<https://leetcode.com/problems/stickers-to-spell-word/description/>

We are given n different types of stickers. Each sticker has a lowercase English word on it.

You would like to spell out the given string target by cutting individual letters from your collection of stickers and rearranging them. You can use each sticker more than once if you want, and you have infinite quantities of each sticker.

Return **the minimum number of stickers that you need to spell out**target. If the task is impossible, return -1.

**Note:** In all test cases, all words were chosen randomly from the 1000 most common US English words, and target was chosen as a concatenation of two random words.

**Example 1:**

**Input:** stickers = ["with","example","science"], target = "thehat"

**Output:** 3

**Explanation:**

We can use 2 "with" stickers, and 1 "example" sticker.

After cutting and rearrange the letters of those stickers, we can form the target "thehat".

Also, this is the minimum number of stickers necessary to form the target string.

**Example 2:**

**Input:** stickers = ["notice","possible"], target = "basicbasic"

**Output:** -1

Explanation:We cannot form the target "basicbasic" from cutting letters from the given stickers.

**Constraints:**

n == stickers.length

1 <= n <= 50

1 <= stickers[i].length <= 10

1 <= target.length <= 15

stickers[i] and target consist of lowercase English letters.

**Attempt 1: 2023-02-11**

**Solution 1: DFS + Memoization (180 min)**

**Basic template: No test case pass**

class Solution {

    public int minStickers(String[] stickers, String target) {

        int n = stickers.length;

        int[][] stickersMap = new int[n][26];

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < stickers[i].length(); j++) {

                int index = stickers[i].charAt(j) - 'a';

                stickersMap[i][index]++;

            }

        }

        return helper(stickersMap, target);

    }

    private int helper(int[][] stickersMap, String target) {

        if(target.equals("")) {

            return 0;

        }

        int min = Integer.MAX\_VALUE;

        int[] targetMap = new int[26];

        for(int i = 0; i < target.length(); i++) {

            int index = target.charAt(i) - 'a';

            targetMap[index]++;

        }

        // Attempt with each sticker in stickers(represented as stickersMap)

        for(int[] stickerMap : stickersMap) {

            StringBuilder sb = new StringBuilder();

            for(int i = 0; i < 26; i++) {

                for(int time = 0; time < Math.max(targetMap[i] - stickerMap[i], 0); time++) {

                    sb.append((char)('a' + i));

                }

            }

            String newTarget = sb.toString();

            int cur = helper(stickersMap, newTarget);

            if(cur != -1) {

                min = Math.min(cur, min);

            }

        }

        int result = (min == Integer.MAX\_VALUE) ? -1 : 1 + min;

        return result;

    }

}

**Promotion 1: Adding condition "Skip words that don't cover the first character of target" (TLE 33/101)**

**Skip words that don't cover the first character of target. That is inspected by the top post. My original approach is to skip stickers don't cover any character of target. The current approach cuts branches in solution tree much earlier thus much faster.**

if(stickerMap[target.charAt(0) - 'a'] <= 0) {

    continue;

}

**The reason why we can promote in this way:**

**If this sticker didn't contains first character but contains the character after it and it is unique, it will be picked in the future anyway (when that character become the first character). So we won't miss case in such a optimization.**

**Refer to**

<https://leetcode.com/problems/stickers-to-spell-word/solutions/108318/c-java-python-dp-memoization-with-optimization-29-ms-c/comments/481298>

class Solution {

    public int minStickers(String[] stickers, String target) {

        int n = stickers.length;

        int[][] stickersMap = new int[n][26];

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < stickers[i].length(); j++) {

                int index = stickers[i].charAt(j) - 'a';

                stickersMap[i][index]++;

            }

        }

        return helper(stickersMap, target);

    }

    private int helper(int[][] stickersMap, String target) {

        if(target.equals("")) {

            return 0;

        }

        int min = Integer.MAX\_VALUE;

        int[] targetMap = new int[26];

        for(int i = 0; i < target.length(); i++) {

            int index = target.charAt(i) - 'a';

            targetMap[index]++;

        }

        // Attempt with each sticker in stickers(represented as stickersMap)

        for(int[] stickerMap : stickersMap) {

            // Skip words that don't cover the first character of target. That is

            // inspected by the top post. My original approach is to skip stickers

            // don't cover any character of target, the current approach cuts

            // branches in solution tree much earlier thus much faster.

