2

Software Biology

…can we doubt that individuals having any advantage, however slight, over others, would have the best chance of surviving and of procreating their kind? On the other hand, we may feel sure that any variation in the least degree injurious would be rigidly destroyed. This preservation of favourable variations and the rejection of injurious variations, I call Natural Selection. **— *Charles Darwin,* On the Origin of Species**

Based on a few simple mechanisms, life adapts species to uncountable niches in an ever-changing environment. A species can be thought of as life's solution to exploiting a niche; it seems "natural", then, that solution-seeking software could benefit from biological techniques. Of course, the software universe is far less complex than the biological world, and no piece of software can — or needs — to incorporate all of life's techniques. Whereas living things need to seek flexible roles in variable environments, computer algorithms often need only to find a specific answer to a fixed question.

Definitions

The quest to apply evolution to software is not new. The University of Michigan's John Holland defined the concept of **genetic algorithms** in a 1975 paper titled *Adaptation in Natural and Artificial Systems.* Reasoning that the robustness of life stemmed from evolution by natural selection, Holland concluded that biology could provide a metaphor for artificial systems. Holland began by codifying the precise mechanisms of biological evolution; he then applied those principles to the development of software.

Computer scientists still debate the precise definition of a genetic algorithm (or GA). In the broadest sense, a GA creates a set of solutions that reproduce based on their fitness in a given environment. The process follows this pattern:

1. An initial population of random solutions is created.
2. Each member of the population is assigned a fitness value based on its evaluation against the current problem.
3. Solutions with a higher fitness value are most likely to parent new solutions during reproduction.
4. The new solution set replaces the old, a *generation* is complete, and the process continues by returning to step 2.

That sequence implements, in a most simplistic way, the concept of survival of the fittest. The reproductive success of a solution is directly tied to the fitness value it is given during evaluation. In this stochastic process, the least-fit solution has a small chance at reproduction while the most-fit solution may not reproduce at all. The outcome of a genetic algorithm is based on probabilities, just as biological success is grounded in chance.

So how can a random process possibly reach any sort of definitive answer to a question? Look to biology for your answer; living things evolved flight, complex senses, intricate societies, and millions of other adaptations specific to various niches and environments. Computer programs operate in an environment that is several orders of magnitude simpler than the biological world; the powerful tools of nature should have no trouble finding solutions to relatively trivial problems.

The standard model for a GA solution is an anonymous bit string — called a chromosome after its biological counterpart — that must be decoded during evaluation. Where DNA uses a base-4 alphabet, binary string uses ones and zeros to encode information. During reproduction, the chromosomes of parent solutions combine and undergo mutation in creating the next generation. In the highly-idealized universe of silicon, we have fine control over this process.

Let's develop a genetic algorithm for a simple problem, and see how these techniques become reality in C++ code.

Components of a Genetic Algorithm

We'll begin with a "black box" that returns a single output value for every input value. No one knows what is going on inside the box, but we do know the following facts:

- Input to the box is a 32-bit signed integer.
- Output from the box is a 32-bit signed integer with a value between 0 and 32.
- Only one, unknown input value generates an output of 32.
- Output values less than 32 may be produced by several different inputs.
- There is no obvious correspondence between input values and their outputs. For example, an input of 32 generates an output of zero, while an input of ten produces an output of four.
- The box cannot be opened or otherwise investigated, other than by feeding it values and examining the associated output.

Our task is to find the input value that produces an output of 32; in other words, we want to *maximize* then output of the black box. For the purposes of this example, I define a **BlackBox** function with the following prototype:

```
long BlackBox(long x);
```

You have no idea how **BlackBox** is implemented; it could, for instance, be defined in a module for which the source code is missing or lost. A brute force approach would test all 4,294,967,296 possible **long** values. Here's just such an algorithm:

```
long n = LONG_MIN;

while (true)
{
    if (BlackBox(n) == 32L)
    {
        cout << n << " generates 32!";
        break;
    }

    if (n = LONG_MAX)
    {
```

```
            cout << "Didn't find 32!"
            break;
        }

        ++n;
    }
```

The problem with a brute force approach isn't that it won't find the answer; assuming that we have our facts correct, one of the input values will generate an answer of 32. Where brute force falls on its face is in how long it might take to find an answer. The loop above might find the answer after a few loops, or after *billions*. Clearly, a more intelligent solution is desired.

