /* now we found target, splice it out */
154 for(level = s->height - 1; level >= 0; level--) {
155     while(s->next[level] && s->next[level]->key < key) {
156         s = s->next[level];
157     }
158
159     if(s->next[level] == target) {
160         s->next[level] = target->next[level];
161     }
162 }
163
164 free(target);
165 }

```

 [skiplist.c](#)

Here is the header file, Makefile, and test code:  [skiplist.h](#),  [Makefile.skiplist](#),  [test\\_skiplist.c](#).

## 3.2. Universal hash families

Randomization can also be useful in hash tables. Recall that in building a hash table, we are relying on the hash function to spread out bad input distributions over the indices of our array. But for any fixed hash function, in the worst case we may get inputs where every key hashes to the same location. **Universal hashing** ( Carter and Wegman, 1979) solves this problem by choosing a hash function at random. We may still get unlucky and have the hash function hash all our values to the same location, but now we are relying on the random number generator to be nice to us instead of the adversary. We can also rehash with a new random hash function if we find out that the one we are using is bad.

The problem here is that we can't just choose a function uniformly at random out of the set of all possible hash functions, because there are too many of them, meaning that we would spend more space representing our hash function than we would on the table. The solution is to observe that we don't need our hash function  $h$  to be truly random; it's enough if the probability of collision  $\Pr[h(x) = h(y)]$  for any fixed keys  $x \neq y$  is  $1/m$ , where  $m$  is the size of the hash table. The reason is that the cost of searching for  $x$  (with chaining) is linear in the number of keys already in the table that collide with it. The expected number of such collisions is the sum of  $\Pr[h(x) =$

$h(y)$  over all keys  $y$  in the table, or  $n/m$  if we have  $n$  keys. So this pairwise collision probability bound is enough to get the desired  $n/m$  behavior out of our table. A family of hash function with this property is called **universal**.

How do we get a universal hash family? For strings, we can use a table of random values, one for each position and possible character in the string. The hash of a string is then the exclusive or of the random values `hashArray[i][s[i]]` corresponding to the actual characters in the string. If our table has size a power of two, this has the universal property, because if two strings  $x$  and  $y$  differ in some position  $i$ , then there is only one possible value of `hashArray[i][y[i]] (mod m)` that will make the hash functions equal.

Typically to avoid having to build an arbitrarily huge table of random values, we only has an initial prefix of the string. Here is a hash function based on this idea, which assumes that the `d` data structure includes a `hashArray` field that contains the random values for this particular hash table:

Toggle line numbers

```

1 static unsigned long
2 hash_function(Dict d, const char *s)
3 {
4     unsigned const char *us;
5     unsigned long h;
6     int i;
7
8     h = 0;
9
10    us = (unsigned const char *) s;
11
12    for(i = 0; i < HASH_PREFIX_LENGTH && us[i] != '\0'; i++) {
13        h ^= d->hashArray[i][us[i]];
14    }
15
16    return h;
17 }
```

A modified version of the `Dict` hash table from `C/HashTables` that uses this hash function is given here: [dict.c](#), [dict.h](#), [test\\_dict.c](#), [Makefile.dict](#).

---

## CategoryProgrammingNotes

1. The actual analysis is pretty complicated, since we are more likely to land in a bigger pile, but it's not hard to show that on average even the bigger pile has no more than  $3/4$  of the elements. (1)

---

2014-06-17 11:57



