Wataru Oshima

CSCI 411 - Advanced Algorithms and Complexity
Assignment 4

September 18, 2022

Solutions to the written portion of this assignment should be submitted via PDF to Blackboard. Make sure to justify your answers. C++ code should be submitted on both Blackboard and turnin. Both parts of the assignment are due before **September 25th at 11:59 pm**.

There will likely be time in class to discuss these problems in small groups and I highly encourage you to collaborate with one another outside of class. However, you must write up your own solutions **independently** of one another. Feel free to communicate via Discord and to post questions on the appropriate forum in Blackboard. Do not post solutions. Also, please include a list of the people you work with at the top of your submission.

Written Problems

1. Consider the following refined notion of bitonicity. Call a sequence $S = [s_1, s_2, \ldots, s_n]$ an *initially increasing bitonic sequence* if there is an index $1 \le i \le n$ such that $s_1 < s_2 < \cdots < s_i$ and $s_i > s_{i+1} > \cdots > s_n$. On the other hand, call S an *initially decreasing bitonic sequence* if there is an index $1 \le j \le n$ such that $s_1 > s_2 > \cdots > s_j$ and $s_j < s_{j+1} < \cdots < s_n$. S is *bitonic* if it is either initially increasing or initially decreasing and bitonic.

   Given a sequence A, we would like to determine its longest bitonic subsequence.

   (a) (5 pts) Find a longest bitonic subsequence of the sequence $A = [5, 8, 8, 3, 4, 1, 7, -3, 2, 9, 12]$. Show your work.

   (b) (15 pts) Describe the optimal substructure of this problem. In particular, define the longest bitonic subsequence of A in terms of the solution for shorter sequences. You may assume that we have a solution for the longest increasing subsequence problem. Justify your answer.

   (c) (10 pts) Write pseudocode for a function `LBS(A)` which returns the length of a longest bitonic subsequence of A. You are given a function `LIS(A)` that returns a list L that is the same length as A such that $L[i]$ is the length of the longest increasing subsequence of A ending at index i.

   (d) (5 pts) Analyze the asymptotic run time of your algorithm.

(a) A[5,8,8,3,4,1,7,-3,2,9,12]

5 8 3 4 -3$_{\text{or}2}$ = 5

8 3 1 -3 2 9 12 = 7

3,1,-3,2,9,12 = 6

7

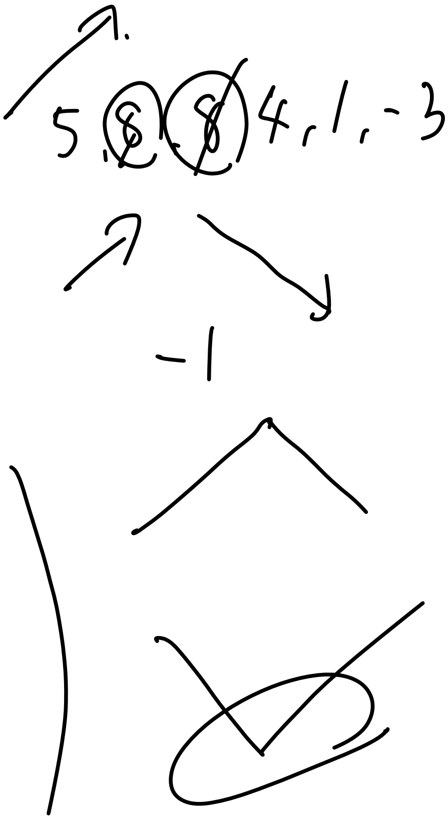
(b) A[5,8,8,3,4,1,7,-3,2,9,12..]