Populations

Let's build a genetic algorithm to maximize the output from **BlackBox**. The first task is to define our solution set, or *population*. Since the input to **BlackBox** is a **long**, our population will consist of an array of **long**s to be tested against the function. Our chromosome, then, is a 32-bit string. We're looking for the largest value returned from **BlackBox** for a given member of our population; therefore, we'll define a second array of **long**s, the same size as the population array, to hold corresponding output values from **BlackBox**. In other words, element five of the fitness array holds the value returned by **BlackBox** when it is given the input value stored in the fifth element of the population array.

Selection by Chance

Assuming that we haven't found the input that produces 32, we now need to produce a new population from the old, using the fitness values to determine reproductive success. The simplest way of accomplishing this is to use a roulette wheel. You've probably seen a standard gambler's roulette wheel, a spinning circle divided into thirty-seven or thirty-eight equal-sized, pie-shaped sections. The croupier sets the wheel spinning and at the same time tosses an marble into the bowl in the direction opposite to that in which the wheel is moving; when the motion of the wheel ceases, the ball comes to rest in one of the numbered sections.

In the case of a genetic algorithm, a roulette wheel selects the chromosomes used in reproduction. The wheel is the fitness array, and the marble is a random integer less than the sum of all fitnesses in the population. To find the chromosome associated with the marble's landing place, the algorithm iterates through the fitness array; if the marble value is less than the current fitness element, the corresponding chromosome becomes a parent. Otherwise, the algorithm subtracts the current fitness value from the marble, and then repeats the process with the next element in the fitness array. Thus the largest fitness values tend to be the most likely resting places for the marble, since they use a larger area of the abstract wheel.

To clarify, lets look at a small example with a hypothetical population of five. Figure 2-1 shows the population and its corresponding fitness values.

Chromosome	Fitness
10110110	20
10000000	5
11101110	15
10010011	8
10100010	12

Figure 2-1 Hypothetical Population and its Fitness

The total fitness of this population is 60. Figure 2-2 is a simple pie chart showing the relative sizes of pie slices as assigned by fitness.

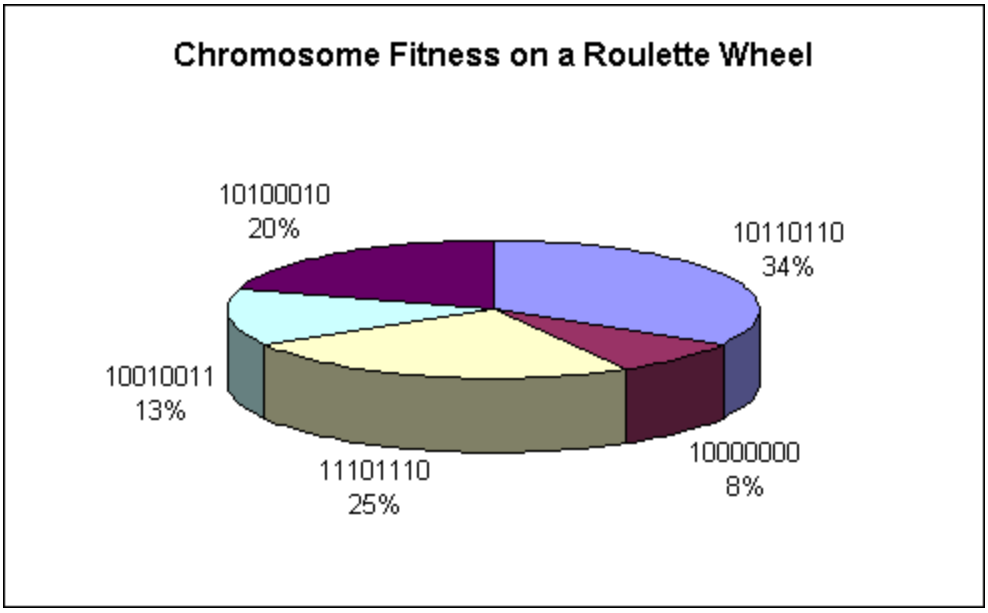

Figure 2-2 A Roulette Wheel of Fitness

In other words, the chromosome 10110110 has a 34% chance of being selected as a parent, whereas 10000000 has only an 8% chance of generating a new chromosome. Selecting five parents requires the algorithm to generate five random numbers, as shown in Figure 2-3.

