<aside> 💡 **Question 1**

Given two strings s1 and s2, return *the lowest **ASCII** sum of deleted characters to make two strings equal*.

Example 1:

Input: s1 = "sea", s2 = "eat"

Output: 231

Explanation: Deleting "s" from "sea" adds the ASCII value of "s" (115) to the sum.

Deleting "t" from "eat" adds 116 to the sum.

At the end, both strings are equal, and 115 + 116 = 231 is the minimum sum possible to achieve this.

</aside>

```
class Solution {
    public int minimumDeleteSum(String s1, String s2) {

        // Make sure s2 is smaller string
        if (s1.length() < s2.length()) {
            return minimumDeleteSum(s2, s1);
        }

        // Case for empty s1
        int m = s1.length(), n = s2.length();
        int[] currRow = new int[n + 1];
        for (int j = 1; j <= n; j++) {
            currRow[j] = currRow[j - 1] + s2.charAt(j - 1);
        }

        // Compute answer row-by-row
        for (int i = 1; i <= m; i++) {

            int diag = currRow[0];
            currRow[0] += s1.charAt(i - 1);

            for (int j = 1; j <= n; j++) {

                int answer;

                // If characters are the same, the answer is top-left-diagonal value
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    answer = diag;
```

```
            }

            // Otherwise, the answer is minimum of top and left values with
            // deleted character's ASCII value
            else {
                answer = Math.min(
                    s1.charAt(i - 1) + currRow[j],
                    s2.charAt(j - 1) + currRow[j - 1]
                );
            }

            // Before overwriting currRow[j] with answer, save it in diag
            // for the next column
            diag = currRow[j];
            currRow[j] = answer;
        }
    }

    // Return the answer
    return currRow[n];
  }
}
```

<aside> 💡 **Question 2**

Given a string s containing only three types of characters: '(', ')' and '*', return true *if* s *is* ***valid***.

The following rules define a **valid** string:

- Any left parenthesis '(' must have a corresponding right parenthesis ')'.
- Any right parenthesis ')' must have a corresponding left parenthesis '('.
- Left parenthesis '(' must go before the corresponding right parenthesis ')'.
- '*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string "".

Example 1:

Input: s = "()"

Output:

true

</aside>

```
class Solution {
    public boolean checkValidString(String s) {
        int lo = 0, hi = 0;
        for (char c: s.toCharArray()) {
            lo += c == '(' ? 1 : -1;
            hi += c != ')' ? 1 : -1;
            if (hi < 0) break;
            lo = Math.max(lo, 0);
        }
        return lo == 0;
    }
}
```

<aside> 💡 **Question 3**

Given two strings word1 and word2, return *the minimum number of* ***steps*** *required to make* word1 *and* word2 *the same*.

In one **step**, you can delete exactly one character in either string.

Example 1:

Input: word1 = "sea", word2 = "eat"

Output: 2

Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".

</aside>

```
public class Solution {
    public int minDistance(String s1, String s2) {
        int[] dp = new int[s2.length() + 1];
        for (int i = 0; i <= s1.length(); i++) {
            int[] temp=new int[s2.length()+1];
            for (int j = 0; j <= s2.length(); j++) {
                if (i == 0 || j == 0)
                    temp[j] = i + j;
                else if (s1.charAt(i - 1) == s2.charAt(j - 1))
                    temp[j] = dp[j - 1];
                else
                    temp[j] = 1 + Math.min(dp[j], temp[j - 1]);
            }
            dp=temp;
        }
        return dp[s2.length()];
    }
}
```

<aside> 💡 **Question 4**

You need to construct a binary tree from a string consisting of parenthesis and integers.

The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root's value and a pair of parenthesis contains a child binary tree with the same structure. You always start to construct the **left** child node of the parent first if it exists.

</aside>

```
class Solution {
    public TreeNode str2tree(String s) {
        return dfs(s);
    }

    private TreeNode dfs(String s) {
        if ("".equals(s)) {
            return null;
        }
        int p = s.indexOf("(");
        if (p == -1) {
            return new TreeNode(Integer.parseInt(s));
        }
        TreeNode root = new TreeNode(Integer.parseInt(s.substring(0, p)));
        int start = p;
        int cnt = 0;
        for (int i = p; i < s.length(); ++i) {
            if (s.charAt(i) == '(') {
                ++cnt;
            } else if (s.charAt(i) == ')') {
                --cnt;
            }
            if (cnt == 0) {
                if (start == p) {
                    root.left = dfs(s.substring(start + 1, i));
                    start = i + 1;
                } else {
                    root.right = dfs(s.substring(start + 1, i));
                }
            }
        }
        return root;
    }
}
```

<aside> 💡 **Question 5**

Given an array of characters chars, compress it using the following algorithm:

Begin with an empty string s. For each group of **consecutive repeating characters** in chars:

- If the group's length is 1, append the character to s.
- Otherwise, append the character followed by the group's length.