            // The reason why we can promote in this way:

            // If this sticker didn't contains first character but contains the

            // character after it and it is unique, it will be picked in the future

            // anyway (when that character become the first character). So we won't

            // miss case in such a optimization.

            if(stickerMap[target.charAt(0) - 'a'] <= 0) {

                continue;

            }

            StringBuilder sb = new StringBuilder();

            for(int i = 0; i < 26; i++) {

                // No need 'if(targetMap[i] > 0)' or 'time < Math.max(targetMap[i] - stickerMap[i], 0)',

                // just 'time < targetMap[i] - stickerMap[i]' is enough

                for(int time = 0; time < targetMap[i] - stickerMap[i]; time++) {

                    sb.append((char)('a' + i));

                }

            }

            String newTarget = sb.toString();

            int cur = helper(stickersMap, newTarget);

            if(cur != -1) {

                min = Math.min(cur, min);

            }

        }

        int result = (min == Integer.MAX\_VALUE) ? -1 : 1 + min;

        return result;

    }

}

**Promotion 2: Adding memo**

class Solution {

    public int minStickers(String[] stickers, String target) {

        int n = stickers.length;

        int[][] stickersMap = new int[n][26];

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < stickers[i].length(); j++) {

                int index = stickers[i].charAt(j) - 'a';

                stickersMap[i][index]++;

            }

        }

        Map<String, Integer> memo = new HashMap<>();

        return helper(stickersMap, target, memo);

    }

    private int helper(int[][] stickersMap, String target, Map<String, Integer> memo) {

        if(target.equals("")) {

            return 0;

        }

        if(memo.containsKey(target)) {

            return memo.get(target);

        }

        int min = Integer.MAX\_VALUE;

        int[] targetMap = new int[26];

        for(int i = 0; i < target.length(); i++) {

            int index = target.charAt(i) - 'a';

            targetMap[index]++;

        }

        // Attempt with each sticker in stickers(represented as stickersMap)

        for(int[] stickerMap : stickersMap) {

            // Skip words that don't cover the first character of target. That is

            // inspected by the top post. My original approach is to skip stickers

            // don't cover any character of target, the current approach cuts

            // branches in solution tree much earlier thus much faster.

            // The reason why we can promote in this way:

            // If this sticker didn't contains first character but contains the

            // character after it and it is unique, it will be picked in the future

            // anyway (when that character become the first character). So we won't

            // miss case in such a optimization.

            if(stickerMap[target.charAt(0) - 'a'] <= 0) {

                continue;

            }

            StringBuilder sb = new StringBuilder();

            for(int i = 0; i < 26; i++) {

                // No need 'if(targetMap[i] > 0)' or 'time < Math.max(targetMap[i] - stickerMap[i], 0)',

                // just 'time < targetMap[i] - stickerMap[i]' is enough

                for(int time = 0; time < targetMap[i] - stickerMap[i]; time++) {

                    sb.append((char)('a' + i));

                }

            }

            String newTarget = sb.toString();

            int cur = helper(stickersMap, newTarget, memo);

            if(cur != -1) {

                min = Math.min(cur, min);

            }

        }

        int result = (min == Integer.MAX\_VALUE) ? -1 : 1 + min;

        memo.put(target, result);

        return result;

    }

}

Time Complexity: O((M \* (N + 26)) \* 2^N)

N - length of target, M - count of stickers, memo can't contain more than 2^N keys- number of ways

you can select different subset of characters from target (next key character are sorted ascendingly)