Random Number	Chromosome Chosen
44	10010011
5	10110110
49	10100010
18	10110110
22	10000000

Figure 2-3 Roulette Selecting Parent Chromosomes

The chromosome with the highest fitness, 10110110, parents two members of the new population; even the chromosome with the lowest fitness will be a parent once. However, the

second most fit chromosome did not reproduce -- an example of the random nature of roulette wheel selection.

Crossover

Crossover is a central feature of genetic algorithms that creates a new chromosome from two parents. Corresponding to biological crossover, the software version combines a pair of parents by randomly selecting a point at which pieces of the parents' bit strings are swapped. Figure 2-4 shows four examples of crossover. The vertical bar in the child chromosome indicates the point of crossover.

Parent Chromosome 1	*Parent Chromosome 2*	*Crossover Point*	*Child (New) Chromosome*
10010011	10110110	3	100|10110
10000000	10110110	6	100000|10
10110110	11101110	2	10|101110
10110110	11101110	5	10110|110

Figure 2-4 Crossover Examples

Crossover allows the mixing of attributes from different chromosomes. In association with crossover, Holland developed the theory of *schema*: sets of bits that provide building blocks for high-fitness chromosomes. Holland determined that crossover mixes schemata from the chromosomes with the highest fitness values, thus increasing the frequency of valuable schema within a population. Some computer scientists characterize genetic algorithms as manipulators of schemata within chromosomes.

Mutation

The final step in reproduction is mutation, which involves the random change of one or more bits in each chromosome of the new population. The primary purpose of mutation is to increase variation into a population; mutation is most important in populations where the initial population may be a small subset of all possible solutions. It is possible, for example, that every instance of an essential bit might be zero in the initial population. In such a case, crossover could never set that bit to one, and the correct solution would not be found without mutation's ability to set that bit "randomly."

Different programmers implement various ways of mutating chromosomes. A common system sets a probability that any given bit will be changed; a test is performed for each bit to see if its value should be changed. Such a system requires the generation of many random numbers, which can degrade program performance. For example, in a population of fifty **long** chromosomes would use 1600 random values during mutation.

I prefer a simpler solution, defining a probability that a bit — any bit — in a chromosome will be randomly changed. This requires the generation of one or two random numbers -- the first to decide if mutation will take place, and the second to determine the bit that changes. To allow for multiple bit changes within a chromosome, I sometimes implement a looping system such as:

```
while (rand_value() < mutationChance)
    mutate(chromosome);
```

Rules of Thumb

Mutation and crossover are known as *reproduction operators*. Holland also defined another reproduction operator, *inversion*, which reverses a segment of bits within a chromosome. Few researchers have examined inversion seriously, and those that have suggest the use of inversion only when the chromosome size is "large." For now, I'll ignore inversion; it will come up again in later chapters, when I describe more sophisticated problems and solutions.

Even if you don't use inversion, you should still keep chromosome length in mind when implementing crossover and mutation. In a four-bit chromosome, only sixteen possibilities exist; crossover probably won't create any new chromosomes in such an environment because all possibilities are likely represented in even a small population. Mutation, in turn, has less effect in longer chromosomes; changing one bit in a 256-bit string has less influence than would flipping a bit in a 16-bit chromosome.

Another thing to keep in mind is the population size of a solution set. The larger the population, the longer it will take to process. Smaller populations, however, lack a robust selection of chromosomes. In designing a genetic algorithm, you need to balance the size of the population against the number of generations required for finding a solution. A population of a hundred chromosomes may find the solution in only ten generations, but each evaluation cycle may take four times as long as it would for a population of twenty chromosomes who find the solution in twenty generations.

Population size has another influence on genetic algorithms: it dilutes the influence of high fitness values on reproductive success. In a population of ten chromosomes, in which one has a fitness of nine and the others a fitness of one, half of all parents will probably be selected from the nine relatively-unfit chromosomes, even though the best chromosome is nine times more fit.

Evaluating chromosomes and calculating fitness is the most time-consuming component of a genetic algorithm. Your goal should be to reduce the number of fitness evaluations, either by reducing the size of the population or by decreasing the number of generations required for finding the solution.

Before getting into the development of code, I'll bring up two more techniques that can enhance the performance of a genetic algorithm. *Elitist selection* always copies the most-fit chromosome into the next generation, guaranteeing that the best solution survives so that a population doesn't "lose ground" in terms of fitness.

Another enhancement is *fitness scaling*. As a population converges on a definitive solution, the difference between fitness values may become very small. That produces a roulette wheel with nearly-equal sections, preventing the best solutions from having a significant advantage in reproductive selection. Fitness scaling solves this problem by adjusting the fitness values to the advantage of the most-fit chromosomes.

