Algorithm HW5 documentation

執行環境：Windows 10

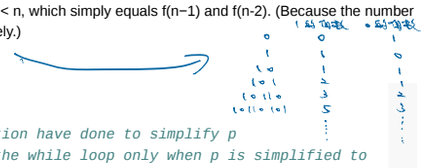
程式語言 C++ 17

v_1

the most naive solution, concat every element before n to obtain F(n), but this fails because string can store at max around 1e5. The Fibonacci Words string is going to exceed when n = 27 or 28. Thus, other ways may needed.

v_2

F(n − 1) to F(n) can be seen as a replacement like follow: every 1 becomes replaced by 10, and every 0 becomes replaced by a 1. In order to compute the number of occurrences of p in F(n), we can apply these rules backwards to obtain a shorter string p′, and the task is now to find the number of occurrences of p′ in F(n − 1) (if p ended in a 1 there are two possibilities for p′ since we don't know whether that last 1 came from a 1 or a 0 in F(n − 1)). Recursing this process, we are ultimately looking for the number of occurences of either 1 or 0 in F(n′) for some n′ < n, which simply equals f(n−1) and f(n-2). (Because the number of occurences or 1 and in F(n) is f(n - 1) and f(n -2) respectively.)



```
int Fibonacci_Words(int n, string& p)
{
    int count = 0; //count how many iteration have done to simplify p
    while(p != "1" && p != "0") //escape the while loop only when p is simplified to
 "1" or "0"
    {
        size_t pos = 0;
        while (pos != p.size()) //simplify p from the beginning of the string
        {
            if(p[pos] == '1' && p[pos + 1] == '0')
            {
                p.erase (p.begin());
                p[pos] = '1';
                pos++;
                cout<< p<< endl;
                //cout<< pos<< endl;
            }
            else if(p[pos] == '1')
            {
                p[pos] = '0';
                pos++;
                cout<< p<< endl;
            }
        }
        if(pos == p.size()) //when reaching to the end of the string, start over.
        {   count++;
            continue;
        }
    }

    if(p == "1")
        return 1;
    else
        return 2;
}
```

However, this method fails when reaching those p which can not translate to "1" or "0"(e.g. "01"). Also, through the backward replacement, some occurences of p in the given string may not be counted in because this method actually break the given string into multiple parts of "10" and "1" and translate backward to simplify.

v_3

denote the occurences of p in F(i) as G(i)
Then G(i)=G(i−1)+G(i−2)+the occurences of p in the middle of F(i−1) and F(i−2)
denote the two shortest Fibonacci Words string that longer than p as a,b. Then the the middle of F(i−1) and F(i−2) would be only two possibilities, which is a+b and b+a. Furthermore, a+b and b+a appears either at the front or the tail of F(i).
Thus, by using the KMP algorithm to find the occurences of a,b,a+b, and b+a in F(i), we can solve G(n) inductively.

$$G(n) = \begin{cases} 0 & (n \le p\text{的長度}) \\ G(n-1) + G(n-2) + \text{Match}(b+a) & (n\text{是b之後的第奇數個串}) \\ G(n-1) + G(n-2) + \text{Match}(a+b) & (n\text{是b之后的第偶數個串}) \end{cases}$$

The KMP matching algorithm

The idea behind KMP is "recovering" from mismatches by using partial match information to skip over positions where the needle can't possibly be found and avoid redoing comparisons where we already know that a prefix of the needle matches the haystack.

If the first k characters of the needle are known to match characters of the haystack starting at position i , then that might imply that there are some positions after i where the needle cannot possibly match in its entirety. For example, if the needle is ABC and the prefix AB is found in the haystack starting at position i , then the needle can't possibly be found at position i+1 , since the (i+1) character is B, which is not A. Likewise, if the needle is AABAAC and the prefix AABAA is found at position i of the haystack, then

the needle can't be found at position i+1 since the needle doesn't start with ABAA
the needle can't be found at position i+2 since the needle doesn't start with BAA
the needle might be found at position i+3 since the needle does start with AA.
In order to find the first position after i where the needle might be found, given that a prefix P of length k of the needle matches the haystack, we need to find the largest m<k such that the prefix of length m of P equals the suffix of length m of P. For example, if the needle is AABAAC as above, and k=5 , then m=2 because the prefix of length 2 of AABAA is also a suffix of AABAA. Having found such m , we know that the first position after i where the needle might possibly occur is i+k−m .

So we go along, comparing characters one by one between the needle and the haystack starting at position i of the haystack, and as soon as we get a mismatch, we let k be the number of characters matched, and skip ahead to i+k−m . We don't re-examine any of the k matched characters in the haystack, since we already know that the last m of those match the first m characters of the needle. Instead we continue at the mismatched character, comparing it to character m+1 of the needle, and proceed until another mismatch occurs, and so on. If the needle is successfully found, it's treated the same way as a mismatch at the (L+1) th character where L is the length of the needle.