Then to compute value for missing key we loop through M stickers and for each sticker it takes at most

max(N, 26) operations to build next key

**Refer to**

<https://leetcode.com/problems/stickers-to-spell-word/solutions/313463/java-dfs-memo/>

input: words, target

output: min combination of words that cover target

similar to min combination of nums that sum up to target with result = {}

= 1 + (min combination of nums that sum up to (target - num) with result = {num}) // but we don't know num, so we guess each num

base case: target = 0

min combination of words that cover target with result = {}

= 1 + (min combination of words that cover (target substract word) with result = {word}) // but we don't know word, so we guess each word

// target substract word: remove `ch` from target if `ch` in word

base case: target is empty

The key to the efficiency of DFS is to cut branches in solution tree, the earlier(the more), the better:

// #0: Skip words that don't cover the first character of target. That is inspected by the top post. My original approach is to skip stickers don't cover any character of target. The current approach cuts branches in solution tree much earlier thus much faster.

// #1: Skip characters that are not in target. That also helps cut branches in solution tree.

class Solution {

private int[][] maps;

private Map<String, Integer> memo;

public int minStickers(String[] words, String target) {

memo = new HashMap<>();

memo.put("", 0);

// Build maps

maps = new int[words.length][26];

for (int i = 0; i < words.length; i++) {

maps[i] = toMap(words[i]);

}

return dfs(target);

}

private int[] toMap(String word) {

int[] dict = new int[26];

for (char ch : word.toCharArray()) {

dict[ch - 'a']++;

}

return dict;

}

private int dfs(String target) {

if (memo.containsKey(target)) {

return memo.get(target);

}

int min = Integer.MAX\_VALUE;

// Build map of target

int[] targetMap = toMap(target);

// For each word

for (int[] map : maps) {

if (map[target.charAt(0) - 'a'] <= 0) { // #0

continue;

}

StringBuilder newTarget = new StringBuilder();

for (int i = 0; i < 26; i++) {

if (targetMap[i] > 0) { // #1

for (int time = 0; time < Math.max(targetMap[i] - map[i], 0); time++) {

newTarget.append((char)('a' + i));

}

}

}

int cur = dfs(newTarget.toString());

if (cur != -1) {

min = Math.min(cur, min);

}

}

int result = (min == Integer.MAX\_VALUE) ? -1 : 1 + min;

memo.put(target, result);

return result;

}

}

**Solution 2: BFS + Level Order Traversal + Bit Manipulation (180 min)**

**A more natural thought for "the minimum number of stickers that you need to spell out target" is using BFS, as BFS naturally used for handling minimum path**

**The critical point of how we construct BFS here is instead of using boolean "visited" array to track if any "sticker" been used or not, the boolean "visited" array here setup to track "target" word characters' covering status in BIT format. Why we have to implement boolean "visited" array in this idea ? Because if still tracking as normal as a "sticker" been used or not won't help, since we can repeatedly using a "sticker", but tracking on how characters in "target" word been covered by "stickers" is a good solution, because the covering status alone with BFS will guarantee current covering status must be the "shortest path" which means use the least "stickers" to approach, and we can mark the covering status with BIT format to avoid duplicate paths reach to same status but repeatedly calculate again, the BIT format will be the most efficient storage choice of status.**

class Solution {

    public int minStickers(String[] stickers, String target) {

        int result = 0;

        int n = target.length();

        // Visited states to avoid repetition, track "target" word characters'

        // covering status in BIT format

        boolean[] visited = new boolean[1 << n];

        Queue<Integer> q = new LinkedList<>();

        // Start with an empty state (no characters covered)

        q.offer(0);

        // Mark the empty state as visited

        visited[0] = true;

        // The solution uses a while-loop that continues until either the queue

        // is empty or the target state ((1 << n) - 1, where all n bits are set

        // meaning all characters of target are obtained) is achieved.

        while(!q.isEmpty()) {

            int size = q.size();

            for(int j = 0; j < size; j++) {

                int curState = q.poll();

                // If all characters in target word are covered, return the result

                if(curState == (1 << n) - 1) {

                    return result;