Windowing is the simplest form of fitness scaling. To implement windowing, begin by computing fitness values as usual, keeping track of the smallest fitness value. Then, subtract the minimum value from all fitness values, thus adjusting the fitness array to a zero-base. You might also want to subtract a number slightly smaller than the minimum fitness, just to ensure that all chromosomes have a chance at reproduction.

Programming genetic algorithms is largely a matter of "feel." You need to see what works, and how, to understand exactly which factors produce the fastest results in a given situation. To aid your exploration, I've defined a set of parameters for solving this chapter's problems. You

can set those parameters to a variety of values, learning how different combinations of options affect performance.

An Implementation

The **OptimizeBlackBox** function has seven parameters that define how the genetic algorithm works: population size, the number of generations to run, the chance of crossover, the rate of mutation, and Boolean values that turn on and off elitist selection and fitness scaling. The program's **main** function passes command-line arguments into these parameters; if the command line lacks the required argument, **main** uses default values.

```
int main(int argc, char* argv[])
{
    if (argc != 7)
        OptimizeBlackBox(1000,50,0.5F,0.1F,true,true,cout);
    else
    {
        OptimizeBlackBox(atoi(argv[1]),
                         atoi(argv[2]),
                         atof(argv[3]),
                         atof(argv[4]),
                         argv[5][0] == '1',
                         argv[6][0] == '1',
                         cout);
    }

    cout << endl;

    return 0;
}
```

Evolving an Answer

With a parameters in hand, it's time to write a genetic algorithm for optimizing the output from the **BlackBox** function. For now, I'll keep the implementation of **BlackBox** a secret; while you may have guessed by now what it is doing, I don't want to give away the optimum solution until you've seen how the genetic algorithm performs.

The algorithm begins by verifying the parameters, replacing any invalid or nonsensical values with defaults.

```
void OptimizeBlackBox
    (
    size_t populationSize,
    size_t numGenerations,
    float  crossoverRate,
    float  mutationRate,
    bool   elitismEnabled,
    bool   fitnessScalingEnabled,
    ostream & display
    )
{
    // display header
    display << "BlackBox Optimization" << endl
            << "---------------------" << endl
            << setprecision(7);

    #ifndef __GNUC__
    display.setf(ios::boolalpha);
```

```
#endif

// adjust any invalid parameters
if (populationSize < 10)
    populationSize = 10;

if (numGenerations < 1)
    numGenerations = 1;

if (crossoverRate < 0.0F)
    crossoverRate = 0.0F;
else
    if (crossoverRate > 1.0F)
        crossoverRate = 1.0;

if (mutationRate < 0.0F)
    mutationRate = 0.0F;
else
    if (mutationRate > 1.0F)
        mutationRate = 1.0;
```

Next, I seed the standard C++ random number generator and define working variables. The program allocates buffers to hold the population, its children, and fitness values. The last stage of initialization is the creation of an starting population of random chromosome values.

```
// initialize psuedo-random number generator
srand((unsigned)time(NULL));

// allocate population and fitness buffers
long * population = new long[populationSize];
long * children   = new long[populationSize];
long * fitness    = new long[populationSize];

// various variables
long  valueMostFit, fitnessHighest, fitnessLowest, mask;
float totalFitness, fitnessAverage;
size_t i, generationCounter, selection, father, mother, start;

// create initial population with random values
for (i = 0; i < populationSize; ++i)
    population[i] = long(rand());

// start with generation zero
generationCounter = 0;
```