$\overset{0\le j<i}{\max(1,\ \text{If}\ \max(\text{s.t}\ A[j]<A[i], L[j]\ )}$

L_{IS} = [1,2,1,1,1,1,1,2,2,4,5]

L_{DS} = [4,4,4,3,3,2,2,1,1,1,1]

L_{BS} = [4,5,5,3,4,2,4,1,2,4,5]



Ain[-5,-8,-8,-3,-4,-1,-7,3,-2,-9,-12]

Lis_in = [1,1,1,2,2,3,2,4,3,1,1]

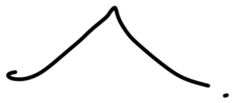
LDS_in = [4,3,3,5,4,4,3,4,3,2,1]

Lbs_in = [4,3,3,6,5,6,5,7,5,2,1]

8,4,1,-3,2,9,12

valley

longest : 7.

Since we need to find the longest sequence of increasing and decreasing order, we set two arrays One, called LIS stores the longest sequence in ascending order. the other one, called LDS, store the longest sequence in descending order. We start checking how long it could be from 0 to i and store it in the array. The difference between the arrays is we start checking the maximum number in forward and backward respectively. And at last after filling up the arrays, we add two numbers from Lis[i] and LDS[i] and minus 1 so that we can check what the longest value is for uphill to downhill such as . And do the same thing as dowhill to uphill " " and the biggest number between them is the number of longest bitonic subsequence.

```
(c) def LBS(A)
        Lis =[]
        Lds =[]
        Lbs = []
        maxNum = 0
        Lis = LIS(A)
        Lds = LIS(A.reverse)  //reversed order
        for i in lengthof A
            Lbs.append((Lis[i]+Lds[i])-1)
        maxNum = max(maxNum, lbs)
        Lis.clear()
        Lds.clear()
        Lbs.clear()
        Ainv = A.inverse  //make them negative
        Lis = LIS(Ainv)
        Lds = LIS(Ainv.inverse)
        for i in length of A
            Lbs.append((Lis[i]+Lds[i])-1)
        maxNum = max(maxNum, Lbs)
        return maxNum
```

(d) $O(|A||A|) = O(|A|^2)$ // $O(n^2)$

2 A= exponential

(a) B= polynomial

a t a c c g
a a c g c a

cost : 6

	_	e	x	p	o	n	e	n	t	i	a	l
_	0	1	2	3	4	5	6	7	8	9	10	11
p	1	1	2	2	3	4	5	6	7	8	9	10
o	2	2	2	3	2	3	4	5	6	7	8	9
l	3	3	3	3	3	3	4	5	6	7	8	8
y	4	4	4	4	4	4	4	5	6	7	8	9
n	5	5	5	5	5	4	5	4	5	6	7	8
o	6	6	6	6	5	5	5	5	5	6	7	8
m	7	7	7	7	6	6	6	6	6	6	7	8
i	8	8	8	8	7	7	7	7	7	6	7	8
a	9	9	9	9	8	8	8	8	8	7	6	7
l	10	10	10	10	9	9	9	9	9	8	7	6

del

ins

A	e x p o n e n _ t i a l
B	_ _ p o l y n o m i a l

(b) Let me set A = "abc", B = "cbda".

The optimal alignment between them is below.

A	a b _ c
B	c b d a

To get this, I set the matrix (A+1) × (B+1) such as below

	_	a	b	c
_				
c				
b				
g				
a				

The first row and column are empty string to make base cases.

For the base cases,

if I set D as a 2-d array for the cost, then

$D[0][0] = 0$ // because it matches

$D[i][0] = i \times del$

↳ because as you go to the next column you have to delete a charcter i times to make it empty.

	_	a	b	c
_	0	1	2	3
c	1			
b	2			
g	3			
a	4			

del

ins

So as D[0][j], which should be

$D[0][j] = j \times ins$ in this case inserting a chaecter.

Then we can start alligning the strings with the base cases.

