SHANGHAI JIAO TONG UNIVERSITY

Algorithm Analysis HW:4

Author: *519030910100 Huangji Wang F1903004*

Course: *Fall 2021, AI2615: Algorithm*
Date: *December 12, 2021*

Problem 1 (15 points)
Given two strings $A = a_1a_2a_3a_4...a_n$ and $B = b_1b_2b_3b_4...b_n$, how to find the longest common subsequence between A and B? In particular, we want to determine the largest k, where we can find two list of indices $i_1 < i_2 < i_3 < ... < i_k$ and $j_1 < j_2 < j_3 < ... < j_k$ with $a_{i1}a_{i2}...a_{ik} = b_{j1}b_{j2}...b_{jk}$. Design a DP algorithm for this task.

To solve the DP problem, I firstly give the sub-problem definition:
$f[i][j]$ defines the longest common subsequence length between $a[1,...,i]$ and $b[1,...,j]$.
The state transition equation can have three situations:

1. Initialization when we haven't start to solve the problem.

   $\forall i = 0$ or $j = 0,\ f[i][j] = 0$

2. Update the length if the last word of our $A' = a_1...a_i$ and $B' = b_1...b_i$ are the same.

   If $a_i == b_j$, then $f[i][j] = f[i-1][j-1] + 1$

3. If the last words are not the same, update the original state.

   Otherwise, $f[i][j] = \max(f[i-1][j], f[i][j-1], f[i][j])$

Analysis for time complexity: Obviously $O(n)$ for initialization and $O(n^2)$ for update operations. Total time complexity: $O(n^2)$

Correctness proof of the DP idea:

1. Base case: Check our initialization: $f[i][0] = f[0][j] = 0$.

2. Inductive hypothesis:

   Each $f[i][j]$ can be solved correctly by $f[i-1][j-1], f[i-1][j], f[i][j-1]$.

3. Induction: for the new $f[i][j]$, we can also obtain it correctly.

   Proof:

   There are two situations to be considered.

   (1) $a_i = b_j$

   Firstly, its impossible that $f[i][j] < f[i-1][j-1]$ because if the longest common subsequences between $A'[1,...,i-1]$ and $B'[1,...,j-1]$ are $a' = i_1i_2...i_k \in A'$

and $b' = j_1j_2...j_k \in B'$, then $a' \in A'[1,...,i-1] \in A'[1,...,i]$ and $b' = j_1j_2...j_k \in B'[1,...,j-1] \in B'[1,...,j]$, which implies that $f[i][j] \geq f[i-1][j-1]$.

Suppose that $f[i][j] > f[i-1][j-1]+1$. Then, we can find that there must exist some longest common subsequence that is larger than $f[i-1][j-1]$. However, each two characters can contribute at most 1 to the value, which yields a contradiction!

(2) $a_i \neq b_j$

Suppose that $f[i][j] > \max\{f[i-1][j], f[i][j-1]\}$. Thus, $f[i][j] > f[i-1][j]$. Then we can find that a_i should be in the longest common subsequence in $f[i][j]$. In the same way, we can find that b_j should be in the longest common subsequence in $f[i][j]$, too. Because they are both the last character in each substring, we can conclude that $a_i = b_j$, which yields a contradiction!

4. Conclusion: for each $f[i][j]$, we can obtain its value correctly by our methods.

Algorithm 1 Find the Longest Common Subsequence

Require: Strings $A = a_1a_2...a_n$ and $B = b_1b_2...b_n$. Size n.

Ensure: The length k of the longest common subsequence.

```
function LCS($StringA, StringB, n$)
    $i \leftarrow 0$
    for $i \leq n$ do
        $f[i][0] \leftarrow 0$
        $f[0][i] \leftarrow 0$
        $i \leftarrow i+1$
    end for
    $i \leftarrow 1$
    for $i \leq n$ do
        $j \leftarrow 1$
        for $j \leq n$ do
            if $a_i == b_j$ then
                $f[i][j] \leftarrow f[i-1][j-1]+1$
            else
                $f[i][j] \leftarrow \max\{f[i-1][j], f[i][j-1]\}$
            end if
            $j \leftarrow j+1$
        end for
        $i \leftarrow i+1$
    end for
    $result \leftarrow f[n][n]$
    return $result$
end function
```

Problem 2 (15 points)

Given two teams A and B, who has already played $i+j$ games, where A won i games and B won j games. We suppose that they both have 0.5 independent probability to win the upcoming games. The team that first win $n \geq \max\{i, j\}$ games will be the final winner. Design a DP algorithm to calculate the probability that A will be the winner.