The main loop looks like this; look over the commented code, and then read my elaborations below.

```
while (true) // loop breaks in the middle
{
    // initialize for fitness testing
    fitnessLowest  = LONG_MAX;
    fitnessHighest = -1L;
    totalFitness   = 0.0F;

    // fitness testing
    for (i = 0; i < populationSize; ++i)
    {
        // call fitness function and store result
        fitness[i] = BlackBox(population[i]);

        // keep track of best fitness
```

```
    if (fitness[i] > fitnessHighest)
    {
        fitnessHighest = fitness[i];
        valueMostFit = population[i];
    }

    // keep track of least fitness
    if (fitness[i] < fitnessLowest)
        fitnessLowest = fitness[i];

    // total fitness
    totalFitness += float(fitness[i]);
}

// make sure we have at least some fitness values
if (totalFitness == 0.0F)
{
    display << "** Population has total fitness of zero –"
               "optimzation terminated." << endl;
    break;
}

// compute average fitness
fitnessAverage = totalFitness / float(populationSize);

// display stats for this generation
display << setw(4) << generationCounter;
display.setf(ios::internal);
display << " best: " << hex << showbase << setfill('0')
        << setw(10) << valueMostFit << setfill(' ');
display.unsetf(ios::internal);
display << ", fitness = " << dec << setw(2) << fitnessHighest
        << ", average fitness = " << fitnessAverage
        << endl;

// jump out if we've found the solution
if (fitnessHighest == 32L)
{
    display << "** Optimization complete!" << endl;
    break;
}

// if enabled, scale fitness values
if (fitnessScalingEnabled)
{
    // ensures that the least fitness is one
    ++fitnessLowest;

    // recalculate total fitness to reflect scaled values
    totalFitness = 0.0F;

    for (i = 0; i < populationSize; ++i)
    {
        // reduce by smallest fitness
        fitness[i]   -= fitnessLowest;
        // square result of above
        fitness[i]   *= fitness[i];
        // add into total fitness
        totalFitness += float(fitness[i]);
    }
}

// exit if this is final generation
if (generationCounter == numGenerations)
```

```
{
    display << "** Done" << endl;
    break;
}

// if elitist selection, replace first item with best
if (elitismEnabled)
{
    children[0] = valueMostFit;
    start = 1;
}
else
    start = 0;

// create new population
for (i = start; i < populationSize; ++i)
{
    // roulette-select parent
    selection = (size_t)(getRandomFloat() * totalFitness);
    father    = 0;

    while (selection > fitness[father])
    {
        selection -= fitness[father];
        ++father;
    }

    // crossover reproduction
    if (getRandomFloat() <= crossoverRate)
    {
        // roulette-select second parent
        selection = (size_t)(getRandomFloat() * totalFitness);
        mother = 0;

        while (selection > fitness[mother])
        {
            selection -= fitness[mother];
            ++mother;
        }

        // mask of bits to be copied from first parent
        mask = 0xFFFFFFFFL << (int)(getRandomFloat() * 32.0F);

        // new string from two parents
        children[i] = (population[father] & mask)
                    | (population[mother] & (~mask));
    }
    else
    {
        // one parent, no crossover reproduction
        children[i] = population[father];
    }

    // mutation
    if (getRandomFloat() < mutationRate)
    {
        // select bit to be changed
        mask = 1L << (long)(getRandomFloat() * 32.0F);

        // flip the bit
        if (children[i] & mask)
            children[i] &= ~mask;
        else
            children[i] |= mask;
```

```
            }
        }

        // exchange old population with new one
        long * temp = children;
        children    = population;
        population  = temp;

        // increment generation
        ++generationCounter;
    }
```

The Standard C/C++ function **rand** only returns an integer value; I use a small inline function, **getRandomFloat**, to generate a random floating-point value between 0 and 1. This is *not* a good way to generate "random" floating-point values. However, this simple technique suffices for now — and in Chapter 3, I'll describe something called a *uniform deviate generator* that is most useful for stochastic computing.

```
inline float getRandomFloat()
{
    return (float(rand()) / float(RAND_MAX));
}
```

Each generation begins with the fitness testing of every chromosome through calls to **BlackBox**. The highest fitness is tracked for reporting purposes, while the least fitness is recorded in case fitness scaling is enabled. In the extremely unlikely event that the population has a total fitness of zero, the routine stops, since you can't optimize a population that has no potential. (I could make comments on certain segments of human society, but I guess I'll be polite… ☺)

After calculating the average fitness of the population and reporting statistics for the current generation, the algorithm performs fitness scaling (assuming it was selected). To adjust the fitness values, I subtract the minimum fitness value, add one, and square the result. Adding one gives every chromosome a chance of reproduction, and squaring the fitness value strengthens the reproductive chances of the most fit chromosomes.

A third loop implements reproduction. Using the roulette wheel technique, the algorithm selects one or two parents (depending on whether or not crossover is enabled). I use one shifted bitmask to implement crossover, and another bitmask in mutating a child chromosome. If elitism is enabled, the algorithm automatically copies the best chromosome of the parent generation into the first element of the new population. Then I copy the new population over the old, and proceed to the next generation.