	_	a	b	c
_	0	1	2	3
c	1	1	2	2
b	2	2	1	2
g	3	3	2	2
a	4	4	3	$\boxed{3}$

del

ins

We can set if statement like

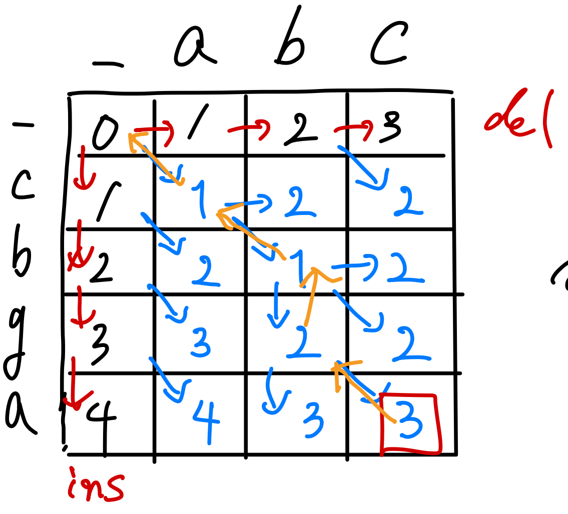
$$D[i][j] = \begin{cases} D[i-1][j-1], & \text{if } A[i]=B[j] \\ \min(D[i][j-1]+ins,\ D[i-1][j]+del,\ D[i-1][j-1]+sub) \end{cases}$$

In words, if they match, we do not have to do so just put the previous number which is located at D[i-1][j-1].

If they do not match, then we have to get the minimum cost of operation among insertion, deletion, substitution.

If we need to insert a character, we have to add the cost of insertion and D[i][j-1] so do deletion & substitution but refer to the cost of deletion and substitution D[i-1][j] and D[i-1][j-1] respectively

And the last element of D, which is D[i][j], is the total cost for edit distance and you can backtrack the arrows to get the optimal allignment.



	_	a	b	c
_	0	1	2	3
c	1	1	2	2
b	2	2	1	2
g	3	3	2	2
a	4	4	3	3

Cost : 3

⇒

A	a	b	_	c
B	c	b	g	a

```
def ediDistance(A, B, ins, del, sub)
  D = [length of A + 1][length of B + 1]
  D[0][0] = 0
  D[[i for i in len(D)]][0] = i × ins
  D[0][[j for j in len(D[0])]] = j × del
  for i ∈ D
    for j ∈ D[i]
      if A[i] == B[j]
        D[i][j] = D[i-1][j-1]
      else
        D[i][j] = min(D[i][j-1]+ins, D[i-1][j]+del, D[i-1][j-1]+sub)
  return D[i][j]
```

(d) $O(|A||B|)$

2. Given two strings A and B, we can transform A to B using insertions, deletions, and substitutions. Each of these operations comes with a prespecified cost. Our goal is to determine the minimum cost of changing A to B. We call this cost the *edit distance* from A to B.

   For example, suppose A = "*ataccg*" and B = "*aacgca*" and insertions, deletions, and substitutions all have cost 1. Then we can transform A to B by (1) deleting the t, (2) inserting a g between the two c's, and (3) substituting the last g for an a. This process has total cost 3 and the edit distance from A to B is 3.

   We can represent this transformation with the following alignment:

A	a	t	a	c	_	c	g
B	a	_	a	c	g	c	a

   Here, insertions are represented by an underscore ("_") in A, deletions are represented by an underscore in B, and substitutions are represented by mismatched characters.

   (a) (5 pts) Determine the edit distance and an optimal alignment between the strings "exponential" and "polynomial". Show your work (draw the resulting alignment).

   (b) (15 pts) Describe the optimal substructure of this problem. In particular, define the edit distance between A and B in terms of the solution for shorter strings. Justify your answer.

   (c) (10 pts) Write pseudocode for a function `editDistance(A, B, ins, del, sub)` which returns the edit distance between A and B given costs for insertions, deletions, and substitutions. Assume that matches have a cost of 0. This function **does not** need to generate an alignment.

   (d) (5 pts) Analyze the asymptotic run time of your algorithm.

Coding Problem

(30 pts) Write a C++ implementation of the pseudocode you developed for problem (2c) modified to return an alignment and submit to Blackboard and to turnin as assignment_4.cpp. You may find the skeleton code in assignment_4_skeleton.cpp on Blackboard helpful.