Note:

To simplify the idea, I denote **a** as the number i while **b** as the number j because i,j are widely used in recursive numbers.

To solve the DP problem, I firstly give the sub-problem definition:

$f[i][j]$ defines the possibility when A wins i games and B wins j games.

The state transition equation can have three situations:

1. Initialization when we haven't start to solve the problem. $f[a][b] = 1$
2. Update the possibility. The situation that A wins i games and B wins j games means A wins the $i+j$-th game after $f[i-1][j]$ or A loses the $i+j$-th game after $f[i][j-1]$
$$f[i][j] = \frac{1}{2}f[i-1][j] + \frac{1}{2}f[i][j-1]$$
3. The possibility for A to be the winner is to consider all A's winning game times n.
$$P(\text{A win}) = \Sigma_{j=b}^{n-1} f[n][j]$$

Analysis for time complexity: Obviously $O(n)$ for initialization and $O(n^2)$ for update operations. Total time complexity: $O(n^2)$

Correctness proof of the DP idea:

Actually, we can find that this problem is a typical **binomial distribution** problem. Thus, if we can ensure the solution of $f[i][j]$ satisfies the **binomial distribution** formula, then our DP algorithm is true.

For this typical Markov Train problem, we have obviously that

$$P(\textbf{A win i and B win j}) = C_{i+j}^{i}(\frac{1}{2})^{i}(\frac{1}{2})^{j} = C_{i+j}^{i}(\frac{1}{2})^{i+j} = \frac{(i+j)!}{i!j!}(\frac{1}{2})^{i+j}$$

Luckily, we can find that $P(\textbf{A win 0 and B win 0}) = C_0^0(\frac{1}{2})^0(\frac{1}{2})^0 = 1 = f[0][0]$. Obviously, each linear recursive formula can have only one general term formula. If the formula of $P(\textbf{A win i and B win j})$ is exactly the formula of $f[i][j]$ then this problem is proved. Suppose $f[i][j] = \frac{(i+j)!}{i!j!}(\frac{1}{2})^{i+j}$, we have that:

$$\frac{1}{2}f[i-1][j] + \frac{1}{2}f[i][j-1] = \frac{1}{2}\frac{(i+j-1)!}{(i-1)!j!}(\frac{1}{2})^{i+j-1} + \frac{1}{2}\frac{(i+j-1)!}{(i)!(j-1)!}(\frac{1}{2})^{i+j-1}$$

$$\frac{1}{2}f[i-1][j] + \frac{1}{2}f[i][j-1] = (\frac{1}{2})^{i+j} * \frac{i*(i+j-1)! + j*(i+j-1)!}{i!j!} = \frac{(i+j)!}{i!j!}(\frac{1}{2})^{i+j} = f[i][j]$$

Therefore, the proof is done. The DP idea is correct. And the possibility of **A win n games** is:

$$P(\textbf{A win}) = \Sigma_{j=b}^{n-1} P(\textbf{A win n and B win j}) = \Sigma_{j=b}^{n-1} f[n][j]$$

Algorithm 2 The possibility for A to be a winner!

Require: Max winning game number: n.

Ensure: The possibility for A to be a winner.

```
 1: function WINNER($n$)
 2:     $f[0][0] \leftarrow 1$
 3:     $i \leftarrow a+1$
 4:     for $i \le n$ do
 5:         $f[i][b] \leftarrow \frac{1}{2} f[i-1][b]$
 6:         $i \leftarrow i+1$
 7:     end for
 8:     $j \leftarrow b+1$
 9:     for $i \le n$ do
10:         $f[a][i] \leftarrow \frac{1}{2} f[a][i-1]$
11:         $i \leftarrow i+1$
12:     end for
13:     $i \leftarrow a+1$
14:     for $i < n$ do
15:         $j \leftarrow b+1$
16:         for $j < n$ do
17:             $f[i][j] \leftarrow \frac{1}{2} f[i-1][j] + \frac{1}{2} f[i][j-1]$
18:             $j \leftarrow j+1$
19:         end for
20:         $i \leftarrow i+1$
21:     end for
22:     $result \leftarrow 0$
23:     $i \leftarrow b$
24:     for $i < n$ do
25:         $f[n][i] \leftarrow \frac{1}{2} f[n-1][i]$
26:         $result \leftarrow result + f[n][i]$
27:         $i \leftarrow i+1$
28:     end for
29:     return $result$
30: end function
```

Problem 3 (20 points)

Formalize the improved DP algorithm for the Longest Increasing Sequence Problem in the lecture, prove its correctness, and analyze its time complexity.

