LeetCode Training Day 13 DP IV

Another set of DP problems is known as **Backpack** problem. It requires us to build up a large number from using the smaller numbers. For such problem, we normally iterate from smaller numbers to large numbers. For example, if we want to know how many pieces we need to build number X, and if we have a piece p, and we will end up dp[X] = dp[X-p] + 1.

## 322. Coin Change

Medium

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

**Example 1:**

**Input:** coins = [1,2,5], amount = 11

**Output:** 3

**Explanation:** 11 = 5 + 5 + 1

**Example 2:**

**Input:** coins = [2], amount = 3

**Output:** -1

**Example 3:**

**Input:** coins = [1], amount = 0

**Output:** 0

**Constraints:**

* 1 <= coins.length <= 12
* 1 <= coins[i] <= 231 - 1
* 0 <= amount <= 104

### Analysis:

On every amount, you can see if it is not visited (-1), then skip it, otherwise we add every type of coin to this value and build the future minimum coin result.

/// <summary>

/// Leet code #322. Coin Change

/// You are given coins of different denominations and a total amount of money

/// amount. Write a function to

/// compute the fewest number of coins that you need to make up that amount.

/// If that amount of money cannot

/// be made up by any combination of the coins, return -1.

/// Example 1:

/// coins = [1, 2, 5], amount = 11

/// return 3 (11 = 5 + 5 + 1)

/// Example 2:

/// coins = [2], amount = 3

/// return -1.

/// Note:

/// You may assume that you have an infinite number of each kind of coin.

/// </summary>

int LeetCodeDP::coinChange(vector<int>& coins, int amount)

{

vector<int> dp(amount + 1, -1);

for (int i = 0; i <= amount; i++)

{

if (i == 0) dp[i] = 0;

else if (dp[i] == -1) continue;

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

{

// take care overflow

if ((long long)i + coins[j] > (long long) amount) continue;

if (dp[i + coins[j]] == -1)

{

dp[i + coins[j]] = dp[i] + 1;

}

else

{

dp[i + coins[j]] = min(dp[i + coins[j]], dp[i] + 1);

}

}

}

return dp[amount];

}

## 518. Coin Change 2

Medium

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the number of combinations that make up that amount*. If that amount of money cannot be made up by any combination of the coins, return 0.

You may assume that you have an infinite number of each kind of coin.

The answer is **guaranteed** to fit into a signed **32-bit** integer.

**Example 1:**

**Input:** amount = 5, coins = [1,2,5]

**Output:** 4

**Explanation:** there are four ways to make up the amount:

5=5

5=2+2+1

5=2+1+1+1

5=1+1+1+1+1

**Example 2:**

**Input:** amount = 3, coins = [2]

**Output:** 0

**Explanation:** the amount of 3 cannot be made up just with coins of 2.

**Example 3:**

**Input:** amount = 10, coins = [10]

**Output:** 1

**Constraints:**

* 1 <= coins.length <= 300
* 1 <= coins[i] <= 5000
* All the values of coins are **unique**.
* 0 <= amount <= 5000

### Analysis:

For this problem, although the code is shorter, but it is more difficult, you should avoid use different coins on every possible amount. It is because in this case, you will consider 1+2 and 2+1 as different ways. The right solution is that you should use one coin all the way down, then you add another coin.

/// <summary>

/// Leet code #518. Coin Change 2

///

/// You are given coins of different denominations and a total amount of

/// money. Write a function to compute the number of combinations that

/// make up that amount. You may assume that you have infinite number of

/// each kind of coin.

///

/// Note: You can assume that

///

/// 1. 0 <= amount <= 5000

/// 2. 1 <= coin <= 5000

/// 3. the number of coins is less than 500

/// 4. the answer is guaranteed to fit into signed 32-bit integer

///

/// Example 1:

/// Input: amount = 5, coins = [1, 2, 5]

/// Output: 4

/// Explanation: there are four ways to make up the amount:

/// 5=5

/// 5=2+2+1

/// 5=2+1+1+1

/// 5=1+1+1+1+1

///

///

/// Example 2:

/// Input: amount = 3, coins = [2]

/// Output: 0

/// Explanation: the amount of 3 cannot be made up just with coins of 2.

///

///

/// Example 3:

/// Input: amount = 10, coins = [10]

/// Output: 1

/// </summary>

int LeetCodeDP::change(int amount, vector<int>& coins)

{

vector<int> dp(amount + 1);

dp[0] = 1;

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

{

for (int j = 0; j <= amount - coins[i]; j++)

{

dp[j + coins[i]] += dp[j];

}

}

return dp[amount];

}

## 494. Target Sum

Medium

You are given an integer array nums and an integer target.

You want to build an **expression** out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

* For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1".

Return the number of different **expressions** that you can build, which evaluates to target.

