Chapter

5 Primitive Types

Representation is the essence of programming.

— *"The Mythical Man Month,"*
F. P. Brooks, 1975

A program updates variables in memory according to its instructions. Variables come in types—a type is a classification of data that spells out possible values for that type and the operations that can be performed on it.

A type can be provided by the language or defined by the programmer. Many languages provide types for Boolean, integer, character and floating point data. Often, there are multiple integer and floating point types, depending on signedness and precision. The width of these types is the number of bits of storage a corresponding variable takes in memory. For example, most implementations of C++ use 32 or 64 bits for an `int`. In Java an `int` is always 32 bits.

5.1 Computing the parity of a word

The parity of a binary word is 1 if the number of 1s in the word is odd; otherwise, it is 0. For example, the parity of 1011 is 1, and the parity of 10001000 is 0. Parity checks are used to detect single bit errors in data storage and communication. It is fairly straightforward to write code that computes the parity of a single 64-bit word.

How would you compute the parity of a very large number of 64-bit words?

Hint: Use a lookup table, but don't use 2^{64} entries!

Solution: The brute-force algorithm iteratively tests the value of each bit, and keeps the running XOR of the bits processed.

```
short Parity(unsigned long x) {
  short result = 0;
  while (x) {
    result ^= (x & 1);
    x >>= 1;
  }
  return result;
}
```

The time complexity is $O(n)$, where n is the word size.

On Page 24 we showed how to erase the lowest set bit in a word in a single operation. This can be used to improve performance in the best- and average-cases.

```
short Parity(unsigned long x) {
  short result = 0;
  while (x) {
    result ^= 1;
    x &= (x - 1);  // Drops the lowest set bit of x.
  }
  return result;
}
```

Let k be the number of bits set to 1 set in a particular word. (For example, for 10001010, $k = 3$.) Then time complexity of the algorithm above is $O(k)$.

The problem statement refers to computing the parity for a very large number of words. When you have to perform a large number of parity computations, and, more generally, any kind of bit fiddling computations, two keys to performance are processing multiple bits at a time and caching results in an array-based lookup table.

First we demonstrate caching. Clearly, we cannot cache the parity of every 64-bit integer. However, when computing the parity of a collection of bits, it does not matter how we group those bits, i.e., the computation is associative. It is feasible to cache the parity of all possible 16-bit words using an array; its length is $2^{16} = 65536$. Therefore, we can compute the parity of a 64-bit integer by grouping its bits into four nonoverlapping 16 bit words, computing the parity of each group, and then computing the parity of these four results.

We illustrate the approach with a lookup table for 2-bit words. The cache is $\langle 0, 1, 1, 0 \rangle$—these are the parities of (00), (01), (10), (11), respectively. To compute the parity of (11001010) we would compute the parities of (11), (00), (10), (10). By table lookup we see these are $0, 0, 1, 1$, respectively, so the final result is the parity of $0, 0, 1, 1$ which is 0.

To lookup the parity of the first two bits in (11101010), we right shift by 6, to get (00000011), and use this as an index into the cache. To lookup the parity of the next two bits, i.e., (10), we right shift by 4, to get (10) in the two least-significant bit places. The right shift does not remove the leading (11)—it results in (00001110). We cannot index the cache with this, it leads to an out-of-bounds access. To get the last two bits after the right shift by 6, we bitwise-AND (00001110) with (00000011) (this is the "mask" used to extract the last 2 bits). The result is (00000010). Similar masking is needed for the two other 2-bit lookups.

```
short Parity(unsigned long x) {
  const int kWordSize = 16;
  const int kBitMask = 0xFFFF;
  return precomputed_parity[x >> (3 * kWordSize)] ^
         precomputed_parity[(x >> (2 * kWordSize)) & kBitMask] ^
         precomputed_parity[(x >> kWordSize) & kBitMask] ^
         precomputed_parity[x & kBitMask];
}
```

The time complexity is a function of the size of the keys used to index the lookup table. Let L be the width of the words for which we cache the results, and n the word size. Since there are n/L terms, the time complexity is $O(n/L)$, assuming word-

level operations, such as shifting, take $O(1)$ time. (This does not include the time for initialization of the lookup table.)

