Problem 1

We use dynamic programming. We pass the string w as an array `W[1..n]` where `n` is the string's length so we could taking the slice of the string as `W[i..j]` (it means the substring from `i` to `j` (inclusive)).

We first store an 2-d array called `IsStringInLArray[1..n, 1..n]` where `IsStringInLArray[i, j]` is a boolean that indicates whether `W[i..j]` (a substring of w from `i` position to `j` position, inclusive) is in the language L.

We then define an 1-d array called `MinCostArray[1..n+1]` for memoization. The `MinCostArray[i]` means the minimum splitting cost (that is, the splitting cost defined in the problem) the substring `W[i..n]`. Few things to notice:

- `MinCostArray[n+1]` is a special case, in this case we want the minimum cost of splitting the substring `W[n+1..n]`, this could be better thought as the minimum cost of splitting the string `ε`, which is just 0.
- If `W[i..n]` is not in L^*, the `MinCostArray[i]` will be `NaN`. This indicates there is no valid splitting.
- Since the `MinCostArray[i]` depends on all `MinCostArray[s+1]` where `i ⩽ s ⩽ n`. That means we need to calculate the `MinCostArray[i]` in the order where `i` goes from `n` down to `1`.
- We define that taking minimum between a valid number `a` and a `NaN` is that number `a`, that is, `min(a, NaN) = min(NaN, a) = a`

We then return `MinCostArray[1]`, this is just what we want, the minimum splitting cost of `W[1..n] = W`. If it's `NaN`, that means $w \notin L^*$.

```
MinCost(W[1..n]):
    if W = ε:
        return cost(0)

    declare IsStringInLArray[1..n, 1..n]

    for i ← 1 to n:
        for j ← i to n:
            IsStringInLArray[i, j] = IsStringInL(W[i..j])

    declare MinCostArray[1..n + 1]

    MinCostArray[n + 1] = 0

    for i ← n to 1:
        MinCostArray[i] ← NaN
        for s ← i to n:
            if IsStringInLArray[i, s]:
                if MinCostArray[s + 1] is not NaN:
                    MinCostArray[i] ← min(
```

```
                        MinCostArray[i],
                        cost(s - i + 1) + MinCostArray[s + 1]
                    )

    return MinCostArray[1]
```

The time complexity of the algorithm is $O(n^3)$. The first part when we calculate the value in the `IsStringInLArray`, we have two `for` loop, and the operation `IsStringInL` in worst case takes $O(n)$ time, therefore we take $n^2 \cdot O(n) = O(n^3)$ for the first part. For the second part when we calculate the value in the `MinCostArray`, we have two `for` loops, and in each loop, all the array access and `min` function takes $O(1)$, so the time complexity for the second part is just $n^2 \cdot O(1) = O(n^2)$ and therefore the total runtime for entire algorithm is $O(n^3)$.

Problem 2

For clarify of the pseudocode, we define a function `split(w)` that take any string:

$$\text{split}(w) = \{(u, v) \mid uv = w\}$$

It's a function that returns all possible split of the string w. To memoize calculations done previously by our `IsStringInRegExp(w, r)`, we have a two dimensional hash map `IsStringInRegExpMemory` where `IsStringInRegExpMemory[w, r]` is the stored calculation of `IsStringInRegExp(w, r)`. If there is nothing stored in the `IsStringInRegExpMemory[w, r]` , then it's `undefined`. Just to make the code neater, we define a function `Store(value)` that write the value into `InStringInRegExpMemory` and then return it input `value`. Just to clarify, the `Store(value)` function is defined inside the function `IsStringInRegExp`, so the `w` and `r` used in the `Store` is just the `w` and `r` in the outer function `IsStringInRegExp` (That is basically what a closure is). If you don't understand, just think `Store` as a function that will (1) store the value it got to the corresponding place in `IsStringInRegExpMemory` (2) then just return the what it received.

Then, the pseudocode is below: (notice the code like `w = ε` is a boolean expression but not a assignment.

```
declare IsStringInRegExpMemory[w, r]

IsStringInRegExp(w, r):

    Store(value):
        IsStringInRegExpMemory[w, r] ← value
        return value

    if IsStringInRegExpMemory[w, r] is not undefined:
        return IsStringInRegExpMemory[w, r]
    if r = ε: return Store(w = ε)
    if r = a: return Store(w = a)
    if r = ∅: return Store(false)
    if r = s + t:
        return Store(IsStringInRegExp(w, s) or IsStringInRegExp(w, t))
    if r = st:
```

```
        foreach (ws, wt) ← split(w):
            if IsStringInRegExp(ws, s) and IsStringInRegExp(wt, t):
                return Store(true)
        return Store(false)
    if r = s*:
        if w = ε: return Store(true)
        else:
            foreach (ws, wr) ← split(w):
                if IsStringInRegExp(ws, s) and IsStringInRegExp(wr, r):
                    return Store(true)
            return Store(false)
    return IsStringInRegExpMap[w, r]
```

The complexity analysis is not required, this is just my notes (not very rigorous) below to help you understand that it's indeed polynomial:

> Since for each recursive call, we are at least making the $|r|$ smaller for the subproblem. (we don't necessarily make $|w|$ smaller since it's possible for it to split into ϵ and itself). So, the recursive call will get to a base case eventually.
>
> For the upper bound of the complexity, since for the parameter w' and r' for the recursive call will satisfy that w' is a substring of w and r' is a substring of r. All the possible parameter that w', r' that the `IsStringInRegExp` going to receive will all be the substring of initial w and r. For w, it has $O(|w|^2)$ possible substrings, same for r. Then, there are thus only $O(|w|^2|r|^2)$ possible parameter combinations that could be passed into the function `IsStringInRegExp`.
>
> In the function, as we could see, the worst case scenario there will be $O(|w|)$ ways to split the string w, then the function body itself only take $O(|w|)$ (think the recursive calls are memoized and are $O(1)$ operations since their time cost will all be considered in the end).
>
> Then, since we need to calculate all the possible $O(|w|^2|r|^2)$ input parameter combination and each case take $O(|w|)$. The total time complexity will be $O(|w|^3|r|^2)$, which is polynomial.