CSE551: Homework 4

(1) Given,

$S = \{s_1, s_2, s_3 \ldots s_n\}$

Is there a solution such that

$$\sum_{i \in I} s_i = \sum_{i \notin I} s_i$$

following Approach can be used to solve this problem.

Since sum of both subsets has to be equal, which means $\sum_{i \in I} s_i = \sum_{i \notin I} s_i = \frac{sum}{2}$

where $sum = \sum_{i \in S} s_i$.

∴ sum % 2 has to be zero, else no sol^n is possible.

Now, if sum / 2 == 0, we will use dynamic programming approach to solve this.

This problem boils down to finding a subset whose sum = $\frac{Sum}{2}$ since automatically the other subset will also have Sum = $\frac{Sum}{2}$ ∴ Total sum of set is sum.

This is similar to ~~partition~~ subset sum problem, finding a subset when sum is $\frac{Sum}{2}$

Here, we will create a table part[n][sum/2] such that the column values would range from $0 \le j \le sum/2$, & Row values will ~~have~~ represent all the elements of subsets. ~~At one~~ Value at part[i][j] will be a boolean value & it will signify that ~~store~~ ~~part~~ whether sum = j is possible from the set which includes all elements from 0 to i in array of elements.

Eg:-

		1	2	3	4	5
2	1	T	F	F	F	F
3	2	T	T	T	F	F
5	3	T	T	T	F	T

Here part[3][3] means whether it is possible to get a sum of 3 ~~by~~ ~~to~~ from the subset of {2, 3, 5} which is true {3}
Similarly part[2][4] means whether it is possible to get sum = 4 from subset of {2, 3} which is false

Our goal is to construct the table above & if the value of part[n][sum/2] = True it means we can construct a subset from set of n elements such that sum = sum/2

Algo to construct the Table :-

if a is the array of elements

$Sum = \sum_{i \in n} a_i$

Let $m = \frac{sum}{2}$

if (sum % 2 != 0)
    No Solution Possible
else
    Construct Table (S, n, m)

Construct Table (S, n, m)
    Part[0][i] = 0 $\forall$ 0 $\leq$ i $\leq$ n
Iterate over every element in set [1 to n]
    here we set values of all those j's to true if subset of set = [S1 ... Si] gives sum = j
    $\therefore \sum_{k \in I} S_k = j$ I = [S1 S2 ... Si]

Iterate over every j column [1 to m]
    if part [i-1][j] = True
        part [i][j] = True
    else if part [i-1] [j - a[i]] = True
        part [i][j] = True.

After all the iterations if the value of part[n][m] = True, it means there is a subset in set whose sum = M which means the other subset will also have sum = M ∴ $M = \frac{sum}{2}$

$$\sum_{i \in I} s_i = \sum_{i \in J} s_i \quad \text{where} \quad I \cup J = S$$

This will give solution to our pblm.
If the value of part[M][n] = false
it means no subset of set has sum = M
which means no disjoint set such that

$$\sum_{i \in I} s_i = \sum_{i \notin I} s_i \text{ is possible.}$$

→ Intuition behind the Table construction

① Firstly we assign all zero's to part[0][i] $1 \le i \le n$ which sum=0 is possible from every possible set since it requires empty set to get sum=0.

② Secondly, Then for the first row we have a single element, therefore we can only get sum = value of that element
Therefore we only update part[1][a[1]]=T
rest all will be false

③ Next element onwards, we see if part[i-1][j] is True, which means

sum = j is possible by using elements from 1 to i-1,
∴ Of course sum = j will also be possible by using elements 1 to i (since 1 to i-1 is a subset of 1 to i)
∴ we update part[i][j] = true.

(4) Now, if ~~part[i][j]~~ = part[i-1][j] = false which means sum = j is not possible by using elements from 1 to i-1, so what if adding element a[i] can lead to sum = j.
∴ we check if ~~sum j - a[i]~~ sum = j - a[i] is possible by using elements 1 to i-1, if that is possible then adding a[i] to (j - a[i]) + a[i] will give j, which means we can include ith element and get sum = j
∴ we update part[i][j] = true if part[i-1][j - a[i]] = true

(5) In this way we construct the table.

We can see from the algo that we ~~iterate twice~~ require 2 for loops to

Construct the Table.
∴ One loop goes from 0 to m & other goes from 0 to n
∴ The complexity of the algo is O(mn) where $m = \frac{sum}{2}$

(2) To find, All LCS of 2 strings using DP.

