

## Part A: Description of Our Algorithm and Analysis of Time Complexity

Let  $A$  be  $a_1a_2\dots a_n$  and  $B$  be  $b_1b_2\dots b_n$ , where  $a_i, b_i = 0$  or  $1$ ,  $i = 1, 2, \dots, n$ . Let  $L(i, j)$ ,  $D(i, j)$  denote the length of the LCS and the number of distinct LCS's, respectively, for the string pairs  $a_1a_2\dots a_i$  and  $b_1b_2\dots b_j$ , where  $1 \leq i \leq n$  and  $1 \leq j \leq n$ . Let  $(A^*, B^*)$  be the optimal string pair which maximizes the number of distinct LCS's and its  $L(n, n) \equiv L$ ,  $D(n, n) \equiv M$ . We use the following steps to get  $(A^*, B^*)$ : (The algorithm description is just descriptive. Please refer to our code for more formal representation.)

1. First, we use the DFS tree algorithm to explore all possible pairs. The following figure is a simplified representation of our tree. A node in Level  $i$  represents a possible combination of length- $i$  strings, and each child in Level  $i$  inherits the first  $(i-1)$  characters of its parent. Because each position of strings has four combinations,  $(0,0)$ ,  $(0,1)$ ,  $(1,0)$ ,  $(1,1)$ , each node has four children.

We start from  $(a_1, b_1) = (0,0)$  and then explore its smallest child  $(a_2, b_2) = (0,0)$ . We continue to explore the smallest child of each node and come to  $(a_n, b_n) = (0,0)$ . The path, which is highlighted in yellow in the following figure, represents a pair of strings which consist of all 0s.

For each node we explore, we use the algorithm described in Step 2 to calculate  $L(i, i)$ ,  $D(i, i)$ , where  $i$  indicates the level. We save the information for further use. Once we get to the leaf node (node in Level  $n$ ), we also calculate its  $L(n, n)$  and  $D(n, n)$ . For the first leaf node, we temporarily set it to be  $(A^*, B^*)$ . Thereafter, if a leaf node has a higher  $D(n, n)$ , we replace the old optimal pair with this one.

Once  $(a_n, b_n) = (0,0)$  is explored, we continue to explore its siblings with different  $(a_n, b_n) = (0,1)$ ,  $(1,0)$ ,  $(1,1)$  combinations, whose first  $(n-1)$  characters are the same. They are highlighted in green in the following figure. Because they inherit the same parent, they can use their parent's information to calculate their own values. Once all children have been explored, we continue to explore the children of their parent's sibling (highlighted in blue in the following figure). Once all siblings' children have been explored, we continue to explore children of the sibling of the siblings' parent. In this way, we can explore all possible pairs.

Level $i \backslash (a_i, b_i)$									
1					(0,0)		(0,1)	(1,0)	(1,1)
2			(0,0)		(0,1)	(1,0)	(1,1)		
3		(0,0)	(0,1)	(1,0)	(1,1)				
...									
$n-1$		(0,0)	(0,1)						
$n$	(0,0)	(0,1)	(1,0)	(1,1)					

However, we find that the recursive method could result in stack overflow when  $n > 7$ . In this case, we use the divide and conquer algorithm to solve the problem. That is, we iteratively generate a set of roots  $(a_1a_2\dots a_p, b_1b_2\dots b_p)$  ( $p = 5$  in our case) and then develop a DFS tree for each root. Besides, we observe that if  $(A^*, B^*)$  starts from  $(1,0)$  or  $(1,1)$ , by swapping 0 and 1, it is still a choice of  $(A^*, B^*)$ , which starts from  $(0,1)$  or  $(0,0)$ . Therefore, for  $n > 7$ , in order to reduce execution time further, we only calculate pairs which start from  $(0,0)$  or  $(0,1)$ .