The XOR of two bits is 1 if both bits are 0 or 1, otherwise it is 1. The XOR of a group of bits is its parity. XOR has the property of being associative (as previously described), as well as commutative, i.e., the order in which we perform the XORs does not change the result. We can exploit this fact to use the CPU's word-level XOR instruction to process multiple bits at a time.

For example, the parity of $\langle b_{63}, b_{62}, \ldots, b_3, b_2, b_1, b_0 \rangle$ equals the parity of the XOR of $\langle b_{63}, b_{62}, \ldots, b_{32} \rangle$ and $\langle b_{31}, b_{30}, \ldots, b_0 \rangle$. The XOR of these two 32-bit values can be computed with a single shift and a single 32-bit XOR instruction. We repeat the same operation on 32-, 16-, 8-, 4-, 2-, and 1-bit operands to get the final result. Note that the leading bits are not meaningful, and we have to explicitly extract the result from the least-significant bit.

We illustrate the approach with an 8-bit word. The parity of (11010111) is the same as the parity of (1101) XORed with (0111), i.e., of (1010). This is turn is the same as the parity of (10) XORed with (10), i.e., of (00). The final result is the XOR of (0) with (0), i.e., 0. Note that the first XOR yields (11011010), and only the last 4 bits are relevant going forward. The second XOR yields (11101100), and only the last 2 bits are relevant. The third XOR yields (1011010). The last bit is the result, and to extract it we have to bitwise-AND with (00000001).

```
short Parity(unsigned long x) {
  x ^= x >> 32;
  x ^= x >> 16;
  x ^= x >> 8;
  x ^= x >> 4;
  x ^= x >> 2;
  x ^= x >> 1;
  return x & 0x1;
}
```

The time complexity is $O(\log n)$, where n is the word size.

Note that we could have combined caching with word-level operations, e.g., by doing a lookup once we get to 16 bits.

5.2 Swap bits

There are a number of ways in which bit manipulations can be accelerated. For example, as described on Page 24, the expression x & $(x - 1)$ clears the lowest set bit in x, and x & ~$(x - 1)$ extracts the lowest set bit of x.

A 64-bit integer can be viewed as an array of 64 bits, with the bit at index 0 corresponding to the least significant bit (LSB), and the bit at index 63 corresponding to the most significant bit (MSB). Implement code that takes as input a 64-bit integer and swaps the bits at indices i and j. Figure 5.1 on the next page illustrates bit swapping for an 8-bit integer.

Hint: When is the swap necessary?

(a) The 8-bit integer 73 can be viewed as array of bits, with the LSB being at index 0.

(b) The result of swapping the bits at indices 1 and 6, with the LSB being at index 0. The corresponding integer is 11.

Figure 5.1: Example of swapping a pair of bits.

Solution: A brute-force approach would be to use bitmasks to extract the *i*th and *j*th bits, saving them to local variables. Consequently, write the saved *j*th bit to index *i* and the saved *i*th bit to index *j*, using a combination of bitmasks and bitwise operations.

The brute-force approach works generally, e.g., if we were swapping objects stored in an array. However, since a bit can only take two values, we can do a little better. Specifically, we first test if the bits to be swapped differ. If they do not, the swap does not change the integer. If the bits are different, swapping them is the same as flipping their individual values. For example in Figure 5.1, since the bits at Index 1 and Index 6 differ, we flip each bit.

In the code below we use standard bit-fiddling idioms for testing and flipping bits. Overall, the resulting code is slightly more succinct and efficient than the brute force approach.

```
long SwapBits(long x, int i, int j) {
  // Extract the i-th and j-th bits, and see if they differ.
  if (((x >> i) & 1) != ((x >> j) & 1)) {
    // i-th and j-th bits differ. We will swap them by flipping their values.
    // Select the bits to flip with bit_mask; flip them using an XOR.
    unsigned long bit_mask = (1L << i) | (1L << j);
    x ^= bit_mask;
  }
  return x;
}
```

The time complexity is $O(1)$, independent of the word size.

