DSA Homework

Katie Foster

Part 1

(a)

s1 = first string
s2 = second string
i = position in s1
j = position in s2

Equations:
The value function represents the minimum edit distance between two strings.

Base cases:
M(i, len(s2)) = len(s1)-len(s2)
M(len(s1), j) = len(s2)-len(s1)

General:
M(i, j) = {
    M(i+1, j+1) if i==j (don't add to edit distance if i and j are equal)
    min(M(i, j+1)+1, M(i+1, j)+1, M(i+1, j+1)+1) if i!=j
    (take minimum of the cost of an insert/delete vs a replacement)

Argue correctness using principle of optimality:

Base case: The base case is achieved when either i or j has reached the end of its respective string. The minimum edit distance that two strings can be from each other is the absolute value of the difference of their lengths. So if there were two strings s1=s and s2=s+"xyz", then the distance between them would be 3, the length of "xyz".

General: The algorithm covers the full set of edit possibilities of: no edit being needed, a substitution being needed, an insertion being needed and a delete being needed. When the characters match, it means no edit is needed, so the algorithm increments to the next index in the string without adding to the edit distance. When the two characters don't match, it means an edit is needed. The algorithm then compares the minimum edit distance between the three possible edits (substitution, insertion, deletion), and selects the edit that would produce the overall minimum edit distance. Then the algorithm adds 1 to account for the cost of the edit. Therefore since it selects the minimum of all possible edits, and covers all possible situations, the algorithm finds the correct minimum edit distance.

(b)

Runtime Analysis:
This algorithm creates a table with size len(s1) by len(s2) where each entry in the table represents a possible edit distance between the two strings. Since the algorithm must fill out each entry in the table and edit the entry, this means that it is performing an O(1) operation O(len(s1)*len(s2)) times. Therefore, the runtime of the algorithm is O(len(s1) *len(s2)).

Part 2

(a)

s1 = pattern string (eg: lemondrop)
s2 = wildcard string (eg: l*dr*p*)
i = position in s1
j = position in s2

Equations:
The value function returns true or false value, which represents whether or not the *s in s2 can be replaced with any string to make s2 match s1.

Base cases:

```
W(j=len(s2)) = {
    // If we get to the end of wildcard string (and does not end in *)
    return True if s1[len(s1)] == s2[len(s2)] and s1[i] is also the end
    return False if s1[len(s1)] != s2[len(s2)]
```

General:

```
W(i, j) = {
    If s1[i] == s2[j], W(i+1, j+1)
    If s1[i] != s2[j], and *, if any(W(i:n, j+1)) == true, return True
    If s1[i] != s2[j], and !*, return False
```

n = length of s1

Argue correctness using principle of optimality:
Base case: The base case happens when we get to the end of the wildcard string and it does not end in a *. In this case, there are only two possibilities: if the last characters in each string match, then the algorithm returns true, and if they do not match, the algorithm returns false.

General: The general case for the algorithm considers the cases of if either s2[j] is a wildcard character, or if j has not gotten to the end of s2 yet. If s2[j] is a wildcard, and j is the end of s2, then it does not matter what is at the end of s1, because the wildcard can be replaced by the end of s1 in any case. The other case is if j has not reached the end of s2 yet. This case is a little bit more complicated, and has three subcases. The first subcase is that the two characters are equal, in which case, the algorithm makes the recursive call, but with i and j incremented. If the two characters are not equal, then either the character in s2 is a wildcard character or it is not. If it is not a wildcard character, the algorithm returns false, since it cannot be replaced with anything. If the character is a wildcard character, then the algorithm makes a recursive call iterating over i, to check if there is any i from the current position of i to the end of s1 that will allow the algorithm to return true. If the algorithm finds an i that returns true, the algorithm will return true because it now knows that such a case exists. If it does not find a case that returns true, the algorithm will return false. Therefore since the algorithm covers all cases, it must be correct.

(b)

<u>Runtime Analysis:</u>

This algorithm creates a table of size len(s1) by len(s2). Since the algorithm goes through each entry in the table, performing an O(1) operation on each entry, the overall runtime of the algorithm is $O(\text{len}(s1)^2 * \text{len}(s2))$.