2. When we reach a node in Level  $i$  (a pair of length- $i$  strings), we use dynamic programming to calculate its  $L(i,i)$  and  $D(i,i)$ . We refer to “Computing the Number of Longest Common Subsequences” published by Professor Ronald I. Greenberg at Loyola University<sup>1</sup> and use the following dynamic programming methods to obtain  $L(i,j)$  and  $D(i,j)$ :

$$\begin{aligned}
 &L(i,j) \\
 &= 0 \text{ if } i = 0 \text{ or } j = 0 \\
 &= L(i-1, j-1) + 1 \text{ if } i, j > 0 \text{ and } a_i = b_j \\
 &= \max(L(i-1, j), L(i, j-1)) \text{ otherwise} \\
 \\
 &D(i,j) \\
 &= 0 \text{ if } i = 0 \text{ or } j = 0 \\
 &= 1 \text{ if } a_i = b_j \text{ and } L(i, j) = 1 \\
 &= D(i-1, j-1) \text{ if } a_i = b_j \text{ and } L(i, j) > 1 \\
 &= D(i-1, j) + D(i, j-1) - D(i-1, j-1) \text{ otherwise}
 \end{aligned}$$

Because in each iteration, only  $i, i-1, j, j-1$ 's information is used, to reduce memory requirement, we use two arrays to calculate the values for each node, one for fixed  $i$  and floating  $j$  and one for floating  $i$  and fixed  $j$ . Besides, we also store  $L(i, i)$  and  $D(i, i)$  for each parent in Level  $i$  so that its child can inherit the information and develop its own  $L(i+1, i+1)$  and  $D(i+1, i+1)$  when one more combination of characters is added.

When  $n \leq 7$ , we use DFS only. Each node takes  $\theta(n)$  time to calculate  $L(i,i)$  and  $D(i,i)$  because it uses its parent's information, and there are  $4^0 + 4^1 + 4^2 + \dots + 4^n = \frac{1-4^{n+1}}{1-4} = c*4^n + d$  ( $c, d$  are constants) nodes in a DFS tree, so the running time is  $\theta(n*4^n)$ .

When  $n > 7$ , first we iteratively generate all combinations of length-5 strings which start from  $(0,0)$  or  $(0,1)$ , which takes  $\theta(2*4^4)$  time. And for each root, we spend  $\theta(5^2)$  time to get  $L(5,5)$ ,  $D(5,5)$  and  $\theta(n*4^{n-5})$  time to develop its leaves, so the running time is  $(\theta(5^2) + \theta(n*4^{n-5})) * \theta(2*4^4) = \theta(n*4^n)$ , still the same.

<sup>1</sup> <https://arxiv.org/abs/cs/0301034>

3. Once we have explored all leaves, we can get  $(A^*, B^*)$  and L, M. To decide its list of distinct LCS's, we follow the steps below:
  - (1) We use a backward version of the dynamic programming method mentioned above to get  $L'$  and  $D'$  matrices where  $L'(0,0) = L$  and  $D'(0,0) = M$ . (Because we will need to copy the first few characters of one string to another string using the C++ function "strncpy",  $L'$  will facilitate the process.) This step takes  $\theta(n^2)$  time.
  - (2) And then we build its Position Matrices  $A\_Position[i][j]$  and  $B\_Position[i][j]$  for  $j = 0, 1$ , where  $i = 1, 2, \dots, n$ .  $A\_Position[i][j]$  means the position of the first  $j$  for  $a_1a_2\dots a_n$ . For example, if  $A$  is "1010", then  $A\_Position[1][0]$  is 2 (the first 0 occupies  $a_2$  for  $a_1a_2a_3a_4$ ) and  $A\_Position[2][1]$  is 3 (the first 1 occupies  $a_3$  for  $a_2a_3a_4$ ). This step takes  $\theta(n)$  time.
  - (3) Finally, we use the DFS tree algorithm similar to that of Step 1 (shown as below) to explore all pairs of length- $L$  strings to find all LCS's of  $(A^*, B^*)$ . Let  $D\_LCS = \{D\_LCS_k = d_1d_2\dots d_L, k = 1, 2, \dots, M\}$  be the set of all distinct LCS's. As with the figure in Step 1, in the following tree, a node in Level  $i$  represents a length- $i$  string which could be a substring of an LCS, and each child in Level  $i$  inherits the first  $(i-1)$  characters of its parent.

```

Explore ( $a_x, b_y$ ) where  $x = A\_Position[0][0]$  and  $y = B\_Position[0][0]$  first
If  $L'(x,y) = L$  (by definition,  $a_x$  and  $b_y$  must be 0), assign 0 to the first character of  $D\_LCS_1$ 
    and then explore ( $a_x, b_y$ ) where  $x = A\_Position[x+1][0]$  and  $y = B\_Position[y+1][0]$ 
        if  $L'(x,y) = L-1$ , assign 0 to the second character of  $D\_LCS_1$ 
        else explore ( $a_x, b_y$ ) where  $x = A\_Position[x+1][1]$  and  $y = B\_Position[y+1][1]$  to see if 1 is
the second character of  $D\_LCS_1$ 
    ...
else explore ( $a_x, b_y$ ) where  $x = A\_Position[0][1]$  and  $y = B\_Position[0][1]$  to see if 1 is the first
character of any LCS
    if  $L'(x,y) = L$ , assign 1 to the first character of  $D\_LCS_1$ 
    and then explore ( $a_x, b_y$ ) where  $x = A\_Position[x+1][0]$  and  $y = B\_Position[y+1][0]$  (always start
from 0, and then 1)
    ...