5.3 Reverse bits

Write a program that takes a 64-bit word and returns the 64-bit word consisting of the bits of the input word in reverse order. For example, if the input is alternating 1s and 0s, i.e., (1010 . . . 10), the the output should be alternating 0s and 1s, i.e., (0101 . . . 01).

Hint: Use a lookup table.

Solution: If we need to perform this operation just once, there is a simple brute-force algorithm: iterate through the 32 least significant bits of the input, and swap each with the corresponding most significant bit, using, for example, the approach in Solution 5.2.

To implement reverse when the operation is to be performed repeatedly, we look more carefully at the structure of the input, with an eye towards using a cache.

Let the input consist of the four 16-bit words y_3, y_2, y_1, y_0, with y_3 holding the most significant bits. Then the 16 least significant bits in the reverse come from y_3. To be precise, these bits appear in the reverse order in which they do in y_3. For example, if y_3 is (1110000000000001), then the 16 LSBs of the result are (1000000000000111).

Similar to computing parity (Problem 5.1 on Page 43), a very fast way to reverse bits for 16-bit words when we are performing many such reverses is to build an array-based lookup-table A such that for every 16-bit number y, $A[y]$ holds the bit-reversed y. We can then form the reverse of x with the reverse of y_0 in the most significant bit positions, followed by the reverse of y_1, followed by the reverse of y_2, followed by the reverse of y_3.

We illustrate the approach with 8-bit words and 2-bit lookup table keys. The table is rev $= \langle (00), (10), (01), (11) \rangle$. If the input is (10010011), its reverse is rev(11), rev(00), rev(01), rev(10), i.e., (11001001).

```
long ReverseBits(long x) {
  const int kWordSize = 16;
  const int kBitMask = 0xFFFF;
  return precomputed_reverse[x & kBitMask] << (3 * kWordSize) |
         precomputed_reverse[(x >> kWordSize) & kBitMask] << (2 * kWordSize) |
         precomputed_reverse[(x >> (2 * kWordSize)) & kBitMask] << kWordSize |
         precomputed_reverse[(x >> (3 * kWordSize)) & kBitMask];
}
```

The time complexity is identical to that for Solution 5.1 on Page 43, i.e., $O(n/L)$, for n-bit integers and L-bit cache keys.

5.4 Find a closest integer with the same weight

Define the *weight* of a nonnegative integer x to be the number of bits that are set to 1 in its binary representation. For example, since 92 in base-2 equals $(1011100)_2$, the weight of 92 is 4.

Write a program which takes as input a nonnegative integer x and returns $y \neq x$ such that y has the same weight as x, and the difference of x and y is as small as possible. You can assume x is not 0, or all 1s. For example, if $x = 6$, you should return 5.

Hint: Start with the least significant bit.

Solution: A brute-force approach might be to try all integers $x-1, x+1, x-2, x+2, \ldots$, stopping as soon as we encounter one with the same weight at x. This performs very poorly on some inputs. One way to see this is to consider the case where $x = 2^3 = 8$. The only numbers with a weight of 1 are powers of 2. Thus, the algorithm will try the following sequence: $7, 9, 6, 10, 5, 11, 4$, stopping at 4 (since its weight is the same as 8's weight). The algorithm tries 2^{3-1} numbers smaller than 8, namely, $7, 6, 5, 4$, and $2^{3-1} - 1$ numbers greater than 8, namely, $9, 10, 11$. This example generalizes. Consider, for example, $x = 2^{30}$. The power of 2 nearest to 2^{30} is 2^{29}. Therefore this computation will evaluate the weight of all integers between 2^{30} and 2^{29} and between 2^{30} and $2^{30} + 2^{29} - 1$, i.e., over one billion integers.

