Question 2

You can implement this using dynamic programming in a divide and conquer fashion with memoization.

```
function stringCanBeA(str) {
  options = computePossible(str);
  return true if ”a” in options else false
}

lookupTable = {} // a map between string and possible options.

function computePossible(str) {
  // No need to memoize smallest problem, overhead on coordination is probably
  // higher than returning str.
  if (length(str) <= 1) {
    return str
  }

  // Memoize function
  if (lookupTable[str]) {
    return lookupTable[str]
  }

  options = []
  for (mid in 1 to length(str)-2) {
    leftOptions = computePossible(str[:mid])
    rightOptions = computePossible(str[mid:])

    for (l in leftOptions) {
      for (r in rightOptions) {
        add multiply(l, r) to options if not present
      }
    }
  }

  lookupTable[str] = options

  return options
}
```

At each step there is n-1 subproblems, however, most of the subproblems are duplicates through memoization. For instance `computePossible(”abc”)` has the subproblems `computePossible(”ab”)` and `computePossible(”bc”)`, which shares half the subproblems with `computePossible(”bcd”)`.

The merge operation at each invocation of `computePossible` has a maximum of 3 elements in `leftOptions` and `rightOptions` with a worst case of $O(9) = O(1)$.

$T(n) = \sum_{i=1}^{n} T(i) + T(n-i)$