<https://leetcode.com/problems/smallest-sufficient-team/description/>

In a project, you have a list of required skills req\_skills, and a list of people. The ith person people[i] contains a list of skills that the person has.

Consider a sufficient team: a set of people such that for every required skill in req\_skills, there is at least one person in the team who has that skill. We can represent these teams by the index of each person.

For example, team = [0, 1, 3] represents the people with skills people[0], people[1], and people[3].

Return *any sufficient team of the smallest possible size, represented by the index of each person*. You may return the answer in **any order**.

It is **guaranteed** an answer exists.

**Example 1:**

**Input:** req\_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],["nodejs","reactjs"]]

**Output:** [0,2]

**Example 2:**

**Input:** req\_skills = ["algorithms","math","java","reactjs","csharp","aws"], people = [["algorithms","math","java"],["algorithms","math","reactjs"],["java","csharp","aws"],["reactjs","csharp"],["csharp","math"],["aws","java"]]

**Output:** [1,2]

**Constraints:**

1 <= req\_skills.length <= 16

1 <= req\_skills[i].length <= 16

req\_skills[i] consists of lowercase English letters.

All the strings of req\_skills are **unique**.

1 <= people.length <= 60

0 <= people[i].length <= 16

1 <= people[i][j].length <= 16

people[i][j] consists of lowercase English letters.

All the strings of people[i] are **unique**.

Every skill in people[i] is a skill in req\_skills.

It is guaranteed a sufficient team exists.

**Attempt 1: 2025-08-30**

**Solution 1: Backtracking + Bit Manipulation (120 min)**

**Style 1: For loop**

class Solution {

    int minSize = Integer.MAX\_VALUE;

    // Stores the best team found so far (array of person indices)

    int[] bestTeam;

    public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

        int n = req\_skills.length;

        // Create mapping from skill name to bit index (0 to n-1)

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

        for(int i = 0; i < n; i++) {

            skillBitMap.put(req\_skills[i], i);

        }

        // Convert each person's skills to a bitmask

        // Each bit represents whether the person has that skill

        int[] peopleMasks = new int[people.size()];

        for(int i = 0; i < people.size(); i++) {

            int mask = 0;

            for(String people\_skill : people.get(i)) {

                // Set the corresponding bit for this skill

                mask |= (1 << skillBitMap.get(people\_skill));

            }

            peopleMasks[i] = mask;

        }

        // Bitmask with all n bits set (represents all skills required)

        int target = (1 << n) - 1;

        // Start backtracking from the first person with no skills covered initially

        helper(peopleMasks, target, 0, new ArrayList<Integer>(), 0);

        return bestTeam;

    }

    // Similar to L77.Combinations, try to find a combination

    private void helper(int[] peopleMasks, int target, int curSkills, List<Integer> team, int index) {

        // Base case: check if current team covers all skills

        if(curSkills == target) {

            if(team.size() < minSize) {

                minSize = team.size();

                bestTeam = team.stream().mapToInt(Integer::intValue).toArray();

            }

            return;

        }

        // Pruning: stop if current team is already larger than best

        if(team.size() >= minSize) {

            return;

        }

        for(int i = index; i < peopleMasks.length; i++) {

            // Critical prune: Only consider this person if they add new skills

            int newSkills = curSkills | peopleMasks[i];

            if(newSkills != curSkills) {

                team.add(i);

                helper(peopleMasks, target, newSkills, team, i + 1);

                team.remove(team.size() - 1);

            }

        }

    }

}

Time Complexity: O(2^n \* n)

Space Complexity: O(n)

**Style 2: Take or not take (Refer to** [L77.Combinations](note://1337216AB2164464BA7362B6FA4FFFC9)**)**

class Solution {

    int minSize = Integer.MAX\_VALUE;

    // Stores the best team found so far (array of person indices)

    int[] bestTeam;

    public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

        int n = req\_skills.length;

        // Create mapping from skill name to bit index (0 to n-1)

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

        for(int i = 0; i < n; i++) {

            skillBitMap.put(req\_skills[i], i);

        }

        // Convert each person's skills to a bitmask

        // Each bit represents whether the person has that skill

        int[] peopleMasks = new int[people.size()];

        for(int i = 0; i < people.size(); i++) {

            int mask = 0;

            for(String people\_skill : people.get(i)) {

                // Set the corresponding bit for this skill

                mask |= (1 << skillBitMap.get(people\_skill));

            }

            peopleMasks[i] = mask;

        }

        // Bitmask with all n bits set (represents all skills required)

        int target = (1 << n) - 1;

        // Start backtracking from the first person with no skills covered initially

        helper(peopleMasks, target, 0, new ArrayList<Integer>(), 0);

        return bestTeam;

    }

    // Similar to L77.Combinations, try to find a combination

    private void helper(int[] peopleMasks, int target, int curSkills, List<Integer> team, int index) {

        // Base case: check if current team covers all skills

        if(curSkills == target) {

            if(team.size() < minSize) {

                minSize = team.size();

                bestTeam = team.stream().mapToInt(Integer::intValue).toArray();

            }

            return;

        }

        // Pruning: stop if we've processed all people or current team is already larger than best

        if(index == peopleMasks.length || team.size() >= minSize) {

            return;

        }

        // Option 1: Skip the current person

        // Move to next person without adding current person to team

        helper(peopleMasks, target, curSkills, team, index + 1);

        // Option 2: Take the current person (if they contribute new skills)

        int newSkills = curSkills | peopleMasks[index];

        // Only consider adding if this person actually adds new skills

        if(newSkills != curSkills) {

            team.add(index);

            helper(peopleMasks, target, newSkills, team, index + 1);

            team.remove(team.size() - 1);

        }

    }

}

Time Complexity: O(2^n \* n)

Space Complexity: O(n)

**Refer to Deepseek**

**Alternative Backtracking Solution (For Reference)**

import java.util.\*;

class Solution {

private int[] bestTeam; // Stores the best team found so far (array of person indices)

private int minSize = Integer.MAX\_VALUE; // Tracks the size of the smallest team found

public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

int n = req\_skills.length;

// Create mapping from skill name to bit index (0 to n-1)

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

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

skillToBit.put(req\_skills[i], i);

}

// Convert each person's skills to a bitmask

// Each bit represents whether the person has that skill

int[] peopleMasks = new int[people.size()];

for (int i = 0; i < people.size(); i++) {

int mask = 0;

for (String skill : people.get(i)) {

// Set the corresponding bit for this skill

mask |= (1 << skillToBit.get(skill));

