# LeetCode\_Day\_38\_DP\_II Knapsack

The original Knapsack problem, is that we have some items with specific weight and value, and we have a bag with limited size of weight, we want to carry as much value as we can.

The extended Knapsack is even more popular, the bag has no limit, even item can be duplicated, but to combine them to generate a specific weight or value how many choices are there?

For both problems the code can be similar, we simply calculate on specific weight, add the current item to the extsing combinations of weight and value, so we generate a new set of weight-value combination for the next item. Please remember without the current item the previous combination should still stand.

## 879. Profitable Schemes

Hard

There are G people in a gang, and a list of various crimes they could commit.

The i-th crime generates a profit[i] and requires group[i] gang members to participate.

If a gang member participates in one crime, that member can't participate in another crime.

Let's call a *profitable scheme* any subset of these crimes that generates at least P profit, and the total number of gang members participating in that subset of crimes is at most G.

How many schemes can be chosen?  Since the answer may be very large, **return it modulo** 10^9 + 7.

**Example 1:**

**Input:** G = 5, P = 3, group = [2,2], profit = [2,3]

**Output:** 2

**Explanation:**

To make a profit of at least 3, the gang could either commit crimes 0 and 1, or just crime 1.

In total, there are 2 schemes.

**Example 2:**

**Input:** G = 10, P = 5, group = [2,3,5], profit = [6,7,8]

**Output:** 7

**Explanation:**

To make a profit of at least 5, the gang could commit any crimes, as long as they commit one.

There are 7 possible schemes: (0), (1), (2), (0,1), (0,2), (1,2), and (0,1,2).

**Note:**

1. 1 <= G <= 100
2. 0 <= P <= 100
3. 1 <= group[i] <= 100
4. 0 <= profit[i] <= 100
5. 1 <= group.length = profit.length <= 100

### Analysis:

Consider each crime is an item and each item can only be used once, the Knapsack is a matrix of number of people used, and how much profit generated, with the count on how many combination of crime schema you can reach this state.

Loop the crime at the outer loop, add it to all the existing crime schema.

Because each crime can commit only once, to avoid double count, we scane reversely.

/// <summary>

/// Leet code #879. Profitable Schemes

///

/// There are G people in a gang, and a list of various crimes they could

/// commit.

///

/// The i-th crime generates a profit[i] and requires group[i] gang

/// members to participate.

///

/// If a gang member participates in one crime, that member can't

/// participate in another crime.

///

/// Let's call a profitable scheme any subset of these crimes that

/// generates at least P profit, and the total number of gang members

/// participating in that subset of crimes is at most G.

///

/// How many schemes can be chosen? Since the answer may be very large,

/// return it modulo 10^9 + 7.

///

/// Example 1:

/// Input: G = 5, P = 3, group = [2,2], profit = [2,3]

/// Output: 2

/// Explanation:

/// To make a profit of at least 3, the gang could either commit crimes

/// 0 and 1, or just crime 1.

/// In total, there are 2 schemes.

///

/// Example 2:

///

/// Input: G = 10, P = 5, group = [2,3,5], profit = [6,7,8]

/// Output: 7

/// Explanation:

/// To make a profit of at least 5, the gang could commit any crimes, as

/// long as they commit one.

/// There are 7 possible schemes: (0), (1), (2), (0,1), (0,2), (1,2), and

/// (0,1,2).

///

///

/// Note:

///

/// 1. 1 <= G <= 100

/// 2. 0 <= P <= 100

/// 3. 1 <= group[i] <= 100

/// 4. 0 <= profit[i] <= 100

/// 5. 1 <= group.length = profit.length <= 100

/// </summary>

int LeetCode::profitableSchemes(int G, int P, vector<int>& group, vector<int>& profit)

{

int result = 0;

int mod = 1000000007;

vector<vector<int>> schemes(G+1, vector<int>(P+1));

schemes[0][0] = 1;

for (size\_t i = 0; i < group.size(); i++)

{

for (int j = G; j >= 0; j--)

{

int g = j + group[i];

if (g > G) continue;

for (int k = P; k >= 0; k--)

{

if (schemes[j][k] == 0) continue;

int p = k + profit[i];

if (p > P) p = P;

int count = schemes[j][k];

schemes[g][p] = (schemes[g][p] + count) % mod;

}

}

}

for (auto j = 0; j <= G; j++)

{

result = (result + schemes[j][P]) % mod;

}

return result;

}

## 1125. Smallest Sufficient Team

Hard

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 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 <= people.length <= 60
* 1 <= people[i].length, req\_skills[i].length, people[i][j].length <= 16
* Elements of req\_skills and people[i] are (respectively) distinct.
* req\_skills[i][j], people[i][j][k] are lowercase English letters.
* Every skill in people[i] is a skill in req\_skills.
* It is guaranteed a sufficient team exists.

### Analysis:

Consider each skill as a bit map, loop the people at outside, add the skill to the existing skill set, if we got a smaller team, we replace the team. The full skill set is the answer.

/// <summary>

/// Leet code #1125. Smallest Sufficient Team

///

/// 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

/// 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. 1 <= req\_skills.length <= 16

/// 2. 1 <= people.length <= 60

/// 3. 1 <= people[i].length, req\_skills[i].length, people[i][j].length <= 16

/// 4. Elements of req\_skills and people[i] are (respectively) distinct.

/// 5. req\_skills[i][j], people[i][j][k] are lowercase English letters.

/// 6. It is guaranteed a sufficient team exists.

/// </summary>

vector<int> LeetCode::smallestSufficientTeam(vector<string>& req\_skills, vector<vector<string>>& people)

{

unordered\_map<string, int> skill\_ids;

size\_t n = req\_skills.size();

for (size\_t i = 0; i < n; i++)

{

skill\_ids[req\_skills[i]] = (1 << i);

}

vector<vector<int>> team((1 << n), vector<int>());

team[0] = vector<int>();

for (size\_t i = 0; i < people.size(); i++)

{

int people\_skill = 0;

for (size\_t j = 0; j < people[i].size(); j++)

{

people\_skill |= skill\_ids[people[i][j]];

}

if (people\_skill == 0) continue;

for (size\_t j = 0; j < team.size(); j++)

{

if ((j != 0) && (team[j].size() == 0))

{

continue;

}

int skill = j | people\_skill;

if (team[skill].size() == 0 ||

team[skill].size() > team[j].size() + 1)

{

team[skill] = team[j];

team[skill].push\_back(i);

}

}

}

return team[(1 << n) - 1];

}