

Exercise 1

The integer division algorithm in school method is implemented in *App.1*.

The core code for integer division is as follows.

```
carry = 0
for i in range(len_lhs - len_rhs, -1, -1): #  $O(n - k)$ 
    ok = True
    if carry == 0:
        for j in range(len_rhs - 1, -1, -1): #  $O(k)$ 
            if res_rem.b[i + j] > rhs.b[j]:
                ok = True
                break
            elif res_rem.b[i + j] < rhs.b[j]:
                ok = False
                break

    if ok:
        res_div.b[i] = 1
        for j in range(0, len_rhs): #  $O(k)$ 
            res_rem.b[i + j] -= rhs.b[j]
            if res_rem.b[i + j] < 0:
                res_rem.b[i + j] += 2
                res_rem.b[i + j + 1] -= 1
    else:
        res_div.b[i] = 0
    carry = res_rem.b[i + len_rhs - 1]
```

The outer loop runs $n - k$ times, and the inner loop runs k times. Inside the loop body are bit operations. So the total algorithm runs in $O(k(n - k))$ operations.

Exercise 2

```
def euclid(a, b):
    while b > 0:
        r = a % b # so  $a = bu + r$ 
        if r == 0:
            return b
        s = b % r # so  $b = rv + s$ 
        a = r
        b = s
    return a
```

It is obvious that $r \leq \lfloor \frac{a}{2} \rfloor$ and $s \leq \lfloor \frac{b}{2} \rfloor$, so every cycle a and b become at most half of the previous value they have.

Suppose $a \geq b$, and let n and m denote the number of bits of a and b , we have $n \geq m$.

Since the integer division makes $O(m(n - m))$ bit operations.

The $\text{gcd}(a, b)$ makes

$$O(m(n - m)) + \frac{1}{2^2}m(n - m) + \frac{1}{2^4}m(n - m) + \dots = O(m(n - m)) = O(n^2)$$

operations.

Exercise 3

```
def combination_rec(n, k):
    if k < 0 or k > n:
        return 0
    elif n == 0:
        return 1

    return combination_rec(n - 1, k) + combination_rec(n - 1, k - 1)

print(combination_rec(7, 3))
```

10

The algorithm recurses $O(nC_n^k)$ times, and the time complexity of bitwise addition is $O(\log_2 2^n) = O(n)$.

So the running time of the algorithm is $O(n^2C_n^k)$.

It is not an efficient algorithm, because the time complexity of the algorithm is exponential.

Exercise 4

The dynamic programming algorithm is implemented as follows.

```
def combination_dp(n, k):
    if k < 0 or k > n:
        return 0
    elif n == 0:
        return 1

    """
    C = [0] * (k + 1)
    for i in range(n + 1):
        C[0] = 1
        for j in range(min(i, k), 0, -1):
            C[j] = C[j] + C[j - 1]
    return C[k]
```

```

"""
C = [None] * (n + 1)
for i in range(n + 1):
    C[i] = [0] * (k + 1)
    C[i][0] = 1
    for j in range(1, min(i + 1, k + 1)):
        C[i][j] = C[i - 1][j] + C[i - 1][j - 1]
return C[n][k]

print(combination_dp(5, 3))

```

10

The outer loop runs n times, and the inner loop runs k times. And the time complexity of bitwise addition is $O(\log_2 2^n) = O(n)$. So the running time of the program is $O(n^2k)$.

It is an efficient algorithm, because the time complexity of the algorithm is polynomial.

Exercise 5

The dynamic programming algorithm is implemented as follows.

```

def combination_dp_mod_2(n, k):
    if k < 0 or k > n:
        return 0
    elif n == 0:
        return 1

    """
    C = [0] * (k + 1)
    for i in range(n + 1):
        C[0] = 1
        for j in range(min(i, k), 0, -1):
            C[j] = (C[j] + C[j - 1]) % 2
    return C[k]
    """

    C = [None] * (n + 1)
    for i in range(n + 1):
        C[i] = [0] * (k + 1)
        C[i][0] = 1
        for j in range(1, min(i + 1, k + 1)):
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % 2
    return C[n][k]

print(combination_dp_mod_2(5, 3))

```

10

The outer loop runs n times, and the inner loop runs k times. And the time complexity of bitwise addition modulo 2 is $O(1)$. So the running time of the program is $O(nk)$.

It is an efficient algorithm, because the time complexity of the algorithm is polynomial.

Exercise 6

A lasso cannot happen.

If we had some $1 \leq i < j$ such that $F'_i = F'_j$ and $F'_{i+1} = F'_{j+1}$, we would have $F'_{i-1} \equiv F'_{i+1} - F'_i \equiv F'_{j+1} - F'_j \equiv F'_{j-1} \pmod{k}$ and $F'_i = F'_j$, so $i-1$ also satisfies the condition. Therefore, the smallest i must be 0, and form a circle.