}

peopleMasks[i] = mask;

}

int target = (1 << n) - 1; // Bitmask with all n bits set (represents all skills required)

List<Integer> currentTeam = new ArrayList<>(); // Current team being constructed

// Start backtracking from the first person with no skills covered initially

backtrack(peopleMasks, 0, target, 0, currentTeam);

return bestTeam;

}

private void backtrack(int[] peopleMasks, int index, int target, int currentSkills, List<Integer> team) {

// Base case: current team covers all required skills

if (currentSkills == target) {

// Update best team if current team is smaller

if (team.size() < minSize) {

minSize = team.size();

// Convert List<Integer> to int[] for storage

bestTeam = team.stream().mapToInt(Integer::intValue).toArray();

}

return;

}

// Pruning: stop if we've processed all people or current team is already larger than best

if (index == peopleMasks.length || team.size() >= minSize) {

return;

}

// Option 1: Skip the current person

// Move to next person without adding current person to team

backtrack(peopleMasks, index + 1, target, currentSkills, team);

// Option 2: Take the current person (if they contribute new skills)

int newSkills = currentSkills | peopleMasks[index];

// Only consider adding if this person actually adds new skills

if (newSkills != currentSkills) {

team.add(index); // Add current person to team

// Recurse to next person with updated skills coverage

backtrack(peopleMasks, index + 1, target, newSkills, team);

team.remove(team.size() - 1); // Backtrack: remove current person

}

}

}

**Key Comments Explained:**

**bestTeam and minSize**: Track the best solution found during backtracking

**Skill Mapping**: Convert skill names to bit indices for efficient bitmask operations

**People Bitmasks**: Each person's skills are represented as a bitmask for fast set operations

**Target**: (1 << n) - 1 creates a bitmask with all n bits set (represents all required skills)

**Backtracking Function**:

**Base Case**: When current team covers all skills, check if it's the smallest team found

**Pruning**: Stop if no more people or current team is already larger than best team

**Skip Option**: Don't include current person

**Take Option**: Include current person only if they add new skills to the team

**Important Optimizations:**

**Bitmask Operations**: Use bitwise OR (|) to combine skills efficiently

**Pruning**: Stop early if current team size exceeds the best found so far

**Skill Check**: Only add a person if they contribute new skills (newSkills != currentSkills)

**Backtracking**: Properly add/remove people from the current team to explore all combinations

This backtracking solution explores all possible team combinations but uses pruning to avoid unnecessary paths. However, for the problem constraints (up to 16 skills but 60 people), the DP solution is generally more efficient as backtracking could be slow in worst-case scenarios.

**I see we are using Option 1 and Option 2 as take or not take style in backtracking method, how about using for loop style ? is that possible or not ?**

// Option 1: Skip the current person

// Move to next person without adding current person to team

backtrack(peopleMasks, index + 1, target, currentSkills, team);

// Option 2: Take the current person (if they contribute new skills)

int newSkills = currentSkills | peopleMasks[index];

// Only consider adding if this person actually adds new skills

if (newSkills != currentSkills) {

team.add(index); // Add current person to team

// Recurse to next person with updated skills coverage

backtrack(peopleMasks, index + 1, target, newSkills, team);

team.remove(team.size() - 1); // Backtrack: remove current person

}

**For Loop Style vs Take/Not Take Style in Backtracking (Concept refer to** [L77.Combinations](note://1337216AB2164464BA7362B6FA4FFFC9)**)**

Yes, you can absolutely use a **for loop style** instead of the explicit take/not take approach. Both are valid backtracking patterns, but they have different characteristics.

**For Loop Style Implementation**

private void backtrack(int[] peopleMasks, int startIndex, int target, int currentSkills, List<Integer> team) {

// Base case: check if current team covers all skills

if (currentSkills == target) {

if (team.size() < minSize) {

minSize = team.size();

bestTeam = team.stream().mapToInt(Integer::intValue).toArray();

}

return;

}

// Pruning: stop if current team is already larger than best

if (team.size() >= minSize) {

return;

}

// For loop style: iterate through remaining people

for (int i = startIndex; i < peopleMasks.length; i++) {

int newSkills = currentSkills | peopleMasks[i];

// Only consider this person if they add new skills

if (newSkills != currentSkills) {

team.add(i); // Add person to team

backtrack(peopleMasks, i + 1, target, newSkills, team);

team.remove(team.size() - 1); // Backtrack

}

}

}

**Key Differences Between the Two Styles**

**Take/Not Take Style (Original)**

// Option 1: Skip current person

backtrack(..., index + 1, currentSkills, team);

// Option 2: Take current person (if useful)

if (newSkills != currentSkills) {

team.add(index);

backtrack(..., index + 1, newSkills, team);

team.remove(...);

}

**Characteristics:**

**Explicit branching**: Clearly shows both possibilities

**Better pruning**: Can skip entire subtrees more easily

**More control**: Can implement different pruning conditions for each branch

**For Loop Style**

for (int i = startIndex; i < peopleMasks.length; i++) {

if (newSkills != currentSkills) {

team.add(i);

backtrack(..., i + 1, newSkills, team);

team.remove(...);

}

}

**Characteristics:**

**More concise**: Single loop instead of two explicit branches

**Natural ordering**: Processes people in sequence

**Easier to understand**: More intuitive for many developers

**Which One is Better?**

**For this specific problem, the take/not take style is actually better because:**

**Better Pruning Opportunity**:

// In take/not take style, we can do this:

if (index == peopleMasks.length || team.size() >= minSize) {

return; // Early termination

}

This pruning happens before considering the current person, which can save more recursive calls.

**Skip Useless People Earlier**:

// We can skip people who don't add new skills without entering the loop

if (newSkills == currentSkills) {

// Skip this person entirely and move to next

backtrack(..., index + 1, currentSkills, team);

} else {

// Consider both options

}

**More Granular Control**:

Can implement different pruning strategies for "take" vs "not take" branches

Easier to add additional optimizations

**Recommendation**

**Stick with the take/not take style for this problem** because:

It allows for more aggressive pruning

It's more efficient for the problem constraints

It provides better control over the backtracking process

However, the for loop style is perfectly valid and might be preferred in scenarios where:

The problem has a natural ordering constraint

You want to avoid duplicate combinations

The code readability is more important than micro-optimizations

Both approaches will solve the problem, but the take/not take style has slight advantages for this specific use case.