                }

                // Try to cover more characters using each sticker

                for(String sticker : stickers) {

                    int nextState = curState;

                    int[] stickerFreq = new int[26];

                    for(char c : sticker.toCharArray()) {

                        stickerFreq[c - 'a']++;

                    }

                    for(int i = 0; i < n; i++) {

                        int targetIndex = target.charAt(i) - 'a';

                        // If the character at position i in target word is not covered and the

                        // sticker has the character, cover it and decrement the sticker's count

                        if((curState & (1 << i)) == 0 && stickerFreq[targetIndex] > 0) {

                            nextState = (nextState | (1 << i));

                            stickerFreq[targetIndex]--;

                        }

                    }

                    // If the next state has not been visited, mark it as visited and add to queue

                    if(!visited[nextState]) {

                        visited[nextState] = true;

                        q.offer(nextState);

                    }

                }

            }

            result++;

        }

        // Return -1 if it's not possible to cover all the characters in the target string

        return -1;

    }

}

Time Complexity: O(2^N \* M \* L \* K)

The breadth of the BFS is 2^N, where n is the length of the target string, because there are 2^N possible states in the worst case (each bit in the bitmask can be either 0 or 1).

For each state, the algorithm iterates over all stickers, and then over each character in the target. Let m be the number of stickers, and k be the average length of a sticker.

For each sticker, it attempts to match characters to the target (L characters at most). During this process, it counts occurrences using a Counter, which takes O(K) time.

Upon matching characters, the algorithm updates the state and checks whether it has been visited before.

Space Complexity: O(2^N)

The visited list, which stores whether a state has been visited (visited has a size of 2^N).

The q queue, which in the worst case could store all possible states (again, 2^N in size).

The Counter instance for each sticker, repeated for every state in the BFS.

However, since the Counter is temporary and only stores up to K elements (where k is the average length of a sticker),

this does not exceed O(K) space at any instance, and thus does not dominate the space complexity.

**Refer to**

<https://algo.monster/liteproblems/691>

**Problem Description**

In this problem, we are provided with n different types of stickers, with each sticker containing a single lowercase English word. Our goal is to construct a specific target string by cutting out individual letters from these stickers and rearranging them. Importantly, we can use any number of stickers, and each type of sticker is available in an infinite quantity.

The key objective is to determine the minimum number of stickers that we need to use to spell out the entire target string. If it's not possible to construct the target string from the stickers provided, the function should return -1.

It is important to note that:

Each sticker can be used multiple times.

The target string is generated by concatenating random words from the 1000 most common US English words.

The task involves both optimisation (minimising the number of stickers used) and combination (cutting and rearranging sticker letters to form the target).

**Intuition**

Finding the minimum number of stickers to form the target string can be broken down into exploring different combinations of sticker usage. The solution employs Breadth-First Search (BFS), a strategy used to traverse or search tree or graph data structures level by level.

Here's how the intuition develops:

**States Representation Using Bit Masking:** Treat each letter in the target string as a bit in a bitmask. If a letter is included, its corresponding bit is set to 1, otherwise it's 0. This allows an efficient representation of which characters are currently used to form the target.

**Using a Queue for BFS:** The queue stores states representing the letters from the target we have constructed at any step. Starting with no letters, the goal is to reach a state where all letters are obtained (all bits set to 1).

**Processing Each Sticker:** The algorithm examines how adding each sticker can change the current state. This involves checking if adding a letter from the sticker can fill in a missing bit in the state.

**Avoiding Revisiting States:** With potentially many stickers and target letters, the number of states can become large. Keeping track of visited states with a "visited" list ensures we don't process the same state multiple times.

**Incrementing the Number of Stickers:** Each level in the BFS represents the use of an additional sticker. The depth of the level (how many stickers used) increases until the target is reached or all options are exhausted.

**Termination and Result:** The search terminates when the full target is reached (all bits set to 1), or when no more states can be developed. The number of stickers (depth of BFS traversal) needed to accomplish the target is the answer, unless it is determined to be impossible, in which case -1 is returned.

In summary, the intuition behind the BFS approach is to explore all possible combinations of stickers incrementally and find the least number of stickers needed to create the target without revisiting the same intermediate states.

