Data Structures and Programming spring 2019

Prog #1

Student: 謝昌諭

ID : b06602052

All below analization of complexity is use N as the length of sequence of requests and M as the length of L (list consisting of M distinct elements).

Implementation

I implement all algorithm by a for loop and a while loop and a special function F which is different from other algorithm. (Pseudo code is as below)

```
…….                    // may have some code to handle some operation to prepare lists
total_count = 0        // total numbers of comparision
for( from 1st to N-th term in sequence of requests ){
    counter = 0        //current index of L. After end, counter+1 means comparision this time
    while( True ){
        if L[counter] is equal to this term
            F(L,L_special,counter)     //rearrange L and L_special(frequency ..etc.)
            /*the code of F() will explain in next part*/
            total_count = (counter + 1)
            break
        else                   //not equal
            counter++          //do next while
            continue
    } //end while
} //end for
```

From above, we can know it cost O(N*M) in if-else, for the worst case is that all while loop running all element in L. F() cost N*O(F), for only last comparision do F().

They all need at least two integers (total_count and counter) So, O(1)

Optimal

1.Time complexity

Calculate frequency of each element in L first. → O(N)

For loop and linear search to sort L by frequency first. → O(M^2)

No F() → 0

Total time complexity : O($M^2 + N * M + N$).

2.Space complexity (ignoring the input array)

List to calculate frequency. → O(M)

Total space complexity : O(M)

3.Concept

Sorting L with the frequencies of occurrences, for it is off-line. The more common term will get more early position, so overally it can decrease compare times. The efficiency will increase as the ratio of numbers is big.

MTF (move-to-front)

1.Time complexity

F() : Do the deletion and insertion. → O(1) (Pseudo code is as below)

```
move_to_front(L,index){
    If(index == 0)              // if it is in first term, no any move
        return None
    temp = L.pop(index)    // get the term and delete it
    L.insert(0,temp)         // insert it in to front
    /*when analizing complexity. View list as Linked list*/
}
```

Total time complexity : O($N * M + \mathrm{N}$).

2.Space complexity (ignoring the input array)

No other space needed

Total space complexity : O(1).

3.Concept

According to the spatial locality and temporal locality, we move accessed term into front, for this number is likely accessed in the future. This way may give a not very often-accessed term biggest priority. It may cause the over rapidly of change. The most common term might be in back, because a short interval which didn't have this term.

BIT

1.Time complexity

Initialize bits for L. → O(M)

F() : Do MTF, if bit is 0 to 1. → O(1) (Pseudo code is same as MTF)

Total time complexity : O($\mathrm{N} * \mathrm{M} + \mathrm{M}$)

2.Space complexity (ignoring the input array)

Bit list to represent elements in L. → O(M)

Total space complexity : O(M)

3.Concept

Same as MTF, but doing move-to-front only when bit change is 0→1. In normal

MTF, although the accessed term is moved into front, it also make another number more behind. Compared with MTF, BIT won't change so often but need more memory to save bits.

Transpose

1.Time complexity

transpose() : Transpose term cost constant time. → O(1)

```
transpose(L,index){
    if(index == 0)            // if it is in first term, no any move
        return None
    temp = L[index - 1]    // change with the former term only one times → constant time
    L[index - 1] = L[index]
    L[index] = temp
}
```

Total time complexity : O($N * M + N$)

2.Space complexity (ignoring the input array)

No other space needed

Total space complexity : O(1)

3.Concept

After end of comparision, only swap one unit in list. This way can increase the priority of terms. But, it is a relative slow transform, for we need to access many times to move it to earlier position.

Frequency count

1.Time complexity

Increase the frequency of accessed term → O(1)

F() : compare with former term. Changed place if larger. → O(M)

```
rearrange(number_L,frequency,index){
    if(index == 0)                          // if it is in first term, no any move
        return None
    for(i from index-1 to 0){               // start from previous element of changed one
        if (frequency[i] < frequency[i + 1]){    //if the order is not legal, then replace it
            /*bothlist list need to be rearrange*/
            temp = frequency[i]
            frequency[i] = frequency[i + 1]
            frequency[i + 1] = temp
            temp = number_L[i]
            number_L[i] = number_L[i + 1]
```

```
                    number_L[i + 1] = temp
                } // end if
            }// end for
        } // end function
```

Total space complexity : O($N * M$)

2.Space complexity (ignoring the input array)

List to represent the frequency of elements in L → O(M)

Total space complexity : O(M)

3.Concept

Use realistic frequencies t oarrange list. This way is more close to actual. When the data increase big and big, it will go accurate and accurate. But,compared with other algorithm, this way need additional memory to save frequencies.

Bonus – self-design algorithm

1.improve method and concept of design

Build M lists to save the frquency count for each numbers. When doing the comparision, we use the corresponding list. That is if previous number is 0, we use L0 to doing comparision. Soppose L0 is [8,0,9,4,5,6,1,3,2,7], this means number most likely appear after 0 is 8.

The concept of this design is that the input data might have some tendency. For example, if the input data is all phone number in Taipei, then the L0[0] is very likely to be 2, because the area number of Taipei is 02. Take another example, the input data is paper about electron on a Journal and we have a L consisting of 0~9 and a~z and space. Le's earlier is very likely l or c, for "electron" is commonly appear in this data.

By above way, we can improve the efficiency of comparision when data have some rule or on the condition which every term will affect each other.

2.Implementation

First, we build 10 pairs of lists like below. (the Li is order of posibility of next term and timesi is corresponding frequency)

```
L0 = [0,1,2,3,4,5,6,7,8,9]
times0 = [0,0,0,0,0,0,0,0,0,0]
L1 = [0,1,2,3,4,5,6,7,8,9]
times1 = [0,0,0,0,0,0,0,0,0,0]
```

Second, we need a update function to resort L and times after finish current comparision. This function swap number and frquency in L and times. So, we can just copy the rearrange() in FC.

When program start running, we randomly choose a list as beginning. After end

of coomparision of first term, we make the next list be L of first term. So, next comparision use the list of previous number.

3.Time complexity

Increase the frequency of accessed term → O(1)

F() : compare with former term. Changed place if larger. → O(M)

Assign list of current number for next operation. → O(1)

Total space complexity : O($N * M$)

4.space complexity

Use 2M list with M terms to save frequency and order. → O(M^2)

Total space complexity : O(M^2)

5.running result

```
bonus: [291, 601, 855, 1097, 1339, 1625, 1905, 2183, 2391, 2681, 2982, 3214, 3503, 3782, 4036, 4349, 4626, 4904, 5171, 5425]
```

This result didn't show very well efficiency on input data of this project. Because the input is pi ,whuch is a unrulely sequence, this algorithm can't do very well than others. From there, we can know this algorithm is not very general use.

Reference

Wikipedia:
https://en.wikipedia.org/wiki/Self-organizing_list?fbclid=IwAR2WDE_Wt7O7takk_du5ukr_hCs1NsUjaBQXRKXqpPta3Rvki2cPenBLX2Q#Other_Methods

stackoverflow
https://stackoverflow.com/questions/32402794/self-organizing-lists-movo-to-front-list-rearrangement

codewars
https://www.codewars.com/kata/self-organizing-lists-move-to-front-transpose-frequency-count?fbclid=IwAR0FoIWaOBE4wj7QiRnzaLu3ukvxAVR6Bz5qssv4H5iYqBlTCByuCimCLgI