**Time and Space complexity for the for loop style backtrack solution ?**

**For Loop Style Backtracking Code (Recap)**

private void backtrack(int[] peopleMasks, int startIndex, int target, int currentSkills, List<Integer> team) {

if (currentSkills == target) {

if (team.size() < minSize) {

minSize = team.size();

bestTeam = team.stream().mapToInt(Integer::intValue).toArray();

}

return;

}

if (team.size() >= minSize) return;

for (int i = startIndex; i < peopleMasks.length; i++) {

int newSkills = currentSkills | peopleMasks[i];

if (newSkills != currentSkills) {

team.add(i);

backtrack(peopleMasks, i + 1, target, newSkills, team);

team.remove(team.size() - 1);

}

}

}

**Time Complexity**

**Worst-Case: O(2^n \* n)**

**Without pruning**: Each person can be included or excluded → 2^n combinations

**For each combination**: Checking skills and maintaining team list takes O(n) time

**Total**: O(2^n \* n)

**Best-Case with Pruning: Much better but still exponential**

**Pruning**: if (team.size() >= minSize) return cuts off many branches

**Skill check**: if (newSkills != currentSkills) avoids redundant additions

**Still exponential** in worst-case, but much better in practice

**Factors Affecting Time Complexity:**

**Number of people (n)**: Exponential growth O(2^n)

**Pruning effectiveness**: Depends on how early we find good solutions

**Skill overlap**: More overlap → more pruning opportunities

**Space Complexity**

**Recursion Stack: O(n)**

Maximum recursion depth = number of people = n

Each stack frame has constant space (references, not copies)

**Team List: O(n)**

The current team list can grow up to size n

However, this is shared/reused through backtracking

**Total Space: O(n)**

Dominated by recursion stack depth

Auxiliary space for team list is O(n)

**Comparison with Take/Not Take Style**

|  |  |  |
| --- | --- | --- |
| Aspect | For Loop Style | Take/Not Take Style |
| Time Complexity | O(2^n \* n) | O(2^n \* n) |
| Space Complexity | O(n) | O(n) |
| Pruning Effectiveness | Good | Slightly Better |
| Code Readability | More intuitive | More explicit |
| Control over branching | Less | More |

**Practical Performance**

For the problem constraints (n ≤ 60 people, but with good pruning):

**Theoretical worst-case**: 2^60 ≈ 1.15e18 operations (impossible)

**Actual with pruning**: Usually completes for n ≤ 20-25

**With aggressive pruning**: Might handle n ≤ 30-35

**Why This Might Time Out**

The for loop style backtracking will **likely timeout** for the actual problem constraints because:

n (number of people) can be up to 60

2^60 is astronomically large (1.15e18 operations)

Even with pruning, it's unlikely to handle n > 30 efficiently

**Recommendation**

For this specific problem, **neither backtracking style is optimal**. The DP solution is preferred because:

n (number of skills) is small (≤16) → 2^16 = 65536 states

Number of people (60) is handled efficiently in O(people \* 2^n) time

DP guarantees finding the optimal solution efficiently

Use backtracking only for:

Small instances (n ≤ 20-25)

Educational purposes

When you want a simpler implementation despite worse time complexity

The DP solution is preferred as it's more efficient for the given constraints, while the backtracking solution may be easier to understand but could be slower for larger inputs.

**Solution 2: DFS + Memoization (180 min)**

**Style 1: Global variable**

class Solution {

    int[] peopleMasks;

    int[] requiredPeople;

    List<Integer>[] requiredTeam;

    public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

        int n = req\_skills.length;

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

        for(int i = 0; i < req\_skills.length; i++) {

            skillBitMap.put(req\_skills[i], i);

        }

        peopleMasks = new int[people.size()];

        for(int i = 0; i < people.size(); i++) {

            int mask = 0;

            for(String people\_skill : people.get(i)) {

                mask |= (1 << skillBitMap.get(people\_skill));

            }

            peopleMasks[i] = mask;

        }

        int totalTasks = 1 << n;

        // requiredPeople[mask] = minimum number of people needed to cover the

// remaining skills (skills not in mask), given that we already have

// skills in mask, -1 indicates this state hasn't been computed yet

        requiredPeople = new int[totalTasks];

        // Mark all states as uncomputed

        Arrays.fill(requiredPeople, -1);

        // requiredTeam[mask] = optimal team needed to cover the remaining skills

        // (skills not in mask), given that we already have skills in mask

        requiredTeam = new ArrayList[totalTasks];

        // Start DFS from the initial state (mask = 0, no skills covered)

        helper(0, n);

        // Extract the result from the computed optimal team

        // requiredTeam[0] contains the optimal team that covers all skills starting from nothing

        List<Integer> resultTeam = requiredTeam[0];

        int[] result = new int[resultTeam.size()];

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

            result[i] = resultTeam.get(i);

        }

        return result;

    }

    private int helper(int mask, int n) {

        // Base case: All skills are covered, no remaining skills (all n bits are set)

        // mask = 111...1 (n times) = (1 << n) - 1

        // No additional people needed - return empty team

        if(mask == (1 << n) - 1) {

            requiredTeam[mask] = new ArrayList<>();

            return 0;

        }

        // Memoization check: if we've already computed this state, return stored result

        if(requiredPeople[mask] != -1) {

            return requiredPeople[mask];

        }

        // Track minimum team size found

        int minPeople = Integer.MAX\_VALUE;

        // Track the actual team that achieves this minimum

        List<Integer> bestTeam = null;

        // Try adding each person to see if they help cover more skills

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

            // Calculate new skill set by adding this person's skills

            int newMask = mask | peopleMasks[i];

            // Only proceed if this person actually adds new skills (avoid redundant additions)

            if(newMask != mask) {

                // Recursively compute the team needed for the new skill set

                int result = helper(newMask, n);

                // If a valid team was found for the new skill set

                if(result != Integer.MAX\_VALUE) {

                    // Check if adding this person gives us a better solution

                    if(result + 1 < minPeople) {

                        minPeople = result + 1;

                        // Create a new team by copying the optimal team for newMask

                        // and adding the current person

                        bestTeam = new ArrayList<>(requiredTeam[newMask]);

                        bestTeam.add(i);

                    }

                }

            }

        }

        // Store computed results in memoization arrays

        requiredPeople[mask] = minPeople;

        requiredTeam[mask] = bestTeam;

        return minPeople;