**Solution Approach**

The solution to this LeetCode problem employs a Breadth-First Search (BFS) approach. For BFS, we typically use a queue data structure, and here's how the provided solution leverages BFS along with other techniques:

**Queue Initialization:** We use Python's deque from the collections module to initialize a queue for our BFS which starts with an initial state of 0 signifying that no letters of the target have been used.

**State Representation:** The state of our approach is represented as an integer where each bit corresponds to a character in the target string. If the i-th bit is set, it means the i-th character of target is already constructed using the stickers.

**Visited States Tracking:** An array vis is initialized to keep track of which states have been visited to avoid redundant processing. Only unvisited states are pushed into the queue. Initially, only the state 0 is marked as visited (True).

**Breadth-First Search: The solution uses a while-loop that continues until either the queue is empty or the target state ((1 << n) - 1, where all n bits are set meaning all characters of target are obtained) is achieved.**

Within the while-loop, a for-loop iterates over the current breadth of the queue, using queue.popleft() to examine states one by one.

For every current state, the algorithm goes through each sticker and calculates the nxt state after possibly using this sticker. It uses a Counter object from Python's collections module to count the availability of characters in the sticker which assists in determining the eligibility of placing a character in the target.

As it goes through the characters in the target, if it finds a character that is not yet used in the current state (checked using bitwise AND operation), and if the character is present in the sticker's Counter, it will update the nxt state by setting the corresponding bit (using bitwise OR operation) and decrementing the character's count in the counter.

**State Transitions and Level Traversal:** Whenever we encounter a new state (one that hasn't been visited), we mark it as visited and add it to the queue for further exploration. After examining all possible sticker applications for a given level, the ans (answer) variable is incremented since it would take one more sticker to potentially reach the target state at the next deeper level of our search.

**Termination:** If at some point the nxt state matches the target (all bits are set), we return the ans which represents the minimum number of stickers needed to achieve the target. If the queue is exhaustively processed without achieving the target, the function returns -1.

**Algorithm Complexity:** The solution is quite comprehensive in that it explores all possible combinations but maintains efficiency by avoiding revisiting states and by processing levels depth by depth, typical to BFS. The time complexity is substantial due to the combinatory nature, but by no means exponential to the point of being unfeasible, as the early termination and visited state pruning substantially reduce the search space.

In conclusion, the provided solution is a clear-cut application of BFS with a bitwise representation of states and an intricate handling of transitions using stickers to efficiently solve the problem of finding the minimum number of stickers to create the target string.

**Example Walkthrough**

Let's consider a simple example to illustrate the solution approach. Suppose we have the following input:

Stickers: ["with", "example", "science"] Target: "wise"

**Queue Initialization:** We start by initializing an empty queue and push the initial state 0 into it, representing that none of the target letters are used yet.

**State Representation:** The target "wise" has four characters, each corresponding to a bit in our bitmask. Thus, "wise" corresponds to 1111 in binary when all characters are used.

**Visited States Tracking:** We have an array vis that starts filled with False values, and we set vis[0] to True to mark our initial state as visited.

**Breadth-First Search:**

We begin our BFS. Initially, the queue contains only the initial state 0 (0000 in binary, since none of the target's letters have been covered).

We dequeue 0 and start examining our stickers. Let's start with the first sticker "with".

The target is "wise", and the sticker "with" can contribute w and i to the target. We then update our state; w is the first bit and i is the second bit. Thus our state, after using the sticker "with", becomes 0110.

We check our vis array and since this state is not visited, we mark it as visited (vis[0110] = True), then enqueue it.

Next, we take the next sticker on the list, "example". It can contribute e. However, since we're considering the BFS level that started with state 0, adding "example" does not help us progress towards our target as we need either s or e at this point. Hence, it doesn't change the state.

Finally, we consider the sticker "science" which can contribute s, i, and e, but we only care about s and e since i is already included from the "with" sticker. The state changes to 1111, which is our goal state.