Once all generations have run, the routine ends by deleting the various buffers.

```
    // delete population and fitness arrays
    delete [] population;
    delete [] children;
    delete [] fitness;
}
```

The output of the above routine looks like this, using the default configuration parameters:

```
BlackBox Optimization
---------------------
 population size: 100
# of generations: 50
  crossover rate: 100%
```

```
  mutation rate: 10%
scaling enabled: true
elitism enabled: true

  0 best: 0x00003842, fitness = 22, average fitness = 16.43
  1 best: 0x0000384a, fitness = 23, average fitness = 18.36
  2 best: 0x0000384a, fitness = 23, average fitness = 20.42
  3 best: 0x000c3842, fitness = 24, average fitness = 21.69
  4 best: 0x000c3842, fitness = 24, average fitness = 22.37
  5 best: 0x0008784a, fitness = 25, average fitness = 22.76
  6 best: 0x0008784a, fitness = 25, average fitness = 23.27
  7 best: 0x0108784a, fitness = 26, average fitness = 23.8
  8 best: 0x0108784a, fitness = 26, average fitness = 24.47
  9 best: 0x1018784a, fitness = 27, average fitness = 25.04
 10 best: 0x1018784a, fitness = 27, average fitness = 25.28
 11 best: 0x1018784a, fitness = 27, average fitness = 25.8
 12 best: 0x108c784a, fitness = 28, average fitness = 25.84
 13 best: 0x108c784a, fitness = 28, average fitness = 26.75
 14 best: 0x118c784a, fitness = 29, average fitness = 27.45
 15 best: 0x118c784a, fitness = 29, average fitness = 27.91
 16 best: 0x119c784a, fitness = 30, average fitness = 28.14
 17 best: 0x119c784a, fitness = 30, average fitness = 28.52
 18 best: 0x119e784a, fitness = 31, average fitness = 28.95
 19 best: 0x119e784a, fitness = 31, average fitness = 29.39
 20 best: 0x119e784a, fitness = 31, average fitness = 29.89
 21 best: 0x11de784a, fitness = 32, average fitness = 30.41
** Optimization complete!
```

A genetic algorithm is a stochastic process that exhibits variable performance. You can run the program a million times, a probably never see the same output twice — but you *will* see the correct result generated in every run. Well, maybe not *every* run; it is very remotely possible that the algorithm could fail to find the correct answer after fifty (or even a million) generations. Think of it this way: It is possible to flip a coin and get "heads" a million times in a row, without ever seeing "tails" — but the likelihood of such is so infinitesimally small that we needn't worry about it.

Some Analysis

After studying many runs of this program, you can see how the various factors affect performance. If the mutation rate is too small or zero, the algorithm will run poorly; but if you set the mutation rate very high (say, above 50%), the algorithm may also be less effective, because mutations obscure useful combinations of genes created by crossover.

Enabling elitist selection improves the speed of the algorithm, ensuring that a solution is found before 400 generations have been run. Combining a low mutation rate with elitist selection further enhances the algorithm's speed. However, the most dramatic increase in performance comes when fitness scaling is employed with mutation and crossover. The reason for faster convergence lies with the nature of the problem being solved. Fitness values returned by **BlackBox** fall into a relatively narrow range, leaving little distinction between different chromosomes. This is known as the *close race* phenomena. The inclusion of fitness scaling adjusts reproductive success in favor of the chromosomes with the highest fitness.

You should also note a relationship between mutation rate and fitness scaling. Without scaling, a low mutation rate is advantageous; once fitness testing skews reproductive success, however, a higher mutation rate proves most useful by increasing variety in the population.

In case you were wondering, the **BlackBox** function compares the input value **x** to a constant named **secret**, returning the number of bits that match between the two values. Only one value can match all 32 bits in **secret**.

```
long BlackBox(long x)
{
    // test value -- the speed of light in meters per second
    static const long secret =  299792458L;

    long fitness  = 0L;
    long mask = 1L;

    // count matching bits
    for (int i = 0; i < 32; ++i)
    {
        if ((x & mask) == (secret & mask))
            ++fitness;

        mask <<= 1;
    }

    // return fitness between 0 and 32
    return fitness;
}
```

The genetic algorithm was looking for a single solution among more than four million — and it can find it in only a few seconds, testing, on average, only a few hundred values.

Onward

In optimizing the output of **BlackBox**, I've demonstrated only the basic principles of genetic algorithms. Now it's time to elaborate and extend these concepts; in the next chapter, I'll build a set of reusable components for GA development.