    }

}

Time Complexity: O(n × 2^m × m)

Space Complexity: O(2^m × m)

Where:

n = number of people (up to 60)

m = number of skills (up to 16)

**Style 2: Local variable (Better way)**

// Helper class to return both team size and team composition

class Result {

    int size;

    List<Integer> team;

    Result(int size, List<Integer> team) {

        this.size = size;

        this.team = team;

    }

}

class Solution {

    int[] peopleMasks;

    public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

        int n = req\_skills.length;

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

        for(int i = 0; i < req\_skills.length; i++) {

            skillBitMap.put(req\_skills[i], i);

        }

        peopleMasks = new int[people.size()];

        for(int i = 0; i < people.size(); i++) {

            int mask = 0;

            for(String people\_skill : people.get(i)) {

                mask |= (1 << skillBitMap.get(people\_skill));

            }

            peopleMasks[i] = mask;

        }

        int totalTasks = 1 << n;

        // Now stores both size and team

        Result[] memo = new Result[totalTasks];

        // Start DFS from the initial state (mask = 0, no skills covered) with memo array as parameter

        Result result = helper(0, n, memo);

        // Convert to output format

        int[] output = new int[result.team.size()];

        for (int i = 0; i < result.team.size(); i++) {

            output[i] = result.team.get(i);

        }

        return output;

    }

    private Result helper(int mask, int n, Result[] memo) {

        // Base case: All skills are covered, no remaining skills (all n bits are set)

        // mask = 111...1 (n times) = (1 << n) - 1

        // No additional people needed - return empty team

        if(mask == (1 << n) - 1) {

            return new Result(0, new ArrayList<>());

        }

        // Memoization check: if we've already computed this state, return stored result

        if(memo[mask] != null) {

            return memo[mask];

        }

        // Track minimum team size found

        int minPeople = Integer.MAX\_VALUE;

        // Track the actual team that achieves this minimum

        List<Integer> bestTeam = null;

        // Try adding each person to see if they help cover more skills

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

            // Calculate new skill set by adding this person's skills

            int newMask = mask | peopleMasks[i];

            // Only proceed if this person actually adds new skills (avoid redundant additions)

            if(newMask != mask) {

                // Recursively compute the team needed for the new skill set

                Result subResult = helper(newMask, n, memo);

                // If a valid team was found for the new skill set

                if(subResult.size != Integer.MAX\_VALUE) {

                    // Check if adding this person gives us a better solution

                    if(subResult.size + 1 < minPeople) {

                        minPeople = subResult.size + 1;

                        // Create a new team by copying the optimal team for newMask

                        // and adding the current person

                        bestTeam = new ArrayList<>(subResult.team);

                        bestTeam.add(i);

                    }

                }

            }

        }

        // Handle case where no valid team was found

        if(minPeople == Integer.MAX\_VALUE) {

            memo[mask] = new Result(Integer.MAX\_VALUE, null);

        } else {

            memo[mask] = new Result(minPeople, bestTeam);

        }

        return memo[mask];

    }

}

Time Complexity: O(n × 2^m × m)

Space Complexity: O(2^m × m)

Where:

n = number of people (up to 60)

m = number of skills (up to 16)

**Solution 3: DP (120 min)**

class Solution {

    public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

        int n = req\_skills.length;

        // Create mapping from skill name to bit index

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

        for (int i = 0; i < n; i++) {

            skillBitMap.put(req\_skills[i], i);

        }

        // Convert each person's skills to a bitmask

        int[] peopleMasks = new int[people.size()];

        for (int i = 0; i < people.size(); i++) {

            int mask = 0;

            for (String skill : people.get(i)) {

                mask |= (1 << skillBitMap.get(skill));

            }

            peopleMasks[i] = mask;

        }

        int totalMasks = 1 << n;

        // dp[mask] = minimum number of people needed to cover remaining skills

        // given that we already have skills in mask

        int[] dp = new int[totalMasks];

        Arrays.fill(dp, Integer.MAX\_VALUE);

        // teamDp[mask] = optimal team needed to cover remaining skills

        List<Integer>[] teamDp = new ArrayList[totalMasks];

        // Base case: all skills covered (mask = 111...1)

        int fullMask = (1 << n) - 1;

        dp[fullMask] = 0;

        teamDp[fullMask] = new ArrayList<>();

        // Process masks in reverse order (from full mask down to 0)

        // This corresponds to the DFS order where we start from full mask and work backwards

        for (int mask = fullMask; mask >= 0; mask--) {

//

            if (dp[mask] == Integer.MAX\_VALUE) continue;

            // Try adding each person (same order as DFS)

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

                int personMask = peopleMasks[i];

                // Calculate the mask we had before adding this person

                // This is the inverse of the DFS logic

                int prevMask = mask & ~personMask;

                // Only proceed if this person actually contributes to the current mask

                // This corresponds to: if (newMask != mask) in DFS

                if (prevMask != mask) {

                    if (dp[prevMask] > dp[mask] + 1) {

                        dp[prevMask] = dp[mask] + 1;

                        teamDp[prevMask] = new ArrayList<>(teamDp[mask]);

                        teamDp[prevMask].add(i);

                    }

                }

            }

        }

        // Extract result for mask = 0 (no skills initially)

        List<Integer> resultTeam = teamDp[0];

        int[] result = new int[resultTeam.size()];

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

            result[i] = resultTeam.get(i);

        }

        return result;

    }

}

**Bitmask Dynamic Programming and its corresponding DFS solution relation**

**Approach**

This is a **bitmask dynamic programming** problem where we need to find the smallest team that covers all required skills. Each person has a set of skills represented as a bitmask, and we need to find the minimum number of people whose combined skills cover all required skills.