- Input will come from cin
  - The first line contains a single integer n indicating the number of examples. $n+1$ lines follow.
  - The second line contains three space separated integers. These represent the cost of an insertion, a deletion, and a substitution respectively and may be positive, negative, or zero. The cost of a match is assumed to be 0.
  - The next n lines each contain two space separated strings, A and B.
- Print output to cout
  - Your output should consist of three lines per example.
  - The first line is the alignment of string A.
  - The second line is the alignment of string B.
  - The third line is the cost or score associated with the alignment.
- If there is ever a choice between multiple actions which would result in different optimal alignments, prefer substitutions to deletions and deletions to insertions (first try substitution, then try deletion, and finally try insertion).

Examples

Example 1:

Input:

```
4
1 1 1
expon poly
snowy sunny
coarse course
ataccg aacgca
```

Expected output:

```
expo_n
__poly
4
snowy
sunny
3
coarse
course
1
atac_cg
a_acgca
3
```

Example 2:

Input:

```
2
3 3 2
ataagcc gtacc
aagtaac gcccgtaa
```

Expected output:

```
ataagcc
gt_a_cc
8
__aagtaac
gcccgtaa_
13
```

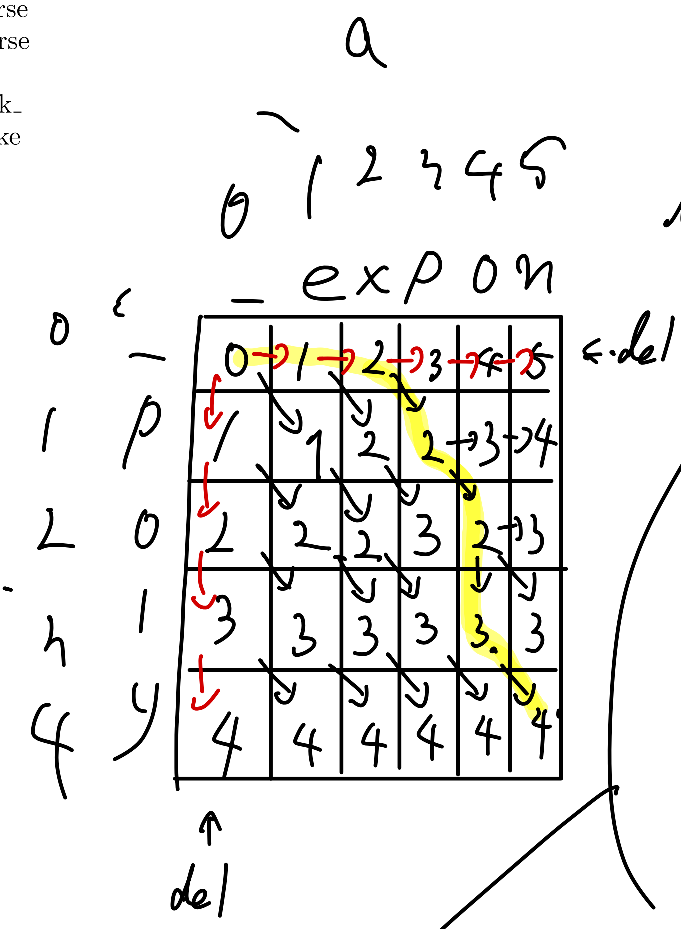
Example 3:

Input:

```
2
1 3 5
coarse course
break brake
```

Expected output:

co_arse
cou_rse
4
break_
br_ake
4



```
Cose = D[lengthA-1][lengB-1]

i = length of A - 1
j = length of B - 1

while(descendent != null)
    if type == sub
        say   i--, j--
    else type == ins
        A.append("_")
        B.append(B[j])
        j--
    else type == dns
        A.append(A[i])
        B.append("_")
        i--
return A.inverse, B.inverse, cose
```

A. put front

B. put it front

possibility

co_arse
cou_rse
4
break_
br_ake
4

sub ≤ ins

& sub ≤ del

sub

ins < sub

sub ≤ del

5 5 3

ins

i d s

3.

del < sub

sub ≤ del

335

del < ins

del

2 1 3

del == ins & sub>del

ins