**Example 1:**

**Input:** nums = [1,1,1,1,1], target = 3

**Output:** 5

**Explanation:** There are 5 ways to assign symbols to make the sum of nums be target 3.

-1 + 1 + 1 + 1 + 1 = 3

+1 - 1 + 1 + 1 + 1 = 3

+1 + 1 - 1 + 1 + 1 = 3

+1 + 1 + 1 - 1 + 1 = 3

+1 + 1 + 1 + 1 - 1 = 3

**Example 2:**

**Input:** nums = [1], target = 1

**Output:** 1

**Constraints:**

* 1 <= nums.length <= 20
* 0 <= nums[i] <= 1000
* 0 <= sum(nums[i]) <= 1000
* -1000 <= target <= 1000

### Analysis:

This problem is also very common, to find out all possible sum, with positive or negative, you can use DFS or recursion. But there is another way to do so. You add calculate all possible result with first number and the use every first result along with second number to get second round of result, and so on, until the last number. Because there are so many intermediate results, we need to use hash table to track them. After every round we assign the new round of result to previous round.

For C++, we use reference as pointers, for Java and C#, it is pointer by default.

/// <summary>

/// Leet code #494. Target Sum

///

/// You are given a list of non-negative integers, a1, a2, ..., an, and a

/// target, S.

/// Now you have 2 symbols + and -. For each integer, you should choose one

/// from + and - as its new symbol.

/// Find out how many ways to assign symbols to make sum of integers equal

/// to target S.

/// Example 1:

/// Input: nums is [1, 1, 1, 1, 1], S is 3.

/// Output: 5

/// Explanation:

///

/// -1+1+1+1+1 = 3

/// +1-1+1+1+1 = 3

/// +1+1-1+1+1 = 3

/// +1+1+1-1+1 = 3

/// +1+1+1+1-1 = 3

///

/// There are 5 ways to assign symbols to make the sum of nums be target 3.

///

/// Note:

/// 1.The length of the given array is positive and will not exceed 20.

/// 2.The sum of elements in the given array will not exceed 1000.

/// 3.Your output answer is guaranteed to be fitted in a 32-bit integer.

/// </summary>

int LeetCodeDP::findTargetSumWays(vector<int>& nums, int S)

{

map<int, int> sum\_way1, sum\_way2;

map<int, int>\* p\_curr = &sum\_way1;

map<int, int>\* p\_next = &sum\_way2;

sum\_way1[0] = 1;

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

{

for (auto itr : \*p\_curr)

{

(\*p\_next)[itr.first + nums[i]] += itr.second;

(\*p\_next)[itr.first - nums[i]] += itr.second;

}

swap(p\_curr, p\_next);

p\_next->clear();

}

return (\*p\_curr)[S];

}

## 377. Combination Sum IV

Medium

Given an array of **distinct** integers nums and a target integer target, return *the number of possible combinations that add up to* target.

The test cases are generated so that the answer can fit in a **32-bit** integer.

**Example 1:**

**Input:** nums = [1,2,3], target = 4

**Output:** 7

**Explanation:**

The possible combination ways are:

(1, 1, 1, 1)

(1, 1, 2)

(1, 2, 1)

(1, 3)

(2, 1, 1)

(2, 2)

(3, 1)

Note that different sequences are counted as different combinations.

**Example 2:**

**Input:** nums = [9], target = 3

**Output:** 0

**Constraints:**

* 1 <= nums.length <= 200
* 1 <= nums[i] <= 1000
* All the elements of nums are **unique**.
* 1 <= target <= 1000

**Follow up:** What if negative numbers are allowed in the given array? How does it change the problem? What limitation we need to add to the question to allow negative numbers?

### Analysis:

This one is simpler than 518 Coin Change 2, because we say 1+2 and 2+1 are different. We simply add every number to previous calculated result. In case we have negative number, we can simply use solution 494.

/// <summary>

/// Leet code #377. Combination Sum IV

///

/// Given an integer array with all positive numbers and no duplicates,

/// find the number of possible combinations that add up to a positive

/// integer target.

///

/// Example:

/// nums = [1, 2, 3]

/// target = 4

/// The possible combination ways are:

/// (1, 1, 1, 1)

/// (1, 1, 2)

/// (1, 2, 1)

/// (1, 3)

/// (2, 1, 1)

/// (2, 2)

/// (3, 1)

/// Note that different sequences are counted as different combinations.

/// Therefore the output is 7.

/// Follow up:

/// What if negative numbers are allowed in the given array?

/// How does it change the problem?

/// What limitation we need to add to the question to allow negative numbers?

/// </summary>

int LeetCodeDP::combinationSum4(vector<int>& nums, int target)

{

vector<int> sum(target + 1);

for (int i = 0; i <= target; i++)

{

if (i == 0) sum[0] = 0;

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

{

if (i - nums[j] < 0) continue;

else if (i == nums[j])

{

sum[i] += 1;

}

else

{

sum[i] += sum[i - nums[j]];

}

}

}

return sum[target];

}

## 628. Maximum Product of Three Numbers

Easy

Given an integer array, find three numbers whose product is maximum and output the maximum product.