**General DP solution (Not mapping to DFS solution, since the internal, external loop order and directions not exactly match)**

import java.util.\*;

class Solution {

public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

int n = req\_skills.length;

int totalSkills = 1 << n;

// Map each skill to a bit index

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

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

skillToBit.put(req\_skills[i], i);

}

// Convert each person's skills to a bitmask

int[] peopleMasks = new int[people.size()];

for (int i = 0; i < people.size(); i++) {

int mask = 0;

for (String skill : people.get(i)) {

mask |= (1 << skillToBit.get(skill));

}

peopleMasks[i] = mask;

}

// dp[mask] = minimum team size to achieve the skill mask

int[] dp = new int[totalSkills];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

// team[mask] = the team that achieves the skill mask (as a bitmask of people indices)

long[] team = new long[totalSkills];

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

int personMask = peopleMasks[i];

for (int j = totalSkills - 1; j >= 0; j--) {

if (dp[j] == Integer.MAX\_VALUE) continue;

int newMask = j | personMask;

if (dp[newMask] > dp[j] + 1) {

dp[newMask] = dp[j] + 1;

team[newMask] = team[j] | (1L << i);

}

}

}

// Convert the team bitmask to array of indices

long resultTeam = team[totalSkills - 1];

int[] result = new int[Long.bitCount(resultTeam)];

int idx = 0;

for (int i = 0; i < people.size(); i++) {

if ((resultTeam & (1L << i)) != 0) {

result[idx++] = i;

}

}

return result;

}

}

**Explanation**

**Skill Mapping**: Convert each required skill to a bit index (0 to n-1)

**People Bitmasks**: Convert each person's skills to a bitmask where each bit represents a skill they have

**Dynamic Programming**:

dp[mask] stores the minimum team size needed to achieve the skill set represented by mask

team[mask] stores the actual team (as a bitmask of person indices) that achieves mask

**DP Transition**: For each person, update all existing states by adding this person's skills

**Result Extraction**: Convert the final team bitmask back to an array of person indices

**Time and Space Complexity**

**Time Complexity**: O(people.length \* 2^n) where n is number of required skills

**Space Complexity**: O(2^n) for DP arrays

This solution efficiently finds the smallest team using bitmask DP, leveraging the fact that n (number of skills) is small (≤16) while the number of people can be larger (≤60).

**Is the Bitmask Dynamic Programming solution for Leetcode 1125 deduced from for loop style backtracking solution ?**

No, the **bitmask dynamic programming solution for LeetCode 1125 is not directly deduced from the for loop style backtracking solution**. They are fundamentally different approaches to solving the problem.

**Key Differences Between the Two Approaches**

**1. For Loop Backtracking (DFS)**

**Approach**: **Explore all possible combinations of people through recursion**

**State**: **Current team composition and skills covered**

**Mechanism**: Depth-first search with pruning

**Complexity**: O(2^n) where n = number of people

**2. Bitmask Dynamic Programming**

**Approach**: **Build solution incrementally using dynamic programming**

**State**: **Bitmask representing skills covered so far**

**Mechanism**: Iterative DP filling with state transitions

**Complexity**: O(n \* 2^m) where m = number of skills

The bitmask DP solution comes from a **different paradigm**

**Why They're Not Directly Related**

**Different State Representation**:

Backtracking: Explicit team list + current skills

DP: Bitmask of skills + team size count

**Different Exploration Order**:

Backtracking: Depth-first, recursive

DP: Bottom-up, iterative

**Different Complexity Factors**:

Backtracking: Exponential in number of people (n)

DP: Exponential in number of skills (m)

**Different Optimization Strategies**:

Backtracking: Pruning, early termination

DP: State compression, efficient transitions

**Relationship Between the Approaches**

While not directly deduced from each other, both solutions share the same **underlying concept**: using bitmasks to represent skill sets. However:

The **backtracking solution** uses bitmasks as part of its state during DFS

The **DP solution** uses bitmasks as the primary state in a bottom-up approach

The DP solution is essentially a **more efficient reorganization** of the solution space that leverages the fact that:

The number of skills (m) is small (≤16)

The number of people (n) is larger but manageable (≤60)

We can use dynamic programming to avoid recomputing overlapping subproblems

So while both use bitmasks, the DP solution is not derived from the backtracking solution but rather represents a fundamentally different algorithmic strategy optimized for the problem constraints.

**But usually DP (bottom-up) approach is derived from a DFS solution, how is the DFS solution for this DP solution ?**

You're absolutely right! Usually, bottom-up DP is derived from a top-down DFS (recursive) solution with memoization. Let me show you the **DFS solution that directly leads to the bitmask DP solution** for LeetCode 1125.

**The DFS solution that Inspires the DP solution**

import java.util.\*;

class Solution {

// Maps each skill name to its corresponding bit index (0 to n-1)

private Map<String, Integer> skillToBit;

// Array storing bitmask representation of each person's skills

// Each bit in the mask represents whether the person has that skill

private int[] peopleMasks;

// Number of required skills

private int n;

// memo[mask] = minimum number of people needed to achieve the skill set represented by 'mask'

// -1 indicates this state hasn't been computed yet

private int[] memo;

// teamMemo[mask] = optimal team needed to cover the remaining skills

// (skills not in mask), given that we already have skills in mask

private List<Integer>[] teamMemo;

public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

// Step 1: Create mapping from skill names to bit indices

// This allows us to represent skills as bits in an integer

skillToBit = new HashMap<>();

n = req\_skills.length;

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

skillToBit.put(req\_skills[i], i); // Skill -> bit position (0-indexed)

}

// Step 2: Convert each person's skills to a bitmask

// Example: if person has skills corresponding to bits 0 and 2, their mask = 101 (binary) = 5

peopleMasks = new int[people.size()];

for (int i = 0; i < people.size(); i++) {

int mask = 0; // Start with empty skill set

for (String skill : people.get(i)) {

// Set the bit corresponding to this skill

mask |= (1 << skillToBit.get(skill));

}

peopleMasks[i] = mask; // Store this person's skill bitmask

}

// Step 3: Initialize memoization arrays

// We have 2^n possible skill combinations (each skill can be present or absent)

int totalMasks = 1 << n; // 2^n

memo = new int[totalMasks];

teamMemo = new ArrayList[totalMasks];

Arrays.fill(memo, -1); // Mark all states as uncomputed

// Step 4: Start DFS from the initial state (mask = 0, no skills covered)

dfs(0);

// Step 5: Extract the result from the computed optimal team

// teamMemo[0] contains the optimal team that covers all skills starting from nothing

List<Integer> resultTeam = teamMemo[0];

int[] result = new int[resultTeam.size()];

for (int i = 0; i < resultTeam.size(); i++) {

result[i] = resultTeam.get(i); // Convert List<Integer> to int[]

}

return result;

}

/\*\*

\* Recursive DFS function that computes the minimum team needed to achieve a given skill set

\* @param mask Current skill set represented as a bitmask

\* @return Minimum number of people needed to achieve this skill set

\*/

private int dfs(int mask) {

// Base case: All skills are covered, no remaining skills (all n bits are set)

