1125. Smallest Sufficient Team II DP II Bit Manipulation

16 July 2023 07:51 AM

1125. Smallest Sufficient Team

Hint

Hard 1K 20

Companies

In a project, you have a list of required skills `req_skills`, and a list of people. The `i`th 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:

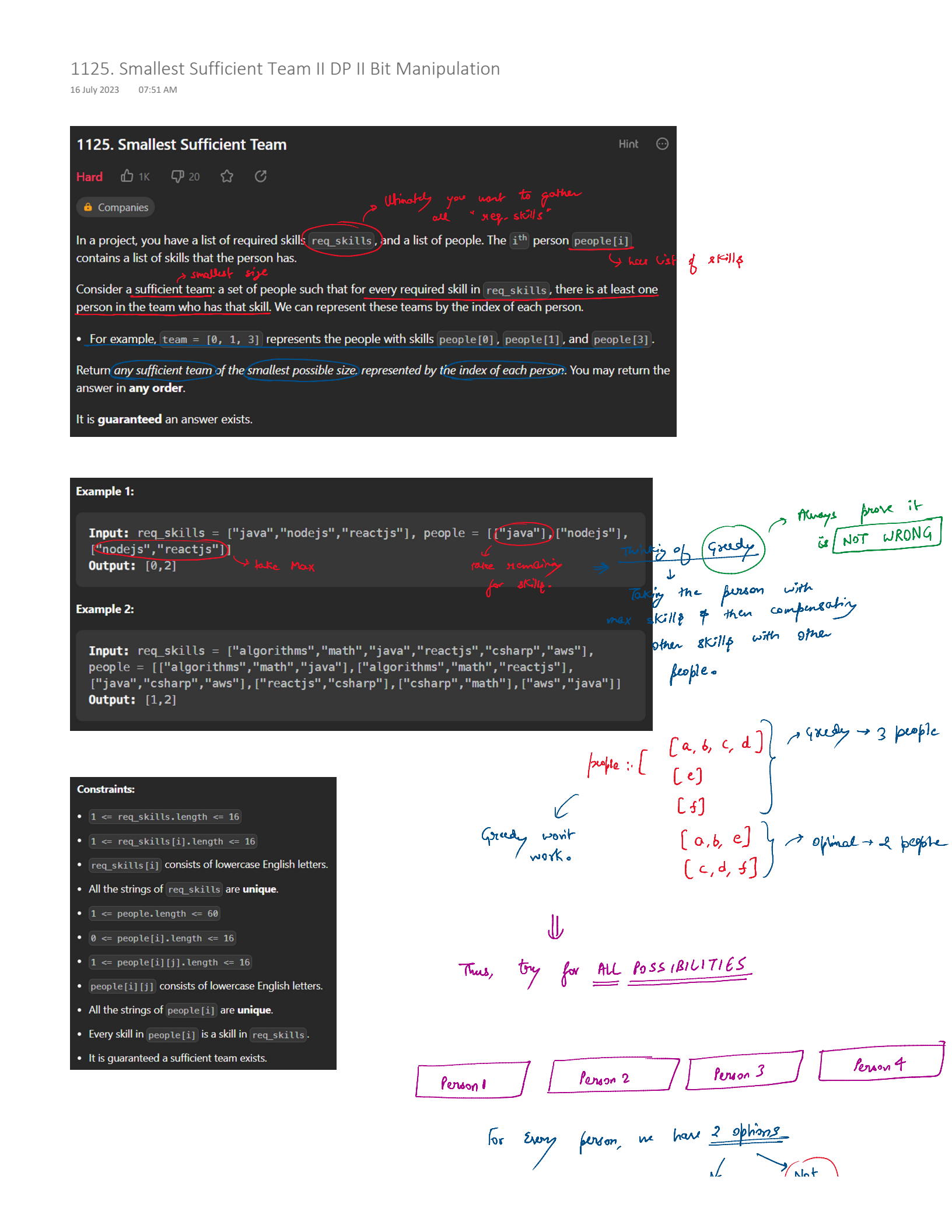
```
Input: req_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],["nodejs","reactjs"]]
Output: [0,2]
```

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.



For every person, we have 2 options

Take it

Not take it → Skip the i^{th} person & go on to next person..

Take what → Take skills

Take skills of that i^{th} Person.