Heuristically, it is natural to focus on the LSB of the input, specifically, to swap the LSB with rightmost bit that differs from it. This yields the correct result for some inputs, e.g., for $(10)_2$ it returns $(01)_2$, which is the closest possible. However, more experimentation shows this heuristic does not work generally. For example, for $(111)_2$ (7 in decimal) it returns $(1110)_2$ which is 14 in decimal; however, $(1011)_2$ (11 in decimal) has the same weight, and is closer to $(111)_2$. This example suggests the correct approach, which is to swap the first two consecutive bits that differ. This approach works because we want to change bits that are as far to the right as possible.

```
unsigned long ClosestIntSameBitCount(unsigned long x) {
  for (int i = 0; i < 63; ++i) {
    if (((x >> i) & 1) != ((x >> (i + 1)) & 1)) {
      x ^= (1UL << i) | (1UL << (i + 1));  // Swaps bit-i and bit-(i + 1).
      return x;
    }
  }

  // Throw error if all bits of x are 0 or 1.
  throw invalid_argument("all bits are 0 or 1");
}
```

The time complexity is $O(n)$, where n is the integer width.

5.5 Compute $x \times y$ without arithmetical operators

Sometimes the processors used in ultra low-power devices such as hearing aids do not have dedicated hardware for performing multiplication. A program that needs to perform multiplication must do so explicitly using lower-level primitives.

Write a program that multiplies two nonnegative integers. The only operators you are allowed to use are

- assignment,
- the bitwise operators », «, |, &, ~, ^ and
- equality checks and Boolean combinations thereof.

You may use loops and functions that you write yourself. These constraints imply, for example, that you cannot use increment or decrement, or test if $x < y$.

Hint: Add using bitwise operations; multiply using shift-and-add.

Solution: A brute-force approach would be to perform repeated addition, i.e., initialize the result to 0 and then add x to it y times. For example, to form 5×3, we would form $0, 5, 10, 15$. The time complexity is very high—as much as $O(2^n)$, where n is the number of bits in the input, and it still leaves open the problem of adding numbers without the presence of an add instruction.

The algorithm taught in grade-school for decimal multiplication does not use repeated addition—it uses shift and add to achieve a much better time complexity. We can do the same with binary numbers—to multiply x and y we initialize the result to 0 and iterate through the bits of x, adding $2^k y$ to the result if bit k of x is 1.

The value $2^k y$ can be computed by left-shifting y by k. Since we cannot use add directly, we must implement it. We apply the grade-school algorithm for addition to the binary case, i.e., compute the sum bit-by-bit, and "rippling" the carry along.

As an example, we show how to multiply, $9 = (1001)_2$ and $13 = (1101)_2$ using the algorithm described above. In the first iteration, since the LSB of 13 is 1, we set the result to $(1001)_2$. The second bit of $(1101)_2$ is 0, so we move on to the third bit. This bit is 1, so we shift $(1001)_2$ to the left by 2 to obtain $(100100)_2$, which we add to $(1001)_2$ to get $(101101)_2$. The fourth and final bit of $(1101)_2$ is 1, so we shift $(1001)_2$ to the left by 3 to obtain $(1001000)_2$, which we add to $(101101)_2$ to get $(1110101)_2 = 117$.

Each addition is itself performed bit-by-bit. For example, when adding $(101101)_2$ and $(1001000)_2$, the LSB of the result is 1 (since exactly one of the two LSBs of the operands is 1). The next bit is 0 (since both the next bits of the operands are 0). The next bit is 1 (since exactly one of the next bits of the operands is 1). The next bit is 0 (since both the next bits of the operands are 1). We also "carry" a 1 to the next position. The next bit is 1 (since the carry-in is 1 and both the next bits of the operands are 0). The remaining bits are assigned similarly.

```
unsigned Multiply(unsigned x, unsigned y) {
  unsigned sum = 0;
  while (x) {
    // Examines each bit of x.
    if (x & 1) {
      sum = Add(sum, y);
    }
    x >>= 1, y <<= 1;
  }
  return sum;
}

unsigned Add(unsigned a, unsigned b) {
  unsigned sum = 0, carryin = 0, k = 1, temp_a = a, temp_b = b;
  while (temp_a || temp_b) {
    unsigned ak = a & k, bk = b & k;
    unsigned carryout = (ak & bk) | (ak & carryin) | (bk & carryin);
    sum |= (ak ^ bk ^ carryin);
    carryin = carryout << 1, k <<= 1, temp_a >>= 1, temp_b >>= 1;
  }
  return sum | carryin;
}
```