Note: this problem is the same as acmOJ **1432. Colorful inversion**(click for more and I have finished it.

To solve the DP problem, I firstly give the sub-problem definition:

$f[i]$ defines the smallest ended value for the longest increasing sequence with $length = i$.

1. Initialization when we haven't start to solve the problem.

   Array f is set to be empty.

2. Find the place pos of a_i in f_i that suits it best. The definition of **best** is:

   (1) $a_i \geq f[pos]$ and $a_i < f[pos-1]$ or (2) $pos = 1$.

3. Update the place of f at the position pos.

   $f[pos] = max(f[pos], a[i])$

Analysis for time complexity: Obviously $O(n)$ for iteration and $O(n)$ for place finding. Using binary search in the place finding can cause $O(\log n)$. At last, the total time complexity: $O(n \log n)$

Correctness proof of the improved DP idea:

Actually, this improved algorithm is based on the Greedy idea. Therefore, we can prove it by Greedy proof methods.

Firstly we try to prove that the array f is monotone. Suppose we have $f[i] > f[j]$ while $i < j$. Then, for the LIS with length j, we can delete $j - i$ values and get the LIS with length $= i$. Due to the definition of LIS, the new LIS satisfies all the elements are less than $f[j] < f[i]$. therefore, we can find one element in the new LIS that satisfies $x < f[j] < f[i]$, which yields a contradiction that $f[i]$ defines the smallest value. Therefore, the array f is monotone.

Secondly we try to prove the correctness of maintaining the array f. We can make sure that there must exist one LIS num with length $= x$ and $f[x-1] < num[i]$. Thus, if we add $num[i]$ at the end of LIS num we can get the new LIS with length $= x + 1$, which ends at $num[i]$. We know that $num[i] < seq[x]$ and thus we update the smallest end value.

In the presented algorithm below, I haven't give the codes about binary_search because it is finished in homework 1 and I take it as one finished tool to be applied.

Algorithm 3 Find the Longest Increasing Sequence by Improved DP

```
Require: Array: $a[1, ..., n]$. Size: $n$.
Ensure: The longest increasing sequence.
function LIS($StringA, StringB, n$)
    $i \leftarrow 0$
    $cnt \leftarrow 1$
    for $i \leq n$ do
        $f[i] \leftarrow 0$
        $i \leftarrow i + 1$
    end for
    $i \leftarrow 1$
    for $i \leq n$ do
        $pos \leftarrow binary\_search(f)$
        if $f[pos] = 0$ then
            $cnt \leftarrow cnt + 1$
        end if
        $f[pos] \leftarrow a[i]$
        $i \leftarrow i + 1$
    end for
    return $f[1, ..., cnt - 1]$
end function
```

Problem 4 (20 points)

Optimal Indexing for A Dictionary Consider a dictionary with n different words $a_1, a_2, ..., a_n$ sorted by the alphabetical order. We have already known the number of search times of each word a_i , which is represented by w_i . Suppose that the dictionary stores all words in a binary search tree T, i.e., each nodes word is alphabetically larger than the words stored in its left subtree and smaller than the words stored in its right subtree. Then, to look up a word in the dictionary, we have to do $l_i(T)$ comparisons on the binary search tree, where $l_i(T)$ is exactly the level of the node that stores a_i (root has level 1). We evaluate the search tree by the total number of comparisons for searching the n words, i.e., $\Sigma_{i=1}^{n} w_i l_i(T)$. Design a DP algorithm to find the best binary search tree for the n words to minimize the total number of comparisons.

To solve the DP problem, I firstly give the sub-problem definition:
$f[i][j]$ defines the smallest cost in the interval of nodes $[i, j]$.