[Convert skills like these below]

we can have 16 strings skills which we can take

we need to take all req_skills in the end

```
Input: req_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],
["nodejs","reactjs"]]
Output: [0,2]
```

0 1 1 | 1 0 0 | 0 1 0

req_skills = 1 1 1

Just making a mask of skills → So, that at every person, we would know if we want that skill or not..

How to make Mask of skills & people :

Example 1:

→ Give indexes to req_skills

0 1 2

```
Input: req_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],
["nodejs","reactjs"]]
Output: [0,2]
```

1 | 10 | 100

index 1 = 10, index 2 = 100

```
12      unordered_map<string, int> skills;
13      for (int i = 0; i < n; ++i)
14          skills[req_skills[i]] = i;
```

110 [OR operation]

```
16      vector<int> people;
17      for (auto &v: people_skills) {
18          int h = 0;
19          for (string &skill: v)
20              h |= 1 << skills[skill];
21
22          people.push_back(h);
23      }
```

get index

left shift

OR operation..

```
25      req_mask = (1 << n) - 1;
26
27      memset(dp, -1, 1 << n + 8);
28      memset(choice, 0, 1 << n + 6);
29
30      solve(people, 0, 0);
```

ultimately requiring every skill

simple Memoization

To keep track if ith person was choosen or not

index

mask

```
48  int solve(vector<int> &people, int i, int mask) {
49      if (mask == req_mask)
50          return 0;
51      if (i == m)
52          return 10000;
53
54      int &ans = dp[mask][i];
55      if (ans != -1)
56          return ans;
57
58      int ans1 = solve(people, i+1, mask | people[i]) + 1;
59      int ans2 = solve(people, i+1, mask);
60
61      if (ans1 < ans2)
62          choice[mask][i] = 1;
63
64      return ans = min(ans1, ans2);
```

if reached required mask return a 0

if reached end return high value

Memoization

adding skills in mask.

include ith person

don't include ith person

return min of both ways

if including ith person is beneficial → Push your choice as (1)

```
32      // build the answer through the choice matrix (see the solve function first)
33      vector<int> ans;
34      for (int i = 0, mask = 0; i < m; ++i) {
35          // did we include the i'th person?
36          if (choice[mask][i]) {
37              ans.push_back(i);
38              mask |= people[i];
39          }
40
41          if (mask == req_mask)
42              break;
43      }
44
45      return ans;
46  }
```

See if you need to include ith person or not

As you reach all req-skills break

return ans

CODE

```
int dp[1 << 16][64];
bool choice[1 << 16][64];
class Solution {
public:
    int req_mask, m;

    vector<int> smallestSufficientTeam(vector<string>& req_skills, vector<vector<string>>&
people_skills) {
        int n = req_skills.size();
        m = people_skills.size();

        unordered_map<string, int> skills;
        for (int i = 0; i < n; ++i)
            skills[req_skills[i]] = i;

        vector<int> people;
        for (auto &v: people_skills) {
            int h = 0;
            for (string &skill: v)
                h |= 1 << skills[skill];

            people.push_back(h);
        }

        req_mask = (1 << n) - 1;

        memset(dp, -1, 1 << n + 8);
        memset(choice, 0, 1 << n + 6);

        solve(people, 0, 0);

        // build the answer through the choice matrix (see the solve function first)
        vector<int> ans;
        for (int i = 0, mask = 0; i < m; ++i) {
            // did we include the i'th person?
            if (choice[mask][i]) {
                ans.push_back(i);
                mask |= people[i];
            }

            if (mask == req_mask)
                break;
        }

        return ans;
    }

    int solve(vector<int> &people, int i, int mask) {
        if (mask == req_mask)
            return 0;
        if (i == m)
            return 10000;

        int &ans = dp[mask][i];
        if (ans != -1)
            return ans;

        int ans1 = solve(people, i+1, mask | people[i]) + 1;
        int ans2 = solve(people, i+1, mask);

        if (ans1 < ans2)
            choice[mask][i] = 1;

        return ans = min(ans1, ans2);
    }
};
```

time: $O(people * 2^{skills})$

space: $O(people * 2^{skills})$

Bottom Up Approach would also be same :)

↳ As you were going to every person, same here go to every person & see its contribution in 2^{skills} possible options...

