Homework 1

Obliviate

March 2021

1 Solution:

Assume that $n \geq k$.

With school method, the number of substraction is at most $O(n-k)$.

And every substraction costs $O(k)$ times because two operators of the substraction both have $O(k)$ bits.

It is noticed that, we don't need to consider the trailing bits when doing one substraction, because they are useless in this step. What we just need to do is to take one bit from the trailing bits before every substraction step. So the complexity is $O(k)$ for each substraction but not $O(n)$.

As the result, the total complexity is $O(k(n-k))$.

2 Solution:

With the result in exercise 1, we have known that the complexity of the school method division is $O(k(n-k))$ but not $O(n^2)$.

Meanwhile, mod operation is similar to division operation. After we do substraction to divide a by b, we also get the remainder. Therefore, the complexity of mod is also $O(k(n-k))$.

And we have $O(k(n-k)) \leq O(n(n-k))$, so we estimate the complexity of mod to be $O(n(n-k))$.

Assuming the number of bits of a and b when doing gcd is the sequence $(n_0, n_1), (n_1, n_2), (n_2, n_3), ...$, so in every step, the complexity is $O(n_0(n_0-n_1)), O(n_1(n_1-n_2)), O(n_2(n_2-n_3)), ...$, i.e. $O(nm)$ where n is the number of bits of the larger number and m is the difference of the number of bits between a and b.

So the total complexity is $O(n_0(n_0-n_1)) + O(n_1(n_1-n_2)) + O(n_2(n_2-n_3)) + ... \leq O(n_0(n_0-n_1)) + O(n_0(n_1-n_2)) + O(n_0(n_2-n_3)) + ... \leq O(n_0^2)$.

Informally, when one division costs $O(nm)$ time, then the number of bits decreases by m after the operation. And we have $O(n)$ bits in total, so the total complexity is $O(n^2)$.

Q.E.D.

3 Solution:

Python code:

```
def binomial_rec (n, k) :
    if (k < 0 or k > n) :
        return 0
```

```
    if (n == 0 or k == 0) :
        return 1
    return binomial_rec(n - 1, k - 1) + binomial_rec(n - 1, k)
```

Let $f(n,k)$ donate the number of times the function is called when we calculate $C(n,k)$.
When $1 \leq k \leq n$, we have:

$$f(n,k) = f(n-1,k-1) + f(n-1,k) + 1$$

$$\Big(f(n,k)+1\Big) = \Big(f(n-1,k-1)+1\Big) + \Big(f(n-1,k)+1\Big)$$

Let $g(n,k) = f(n,k) + 1$.

$$g(n,k) = g(n-1,k-1) + g(n-1,k)$$

Boundary condition is $g(n,k) = 2$ when $k > n$ or $k < 1$

$$g(n,k) = 2C(n+1,k)$$

$$f(n,k) = g(n,k) - 1 = 2C(n+1,k) - 1$$

Because $\sum_{i=0}^{n} C(n,i) = 2^n$, $C(n,k) \leq 2^n$, so the bit length of $C(n,k)$ is no more than n. Therefore, we can assume that the addition of $C(n-1,k-1)$ and $C(n-1,k)$ costs $O(n)$. As the result, the total complexity of the recursive algorithm computing $C(n,k)$ is $O(C(n,k) \cdot n)$.
Consider that the output size is $O(n)$. Since $C(n,k) \cdot n$ isn't a polynomial of n, this algorithm is not efficient.

4 Solution:
Python code:

```
def C(n, k):
    f = [0] * (n + 1) # To save the space, we can reuse f.
    f[0] = 1
    for i in range(1, n + 1):
        j = min(i, k)
        while j >= 1:
            f[j] += f[j - 1]
            j = j - 1
    return f[k]
```

Just the same as Exercise 3, we assume that the addition of $C(i-1,j-1)$ and $C(i-1,j)$ costs $O(i)$. So the complexity is $O(k \cdot \sum_{i=1}^{n} i) = O(n^2 k)$.
Consider that the output size is $O(n)$. Since $n^2 k$ is a polynomial of n, this algorithm is efficient.

5 Solution:
Python code:

```
def C_module2 (n, k):
    f = [0] * (n + 1)
    f[0] = 1
    for i in range(1, n + 1):
        j = min(i, k)
        while j >= 1:
            f[j] ^= f[j - 1]
            j = j - 1
    return f[k]
```

The complexity is at least $O(nk)$ since $n \geq k$ and we have to fill $O(nk)$ blanks in the two-dimension-array.

It is not polynomial in $O(\log n)$, so it is not efficient.

However, we have Lucas theorem so we have the simple equation:

$$\binom{n}{k} \bmod 2 = [\text{k is subset of n in binary expression}]$$

This costs exactly $O(\log n)$ time, which is obviously efficient.

6 Solution:

Python code:

```
def mult(x, y, k):
    return [[(x[0][0] * y[0][0] + x[0][1] * y[1][0]) % k, (x[0][0] * y[0][1] + x[0][1] * y
        [1][1]) % k],
    [(x[1][0] * y[0][0] + x[1][1] * y[1][0]) % k, (x[1][0] * y[0][1] + x[1][1] * y[1][1]) % k]]

def quick_power(x, y, k):
    ret = [[1, 0], [0, 1]]
    while y:
        if y & 1 == 1: ret = mult(ret, x, k)
        x = mult(x, x, k)
        y >>= 1
    return ret

def fib(n):
    return mult(([[1, 1], [0, 0]]), quick_power([[1, 1], [1, 0]], n - 1, k), k)[0][0]
```

Since we fixed k as a constant number, the matrix multiplication and addition and other basic operations cost $O(1)$ time now because every number is less than k due to the modulus operation.

Obviously, when $n \geq 2$, we have:

$$(F_n, F_{n-1}) = (F_{n-1}, F_{n-2}) \begin{pmatrix} 1,1 \\ 1,0 \end{pmatrix} = (F_1, F_0) \begin{pmatrix} 1,1 \\ 1,0 \end{pmatrix}^{n-1}$$

So the basic operations like matrix multiplication cost $O(1)$ and total complexity of matrix multiplication method is exactly $O(\log n)$.