We know how DP can be used to find LCS of 2 strings.
Following is the algo & example to obtain LCS of 2 strings.

```
LCS(S, T)
    c[i][0] = 0      1 ≤ i ≤ m
    c[0][i] = 0      1 ≤ i ≤ n
    m = len(S)
    n = len(T)
for i in range (1, m)
    for j in range (1, n)
        if S[i] == T[j]
            c[i][j] = c[i-1][j-1] + 1
        else
            c[i][j] = max(c[i-1][j], c[i][j-1])
```

Return c

Here c is the table that contains length of LCS at c[m][n]

Eg:-

abcdaf, acdbcf

LCS: ~~acaf~~ abcf

Table:

		a	b	c	d	a	f
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
b	0	1	1	2	2	2	2
c	0	1	1	2	3	3	3
d	0	1	1	2	3	3	3
b	0	1	2	2	3	3	3
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

length of LCS = 4 ∵ c[6][6] = 4

To find the subsequence we traverse as shown in the table above

```
ReadLCS(c, i, j)
    if i == 0 or j == 0:
        return ""
    else if S[i] == T[j]:
        return ReadLCS(c, i-1, j-1) + S[i]
    else:
```

```
if c[i-1][j] ≥ c[i][j-1]
    return ReadLCS (c, i-1, j)
else
    return ReadLCS (c, i, j-1)
```

Intuition behind it

(1) Firstly, if we get $s[i] == t[j]$ it means that element has to be included (since its equal) in the set
∴ we add that to set & Recursively iterate for remaining $i-1$ & $j-1$ elements

(2) If $s[i] != t[j]$, it means the value has come either from Top or left whichever is Maximum.
∴ we check which is max $c[i-1][j]$ or $c[i][j-1]$ & accordingly go in that direction, & if both are equal we can go in any direction; here I choose to go left by default ∴ ≥ I have written ≥ in $c[i-1][j] \geq c[i][j-1]$ which mean go left.

Now, based on above algo, we have

seen that when $S[i] != T[j]$ & $C[i-1][j] = C[i][j-1]$ we can go in any direction to find the subsequence, but here is the trick. This direction defines what subsequence we get. Going in diff directions may result in different LCS.

We will take the example given above

abdaf & acdbcf

we have abdf as LCS

But there is no another LCS as well

acdf

We will rewrite the Table & traverse this sequence

		a	b	c	d	e	f
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
d	0	1	1	2	3	3	3
b	0	1	2	2	3	3	3
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

Path B

Path A

We see that

going by Path B gives a cd f &

going by Path A gives a b c f

So, we have find such different paths to get different LCS.

∴ If both values are equal

C[i-1][j] == C[i][j-1] we traverse recursively to both the directions to find different paths & eventually different LCS.

Algo:-

```
ReadAllLCS(C, i, j)
    if i==0 or j==0:
        return ""
    elif S[i]==T[j]
        return ReadAllLCS(C, i-1, j-1) +
                                S[i-1]
    else:
        if C[i-1][j] > C[i][j]
            return
```

```
ReadAllLCS(C, i, j)
    if i==0 or j==0
        return ""
    elif S[i] == T[j]:
        return set([z + S[i] for z in
                ReadAllLCS(C, i-1, j-1)])
    else:
        R = set()
```

(11)

(1)

if c[i-1][j] ≥ c[i][j-1]
    R.update(ReadAllLCS(c, i-1, j))
else
if c[i][j-1] ≥ c[i-1][j]
    R.update(ReadAllLCS(c, i, j-1))
Return R.

(2)

Here we see that for we have written equal to sign for both if's (1) & (2) to ensure that we traverse in both directions if values are equal. this gives us multiple LCS.

- R.update(*) adds * to set.

R.Set

∴ Using above algo by recursively iterating for all possible LCS, we find the solution to given problem.

(3) To convert a string to another string, we can insert, remove or replace any characters.
To find minimum such operations. Here we can use DP which will require O(mn) complexity
m = length of string 1
n = length of string 2

Example:

			a	b	c	d	e	f	(str1)
		0	1	2	3	4	5	6	
	a	1	0	1	2	3	4	5	
(str2)	b	2	1	1	2	3	4	5	
	c	3	2	2	1	2	3	4	
	e	4	3	3	2	2	2	3	
	d	5	4	4	3	2	3	3	

The construct of table is explained below: str1[i] == str2[j]

if T[i][j] == T[i-1][j-1]

T[i][j] = T[i-1][j-1]

This means if two characters are same in both strings, we don't need to perform any operations. The no. of operations would be same as converting str1[1:i-1] to str2[1:j-1]

If 2 characters aren't equal, we can either update, delete or substitute characters.