1. Initialization when we haven't start to solve the problem.

   $\forall i \in [1, n], f[i][i] = w_i, sum[i] = sum[i-1] + w[i], sum[i, j] = sum[j] - sum[i-1]$

2. Choose one node to be the sub-root of the tree and calculate its cost.

   $f[i][j] = \max_{k=i+1 to j-1} f[i][k-1] + f[k+1][j] + sum[i, j]$

3. Actually, the boundary node i and j can also be the root.

   $f[i][j] = \max(f[i][j], f[i+1][j] + sum[i, j], f[i][j-1] + sum[i, j]$

Analysis for time complexity: Obviously $O(n^2)$ for iteration and $O(n)$ for k iterations in the interval$[i, j]$. At last, the total time complexity: $O(n^3)$

Correctness proof of the DP idea:

I prove the DP idea by two parts. The first is the transformation equation proof, and the second is the sequence equation proof.

1. Sequence order is proved by induction.

   We should prove that if we obey the transformation sequence order, none of the transformation in any iteration is missed.

   Base case: $f[i][i] = 0$.

   Inductive hypothesis:

   Each $f[i][j]$ can be solved correctly by $f[i][k-1], f[k+1][j]$ when $j - i \leq k \in N$.

   Induction:

   For the new $f[i][j](j - i = k + 1)$, it can be solved by the previous $f[i][k-1], f[k+1][j]$.

   Proof:

   $f[i][j] = \max_{k=i+1 to j-1} f[i][k-1] + f[k+1][j] + sum[i, j]$. Notice that even if we don't know $f[i][j]$ when $j - i = k + 1$, we know all the $f[i][k-1]$ and $f[k+1][j]$ because $\max(j - (k+1), k - 1 - i) \leq k$ by definition. Therefore, $f[i][j]$ can also be solved by the previous $f[i][k-1], f[k+1][j]$.

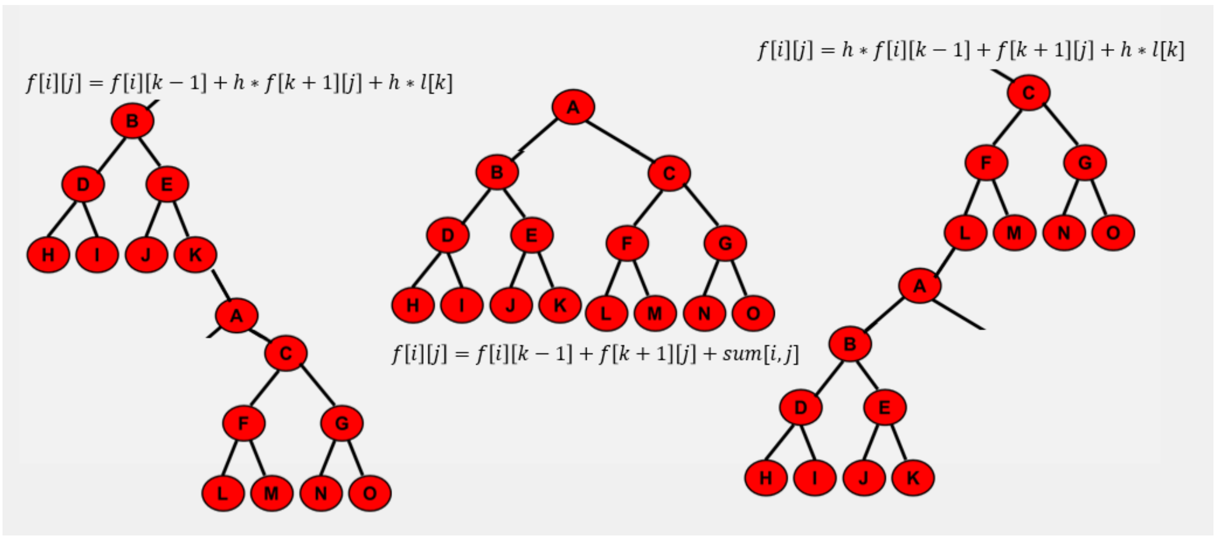
2. Transformation equation proof.

   We prove it by induction and contradiction. To simplify the proof, we just consider the induction step proof.