// mask = 111...1 (n times) = (1 << n) - 1

// No additional people needed - return empty team

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

teamMemo[mask] = new ArrayList<>(); // Empty team needed (already have all skills)

return 0; // No additional people needed

}

// Memoization check: if we've already computed this state, return stored result

if (memo[mask] != -1) {

return memo[mask];

}

int minPeople = Integer.MAX\_VALUE; // Track minimum team size found

List<Integer> bestTeam = null; // Track the actual team that achieves this minimum

// Try adding each person to see if they help cover more skills

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

// Calculate new skill set by adding this person's skills

int newMask = mask | peopleMasks[i];

// Only proceed if this person actually adds new skills (avoid redundant additions)

if (newMask != mask) {

// Recursively compute the team needed for the new skill set

int result = dfs(newMask);

// If a valid team was found for the new skill set

if (result != Integer.MAX\_VALUE) {

// Check if adding this person gives us a better team

if (result + 1 < minPeople) {

minPeople = result + 1;

// Create a new team by copying the optimal team for newMask

// and adding the current person

bestTeam = new ArrayList<>(teamMemo[newMask]);

bestTeam.add(i);

}

}

}

}

// Store computed results in memoization arrays

memo[mask] = minPeople;

teamMemo[mask] = bestTeam;

return minPeople;

}

}

**Key Points Explained:**

**Bitmask Representation**: Each skill is represented by a bit (0 = absent, 1 = present)

**Memoization**: Stores both the team size and the actual team composition for each state

**DFS Recursion**: Explores all possible ways to add people to cover missing skills

**Base Case**: When all skills are covered (all bits set), no more people needed

**State Transition**: For each person, compute new skill set and recursively find optimal team

**Optimal Team Tracking**: Keeps track of both the size and composition of the best team

Let me explain this base case in detail with a concrete example:

**Understanding the Base Case**

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

teamMemo[mask] = new ArrayList<>(); // Empty team

return 0; // No people needed

}

**What does (1 << n) - 1 mean?**

n = number of required skills (e.g., 3 skills: "java", "nodejs", "reactjs")

1 << n = 1 << 3 = 8 (binary: 1000)

(1 << n) - 1 = 8 - 1 = 7 (binary: 111)

So (1 << n) - 1 creates a bitmask with all n bits set to 1.

**Example with n=3:**

Skill mapping:

"java" → bit 0

"nodejs" → bit 1

"reactjs" → bit 2

(1 << 3) - 1 = 7 (binary: 111)

This means: all 3 skills are covered

**How teamMemo Works**

**teamMemo[mask]** = the list of person indices that are **still required** to achieve all skills, **given that we already have the skills represented by mask**.

Or more precisely:

**teamMemo[mask]** = the optimal team needed to **cover the remaining skills** (the skills not yet in **mask**), when we **already have** the skills in **mask**.

**Why This is More Accurate:**

**It's about the "remaining work"**: The function doesn't build a team that achieves **mask**, but rather a team that completes the missing skills.

**Base case makes sense**: When **mask** has all skills (**111...1**), no more people are required → empty team.

**Recursive logic matches**: When we call **dfs(mask)**, we're asking: "Given that we already have skills **mask**, what's the smallest team needed to get the rest?"

**Base Case Explanation:**

When mask == (1 << n) - 1 (all skills covered):

teamMemo[mask] = new ArrayList<>() → **empty team** (no people needed)

return 0 → **0 people** required

This makes sense because:

If we already have all skills covered (mask = 111), we don't need to add any more people

The team required is empty ([])

The number of people needed is 0

**Concrete Example Walkthrough**

**Input:**

req\_skills = ["java","nodejs","reactjs"]

people = [["java"],["nodejs"],["nodejs","reactjs"]]

**Skill mapping:**

"java" → bit 0

"nodejs" → bit 1

"reactjs" → bit 2

**People bitmasks:**

Person 0: ["java"] → 001 (binary) = 1

Person 1: ["nodejs"] → 010 (binary) = 2

Person 2: ["nodejs","reactjs"] → 110 (binary) = 6

**Base case trigger:**

When mask = 111 (binary) = 7 (all skills covered)

teamMemo[7] = [] (empty list)

Returns 0

**How it builds up:**

Someone calls dfs(110) (skills: nodejs + reactjs, missing java)

dfs(110) tries adding Person 0 (java): 110 | 001 = 111 (all skills)

dfs(111) is base case → returns 0 with team []

dfs(110) now knows: adding Person 0 gives team size 1 (0 + 1)

teamMemo[110] = [0] (just Person 0)

Returns 1

This propagates back up the recursion tree until we get the optimal team for the starting state mask = 0.

**Visual Representation**

teamMemo array index = skill mask (binary) = skills covered

teamMemo[0] (000) = [0, 2] // Needs java + (nodejs+reactjs)

teamMemo[1] (001) = [0] // Only java covered

teamMemo[2] (010) = [1] // Only nodejs covered

teamMemo[3] (011) = [0, 1] // java + nodejs

teamMemo[4] (100) = [2] // Only reactjs covered (if it existed)

teamMemo[6] (110) = [0, 2] // nodejs + reactjs + java

teamMemo[7] (111) = [] // BASE CASE: all skills covered

The base case establishes that when we already have all skills, we need 0 people and an empty team. This serves as the foundation that the recursion builds upon.

**How This DFS Translates to Bottom-Up DP**

**1. State Definition**

**DFS state**: dfs(mask) = min people needed to cover skill set mask

**DP state**: dp[mask] = min people needed to cover skill set mask

**2. Base Case**

**DFS**: mask == (1<<n)-1 → return 0

**DP**: dp[(1<<n)-1] = 0

**3. State Transition**

**DFS**: For each person, newMask = mask | personMask, then dfs(newMask) + 1

**DP**: For each mask, for each person, dp[newMask] = min(dp[newMask], dp[mask] + 1)

**4. Translation to Bottom-Up**

The bottom-up DP is essentially the **iterative version** of the DFS with memoization:

// Initialize DP array (similar to memo initialization)

int[] dp = new int[1 << n];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0; // Base case: no skills needed

// Iterate through all masks (similar to DFS recursion)

for (int mask = 0; mask < (1 << n); mask++) {

if (dp[mask] == Integer.MAX\_VALUE) continue;

// Try adding each person (similar to DFS for loop)