**Example 1:**

**Input:** [1,2,3]

**Output:** 6

**Example 2:**

**Input:** [1,2,3,4]

**Output:** 24

**Note:**

1. The length of the given array will be in range [3,104] and all elements are in the range [-1000, 1000].
2. Multiplication of any three numbers in the input won't exceed the range of 32-bit signed integer.

### Analysis:

The maximum product of 3 number comes from the maximum and minimum product of 2 numbers and the maximum or minimum product of 2 number comes from the maximum and minum product 1 number. This is because the product of two mnimum negative numbers can become the maximum product. We keep the result from round 3, round 2 then round 1 to avoid data get polluted in current round.

/// <summary>

/// Leet code #628. Maximum Product of Three Numbers

///

/// Given an integer array, find three numbers whose product is maximum

/// and output the maximum product.

///

/// Example 1:

/// Input: [1,2,3]

/// Output: 6

///

/// Example 2:

/// Input: [1,2,3,4]

/// Output: 24

///

/// Note:

/// 1.The length of the given array will be in range [3,10^4] and all

/// elements are in the range [-1000, 1000].

/// 2.Multiplication of any three numbers in the input won't exceed the

/// range of 32-bit signed integer.

/// </summary>

int LeetCode::maximumProduct(vector<int>& nums)

{

vector<pair<int, int>> product\_array = vector<pair<int, int>>(3, make\_pair(1, 1));

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

{

for (int k = (int)product\_array.size() - 1; k >= 0; k--)

{

if (i <= (size\_t)k)

{

product\_array[k].first \*= nums[i];

product\_array[k].second \*= nums[i];

}

else if (k == 0)

{

product\_array[k].first = min(product\_array[k].first, nums[i]);

product\_array[k].second = max(product\_array[k].second, nums[i]);

}

else

{

product\_array[k].first = min(product\_array[k].first, min(nums[i] \* product\_array[k - 1].first, nums[i] \* product\_array[k - 1].second));

product\_array[k].second = max(product\_array[k].second, max(nums[i] \* product\_array[k - 1].first, nums[i] \* product\_array[k - 1].second));

}

}

}

return product\_array[product\_array.size() - 1].second;

}

## 474. Ones and Zeroes

Medium

In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

For now, suppose you are a dominator of **m** 0s and **n** 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.

Now your task is to find the maximum number of strings that you can form with given **m** 0s and **n** 1s. Each 0 and 1 can be used at most **once**.

**Note:**

1. The given numbers of 0s and 1s will both not exceed 100
2. The size of given string array won't exceed 600.

**Example 1:**

**Input:** Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3

**Output:** 4

**Explanation:** This are totally 4 strings can be formed by the using of 5 0s and 3 1s, which are “10,”0001”,”1”,”0”

**Example 2:**

**Input:** Array = {"10", "0", "1"}, m = 1, n = 1

**Output:** 2

**Explanation:** You could form "10", but then you'd have nothing left. Better form "0" and "1".

### Analysis:

This problem is different from #377 in the case that each item can only be used once, so we should put the loop for items in the outer loop and add it to the the existing results, to avoid double count the item itself, we scan from end to the start.

/// <summary>

/// Leet code #474. Ones and Zeroes

///

/// In the computer world, use restricted resource you have to generate

/// maximum benefit is what we always want to pursue.

/// For now, suppose you are a dominator of m 0s and n 1s respectively.

/// On the other hand,

/// there is an array with strings consisting of only 0s and 1s.

/// Now your task is to find the maximum number of strings that you can

/// form with given m 0s and n 1s.

/// Each 0 and 1 can be used at most once.

/// Note:

/// The given numbers of 0s and 1s will both not exceed 100

/// The size of given string array won't exceed 600.

/// Example 1:

/// Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3

/// Output: 4

///

/// Explanation: This are totally 4 strings can be formed by the using of 5 0s

/// and 3 1s, which are “10,”0001”,”1”,”0”

///

/// Example 2:

/// Input: Array = {"10", "0", "1"}, m = 1, n = 1

/// Output: 2

///

/// Explanation: You could form "10", but then you'd have nothing left.

/// Better form "0" and "1".

/// </summary>

int LeetCode::findMaxOneZeroForm(vector<string>& strs, int m, int n)

{

// we end by m, n, and 0, 0 is a virtual start

int result = 0;

vector<vector<int>> dp(m+1, vector<int>(n+1));

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

{

string str = strs[i];

int zero = std::count(str.begin(), str.end(), '0');

int one = std::count(str.begin(), str.end(), '1');

// scan from end to start so we do not duplicate new item itself

for (int j = m; j >= zero; j--)

{

for (int k = n; k >= one; k--)

{

dp[j][k] = max(dp[j][k], dp[j - zero][k - one] + 1);

result = max(result, dp[j][k]);

}

}

}

return result;

}

# Advanced Problems

## 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 backpack 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];

}