The time complexity of addition is $O(n)$, where n is the width of the operands. Since we do n additions to perform a single multiplication, the total time complexity is $O(n^2)$.

5.6 Compute x/y

Given two positive integers, compute their quotient, using only the addition, subtraction, and shifting operators.

Hint: Relate x/y to $(x - y)/y$.

Solution: A brute-force approach is to iteratively subtract y from x until the remaining term is less than y. The number of such subtractions is exactly the quotient, x/y. The complexity of the brute-force approach is very high, e.g., when $y = 1$ and $x = 2^{31} - 1$, it will take $2^{31} - 1$ iterations.

A better approach is to try and get more work done in each iteration. For example, we can compute the largest k such that $2^k y \leq x$, subtract $2^k y$ from x, and add 2^k to the quotient. For example, if $x = (1011)_2$ and $y = (10)_2$, then $k = 2$. We subtract $(1000)_2$ from $(1011)_2$ to get $(11)_2$, add $(100)_2$ to the quotient, and continue by updating x to $(11)_2$.

The advantage of using $2^k y$ is that it can be computed very efficiently using shifting, and x is at least halved in each iteration. If it takes n bits to represent x/y, there are $O(n)$ iterations. If the largest k such that $2^k y \leq x$ is computed by iterating through k, each iteration has time complexity $O(n)$. This leads to an $O(n^2)$ algorithm.

A better way to find the largest k in each iteration is to recognize that it keeps decreasing. Therefore, instead of testing in each iteration whether $2^0 y, 2^1 y, 2^2 y, \ldots$ is less than or equal to x, after we initially find the largest k such that $2^k y \leq x$, in subsequent iterations we test $2^{k-1} y, 2^{k-2} y, 2^{k-3} y, \ldots$ with x.

For the example given earlier, after setting the quotient to $(100)_2$ we continue with $(11)_2$. Now the largest k such that $2^k y \leq (11)_2$ is 0, so we add $2^0 y = (10)_2$ to the quotient, which is now $(101)_2$. We continue with $(11)_2 - (10)_2 = (1)_2$. Since $(1)_2 < y$, we are done—the quotient is $(101)_2$ and the remainder is $(1)_2$.

```
unsigned Divide(unsigned x, unsigned y) {
  unsigned result = 0;
  int power = 32;
  unsigned long long y_power = static_cast<unsigned long long>(y) << power;
  while (x >= y) {
    while (y_power > x) {
      y_power >>= 1;
      --power;
    }

    result += 1U << power;
    x -= y_power;
  }
  return result;
}
```

In essence, the program applies the grade-school division algorithm to binary numbers. With each iteration, we process an additional bit, so assuming individual shift and add operations take $O(1)$ time, the time complexity is $O(n)$.

5.7 Compute x^y

Write a program that takes a double x and an integer y and returns x^y. You can ignore overflow and underflow.

Hint: Exploit mathematical properties of exponentiation

Solution: First, assume y is nonnegative. The brute-force algorithm is to form $x^2 = x \times x$, then $x^3 = x^2 \times x$, and so on. This approach takes $y - 1$ multiplications, which is $O(2^n)$, where n is the number of bits in y.