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

int newMask = mask | peopleMasks[i];

if (dp[newMask] > dp[mask] + 1) {

dp[newMask] = dp[mask] + 1;

}

}

}

**5. DP Solution exactly corresponding to DFS Solution**

Based on your DFS solution, here's the DP solution that maintains the exact same logic and iteration order:

import java.util.\*;

class Solution {

public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

int n = req\_skills.length;

// Create mapping from skill name to bit index

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

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

skillBitMap.put(req\_skills[i], i);

}

// Convert each person's skills to a bitmask

int[] peopleMasks = new int[people.size()];

for (int i = 0; i < people.size(); i++) {

int mask = 0;

for (String skill : people.get(i)) {

mask |= (1 << skillBitMap.get(skill));

}

peopleMasks[i] = mask;

}

int totalMasks = 1 << n;

// dp[mask] = minimum number of people needed to cover remaining skills

// given that we already have skills in mask

int[] dp = new int[totalMasks];

Arrays.fill(dp, Integer.MAX\_VALUE);

// teamDp[mask] = optimal team needed to cover remaining skills

List<Integer>[] teamDp = new ArrayList[totalMasks];

// Base case: all skills covered (mask = 111...1)

int fullMask = (1 << n) - 1;

dp[fullMask] = 0;

teamDp[fullMask] = new ArrayList<>();

// Process masks in reverse order (from full mask down to 0)

// This corresponds to the DFS order where we start from full mask and work backwards

for (int mask = fullMask; mask >= 0; mask--) {

// Avoid compute based on unreachable 'mask', if not adding

// this check, since teamDp only initialize for 'fullMask'

// as 'new ArrayList<>()', if hit not initialized 'mask',

// the NullPointerException error will come out

if (dp[mask] == Integer.MAX\_VALUE) continue;

// Try adding each person (same order as DFS)

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

int personMask = peopleMasks[i];

// Calculate the mask we had before adding this person

// This is the inverse of the DFS logic

int prevMask = mask & ~personMask;

// Only proceed if this person actually contributes to the current mask

// This corresponds to: if (newMask != mask) in DFS

if (prevMask != mask) {

if (dp[prevMask] > dp[mask] + 1) {

dp[prevMask] = dp[mask] + 1;

teamDp[prevMask] = new ArrayList<>(teamDp[mask]);

teamDp[prevMask].add(i);

}

}

}

}

// Extract result for mask = 0 (no skills initially)

List<Integer> resultTeam = teamDp[0];

int[] result = new int[resultTeam.size()];

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

result[i] = resultTeam.get(i);

}

return result;

}

}

**How This Exactly Corresponds to Your DFS:**

**1. State Definition:**

**DFS**: requiredPeople[mask] = min people needed from state mask

**DP**: dp[mask] = min people needed from state mask

**2. Base Case:**

**DFS**: mask == (1<<n)-1 → return 0 with empty team

**DP**: dp[fullMask] = 0 with empty team

**3. Iteration Order:**

**DFS**: Processes states recursively from mask=0 towards fullMask

**DP**: Processes states iteratively from fullMask down to 0 (reverse order)

**4. Person Processing:**

**DFS**: for(int i = 0; i < peopleMasks.length; i++) (forward order)

**DP**: for (int i = 0; i < peopleMasks.length; i++) (same forward order)

**5. Skill Update Logic:**

**DFS**: newMask = mask | peopleMasks[i] (add skills)

**DP**: prevMask = mask & ~personMask (remove skills - inverse operation)

**6. Usefulness Check:**

**DFS**: if (newMask != mask) (person adds new skills)

**DP**: if (prevMask != mask) (person contributes to current skills)

**7. Team Building:**

**DFS**: Copy team from newMask and add current person

**DP**: Copy team from mask and add current person to prevMask

**Why Reverse Iteration?**

The DP uses reverse iteration because:

**DFS goes forward**: from mask=0 → fullMask (adding skills)

**DP goes backward**: from fullMask → mask=0 (removing skills)

**Same logic**: Both approaches explore the same state transitions, just in opposite directions

**Dependency order**: To compute dp[prevMask], we need to know dp[mask] first

This DP solution exactly mirrors your DFS logic but uses iterative bottom-up processing instead of recursive top-down, making it more efficient while maintaining the same algorithmic structure.

**Actually if we not have this statement 'if (dp[mask] == Integer.MAX\_VALUE) continue;', we will encounter issue when hit line 'teamDp[prevMask] = new ArrayList<>(teamDp[mask]);', the error is below:**

java.lang.NullPointerException: Cannot invoke "java.util.Collection.toArray()" because "c" is null

at line 181, java.base/java.util.ArrayList.<init>

at line 55, Solution.smallestSufficientTeam

at line 56, \_\_DriverSolution\_\_.\_\_helper\_\_

at line 89, \_\_Driver\_\_.main

You're absolutely right! The line if (dp[mask] == Integer.MAX\_VALUE) continue; is **essential** to prevent NullPointerException. Let me explain why:

**The Problem Without the Check**

for (int mask = fullMask; mask >= 0; mask--) {

// WITHOUT THIS CHECK: if (dp[mask] == Integer.MAX\_VALUE) continue;

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

int prevMask = mask & ~personMask;

teamDp[prevMask] = new ArrayList<>(teamDp[mask]); // ← NULL POINTER EXCEPTION HERE

}

}

**Why the Exception Occurs:**

**Initialization**: teamDp array is initialized but all elements are null

List<Integer>[] teamDp = new ArrayList[totalMasks]; // All elements are null

**Base Case**: Only teamDp[fullMask] is initialized to empty list

teamDp[fullMask] = new ArrayList<>();

**The Issue**: When dp[mask] == Integer.MAX\_VALUE, it means:

This state mask is **unreachable** from the base case

teamDp[mask] is still null (was never initialized)

Trying to copy new ArrayList<>(null) throws NullPointerException

**How It Maps to DFS:**

In the DFS solution, this is handled naturally by the memoization check:

private int helper(int mask, int n) {

if (requiredPeople[mask] != -1) { // ← Prevents recomputation of unreachable states

return requiredPeople[mask];

}

// ... only reachable states are computed

}

The DFS never tries to compute unreachable states because the memoization check returns early.