**State Transitions and Level Traversal:** We found our target state at the first BFS level with just two stickers used: with and science. Therefore, we would not process the queue further and our ans would be 2.

**Termination:** Our traversal found the target state 1111, and we can return our answer 2. If we had not found our target, we would continue processing the queue, systematically using stickers at each level, and incrementing our sticker count until we either find the target or determine it is impossible.

In this example, our process was fairly straightforward as the target was reachable with a small number of sticker applications. In more complex cases, the breadth-first search would evaluate many more combinations by incrementing the state bit by bit until the target is fully constructed or found unattainable.

**Java Solution**

class Solution {

    public int minStickers(String[] stickers, String target) {

        // Initialize a queue to store the states of characters from the target string

        Deque<Integer> queue = new ArrayDeque<>();

        queue.offer(0); // Start with an empty state (no characters covered)

        int answer = 0; // Number of stickers used

        int n = target.length(); // Length of the target string

        boolean[] visited = new boolean[1 << n]; // Visited states to avoid repetition

        visited[0] = true; // Mark the empty state as visited

        // While there are states in the queue, continue processing

        while (!queue.isEmpty()) {

            // Process all states at the current level

            for (int t = queue.size(); t > 0; --t) {

                // Get and remove the current state from the queue

                int currentState = queue.poll();

                // If all characters are covered, return the count

                if (currentState == (1 << n) - 1) {

                    return answer;

                }

                // Try to cover more characters using each sticker

                for (String sticker : stickers) {

                    int nextState = currentState; // Start with the current state

                    int[] charCount = new int[26]; // Frequency of each character in sticker

                    for (char c : sticker.toCharArray()) {

                        ++charCount[c - 'a'];

                    }

                    // Try to match sticker characters with uncovered characters in the target

                    for (int i = 0; i < n; ++i) {

                        int targetCharIndex = target.charAt(i) - 'a';

                        // If the character at position i is not covered and

                        // the sticker has the character, cover it and decrement the sticker's count

                        if ((nextState & (1 << i)) == 0 && charCount[targetCharIndex] > 0) {

                            nextState |= 1 << i;

                            --charCount[targetCharIndex];

                        }

                    }

                    // If the next state has not been visited, mark it as visited and add to queue

                    if (!visited[nextState]) {

                        visited[nextState] = true;

                        queue.offer(nextState);

                    }

                }

            }

            // Increment the count after processing all states at the current level

            ++answer;

        }

        // Return -1 if it's not possible to cover all the characters in the target string

        return -1;

    }

}

**Time and Space Complexity**

The provided code performs a BFS (Breadth-First Search) over the states representing which characters of the target string have been covered by stickers. Each state is a bitmask of length n (where n is the length of the target string), with each bit representing whether the corresponding character in the target has been matched by a sticker.

**Time Complexity**

The time complexity of the algorithm can be estimated by considering the following:

The breadth of the BFS is 2^n, where n is the length of the target string, because there are 2^n possible states in the worst case (each bit in the bitmask can be either 0 or 1).

For each state, the algorithm iterates over all stickers, and then over each character in the target. Let m be the number of stickers, and k be the average length of a sticker.

For each sticker, it attempts to match characters to the target (n characters at most). During this process, it counts occurrences using a Counter, which takes O(k) time.

Upon matching characters, the algorithm updates the state and checks whether it has been visited before.

Therefore, the worst-case time complexity is O(2^n \* m \* n \* k). This presumes that we could match every character in target with a character in a sticker every time, which represents the worst-case scenario for the number of operations needed.

**Space Complexity**

The space complexity of the algorithm is determined by:

The vis list, which stores whether a state has been visited (vis has a size of 2^n).

The q queue, which in the worst case could store all possible states (again, 2^n in size).

The Counter instance for each sticker, repeated for every state in the BFS. However, since the Counter is temporary and only stores up to k elements (where k is the average length of a sticker), this does not exceed O(k) space at any instance, and thus does not dominate the space complexity.

So the dominant factor is the space required to store all states, which results in a space complexity of O(2^n).