```

[illegible]









Once we reach a leaf, we obtain an LCS. And then we visit its sibling to see if it is an LCS as well. For example, if "0000" is an LCS, we continue to check if "0001" (the sibling) is also an LCS. If yes, then we copy the string of its parent and add 1 to the string to compose the LCS. After finishing exploring all children, we explore the children of their parent's sibling. In this way, we can systematically explore all possible LCS's and find all distinct LCS's. Because for each LCS, we spend  $\theta(L)$  time to generate the string, this step takes  $\theta(LM)$  time.

Therefore, the total running time of our algorithm is  $\theta(n \cdot 4^n) + \theta(n^2) + \theta(n) + \theta(LM)$ . Since  $L \leq n$  and  $M \leq 2^L \leq 2^n \leq 4^n$ ,  $LM \leq n \cdot 4^n$ , the formula can be simplified as  $\theta(n \cdot 4^n)$ .

## Part B: Organization of Our Source Code

The components of our source code are shown in the following figure:

- **demo.cpp**: main function and used to run the four cases of  $n = 4, 7, 10, 12$ . Can run the program by typing “./demo” in the command line.
- **strings.h** and **strings.cpp**: define the class **STRINGS** which represents a pair of length- $n$  strings
- **test.h** and **test.cpp**: define the class **TEST** which is used to generate  $(A^*, B^*)$  for a given  $n$
- **timer.h** and **timer.cpp**: provided by the Professor as a tool to measure execution time
- **makefile**: used to compile our program

 demo	2017/12/2 下午 02:28	C++ Source File
 makefile	2017/11/23 下午 12:...	檔案
 strings	2017/12/2 下午 02:51	C++ Source File
 strings	2017/12/2 下午 02:51	C Header File
 test	2017/12/2 下午 02:55	C++ Source File
 test	2017/12/2 上午 01:09	C Header File
 timer	2017/11/22 下午 09:...	C++ Source File
 timer	2017/11/22 下午 10:...	C Header File

## Part C: Output Produced by the Execution of Our Program

(Screen shots from Openlab are also attached below.)

Output produced by the execution of our program for **n = 4**:

(1) the determined binary strings A and B:

A: 0110

B: 1001

(2) the number of distinct LCS's:

4

(3) the list of those LCS's:

#1 00

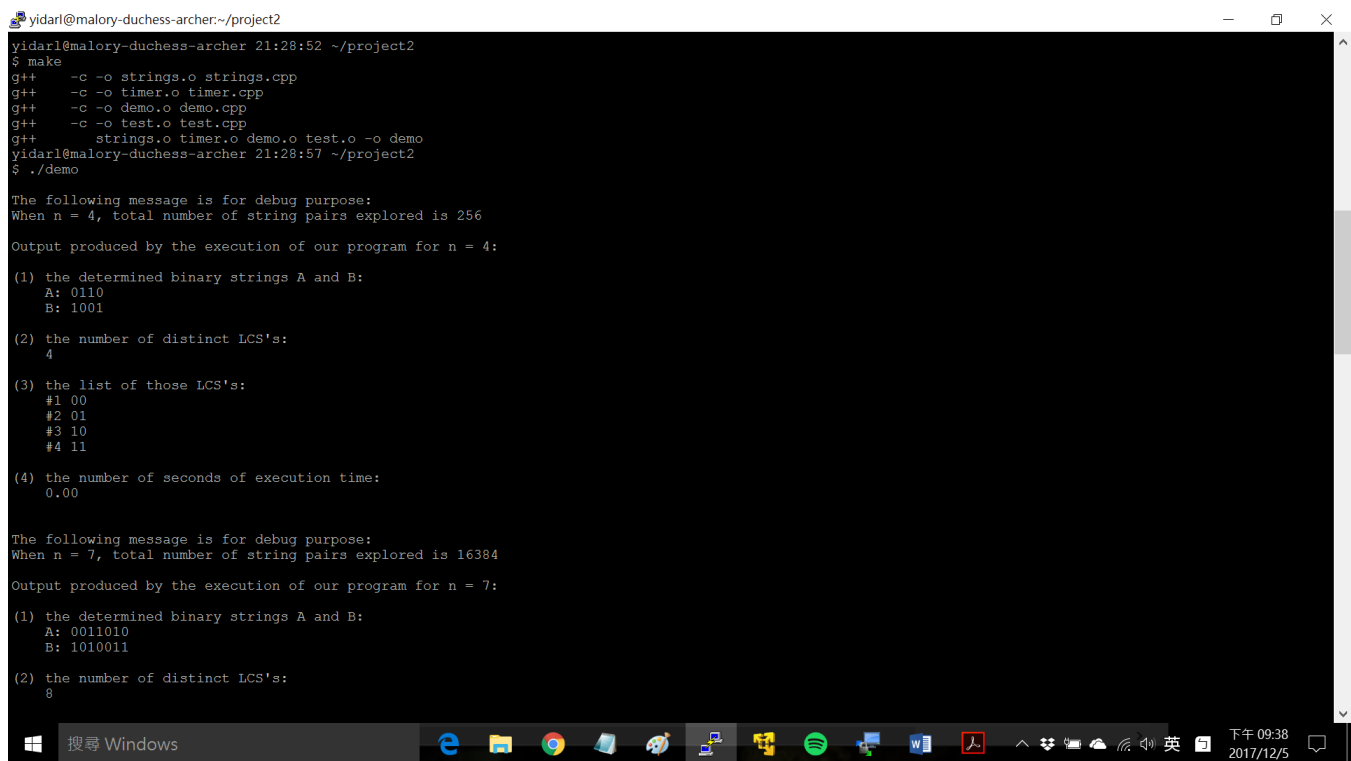
#2 01

#3 10

#4 11

(4) the number of seconds of execution time:

0.00



```
yidarl@malory-duchess-archer:~/project2
$ make
g++ -c -o strings.o strings.cpp
g++ -c -o timer.o timer.cpp
g++ -c -o demo.o demo.cpp
g++ -c -o test.o test.cpp
g++ strings.o timer.o demo.o test.o -o demo
yidarl@malory-duchess-archer:~/project2
$ ./demo

The following message is for debug purpose:
When n = 4, total number of string pairs explored is 256

Output produced by the execution of our program for n = 4:

(1) the determined binary strings A and B:
A: 0110
B: 1001

(2) the number of distinct LCS's:
4

(3) the list of those LCS's:
#1 00
#2 01
#3 10
#4 11

(4) the number of seconds of execution time:
0.00

The following message is for debug purpose:
When n = 7, total number of string pairs explored is 16384

Output produced by the execution of our program for n = 7:

(1) the determined binary strings A and B:
A: 0011010
B: 1010011

(2) the number of distinct LCS's:
8
```

Output produced by the execution of our program for **n = 7**:

(1) the determined binary strings A and B:

A: 0011010

B: 1010011

(2) the number of distinct LCS's:

8

(3) the list of those LCS's:

#1 0001

#2 0011

#3 0100

#4 0101

#5 0111

#6 1010

#7 1100

#8 1101

(4) the number of seconds of execution time:

0.12

```
yidarl@malory-duchess-archer:~/project2
The following message is for debug purpose:
When n = 7, total number of string pairs explored is 16384

Output produced by the execution of our program for n = 7:

(1) the determined binary strings A and B:
A: 0011010
B: 1010011

(2) the number of distinct LCS's:
8

(3) the list of those LCS's:
#1 0001
#2 0011
#3 0100
#4 0101
#5 0111
#6 1010
#7 1100
#8 1101

(4) the number of seconds of execution time:
0.12

The following message is for debug purpose:
When n = 10, total number of string pairs explored is 524288

Output produced by the execution of our program for n = 10:

(1) the determined binary strings A and B:
A: 0101010101
B: 1001100110

(2) the number of distinct LCS's:
20

(3) the list of those LCS's:
#1 0010010
#2 0010011
#3 0010110
#4 0011001
#5 0011010
```

Output produced by the execution of our program for **n = 10**:

