# Solutions to Exercises 2

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

**2B** The midpoint algorithm has time complexity $O(1)$.

**2C** omitted

**2D** 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 $O(n^3)$.

**2E** To yield a (positive) integer *n* to base *2*:

1. Set *s* to the empty string "".
2. Set p to n
3. Repeat the following until *p* = 0:
   3.1. Let *d* be the digit corresponding to (*p* modulo *2*).
   3.2. Insert *d* at the front of *s*.
   3.3. Divide *p* by *2*.
4. Terminate yielding *s*.

Every time we repeat step 3 we perform 3 constant time operations. We repeat step 3 log n times. So (ignoring constants) this algorithm's time complexity is $O(\log(n))$.

**2F** To yield a (positive) integer *n* to base *2*:

1. Set *s* to the empty string "".
2. If n<2
   2.1. insert n at the front of s
   2.2. Terminate yielding s
3. else
   3.1. Let *d* be the digit corresponding to (*p* modulo *2*).
   3.2. Terminate yielding binary((n-d)/2) + d

By a similar argument to 2E, this algorithm's time complexity is $O(\log(n))$.

**2G** omitted

**2H** The factorial algorithm (recursive version) 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, …, *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;
}
```

**2J** 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;
  }
}
```