CS225 HW5

EXERCISE 1. For the KMP algorithm design an algorithm to compute the sequence *NEXT* in time in $O(m)$, where m is the length of the pattern sequence.

HJD

D-7 ⇒ E-3 ?

total points: 10

For elementary KMP algorithm, the NEXT array means if the current char in the long string is not fit in the pattern string, then swich to a particular string.

So the calculation will be two pointers for the pattern string. Because of the NEXT array's characteristic, we can figure out that, the NEXT value of the first node and the second node is always 0 and 1.

Shift pointer to the first node, the other node to the third node, then we can recursively execute the precalculation for KMP algorithm, which is to define NEXT values for the pattern string.

Mainly there is two situations:

If the first pointer value is the same as the second pointer value, first set the NEXT value of second char to be first pointer value. *Then to both two pointers should plus one as to define the next node NEXT value*, which means to recursively execute the NEXT calculation.

The correctness of the calculation is because that if the two pointer are the same, we can see that the **path pointer 1 past can be the same as the pointer 2 past**, so if failed, we can go back to the forward values to check if the string fits to the pattern. This method reduced the switch time, improving the efficiency.

If the first pointer value is not the same as the second pointer value, also, first set the NEXT value of second char to be first pointer value. **then set first pointer back to the first node position**, because the pattern fitting failed, **we should go back to check the first node as the current situation cannoot be fit.**

The C++ code to calculate the elementary NEXT value shows as follows:

```
#include <iostream>
#include <stdlib.h>
template<class T> class NEXT
{
public:
    NEXT(int size = sizeof(p));
    T getitem(int index);
    T setitem(int index, T value);
    void getNEXT(){
        int i = 1;
        int j = 3;
        setitem(1,0);
        setitem(2,1);
        while(j <= numitems){
            //first situation.
            if (p[j-1]==p[i-1]){
                setitem(j,i);
```

```
                i++;
                j++;
            }else{
            //second situation
                setitem(j,i);
                i = 1;
                j++;
            }
        }
    };
private:
    int numitems;
    char *p;
    T* reparray;
}
```

And for **Klaus-Dieter's Optimzed KMP algorithm**, Prof. KD optimzed the algorithm to be more efficient. considering the situation that:

if the former path is the same as the latter path for the pattern, in another words, the pattern like: ab~~ab (~ is not a particular char but to be any other.). :

Then in the unoptimzed algorithm, *when checking the last B failed, we will go to check the first B, which is meaningless, because they are the same.* **So, KD let the NEXT[second pointer] = NEXT[first pointer] to reduce the complexity of useless switching.**

The optimzed algorithm C++ implementation shows as follows:

```
void KMPlist<T>::create_next(int *next){
    next[1] = 0;
    next[2] = 1;
    int i = 2;
    int j = 1;
    //reprarray[] is the string value.
    //and next[] is the next value.
    while (i < numitems){
        if (j == 0 || reprarray[i - 1] == reprarray[j - 1]){
            i++;
            j++;
            //here means if they are the same,
            //then the next(j) will be next(i) rather than i.
            if (reprarray[i - 1] != reprarray[j - 1]){
                next[i] = j;
            }else{
                next[i] = next[j];
            }
        //the other condition is the same
        //as elementary KMP.
        }else{
```

```
            j = next[j];
        }
    }
}
```

EXERCISE 2. Modify the KMP algorithm such that it will find all occurrences of a pattern sequence P in target sequence S in time in $O(n)$, where n is the length of S.

HJD

total points: 8

As we know that: the KMP algorithm is to loop through the S to find out one occurrence of pattern P. if so, return 1.

So let's just modify the loop stucturre to be:

If program find a pattern, record the position and continue to shift until the second pointer reached the end.

If there is no pattern, return zero.

The complexity should be definitely O(n) as we loop through the string S.

CYT&WJ

Ex3. Represent polynomial $P(x) = a_0 + a_1 x + a_2 x^2 + \dots a_d x^d$ by tuple $[a_d, a_{d-1}, \dots, a_1, a_0]$

(i) Obviously $n_1, n_2, \dots, n_d$ are roots of $P(x)$

Thus, $P(x) = a_d \prod_{k=1}^{d} (x - n_k)$ by properties of polynomial's roots.

For each n_k, $1 \le k \le n$, make polynomial $E_k(x) = [1, -a_k]$.

Start from $E_1(x)$, let $P_1(x) = E_1(x)$

Then, for $E_k(x)$, $2 \le k \le d$, execute $P_k(x) = P_{k-1}(x) \cdot E_k(x)$ which needs $O(k-1)$

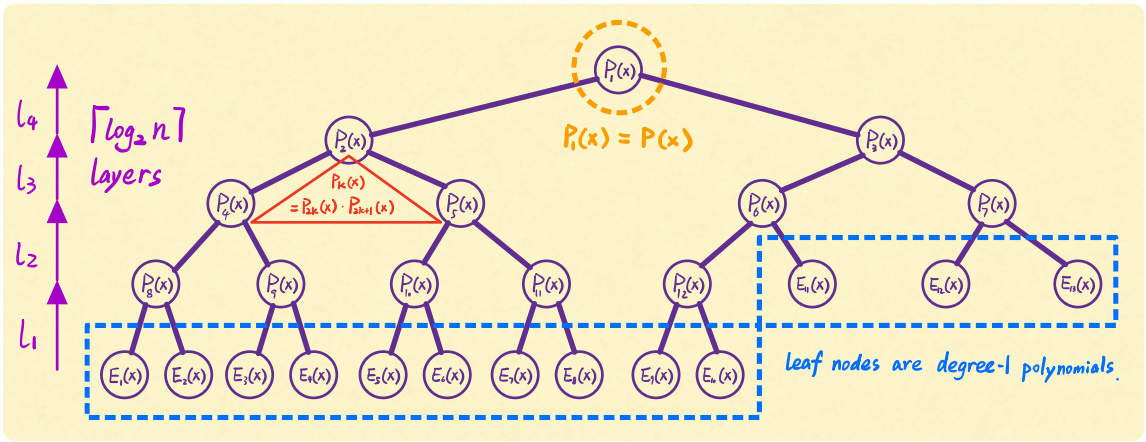
Thus the total complexity is $\sum_{k=2}^{d} (k-1) = \frac{d(d-1)}{2} \in O(d^2)$

(ii) Like in (i), make $E_k(x) = [1, -a_k]$ for each x_k, $1 \le k \le n$

Build a binary tree, with each leaf node one of E_k.

For every other node, its polynomial is the product of two child nodes.

Therefore the root node is the product of all $E_k(x)$, which is $P(x)$.



Calculate from the bottom of the tree.

Each layer requires complexity $\lceil \frac{n}{2^i} \rceil \cdot O(2^{i-1} \log_2(2^{i-1})) \le n(i-1)$

Total complexity $\sum_{i=1}^{\lceil \log_2 n \rceil} n(i-1) = n \cdot \frac{(\lceil \log_2 n \rceil - 1)\lceil \log_2 n \rceil}{2} < n(\log_2 n)^2 \in O(n \cdot \log^2 n)$