Solutions to Exercises in Chapter 2

2.1 In the test case $b = 2$, $n = 11$, the simple power algorithm performs 11 multiplications, while the smart power algorithm performs 7 multiplications.

2.2 Algorithm 1.1 has time complexity $O(1)$.

2.4 The `matrixAdd` method performs n^2 additions. Its time complexity is $O(n^2)$.

The `matrixMult` method performs n^3 additions and n^3 multiplications. Its time complexity is $O(n^3)$.

2.5 To analyze Algorithm 2.16, count the number of characters required to render *i* to base *r*. If *i* is positive, the number of characters is $\log_r i + 1$. If *i* is negative, the number of characters is $\log_r(\text{abs}(i)) + 2$ (the extra character being '–'). The time complexity is $O(\log(\text{abs}(i)))$.

2.6 To print a given integer *i* to base *r*:

1. Set *s* to the empty string "".
2. Set *p* to the absolute value of *i*.
3. Repeat the following until $p = 0$:
   3.1. Let *d* be the digit corresponding to (*p* modulo *r*).
   3.2. Prepend *d* to *s*.
   3.3. Divide *p* by *r*.
4. If $i < 0$, prepend '–' to *s*.
5. Print *s*.
6. Terminate.

This algorithm's time complexity is $O(\log(\text{abs}(i)))$.

2.7 To find the GCD of positive integers *m* and *n* (recursive version):

1. Let *p* be the greater and *q* the lesser of *m* and *n*.
2. If *p* is a multiple of *q*:
   2.1. Terminate with answer *q*.
3. If *p* is not a multiple of *q*:
   3.1. Let *g* be the GCD of *q* and (*p* modulo *q*).
   3.2. Terminate with answer *g*.

2.8 Algorithm 2.21 performs *n* multiplications. Its time complexity is $O(n)$.

Method to calculate the factorial of `n` (recursive version):

```
static int factorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

To calculate the factorial of *n* (non-recursive version):

1. Set *f* to 1.
2. For $i = 1, \ldots, n$, repeat:
   2.1. Multiply *f* by *i*.
3. Terminate with answer *f*.

Method to calculate the factorial of `n` (non-recursive version):

```
static int factorial (int n) {
   int f = 1;
   for (int i = 1; i <= n; i++)
      f *= i;
   return f;
}
```

2.9 Let the Fibonacci function be *fib*(*n*). Tabulate the first few Fibonacci numbers, and the ratios of consecutive numbers:

n	0	1	2	3	4	5	6	7	8
fib(*n*)	1	1	2	3	5	8	13	21	34
fib(*n*)/*fib*(*n*–1)		1.00	2.00	1.50	1.67	1.60	1.63	1.62	1.62

Thus we can see that $fib(n) \approx cb^n$, where $b \approx 1.62$ and $c \approx 0.72$.

Suppose that Algorithm 2.22 performs *adds*(*n*) additions. It is easy to see that $adds(n) = fib(n) - 1 \approx cb^n - 1$. The algorithm's time complexity is therefore $O(b^n)$.

To calculate the Fibonacci number of *n* (non-recursive version):

1. If $n \leq 1$:
   1.1. Terminate with answer 1.
2. If $n > 1$:
   2.1. Set *oldfib* to 1, and set *fib* to 1.
   2.2. For *i* = 2, …, *n*, repeat:
      2.2.1. Set *oldfib* and *fib* to *fib* and *oldfib*+*fib*, respectively.
   2.3. Terminate with answer *fib*.

Method to calculate the Fibonacci number of `n` (recursive version):

```
static int fibonacci (int n) {
   if (n <= 1)
      return 1;
   else
      return fibonacci(n-1) + fibonacci(n-2);
}
```

Method to calculate the Fibonacci number of `n` (non-recursive version):

```
static int fibonacci (int n) {
   if (n <= 1)
      return 1;
   else {
      int oldfib = 1, fib = 1;
      for (int i = 2; i <= n; i++) {
         int newfib = oldfib + fib;
         oldfib = fib;  fib = newfib;
      }
      return fib;
}
```

2.10 Outline of program:

```
static void moveTower (int n,
               int source, int dest) {
  if (n == 1)
    moveDisk(source, dest);
  else {
    int spare = 6 - source - dest;
    moveTower(n-1, source, spare);
    moveDisk(source, dest);
    moveTower(n-1, spare, dest);
  }
}

static void moveDisk (int source, int dest) {
  System.out.println("Move disk from " + source
       + " to " + dest);
}
```

To make the program count the moves, modify `moveTower` to return the required number of moves, as follows:

```
static int moveTower (int n,
               int source, int dest) {
  if (n == 1) {
    moveDisk(source, dest);
    return 1;
  } else {
    int spare = 6 - source - dest;
    int moves1 = moveTower(n-1, source, spare);
    moveDisk(source, dest);
    int moves2 = moveTower(n-1, spare, dest);
    return moves1 + 1 + moves2;
  }
}
```