完全背包：518. 零钱兑换 II

问题描述

我们有硬币的面值为 `[1, 2, 5]`，目标金额为 `10`。问题要求找出多少种方法能够组合这些硬币，凑成 `10`。

动态规划解法

动态规划的 `dp[j]` 表示用硬币组合成金额 `j` 的方案数。

例子：使用 1, 2, 5 的硬币，目标金额为 10

1. **初始化** dp **数组**：

   - `dp[0] = 1`，因为凑成金额为 `0` 的方案只有一种，就是不使用任何硬币。
   - 其他所有 `dp[i]` 初始化为 `0`，因为初始时还没有计算出任何组合方式。

初始 dp 数组：

```plaintext
dp = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

2. **遍历每种硬币**：

   - **硬币** 1：
     - 对于每个金额 `j`，从 `1` 开始一直到 `10`，计算可以用硬币 `1` 凑成金额 `j` 的方案数。

```plaintext
dp[1] += dp[1 - 1] => dp[1] = 1
dp[2] += dp[2 - 1] => dp[2] = 1
dp[3] += dp[3 - 1] => dp[3] = 1
dp[4] += dp[4 - 1] => dp[4] = 1
dp[5] += dp[5 - 1] => dp[5] = 1
dp[6] += dp[6 - 1] => dp[6] = 1
dp[7] += dp[7 - 1] => dp[7] = 1
dp[8] += dp[8 - 1] => dp[8] = 1
dp[9] += dp[9 - 1] => dp[9] = 1
dp[10] += dp[10 - 1] => dp[10] = 1
```

dp 数组更新为：

```plaintext
dp = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

   - **硬币** 2：
     - 对于每个金额 `j` 从 `2` 开始，计算可以用硬币 `2` 凑成金额 `j` 的方案数。

```plaintext
dp[2] += dp[2 - 2] => dp[2] = 2
dp[3] += dp[3 - 2] => dp[3] = 2
dp[4] += dp[4 - 2] => dp[4] = 3
dp[5] += dp[5 - 2] => dp[5] = 3
dp[6] += dp[6 - 2] => dp[6] = 4
dp[7] += dp[7 - 2] => dp[7] = 4
dp[8] += dp[8 - 2] => dp[8] = 5
dp[9] += dp[9 - 2] => dp[9] = 5
dp[10] += dp[10 - 2] => dp[10] = 6
```

dp 数组更新为：

```plaintext
dp = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6]
```

   - **硬币** 5：
     - 对于每个金额 `j` 从 `5` 开始，计算可以用硬币 `5` 凑成金额 `j` 的方案数。

```plaintext
dp[5] += dp[5 - 5] => dp[5] = 4
dp[6] += dp[6 - 5] => dp[6] = 5
dp[7] += dp[7 - 5] => dp[7] = 6
dp[8] += dp[8 - 5] => dp[8] = 7
dp[9] += dp[9 - 5] => dp[9] = 8
dp[10] += dp[10 - 5] => dp[10] = 10
```

dp 数组最终更新为：

```plaintext
dp = [1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 10]
```

结果：

- 通过 dp[10]，我们知道有 **10 种方式**可以使用硬币 [1, 2, 5] 凑成金额 10。

处理大数溢出问题

现在，假设你面对一个非常大的金额和硬币列表，导致计算过程中出现了**整数溢出**，例如当总的硬币组合数超过了 long long 的范围。这时，可能出现你提到的错误：

plaintext 复制

```
runtime error: signed integer overflow: 482256519916909723 + 8741127657625636188 cannot be represented in typ
```

这意味着即使使用 long long 也无法处理如此大的数值。

解决办法

1. **使用 __int128 处理极大数值**

如果你使用的编译器支持 __int128 类型，问题可以通过将 long long 替换为 __int128 来解决。假设我们将 dp 数组的类型改为 __int128。

修改后的代码：

cpp 复制

```cpp
#include <iostream>
#include <vector>

class Solution {
public:
    int change(int amount, std::vector<int>& coins) {
        int n = amount;
        std::vector<__int128> dp(n + 1, 0);  // 使用 __int128 处理大数
        dp[0] = 1;

        for (int coin : coins) {
            for (int j = coin; j <= n; j++) {
                dp[j] += dp[j - coin];
            }
        }

        return static_cast<int>(dp[n]);  // 返回结果时转换为 int
    }
};
```

示例：

假设 amount = 100000, coins = [1, 2, 5, 10, 20, 50, 100]，即使组合数超过了 long long 的范围，__int128 也能够处理超大数值。

- 通过使用 __int128 类型，可以处理比 long long 更大的数，防止溢出。
- 结果计算完毕后，将结果强制转换为 int 返回。

小结

- **问题场景**：当问题涉及非常大的数字时，int 和 long long 会遇到溢出问题。
- **解决方法**：使用更大的数据类型 __int128 或其他大数处理库。
- **举例**：零钱兑换问题，即使硬币面额多且目标金额大，通过动态规划和大数类型，可以安全计算组合数。

```cpp
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        int n = amount;
        vector<double> dp(n + 1, 0);  // 使用 double 处理极大数
        dp[0] = 1;

        for (int coin : coins) {
            for (int j = coin; j <= n; j++) {
                dp[j] += dp[j - coin];
            }
        }

        return static_cast<int>(dp[n]);
    }
};
```