**The Fix in DP:**

The line if (dp[mask] == Integer.MAX\_VALUE) continue; ensures that:

**Only process reachable states**: States that have valid values in dp[mask]

**Avoid null references**: teamDp[mask] is guaranteed to be initialized for these states

**Maintain correctness**: Only propagate from valid, reachable states

**Complete Correct Flow:**

// Initialize

dp[fullMask] = 0;

teamDp[fullMask] = new ArrayList<>(); // ✓ Initialized

for (int mask = fullMask; mask >= 0; mask--) {

if (dp[mask] == Integer.MAX\_VALUE) continue; // ✓ Skip unreachable states

for (int person : people) {

int prevMask = mask & ~personMask;

// teamDp[mask] is guaranteed to be non-null here

teamDp[prevMask] = new ArrayList<>(teamDp[mask]); // ✓ Safe

teamDp[prevMask].add(i);

}

}

This check is **not just an optimization** - it's **essential for correctness** to avoid null pointer exceptions and ensure the algorithm only processes valid, reachable states.

**If you want to avoid global variables and pass memo and teamMemo as parameters, you'll need to return both the team size and the team composition from the DFS method. Here's how it would look:**

If you want to avoid global variables and pass memo and teamMemo as parameters, you'll need to return both the team size and the team composition from the DFS method. Here's how it would look:

**Modified Solution with Parameters**

import java.util.\*;

class Solution {

private Map<String, Integer> skillToBit;

private int[] peopleMasks;

private int n;

// Helper class to return both team size and team composition

class Result {

int size;

List<Integer> team;

Result(int size, List<Integer> team) {

this.size = size;

this.team = team;

}

}

public int[] smallestSufficientTeam(String[] req\_skills, List<List<String>> people) {

// Step 1: Create skill mapping

skillToBit = new HashMap<>();

n = req\_skills.length;

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

skillToBit.put(req\_skills[i], i);

}

// Step 2: Convert people to bitmasks

peopleMasks = new int[people.size()];

for (int i = 0; i < people.size(); i++) {

int mask = 0;

for (String skill : people.get(i)) {

mask |= (1 << skillToBit.get(skill));

}

peopleMasks[i] = mask;

}

// Step 3: Initialize memoization arrays as parameters

int totalMasks = 1 << n;

Result[] memo = new Result[totalMasks]; // Now stores both size and team

// Step 4: Start DFS with memo array as parameter

Result result = dfs(0, memo);

// Step 5: Convert to output format

int[] output = new int[result.team.size()];

for (int i = 0; i < result.team.size(); i++) {

output[i] = result.team.get(i);

}

return output;

}

private Result dfs(int mask, Result[] memo) {

// Base case: All skills covered

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

return new Result(0, new ArrayList<>());

}

// Memoization check

if (memo[mask] != null) {

return memo[mask];

}

int minSize = Integer.MAX\_VALUE;

List<Integer> bestTeam = null;

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

int newMask = mask | peopleMasks[i];

if (newMask != mask) {

Result subResult = dfs(newMask, memo);

if (subResult.size != Integer.MAX\_VALUE) {

if (subResult.size + 1 < minSize) {

minSize = subResult.size + 1;

// Create new team by copying and adding current person

bestTeam = new ArrayList<>(subResult.team);

bestTeam.add(i);

}

}

}

}

// Handle case where no valid team was found

if (minSize == Integer.MAX\_VALUE) {

memo[mask] = new Result(Integer.MAX\_VALUE, null);

} else {

memo[mask] = new Result(minSize, bestTeam);

}

return memo[mask];

}

}

**Key Changes:**

**Result class**: Stores both the team size and team composition

**Single memo array**: Result[] memo replaces both memo[] and teamMemo[]

**Parameter passing**: memo array is passed as parameter to dfs()

**Return type**: dfs() now returns a Result object instead of just an int

**Alternative: Using Pair Class (Java 14+)**

If you're using Java 14+, you can use records for cleaner code:

record TeamResult(int size, List<Integer> team) {}

private TeamResult dfs(int mask, TeamResult[] memo) {

// Same logic, but using record instead of custom class

}

**Why This Approach is Better:**

**No global state**: All data is passed explicitly as parameters

**Thread safety**: No shared mutable state between method calls

**Cleaner encapsulation**: Each method call is self-contained

**Easier testing**: Can test DFS method in isolation with different memo arrays

However, note that this approach still has the same **performance characteristics** as the original solution - it just organizes the state differently. The time and space complexity remain the same.

**Time and Space Complexity Analysis for DFS Solution**

**Time Complexity: O(n × 2^m × m)**

Where:

n = number of people (up to 60)

m = number of skills (up to 16)

**Breakdown:**

**Number of states**: 2^m (all possible skill combinations)

**For each state**: Process up to n people

**For each person**: Check skill overlap (O(m) operations for bitwise OR and comparison)

**Team copying**: O(m) in worst case (team size ≤ m)

**Total**: O(2^m × n × m)

**Space Complexity: O(2^m × m)**

Where:

m = number of skills (up to 16)

**Breakdown:**

**Memo array**: 2^m states × (team size up to m) = O(2^m × m)

**Recursion stack**: O(m) depth (one level per skill bit set)

**Skill mappings**: O(m + n) for skill and people bitmasks

**Total**: O(2^m × m) (dominated by memo storage)

**Concrete Numbers:**

m = 16 → 2^16 = 65,536 states

n = 60 people

**Operations**: 65,536 × 60 × 16 ≈ **63 million operations**

**Memory**: 65,536 × 16 × 4 bytes ≈ **4 MB** (for team storage)

**Why This is Feasible:**

**Exponential in skills (m)**, not people (n)

m = 16 is manageable (2^16 = 65,536)

n = 60 is reasonable (60 people to process per state)

**Bitmask operations** are very fast (O(1) for most operations)

**Comparison with Backtracking Without Memo:**

**Without memo**: O(2^n) = O(2^60) ≈ **1.15e18 operations** (impossible)

**With memo**: O(n × 2^m × m) ≈ **63 million operations** (feasible)

The memoization reduces the complexity from exponential in people to exponential in skills, which makes the problem solvable within the constraints.

**Refer to**

[L77.Combinations](note://1337216AB2164464BA7362B6FA4FFFC9)

[L1986.Minimum Number of Work Sessions to Finish the Tasks (Ref.L473,L698)](note://WEB89175f2f8ba2facb8115b472227a272b)

[L2397.Maximum Rows Covered by Columns (Ref.L473)](note://WEB050bb59c3f216a8c91af09746dbf6efe)