The key to efficiency is to try and get more work done with each multiplication, thereby using fewer multiplications to accomplish the same result. For example, to compute 1.1^{21}, instead of multiplying by 1.1 20 times, we could multiply by $1.1^2 = 1.21$ 10 times for a total of 11 multiplications (one to compute 1.1^2, and 10 additional multiplications by 1.21). We can do still better by computing 1.1^3, 1.1^4, etc.

When y is a power of 2, the best approach is iterated squaring, i.e., $x, x^2, (x^2)^2 = x^4, (x^4)^2 = x^8, \ldots$. To develop an algorithm that works for general y, it is instructive to look at the binary representation of y, as well as properties of exponentiation, specifically $x^{y_0+y_1} = x^{y_0} \cdot x^{y_1}$.

We begin with some small concrete instances, first assuming that $y \geq 0$. For example, $x^{(1010)_2} = x^{(101)_2+(101)_2} = x^{(101)_2} \times x^{(101)_2}$. Similarly, $x^{(101)_2} = x^{(100)_2+(1)_2} = x^{(100)_2} \times x = x^{(10)_2} \times x^{(10)_2} \times x$.

Generalizing, if the least significant bit of y is 0, the result is $(x^{y/2})^2$; otherwise, it is $x \times (x^{y/2})^2$. This gives us a recursive algorithm for computing x^y when $y > 0$.

The only change when y is negative is replacing x by $1/x$ and y by $-y$. In the implementation below we replace the recursion with a while loop to avoid the overhead of function calls.

```
double Power(double x, int y) {
  double result = 1.0;
  long long power = y;
  if (y < 0) {
    power = -power, x = 1.0 / x;
  }
  while (power) {
    if (power & 1) {
      result *= x;
    }
    x *= x, power >>= 1;
  }
  return result;
}
```

The number of multiplications is at most twice the index of y's MSB, implying an $O(n)$ time complexity.

5.8 Reverse digits

Write a program which takes an integer and returns the integer corresponding to the digits of the input written in reverse order. For example, the reverse of 42 is 24, and the reverse of −314 is −413.

Hint: How would you solve the same problem if the input is presented as a string?

Solution: The brute-force approach is to convert the input to a string, and then compute the reverse from the string by traversing it from back to front. For example,

$(1100)_2$ is the decimal number 12, and the answer for $(1100)_2$ can be computed by traversing the string "12" in reverse order.

Closer analysis shows that we can avoid having to form a string. Consider the input 1132. The first digit of the result is 2, which we can obtain by taking the input modulo 10. The remaining digits of the result are the reverse of 1132/10 = 113. Generalizing, let the input be k. If $k \geq 0$, then k mod 10 is the most significant digit of the result and the subsequent digits are the reverse of $\frac{k}{10}$. Continuing with the example, we iteratively update the result and the input as 2 and 113, then 23 and 11, then 231 and 1, then 2311.

For general k, we record the sign of k and operate on its absolute value, and reapply the sign.

```
long Reverse(int x) {
  bool is_negative = x < 0;
  long result = 0, x_remaining = abs(x);
  while (x_remaining) {
    result = result * 10 + x_remaining % 10;
    x_remaining /= 10;
  }
  return is_negative ? -result : result;
}
```

The time complexity is $O(n)$, where n is the number of digits in k.

5.9 Check if a decimal integer is a palindrome

A palindromic string is one which reads the same forwards and backwards, e.g., "redivider". In this problem, you are to write a program which determines if the decimal representation of an integer is a palindromic string. For example, your program should return true for the inputs $0, 1, 7, 11, 121, 333$, and 2147447412, and false for the inputs $-1, 12, 100$, and 2147483647.

Write a program that takes an integer and determines if that integer's representation as a decimal string is a palindrome.

Hint: It's easy to come up with a simple expression that extracts the least significant digit. Can you find a simple expression for the most significant digit?

Solution: First note that if the input is negative, then its representation as a decimal string cannot be palindromic, since it begins with a $-$.

A brute-force approach would be to convert the input to a string and then iterate through the string, pairwise comparing digits starting from the least significant digit and the most significant digit, and working inwards, stopping if there is a mismatch. The time and space complexity are $O(n)$, where n is the number of digits in the input.