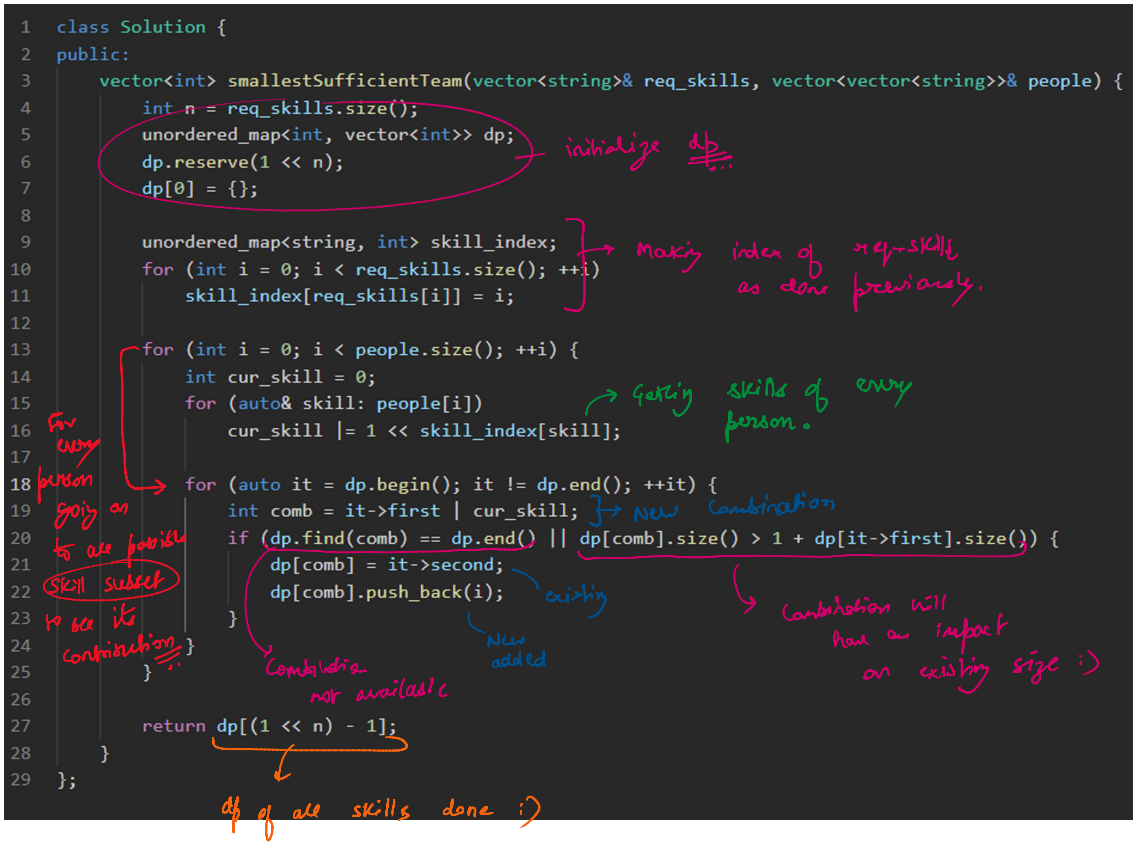
if req_skills = 5

⇒ total option = 2^5

i.e.

$[0, 2^5 - 1]$

[00000, ... , 11111]



time: $O(\text{people} + 2^{\text{skills}})$

space: $O(2^{\text{skills}})$

CODE

```
class Solution {
public:
    vector<int> smallestSufficientTeam(vector<string>& req_skills, vector<vector<string>>&
people) {
        int n = req_skills.size();
        unordered_map<int, vector<int>> dp;
        dp.reserve(1 << n);
        dp[0] = {};
        unordered_map<string, int> skill_index;
        for (int i = 0; i < req_skills.size(); ++i)
            skill_index[req_skills[i]] = i;

        for (int i = 0; i < people.size(); ++i) {
            int cur_skill = 0;
            for (auto& skill: people[i])
                cur_skill |= 1 << skill_index[skill];

            for (auto it = dp.begin(); it != dp.end(); ++it) {
                int comb = it->first | cur_skill;
```

```
                if (dp.find(comb) == dp.end() || dp[comb].size() > 1 + dp[it->first].size())
{
                    dp[comb] = it->second;
                    dp[comb].push_back(i);
                }
            }
        }

        return dp[(1 << n) - 1];
    }
};
```