We'll compare the costs of all & take the minimum

$$T[i][j] = \min(T[i-1][j], T[i][j-1], T[i-1][j-1]) + 1$$

Here,

if we copy value from T[i-1][j]+1 to T[i][j] it means we are inserting a new character, if we use T[i][j-1]+1 & for T[i][j] it means we are deleting a character & if we use T[i-1][j-1]+1, it means we are substituting a value.

The algo is as follows :-

```
for i in range(0, n+1):
    for j in range(0, m+1):
        if (i==0)
            T[i][j] = j
        else if (j==0)
            T[i][j] = i

        elseif (str[i] == str[j])
            T[i][j] = T[i-1][j-1]

        else
            T[i][j] = min(T[i-1][j], T[i][j-1], T[i-1][j-1]) + 1
```

Complexity = O(mn)

The above algorithm converts str1 of length m to str2 of length n.

↓ $T[i][j] = T[i-1][j] + 1$ implies Insert

→ $T[i][j] = T[i][j-1] + 1$ implies delete

↘ $T[i][j] = T[i-1][j-1] + 1$ implies substitute

Since while inserting we are adding one more character, & going from i-1 to i means one character is increased in str2, therefore too match in str1, we add one character.

Going from j-1 to j means we are adding one character to str1, but str2 ~~has~~ ~~is unchanged~~ as is unchanged. ∴ we need to delete this character.

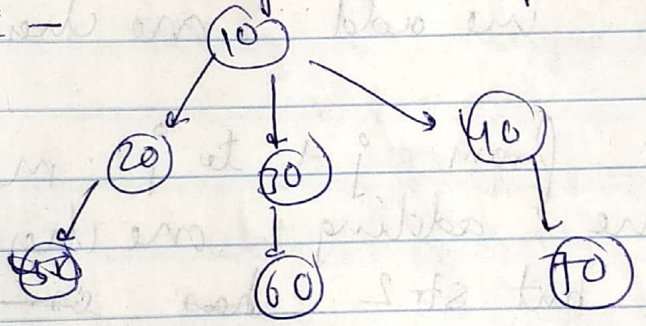
While ~~drago~~ going diagonally we add ~~both~~ characters to both strings, ∴ If current characters are same, we don't need to do any operation & copy the value from $T[i-1][j-1]$ to $T[i][j]$

But if the two characters are different we will replace char from str1 to the character in str2

$T[i][j] = T[i-1][j-1] + 1$

(4) Largest Independent set of a tree is the set that includes nodes such that no 2 nodes have common edges between them. And such a set with max no of nodes is called largest Independent set

Eg:-



Largest Independent Set = {10, 50, 60, 70}

Here we can see that if a node x is included in the set, its children can't be included, only grandchildren can be, since children shares common edge but grand children don't.

So, the idea here is that if x is a member of LIS, then
LISS = 1 + ~~sum of~~ LISS of all grandchildren of x.

& if x is not a member of LIS then
LISS = Sum of LISS of all children

LISS → largest Independent set size
LIS → largest Independent set.

∴ LISS(x) = Max{(1 + sum of LISS of all grandchildren of x), sum of LISS for all children of x}

Solving this Recursively, will lead to calculating repeated sub problems again & again, therefore we store solutions to such problems using DP as given below.

We will both Recursive & ~~dp~~ DP solutions to this.

Recursive Solution:-

Let node be the class which has data, left pointer to left node & right pointer to right node.

```
class node {
    int data;
    node* left, *right
}

LISS (node *root)
    if (root == null)
        return 0;
    size_excl = LISS(root->left) +
                LISS(root->right);
```

(This calculate size of LISS excluding current node)

```
size_incl = 1
if (root->left)
    size_incl += LISS(root->left->left) +
                 LISS(root->left->right)
if (root->right)
    size_incl += LISS(root->right->left) +
                 LISS(root->right->right)
return max(size_incl, size_excl)
```

In the above code, we will calculate LISS of same node again & again & Instead of this we will store LISS of node as one more parameter of class node & use that if its already computed

```
class node {
    int data;
    node* left, *right
    int LISS
}
```

DP solution:

```
LISS( node *root)
    if (root == null)
        return 0;
    if (root->liss)
        return root->liss
```

(this is the set that distinguishes DP from Recursive sol", using the stored value instead of computing again)

```
if (root->left == null && root->right == null)
    return (root->liss=1);

liss_end = LISS(root->left) +
           LISS(root->right)

liss_incl = 1
if (root->left)
    liss_incl += LISS(root->left->left) +
                 LISS(root->left->right)
if (root->right)
    liss_end += LISS(root->right->left) +
                LISS(root->right->right)

root->liss = max(liss_incl, liss_end)
return root->liss;
```

(lastly before returning we store liss value & then return)

The Time Complexity drastically Reduces from exponential to linear.

O(n) where n is no of nodes in given tree.