We can avoid the $O(n)$ space complexity by directly extracting the digits from the input. The number of digits, n, in the input's string representation is the log (base 10) of the input value, x. To be precise, $n = \lfloor \log_{10} x \rfloor + 1$. Therefore, the least significant digit is x mod 10, and the most significant digit is $x/10^{n-1}$. In the program below, we iteratively compare the most and least significant digits, and then remove them

from the input. For example, if the input is 151751, we would compare the leading and trailing digits, 1 and 1. Since these are equal, we update the value to 5175. The leading and trailing digits are equal, so we update to 17. Now the leading and trailing are unequal, so we return false. If instead the number was 157751, the final compare would be of 7 and 7, so we would return true.

```
bool IsPalindrome(int x) {
  if (x < 0) {
    return false;
  } else if (x == 0) {
    return true;
  }

  const int kNumDigits = static_cast<int>(floor(log10(x))) + 1;
  int msd_mask = static_cast<int>(pow(10, kNumDigits - 1));
  for (int i = 0; i < (kNumDigits / 2); ++i) {
    if (x / msd_mask != x % 10) {
      return false;
    }
    x %= msd_mask;  // Remove the most significant digit of x.
    x /= 10;  // Remove the least significant digit of x.
    msd_mask /= 100;
  }
  return true;
}
```

The time complexity is $O(n)$, and the space complexity is $O(1)$.

5.10 Generate uniform random numbers

This problem is motivated by the following scenario. Five friends have to select a designated driver using a single unbiased coin. The process should be fair to everyone.

How would you implement a random number generator that generates a random integer i between a and b, inclusive, given a random number generator that produces zero or one with equal probability? All values in $[a, b]$ should be equally likely.

Hint: How would you mimic a three-sided coin with a two-sided coin?

Solution: Note that it is easy to produce a random integer between 0 and $2^i - 1$, inclusive: we just concatenate i bits produced by the random number generator. For example, two calls to the random number generator produce $(00)_2, (01)_2, (10)_2, (11)_2$ and all of these are equally likely. These four possible outcomes encode the four integers $0, 1, 2, 3$.

For the general case, first note that it is equivalent to produce a random integer between 0 and $b - a$, inclusive, since we can simply add a to the result. If $b - a$ is equal to $2^i - 1$, for some i, then we can just use the approach in the previous paragraph.

If $b - a$ is not of the form $2^i - 1$, we find the smallest number of the form $2^i - 1$ that is greater than $b - a$. We generate an i-bit number as before. This i-bit number may or may not lie between 0 and $b - a$, inclusive. If it is within the range, we return it—all

such numbers are equally likely. If it is not within the range, we try again with i new random bits. We keep trying until we get a number within the range.

For example, to generate a random number corresponding to a dice roll, i.e., a number between 1 and 6, we begin by making three calls to the random number generator (since $2^2 - 1 < (6 - 1) \leq 2^3 - 1$). If this yields one of $(000)_2, (001)_2, (010)_2, (011)_2, (100)_2, (101)_2$, we return 1 plus the corresponding value. Observe that all six values between 1 and 6, inclusive, are equally likely to be returned. If the three calls yields one of $(110)_2, (111)_2$, we make three more calls. Note that the probability of having to try again is 2/8, which is less than half. Since successive calls are independent, the probability that we require many attempts diminishes very rapidly, e.g., the probability of not getting a result in 10 attempts is $(2/8)^{10}$ which is less than one-in-a-million.

```
int UniformRandom(int lower_bound, int upper_bound) {
  int number_of_outcomes = upper_bound - lower_bound + 1, result;
  do {
    result = 0;
    for (int i = 0; (1 << i) < number_of_outcomes; ++i) {
      // ZeroOneRandom() is the provided random number generator.
      result = (result << 1) | ZeroOneRandom();
    }
  } while (result >= number_of_outcomes);
  return result + lower_bound;
}
```

