Assignment 5

Due on Tuesday, 5 March, by 11.59pm

Problem 1

(8 pts)

(a) **(6 pts)**
A naive solution would be to split into 3 cases, one for each of the 3 possible operations
(1) Including characters in both strings(Until no more characters in one of the strings)
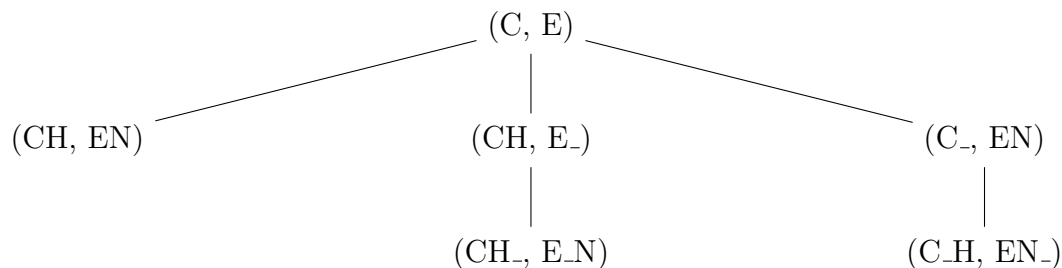(2) Include character of first string and add gap to second string(Until no more characters in first string)
(3) Include character of second string and add gap to first string(Until no more characters in second string)
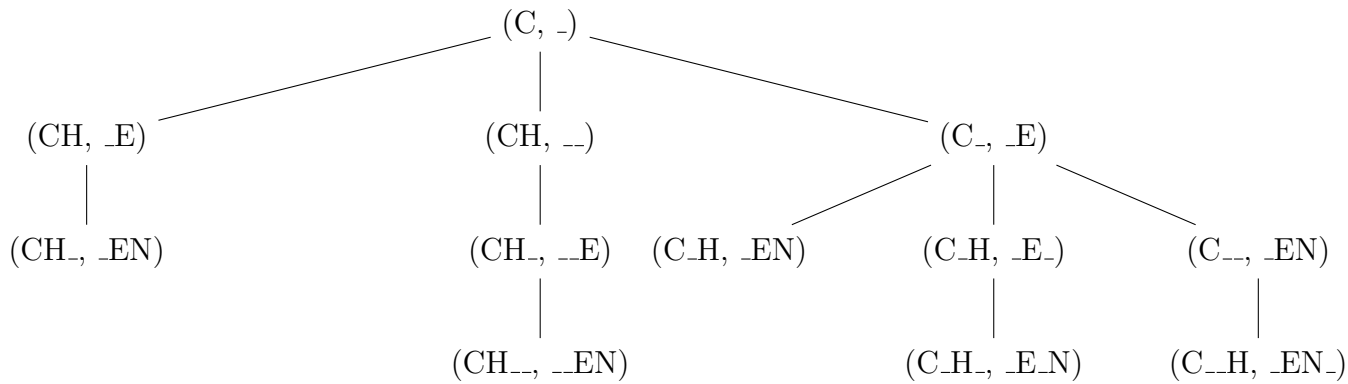OPT(i, j) = $\min(\alpha_{x_i y_i} + OPT(i-1, j-1),\ \delta + OPT(i-1, j),\ \delta + OPT(i, j-1))$
This yields a time complexity of $O(3^{m+n})$
Example: String$_1$: CH, String$_2$: EN
We have 3 possibilities for our first layer, (C, E), (C, _), (_, E)

```
                        (C, E)
              /            |            \
       (CH, EN)        (CH, E_)        (C_, EN)
                           |                |
                      (CH_, E_N)       (C_H, EN_)
```

We have 3 leaves, (CH, EN), (CH_, E_N), (C_H, EN_)

(C, ‿)

(CH, ‿E)   (CH, ‿‿)   (C‿, ‿E)

(CH‿, ‿EN)   (CH‿, ‿‿E)   (C‿H, ‿EN)   (C‿H, ‿E‿)   (C‿‿, ‿EN)

(CH‿‿, ‿‿EN)   (C‿H‿, ‿E‿N)   (C‿‿H, ‿EN‿)

We have 5 leaves, (CH‿, ‿EN), (C‿H, ‿EN), (C‿H‿, ‿E‿N), (C‿‿H, ‿EN‿)

(‿, E)

(‿C, EN)   (‿C, E‿)   (‿‿, EN)

(‿CH, EN‿)   (‿CH, E‿N)   (‿CH, E‿‿)   (‿C‿, E‿N)   (‿‿C, EN‿)
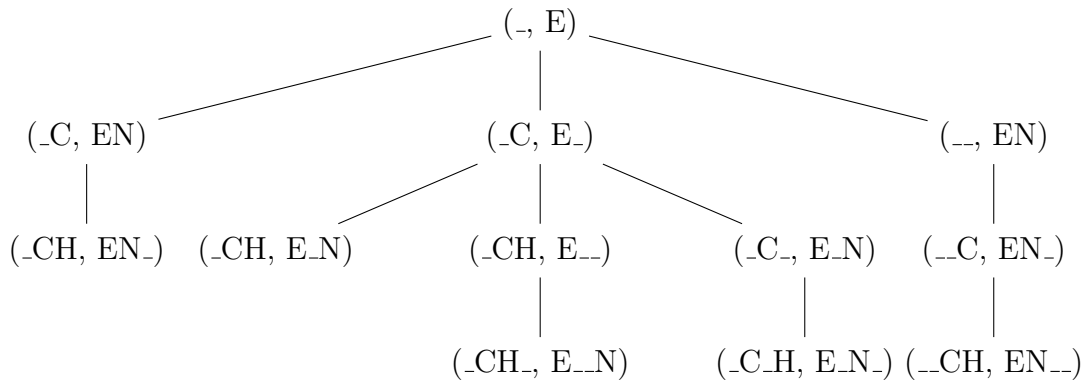
(‿CH‿, E‿‿N)   (‿C‿H, E‿N‿) (‿‿CH, EN‿‿)