(1) the determined binary strings A and B:

A: 0101010101

B: 1001100110

(2) the number of distinct LCS's:

20

(3) the list of those LCS's:

#1 0010010

#2 0010011

#3 0010110

#4 0011001

#5 0011010

#6 0011011

#7 0100110

#8 0110010

#9 0110011

#10 0110110

#11 1001001

#12 1001010

#13 1001011

#14 1001101

#15 1010010

#16 1010011

#17 1010110

#18 1011001

#19 1011010

#20 1011011

(4) the number of seconds of execution time:

2.79

```
yidarl@malory-duchess-archer:~/project2
The following message is for debug purpose:
When n = 10, total number of string pairs explored is 524288

Output produced by the execution of our program for n = 10:

(1) the determined binary strings A and B:
A: 0101010101
B: 1001100110

(2) the number of distinct LCS's:
20

(3) the list of those LCS's:
#1 0010010
#2 0010011
#3 0010110
#4 0011001
#5 0011010
#6 0011011
#7 0100110
#8 0110010
#9 0110011
#10 0110110
#11 1001001
#12 1001010
#13 1001011
#14 1001101
#15 1010010
#16 1010011
#17 1010110
#18 1011001
#19 1011010
#20 1011011

(4) the number of seconds of execution time:
2.79

The following message is for debug purpose:
When n = 12, total number of string pairs explored is 8388608

Output produced by the execution of our program for n = 12:

(1) the determined binary strings A and B:
```



Output produced by the execution of our program for **n = 12**:

(1) the determined binary strings A and B:

A: 001100011001

B: 101010101010

(2) the number of distinct LCS's:

36

(3) the list of those LCS's:

#1 00100010  
#2 00100100  
#3 00100101  
#4 00100110  
#5 00101001  
#6 00101100  
#7 00101101  
#8 00110010  
#9 00110100  
#10 00110101  
#11 00110110  
#12 01000100  
#13 01000101  
#14 01000110  
#15 01001001  
#16 01001100  
#17 01001101  
#18 01011001  
#19 01100010  
#20 01100100  
#21 01100101  
#22 01100110  
#23 01101001  
#24 01101100  
#25 01101101  
#26 10001001  
#27 10001100  
#28 10001101  
#29 10011001  
#30 11000100  
#31 11000101  
#32 11000110  
#33 11001001  
#34 11001100  
#35 11001101  
#36 11011001

(4) the number of seconds of execution time:

38.47

```
yidarl@malory-duchess-archer:~/project2
The following message is for debug purpose:
When n = 12, total number of string pairs explored is 8388608

Output produced by the execution of our program for n = 12:

(1) the determined binary strings A and B:
A: 001100011001
B: 101010101010

(2) the number of distinct LCS's:
36

(3) the list of those LCS's:
#1 00100010
#2 00100100
#3 00100101
#4 00100110
#5 00101001
#6 00101100
#7 00101101
#8 00110010
#9 00110100
#10 00110101
#11 00110110
#12 01000100
#13 01000101
#14 01000110
#15 01001001
#16 01001100
#17 01001101
#18 01011001
#19 01100010
#20 01100100
#21 01100101
#22 01100110
#23 01101001
#24 01101100
#25 01101101
#26 10001001
#27 10001100
#28 10001101
#29 10011001
#30 11000100
#31 11000101

Windows 搜索 Windows
```

```
yidarl@malory-duchess-archer:~/project2

(3) the list of those LCS's:
#1 00100010
#2 00100100
#3 00100101
#4 00100110
#5 00101001
#6 00101100
#7 00101101
#8 00110010
#9 00110100
#10 00110101
#11 00110110
#12 01000100
#13 01000101
#14 01000110
#15 01001001
#16 01001100
#17 01001101
#18 01011001
#19 01100010
#20 01100100
#21 01100101
#22 01100110
#23 01101001
#24 01101100
#25 01101101
#26 10001001
#27 10001100
#28 10001101
#29 10011001
#30 11000100
#31 11000101
#32 11000110
#33 11001001
#34 11001100
#35 11001101
#36 11011001

(4) the number of seconds of execution time:
38.47

yidarl@malory-duchess-archer 21:29:50 ~/project2
$
```

## **Part D: References**

Ronald I. Greenberg. Computing the Number of Longest Common Subsequences. arXiv:cs/0301034v1 [cs.DS] 29 Jan 2003.