To analyze the time complexity, let $t = b - a + 1$. The probability that we succeed in the first try is $t/2^i$. Since 2^i is the smallest power of 2 greater than or equals to t, it must be less than $2t$. (An easy way to see this is to consider the binary representation of t and $2t$.) This implies that $t/2^i > t/2t = (1/2)$. Hence the probability that we do not succeed on the first try is $1 - t/2^i \leq 1/2$. Since successive tries are independent, the probability that more than k tries are needed is less than or equal to $1/2^k$. Hence, the expected number of tries is not more than $1 + 2(1/2)^1 + 3(1/2)^2 + \ldots$. The series converges, so the number of tries is $O(1)$. Each try makes $\lceil \lg(b - a + 1) \rceil$ calls to the 0/1-valued random number generator. Assuming the 0/1-valued random number generator takes $O(1)$ time, the time complexity is $O(\lg(b - a + 1))$.

5.11 Rectangle intersection

This problem is concerned with rectangles whose sides are parallel to the X-axis and Y-axis. See Figure 5.2 on the facing page for examples.

Write a program which tests if two rectangles have a nonempty intersection. If the intersection is nonempty, return the rectangle formed by their intersection.

Hint: Think of the X and Y dimensions independently.

Solution: Since the problem leaves it unspecified, we will treat the boundary as part of the rectangle. This implies, for example, rectangles A and B in Figure 5.2 on the next page intersect.

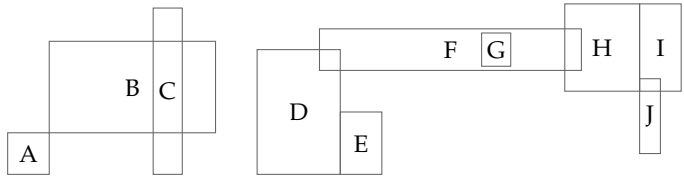

Figure 5.2: Examples of XY-aligned rectangles.

There are many qualitatively different ways in which rectangles can intersect, e.g., they have partial overlap (D and F), one contains the other (F and G), they share a common side (D and E), they share a common corner (A and B), they form a cross (B and C), they form a tee (F and H), etc. The case analysis is quite tricky.

A better approach is to focus on conditions under which it can be guaranteed that the rectangles do *not* intersect. For example, the rectangle with left-most lower point $(1, 2)$, width 3, and height 4 cannot possibly intersect with the rectangle with left-most lower point $(5, 3)$, width 2, and height 4, since the X-values of the first rectangle range from 1 to $1 + 3 = 4$, inclusive, and the X-values of the second rectangle range from 5 to $5 + 2 = 7$, inclusive.

Similarly, if the Y-values of the first rectangle do not intersect with those of the second rectangle, we are certain the two rectangles do not intersect.

Equivalently, if the set of X-values for the rectangles intersect and the set of Y-values for the rectangles intersect, then all points with those X- and Y-values are common to the two rectangles, so there is a nonempty intersection.

```
struct Rectangle {
  int x, y, width, height;
};

Rectangle IntersectRectangle(const Rectangle& R1, const Rectangle& R2) {
  if (IsIntersect(R1, R2)) {
    return {max(R1.x, R2.x), max(R1.y, R2.y),
            min(R1.x + R1.width, R2.x + R2.width) - max(R1.x, R2.x),
            min(R1.y + R1.height, R2.y + R2.height) - max(R1.y, R2.y)};
  }
  return {0, 0, -1, -1};  // No intersection.
}

bool IsIntersect(const Rectangle& R1, const Rectangle& R2) {
  return R1.x <= R2.x + R2.width && R1.x + R1.width >= R2.x &&
         R1.y <= R2.y + R2.height && R1.y + R1.height >= R2.y;
}
```

The time complexity is $O(1)$, since the number of operations is constant.

Variant: Given four points in the plane, how would you check if they are the vertices of a rectangle?

Variant: How would you check if two rectangles, not necessarily aligned with the X and Y axes, intersect?