**CS 331: Algorithms and Complexity (Spring 2024)**
**Unique Number: 50930, 50935 50940, 50945**

**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)
Let `obj` be the the cost for comparing characters and `gap` be the cost of adding a gap

```
def align(i, j):
    if i == s1.length and j == s2.length:
        return 0
    elif i == s1.length:
        return gap(s2[j]) + align(i, j + 1))
    elif j == s2.length:
        return gap(s1[i]) + align(i + 1, j))

    (1) both = align(i + 1, j + 1) + obj(s1[i], s2[j])
    (2) first = align(i + 1, j) + gap(s1[i])
    (3) second = align(i, j + 1) + gap(s2[j])
    return min(both, first, second)
```

This yields a time complexity of $O(3^{m+n})$ since there are 3 recursive calls per function call

**(b) (2 pts)**

1.

| _ | A | L | G | O |
|---|---|---|---|---|
| T | 1 | 2 | 3 | 4 |
| E | 2 | 2 | 3 | 4 |
| S | 3 | 3 | 3 | 4 |
| T | 4 | 4 | 4 | 4 |

The optimal alignment is zero gaps:
ALGO
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 $\sum_{j=1}^{n} j \approx \frac{j^2}{2}$

**(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 as the `min_sections` variable will stay 1 as it's the minimum
By definition, OPT(1) = 1 = f(1)
Therefore, our base case is correct
Inductive Hypothesis: Assume f(i) = OPT(i) = sections[i] for all i $\leq$ k
Inductive Step: 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)
If the new character can't be used as a palindrome, then the minimum number of sections is just the previous string with this new character as a standalone
This yields the same result as the algorithm
If the new character can be used as a palindrome, then the minimum number of sections would be the minimum number of sections of the string before the start of this new palindrome plus 1
These is the logic of our algorithm, and it matches the optimal solution
Therefore, f(k + 1) = OPT(k + 1)
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 subtrees 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 memoized top-down approach 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, {e_i, e_e})
def max_enjoyment(person, canInclude) -> (int, List):
```

```
if person in Map:
    return Map(person, canInclude ? max(e_i, e_e) : e_e)

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(OPT(i - 1), OPT(i - 2) + enjoyment$_i$)
We compute each peron's enjoyment included and excluded once, so the time complexity 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(0, 0 + enjoyment$_i$) = enjoyment$_i$
Therefore, our base case is correct
Inductive Hypothesis: Assume f(i) = OPT(i) for i $\leq$ k
Inductive Step: Show f(k + 1) = OPT(k + 1)
We know OPT(k + 1) = max(OPT(k), OPT(k - 1) + enjoyment$_{k+1}$)
Our algorithm computes $e_e$ = f(k) and $e_i$ = f(k - 1) + enjoyment$_{k+1}$
It then returns max($e_e$, $e_i$) since we allow the inclusion of the current person
Since f(k) = OPT(k) and f(k - 1) = OPT(k - 1), then f(k + 1) = max(OPT(k), OPT(k - 1 ) + enjoyment$_{k+1}$)
Therefore, f(k + 1) = OPT(k + 1)
Therefore, our algorithm produces the optimal solution