We have 5 leaves, (‿CH, EN‿), (‿CH, E‿N), (‿CH‿, E‿‿N), (‿C‿H, E‿N‿), (‿‿CH, EN‿‿)

(b) (2 pts)

	‿	A	L	G	O
T	1	2	3	4
E	2	2	3	4
S	3	3	3	4
T	4	4	4	4

1.

The optimal alignment is zero gaps:

Alignment$_1$: ALGO

Alignment$_2$: TEST

2. The cost is the value in the bottom right corner, 4

Cost is 4 since there are 4 sets of characters that are different

Problem 2

(12 pts)

(a) Indices (i, j) store the minimum number of cuts needed to make the substring from i to j a set of palindromes.

```
def min_palindrome(s):
    sections = [∞] * len(s)
    for j in [1, len(s)]:
        min_sections = j
        for i in [1, j]:
            if isPali(i, j):
                min_sections = min(min_sections, sections[i - 1] + 1)
        sections[j] = min_sections
```

I'll test by running it on the string 'coffee'
Each iteration runs with a fixed endpoint j and a increasing start point i.

First iteration: $(1 \to 1, 1)$

1	∞	∞	∞	∞	∞

Second iteration: $(1 \to 2, 2)$

1	2	∞	∞	∞	∞

Third iteration: $(1 \to 3, 3)$

1	2	3	∞	∞	∞

Fourth iteration: $(1 \to 4, 4)$

1	2	3	3	∞	∞

Fifth iteration: $(1 \to 5, 5)$

1	2	3	3	4	∞

Sixth iteration: $(1 \to 6, 6)$

1	2	3	3	4	4

Let $n = w.length$
The time complexity is $O(n^2)$, as each iteration of the outer loop goes from i + 1 to
the end of the array, resulting in $\sum_{j=1}^{n} j$

(b) Assume S is 1-indexed
We either add the new character as a standalone or use it as a palindrome with the
previous characters in the string.
OPT(i) = min(OPT(i - 1) + 1, min(OPT(j - 1) + 1 for j in [1, i - 1] if isPali(j, i)))
Claim: sections[i] = minimum palindromic substrings of [1, i]
Base Case: i = 1
Our algorithm is correct since for the single first character, it'll return 1
By definition, OPT(1) = 1 = f(1)
Therefore, our base case is correct
Inductive Step: Assume f(i) = OPT(i) = sections[i] for all i ≤ k
Show f(k + 1) is correct
We know OPT(k + 1) = min(OPT(k) + 1, min(OPT(j - 1) + 1 for j in [1, k] if isPali(j,
k + 1)))
f(k + 1) = min(sections[i - 1] + 1) for i in [1, k + 1] if isPali(i, k + 1)
These are the same, so our algorithm is correct
Therefore, our algorithm produces the optimal solution

Problem 3

(10 pts)

(a) Since we can't include the direct children of a manager, then we need to include sub-
trees another layer deeper.
The optimal solution could be to include the current person, but if the manager was
accounted for, more enjoyment could be gained.
This isn't greedy since local best choices don't always lead to the global best choice.

(b) I used memoization to store the maximum enjoyment for each person.
The idea is to store the two maximum enjoyments for each person in a map, one for
the person potentially included in the enjoyment and one for the person not allowed
in the enjoyment.
e_i = enjoyment if person can be included and its associated list of people
e_e = enjoyment if person can't be included and its associated list of people

```
Map(person, {true: e_i, false: e_e}) = {}
def max_enjoyment(person, canInclude) -> int:
    if person in Map:
        return Map(person, canInclude)

    ci = person.enjoyment
    for child in person.children:
        ci += max_enjoyment(child, false)

    ce = 0
    for child in person.children:
        ce += max_enjoyment(child, true)
    Map(person, {ci, ce})
    return canInclude ? max(ci, ce) : ce
```

Let i be the level where i = 0 is the bottom most person
The optimal solution is OPT(i) = max(sum(OPT(i - 1)), sum(OPT(i - 2)) + enjoyment$_i$)
We compute each peron's enjoyment included and excluded once, so the time complex-
ity is $O(n)$, where n = number of people
Now, I'll show the algorithm is optimal
Base Case: person is bottom of tree, show f(0) = OPT(0)
Then, e_i = person.enjoyment and e_e = 0
This is optimal since OPT(i) = max(sum(0), sum(0) + enjoyment$_i$) = max(0, enjoyment$_i$)
= enjoyment$_i$
Therefore, our base case is correct
Inductive Hypothesis: Assume f(i) = OPT(i) for i ≤ k
Inductive Step: Show f(k + 1) = OPT(k + 1)
Our algorithm computes sum(f(k)) and sum(f(k - 1)) + enjoyment$_{k+1}$

It then returns the maximum of the two since we allow the inclusion of the current person

OPT(k + 1) = max(sum(OPT(k)), sum(OPT(k - 1)) + enjoyment$_{k+1}$)

Therefore, the two are the same, so our algorithm is correct

Therefore, our algorithm produces the optimal solution