Here's the code for finding a fibonacci period:

```
def fibonacci_period_mod_k(k):
    period = 0
    f0, f1 = 0, 1 % k

    while True:
        period += 1
        f0, f1 = f1, (f0 + f1) % k

        if (f0, f1) == (0, 1 % k):
            break

    return period

print(f"period = {fibonacci_period_mod_k(3)}")
```

```
period = 8
```

Appendix

Appendix 1. Euclidean algorithm

```
def bit_len(n):
    """
    Return bit length of n.
    Time complexity:  $O(\log n)$ 
    """
    result = 0
    while n > 0:
        n >>= 1
        result += 1

    return result


class Binary:
    def __init__(self, value=None):
        if isinstance(value, int):
            n_bits = bit_len(value)
            self.b = [0] * n_bits
            for i in range(n_bits):
                self.b[i] = value % 2
                value //= 2
        elif isinstance(value, list):
            self.b = value

    def __str__(self):
        integer = 0
        for num in self.b[::-1]:
            integer = integer << 1 | num
        return str(integer)

    def copy(self):
        result = Binary()
        result.b = self.b.copy()
        return result

    def len(self):
        return len(self.b)

    def add(self, rhs):
        if not isinstance(rhs, Binary):
            raise TypeError("rhs is not Binary")

        len_lhs, len_rhs = self.len(), rhs.len()

        result = []

        carry = 0
```

```

for i in range(max(len_lhs, len_rhs)):
    s = 0
    if i < len_lhs:
        s += self.b[i]
    if i < len_rhs:
        s += rhs.b[i]
    result.append(s + carry)

    if result[-1] < 2:
        carry = 0
    else:
        carry = 1
        result[-1] -= 2

if carry > 0:
    result.append(carry)

return Binary(result)

def _div_rem(self, rhs):
    if not isinstance(rhs, Binary):
        raise TypeError("rhs is not Binary")

    len_lhs, len_rhs = self.len(), rhs.len()

    res_div, res_rem = Binary(0), self.copy()

    res_div.b = [0] * (len_lhs - len_rhs + 1)

    carry = 0
    for i in range(len_lhs - len_rhs, -1, -1): #  $O(n - k)$ 
        ok = True
        if carry == 0:
            for j in range(len_rhs - 1, -1, -1): #  $O(k)$ 
                if res_rem.b[i + j] > rhs.b[j]:
                    ok = True
                    break
                elif res_rem.b[i + j] < rhs.b[j]:
                    ok = False
                    break

            if ok:
                res_div.b[i] = 1
                for j in range(0, len_rhs): #  $O(k)$ 
                    res_rem.b[i + j] -= rhs.b[j]
                    if res_rem.b[i + j] < 0:
                        res_rem.b[i + j] += 2
                        res_rem.b[i + j + 1] -= 1
            else:
                res_div.b[i] = 0
                carry = res_rem.b[i + len_rhs - 1]

    while res_div.b and res_div.b[-1] == 0:

```

```

        res_div.b.pop()

    while res_rem.b and res_rem.b[-1] == 0:
        res_rem.b.pop()

    return res_div, res_rem

def div(self, rhs):
    if not isinstance(rhs, Binary):
        raise TypeError("rhs is not Binary")

    return self._div_rem(rhs)[0]

def rem(self, rhs):
    if not isinstance(rhs, Binary):
        raise TypeError("rhs is not Binary")

    return self._div_rem(rhs)[1]

def greaterthan(self, rhs):
    if not isinstance(rhs, Binary):
        raise TypeError("rhs is not Binary")

    len_lhs, len_rhs = self.len(), rhs.len()

    if len_lhs != len_rhs:
        return len_lhs > len_rhs

    for i in range(len_rhs - 1, -1, -1):
        if self.b[i] != rhs.b[i]:
            return self.b[i] > rhs.b[i]

    return False

def equals(self, rhs):
    if not isinstance(rhs, Binary):
        raise TypeError("rhs is not Binary")

    len_lhs, len_rhs = self.len(), rhs.len()

    if len_lhs != len_rhs:
        return False

    for i in range(len_rhs):
        if self.b[i] != rhs.b[i]:
            return False

    return True

def euclid(a, b):
    a, b = Binary(a), Binary(b)
    while b.greaterthan(Binary(0)):

```

```
    r = a.rem(b) # so a = bu + r
    if r.equals(Binary(0)):
        return b
    s = b.rem(r) # so b = rv + s
    a = r
    b = s
    return a

print(euclid(10, 3))
```

4