The compressed string s **should not be returned separately**, but instead, be stored **in the input character array chars**. Note that group lengths that are 10 or longer will be split into multiple characters in chars.

After you are done **modifying the input array,** return *the new length of the array*.

You must write an algorithm that uses only constant extra space.

Example 1:

Input: chars = ["a","a","b","b","c","c","c"]

Output: Return 6, and the first 6 characters of the input array should be:
["a","2","b","2","c","3"]

Explanation:

The groups are "aa", "bb", and "ccc". This compresses to "a2b2c3".

</aside>

```
class Solution {
    public int compress(char[] chars) {
        int i = 0, res = 0;
        while (i < chars.length) {
            int groupLength = 1;
            while (i + groupLength < chars.length && chars[i + groupLength] == chars[i]) {
                groupLength++;
            }
            chars[res++] = chars[i];
            if (groupLength > 1) {
                for (char c : Integer.toString(groupLength).toCharArray()) {
                    chars[res++] = c;
                }
            }
            i += groupLength;
        }
        return res;
    }
}
```

<aside> **Question 6**

Given two strings s and p, return *an array of all the start indices of* p*'s anagrams in* s. You may return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: s = "cbaebabacd", p = "abc"

Output: [0,6]

Explanation:

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

</aside>

```
class Solution {
    public List<Integer> findAnagrams(String s, String p) {
        List<Integer> res = new ArrayList<>();
        if (s.length() < p.length()) return res;
        int[] map = new int[128];
        for (char c : p.toCharArray()) {
            map[c]++;
        }
        // two pointers to track the window
        int left = 0, right = 0;
        char[] chars = s.toCharArray();
        int m = s.length(), n = p.length(), count = n;
        while (right < m) {
            if (map[chars[right++]]-- > 0) count--;
            if (count == 0) res.add(left);
            if (right - left == n && map[chars[left++]]++ >= 0) count++;
        }
        return res;
    }
}
```

<aside> 💡 **Question 7**

Given an encoded string, return its decoded string.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like 3a or 2[4].

The test cases are generated so that the length of the output will never exceed 105.

Example 1:

Input: s = "3[a]2[bc]"

Output: "aaabcbc"

</aside>

```
class Solution {
  public String decodeString(String s) {
    Deque<Character> queue = new LinkedList();
    for (char c: s.toCharArray()) {
      queue.offer(c);
    }
    return bfs(queue);
  }

  private String bfs(Deque<Character> queue) {
    StringBuilder sb = new StringBuilder();
    int num = 0;
    while (!queue.isEmpty()) {
      char c = queue.poll();
      // 1. digit
      if (Character.isDigit(c)) {
        num = num * 10 + c - '0';
      }
      // 2. `[` 进入bfs递归
      else if (c == '[') {
        String sub = bfs(queue);
        for (int i = 0; i < num; i++) {
          sb.append(sub);
        }
        num = 0;
      }
```

```
      // 3. `]` break
      else if (c == ']') {
        break;
      }
      // 4. letter append
      else {
        sb.append(c);
      }
    }

    return sb.toString();
  }
}
```

<aside> 💡 **Question 8**

Given two strings s and goal, return true *if you can swap two letters in* s *so the result is equal to* goal*, otherwise, return* false*.*

Swapping letters is defined as taking two indices i and j (0-indexed) such that i != j and swapping the characters at s[i] and s[j].

- For example, swapping at indices 0 and 2 in "abcd" results in "cbad".

Example 1:

Input: s = "ab", goal = "ba"

Output: true

Explanation: You can swap s[0] = 'a' and s[1] = 'b' to get "ba", which is equal to goal.

</aside>

```
class Solution {
    public boolean buddyStrings(String s, String goal) {
        if (s.length() != goal.length()) {
            return false;
        }

        if (s.equals(goal)) {
            // If we have 2 same characters in string 's',
            // we can swap them and still the strings will remain equal.
            int[] frequency = new int[26];
            for (char ch : s.toCharArray()) {
                frequency[ch - 'a'] += 1;
                if (frequency[ch - 'a'] == 2) {
                    return true;
                }
            }
            // Otherwise, if we swap any two characters, it will make the strings unequal.
            return false;
        }

        int firstIndex = -1;
        int secondIndex = -1;

        for (int i = 0; i < s.length(); ++i) {
            if (s.charAt(i) != goal.charAt(i)) {
```

```
            if (firstIndex == -1) {
                firstIndex = i;
            } else if (secondIndex == -1) {
                secondIndex = i;
            } else {
                // We have at least 3 indices with different characters,
                // thus, we can never make the strings equal with only one swap.
                return false;
            }
        }
    }

    if (secondIndex == -1) {
        // We can't swap if the character at only one index is different.
        return false;
    }

    // All characters of both strings are the same except two indices.
    return s.charAt(firstIndex) == goal.charAt(secondIndex) &&
           s.charAt(secondIndex) == goal.charAt(firstIndex);
  }
}
```