LeetCode Training Day 14 DP IV Knapsack

Another set of DP problems is known as **Knapsack** problem. It requires us to build up a large number from using the smaller numbers. For such problems, 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.

The Knapsack problem can also be solved in a way of using two hash set, these hash set can be considered as two bags, starting with one empty bag, you put the first item, then you generate a bag with the weight and value of this item, then you add second item and generate combined weight and value, (please remember each item can always stand as itself without adding to others). After every iteration you add the new bag with the combination of new items to the old bag.

It is worth to note that if your total sum of number is within a very limited scope you can use array to replace the hash set, in this case when you add a new item please make sure you do from back (high number) to front (low number) to avoid new item get double count.

## 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];

}

## 416. Partition Equal Subset Sum

Medium

Given a **non-empty** array nums containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

**Example 1:**

**Input:** nums = [1,5,11,5]

**Output:** true

**Explanation:** The array can be partitioned as [1, 5, 5] and [11].

**Example 2:**

**Input:** nums = [1,2,3,5]

**Output:** false

**Explanation:** The array cannot be partitioned into equal sum subsets.

**Constraints:**

* 1 <= nums.length <= 200
* 1 <= nums[i] <= 100

### Analysis:

The solution is to find out if we use the existing numbers, can we build the sum which is half of the total sum from all numbers. We can use Knapsack strategy to build up the sum.

/// <summary>

/// Leet code #416. Partition Equal Subset Sum

///

/// Given a non-empty array containing only positive integers, find if

/// the array can be partitioned into two subsets such that the sum of

/// elements in both subsets is equal.

/// Note:

/// Each of the array element will not exceed 100.

/// The array size will not exceed 200.

/// Example 1:

/// Input: [1, 5, 11, 5]

/// Output: true

/// Explanation: The array can be partitioned as [1, 5, 5] and [11].

/// Example 2:

/// Input: [1, 2, 3, 5]

/// Output: false

/// Explanation: The array cannot be partitioned into equal sum subsets.

/// </summary>

bool LeetCodeDP::canPartition(vector<int>& nums)

{

int sum = 0;

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

{

sum += nums[i];

}

if (sum % 2 == 1) return false;

sum = sum / 2;

unordered\_set<int> curr;

unordered\_set<int> next;

curr.insert(0);

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

{

for (auto s : curr)

{

int n = s + nums[i];

if (n == sum) return true;

else if (n > sum) continue;

else

{

next.insert(n);

}

}

curr.insert(next.begin(), next.end());

}

return false;

}

## 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;

}

## 1049. Last Stone Weight II

Medium

You are given an array of integers stones where stones[i] is the weight of the ith stone.

We are playing a game with the stones. On each turn, we choose any two stones and smash them together. Suppose the stones have weights x and y with x <= y. The result of this smash is:

* If x == y, both stones are destroyed, and
* If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.

At the end of the game, there is **at most one** stone left.

Return *the smallest possible weight of the left stone*. If there are no stones left, return 0.

**Example 1:**

**Input:** stones = [2,7,4,1,8,1]

**Output:** 1

**Explanation:**

We can combine 2 and 4 to get 2, so the array converts to [2,7,1,8,1] then,

we can combine 7 and 8 to get 1, so the array converts to [2,1,1,1] then,

we can combine 2 and 1 to get 1, so the array converts to [1,1,1] then,

we can combine 1 and 1 to get 0, so the array converts to [1], then that's the optimal value.

**Example 2:**

**Input:** stones = [31,26,33,21,40]

**Output:** 5

**Example 3:**

**Input:** stones = [1,2]

**Output:** 1

**Constraints:**

* 1 <= stones.length <= 30
* 1 <= stones[i] <= 100

### Analysis:

First you need to think the problem in a different way. If there is only one stone, then the answer is itself. If there are multiple stones, we calculate the sum of some stones with the amount of less than half and find the maximum of it. Why we do so, because if we divide the all stones in two group. One with sum of more than half, another is less than half, then the difference is one of the answer candidates. The right answer is the minimum of such difference.

/// <summary>

/// Leet code #1049. Last Stone Weight II

///

/// We have a collection of rocks, each rock has a positive integer weight.

///

/// Each turn, we choose any two rocks and smash them together. Suppose

/// the stones have weights x and y with x <= y. The result of this smash is:

///

/// If x == y, both stones are totally destroyed;

/// If x != y, the stone of weight x is totally destroyed, and the stone of

/// weight y has new weight y-x.

/// At the end, there is at most 1 stone left. Return the smallest possible

/// weight of this stone (the weight is 0 if there are no stones left.)

///

///

/// Example 1:

///

/// Input: [2,7,4,1,8,1]

/// Output: 1

/// Explanation:

/// We can combine 2 and 4 to get 2 so the array converts to [2,7,1,8,1] then,

/// we can combine 7 and 8 to get 1 so the array converts to [2,1,1,1] then,

/// we can combine 2 and 1 to get 1 so the array converts to [1,1,1] then,

/// we can combine 1 and 1 to get 0 so the array converts to [1] then that's the

/// optimal value.

///

/// Note:

///

/// 1. 1 <= stones.length <= 30

/// 2. 1 <= stones[i] <= 100

/// </summary>

int LeetCodeDP::lastStoneWeightII(vector<int> &stones)

{

int sum = 0;

if (stones.size() == 1) return stones[0];

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

{

sum += stones[i];

}

vector<int> dp(sum / 2 + 1);

dp[0] = 1;

for (int a : stones)

{

for (int i = sum / 2; i >= a; --i)

{

dp[i] = dp[i] + dp[i - a];

}

}

for (int i = sum / 2; i > 0; --i)

{

if (dp[i]) return sum - i - i;

}

return 0;

}