   Suppose we don't choose the one node that is the sub-root of the tree at at least one step. Then it means we choose one node($idx = k$) that is in the middle. Thus, the cost before containing the node will be $f[1][k-1]$ and $f[k+1][n]$. If we set the node at the rightest part of its left tree, the cost will be $f[1][k-1] + h \cdot f[k+1][n] + h \cdot l[k]$. If we set the node at the leftest part its right tree, the cost will be $h \cdot f[1][k-1] + f[k+1][n] + h \cdot l[k]$. We cannot set the node as the root of its left tree and right tree otherwise its the same as the iteration step. However, these two cost can be must huger than I think, which yields a contradiction that we should minimize the total number of comparisons.

   Therefore, our transformation equation is true.

   I try to explain my proof method based on the graph presented below.



$f[i][j] = f[i][k-1] + h * f[k+1][j] + h * l[k]$
$f[i][j] = f[i][k-1] + f[k+1][j] + sum[i,j]$
$f[i][j] = h * f[i][k-1] + f[k+1][j] + h * l[k]$
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

Figure 1: The explanation of my proof method.

Actually, the iteration order is a little different from the other 2-dim DP problems. It goes like:

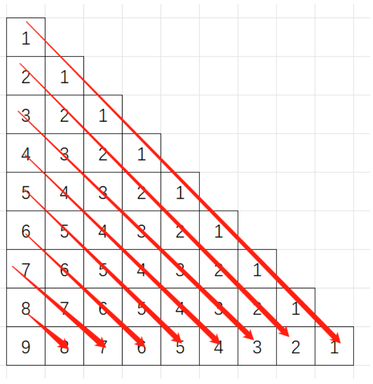


Figure 2: The iteration order of my algorithm.

Algorithm 4 Optimal Indexing for A Dictionary

Require: Array: $a[1,...,n], w[1,...,n]$. Size: n.

Ensure: The shortest total number of comparisons.

```
1: function OID($ArrayA, ArrayW, n$)
2:     $i \leftarrow 1$
3:     for $i \leq n$ do
4:         $f[i][i] \leftarrow w_i$
5:         $i \leftarrow i+1$
6:     end for
7:     $j \leftarrow 1$
8:     for $j \leq n-1$ do
9:         $i \leftarrow 1$
10:        for $i \leq n-j$ do
11:            $k \leftarrow i+1$
12:            for $k \leq i+j-1$ do
13:                $f[i][i+j] \leftarrow \max(f[i][i+j], f[i][k-1]+f[k+1][i+j]+sum[i,i+j])$
14:                $k \leftarrow k+1$
15:            end for
16:            $f[i][i+j] \leftarrow \max(f[i][i+j], f[i+1][i+j]+sum[i,i+j], f[i][i+j-1]+sum[i,i+j])$
17:            $i \leftarrow i+1$
18:        end for
19:        $j \leftarrow j+1$
20:    end for
21:    return $f[1][n]$
22: end function
```

Problem 5 (30 points)

Precedence Constrained Unit Size Knapsack Problem Let us recap the classic knapsack problem (unit size): We are given a knapsack of capacity W and n items, each item i has the same size $s_i = 1$ and is associated with its value v_i . The goal is to find a set of items S (each item can be selected at most once) with total size at most W (i.e., $\Sigma_{i\in S}s_i = |S| \leq W$) to maximize the total value (i.e., $\Sigma_{i\in S}v_i$). Now we are given some additional precedence constraints among items, for example, we may have that item i can only be selected if we have selected item j. These constraints are presented by a tree $G = (V, E)$, which says that a vertex can only be selected if we have selected its parent, (i.e., we need to select all his ancestors). The task is also to maximize the total value we can get, but not violating the capacity constraint and any precedence constraints.

(a) (20 points) Design a DP algorithm for it in $O(nW^2)$ time.

(b) (10 points) Can you improve the algorithm and analysis to make it run in $O(n^2)$ time? (When W is sufficient large like $W = O(n)$, it is an improvement.)

Tips:

- You can write down your best algorithm even if you can not get the required running time.
- You can improve the running time to $O(nW)$.
- You can directly finish question (b) (30 points).
- You may use the following observation: For a subtree T, we can at most select $|T|$ items.

To solve the DP problem, I firstly give the sub-problem definition:

$f[i][j]$ defines the maximum cost when the root is i and the capacity cost is j. Then there are some situations. Denote $w[i]$ as $v[i]$ because I will use v to denote the vertex.

The correctness proof is almost the same as presented in Problem 4. Therefore, I don't put it again.

1. Initialization when we haven't start to solve the problem.

   $\forall i \in V, f[i][0] = 0$. All the leaves(By DFS) should have $f[i][1] = w[i]$

2. Iteration all the root based on the DFS(Finish_time) order and explore all the child.

   $f[i][j] = \max_{\text{v is i's son}}(f[i][j], f[i][j-k-1] + f[v][k] + w[i])$

However, due to the reason that this DFS method goes on the tree, the time complexity is terribly $O(nW^2)$. $O(W^2)$ for iteration over capacity and $O(n)$ for tree DFS cost. I find that the cost for each iteration over capacity(explore all the sub-tree of each node) is too huge and I hope to simplify this complexity. To do this method, I consider the level order situation and sub-problem: $f[i][j]$ defines the whole tree rooted at i and all its chosen nodes in its son costs capacity j. The transmission formula goes like: $f[i][j] =$

$\max(f[brother][j], \max_{x+y=j-1}(f[from][x] + f[to][y] + v[i]))$. Therefore, the algorithm goes like:

Algorithm 5 Precedence Constrained Unit Size Knapsack Problem

```
Require: Graph $G = (V, E)$.
Ensure: The maximum value of knapsack problem.
 1: function INITIALIZE($Array : brother, son$)
 2:     Array brother, son ← DFS in Graph $G = (V, E)$
 3:     Knapsack(root)
 4:     return $f[root][W]$
 5: end function
 6: function KNAPSACK($now$)
 7:     $f[now][0] \leftarrow 0, f[now][1] = v[now]$
 8:     $bro \leftarrow brother[now], s \leftarrow son[now]$
 9:     if s is not empty then
10:         Knapsack(s)
11:     end if
12:     if bro is not empty then
13:         Knapsack(bro)
14:     end if
15:     $i \leftarrow 1$
16:     for $i \leq n$ do
17:         $f[now][i] \leftarrow f[bro][i]$
18:         $i \leftarrow i + 1$
19:     end for
20:     $x \leftarrow 0$
21:     for $x \leq len(s)$ do
22:         $y \leftarrow 0$
23:         for $y \leq len(bro)$ do
24:             $j = x + y + 1$
25:             $f[i][j] = \max(f[i][j], f[s][x] + f[bro][y] + v[now])$
26:             $y \leftarrow y + 1$
27:         end for
28:         $x \leftarrow x + 1$
29:     end for
30: end function
```

Luckily, this problem has the same size $s_i = 1$. If not, this problem is harder!

To prove the time complexity, I learn the idea from my collaborator. We can make sure that each recursive function has the same time complexity formula but different parameters. For the function above, we can firstly find that the complexity for the last loop is $O(len(s) \cdot len(bro))$. The first loop costs $O(W)$ actually. and to do iteration over each node has at most $O(len(s)^2)$ because we just search all its nodes for square times based on the last loop.

Therefore, the time complexity can have the property that

$$O(len(s)^2) + O(len(bro)^2) + O(len(s) \cdot len(bro)) = C(len(s)^2 + len(bro)^2 + len(s))$$

$$C(len(s)^2 + len(bro)^2 + len(s)) \leq C(len(s)^2 + len(bro)^2 + 2len(s)) = C(len(s) + len(bro))^2$$

$$C(len(s) + len(bro))^2 = O((len(s) + len(bro))^2)$$

$$O(len(s)^2) + O(len(bro)^2) + O(len(s) \cdot len(bro)) \leq O((len(s) + len(bro))^2)$$

In the same way, we can conclude that:

$$O((len(s) + len(bro))^2) \leq O((len(s) + len(bro) + 1)^2) = O(len(now)^2)$$

Actually, the maximum scale of *root* cannot be larger than V, which means the total time complexity is $O(n^2)$.

But, we should note that there is one missing loop: $O(W)$ in the 17-th row. Anyway, we can make sure that $W = \min(W, n)$ because otherwise it is useless. Therefore, we can make sure that $O(W) = O(n)$. However, the iteration will not work on this recursive function obviously because its prime is low enough. At last, the total time complexity is $O(n^2)$.

Problem 6

How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

I take 8 hours in total to finish the assignment(include thinking and discussing).

Difficulty: 3(Easy to find the solution if we can find correct iteration formula).

Although this is easy to find, and obviously the formula is true, I cannot find a correct way to prove it.

Collaborators: Xiangge Huang in Problem 5.