二维费用背包： 474. 一和零

代码解析

1. dp 数组的定义

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

- 这里 dp[i][j][k] 表示使用前 i 个字符串，且最多使用 j 个 0 和 k 个 1 时，所能选出的最大字符串数量。
- len 是字符串数组的长度，m 是可用的 0 的数量上限，n 是可用的 1 的数量上限。

2. 外层循环

```cpp
for (int i = 1; i <= len; i++) {
    int a = 0, b = 0;
    for (auto s : strs[i - 1]) {
        if (s == '0') a++;
        else b++;
    }
    // 对每个字符串计算它包含的 '0' 和 '1' 的数量
}
```

- i 从 1 到 len，表示我们在逐一考虑第 i 个字符串 (strs[i-1])。
- a 表示当前字符串中 0 的数量，b 表示当前字符串中 1 的数量。这个通过遍历字符串 strs[i-1] 得到。

3. 计算最大值

```cpp
for (int j = 0; j <= m; j++) {
    for (int k = 0; k <= n; k++) {
        dp[i][j][k] = dp[i - 1][j][k];  // 不选第i个字符串
        if (j >= a && k >= b) {  // 如果能够选第i个字符串
            dp[i][j][k] = max(dp[i][j][k], dp[i - 1][j - a][k - b] + 1);  // 选第i个字符串，更新dp
        }
    }
}
```

- 我们对于每个 j (可用 0 的数量) 和每个 k (可用 1 的数量)，选择是否使用第 i 个字符串。
- dp[i][j][k] = dp[i - 1][j][k]：表示不选择第 i 个字符串时，最大可以选多少个字符串 (继承上一个状态)。
- if (j >= a && k >= b)：只有当当前的 0 和 1 的限制足够选用当前字符串时，才能考虑选择这个字符串。
- dp[i][j][k] = max(dp[i][j][k], dp[i - 1][j - a][k - b] + 1)：如果选择了第 i 个字符串，则更新 dp，从 dp[i-1][j-a][k-b] 加 1 得到新的状态。

4. 返回结果

```cpp
return dp[len][m][n];
```

- 返回 dp[len][m][n]，表示我们最多可以使用 m 个 0 和 n 个 1，能够挑选出的最多的字符串数量。

举例说明

假设 strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3。

- strs 中的每个字符串如下：
  - "10"：1 个 0，1 个 1。
  - "0001"：3 个 0，1 个 1。
  - "111001"：2 个 0，4 个 1。
  - "1"：0 个 0，1 个 1。
  - "0"：1 个 0，0 个 1。

我们希望找到能够选出的最大字符串数量，满足最多使用 5 个 0 和 3 个 1。

DP 状态转移

1. **初始化**：
   - dp 数组大小为 (len+1) x (m+1) x (n+1)，表示选取前 i 个字符串，最多使用 j 个 0 和 k 个 1 的情况下最多能选的字符串数量。
2. **第一步，处理第一个字符串 "10"(1 个 0，1 个 1)**：
   - 我们可以选择或不选择它：
     - 如果不选择它，dp[1][j][k] = dp[0][j][k]，即不做更新。
     - 如果选择它，dp[1][j][k] = dp[0][j-1][k-1] + 1，因为我们用了 1 个 0 和 1 个 1。

- 因此，dp[1][m][n] = 1，如果选择了它。

3. **第二步，处理第二个字符串 "0001"（3 个 0，1 个 1）**：
   - 类似地，如果不选择它，dp[2][j][k] = dp[1][j][k]。
   - 如果选择它，dp[2][j][k] = dp[1][j-3][k-1] + 1。
4. **依次处理剩下的字符串。**

经过一轮循环后，dp[len][m][n] 代表我们最多可以使用 m 个 0 和 n 个 1 时，可以挑选出的最大字符串数量。

时间复杂度

- 外层循环是 O(len)，其中 len 是字符串的数量。
- 内层两个循环分别是 O(m) 和 O(n)，分别遍历最多的 0 和 1 的数量。
- 因此，总的时间复杂度是 O(len * m * n)。

总结

这段代码使用了动态规划的思想，处理类似于背包问题的选择问题。通过构建 dp 数组，并逐步对每个字符串进行选择和更新，能够最终找到

```cpp
//优化前
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        int len = strs.size();
        vector<vector<vector<int>>> dp(len+1,vector<vector<int>>(m+1,vector<int>(n+1)));
        for(int i = 1;i <=len ;i++)
        {
            int a = 0,b = 0;
            for(auto s : strs[i-1])
            {
                if(s == '0') a++;
                else b++;
            }
            for(int j = 0;j <= m;j++)
            {
                for(int k = 0;k <= n;k++)
                {
                    dp[i][j][k] = dp[i-1][j][k];
                    if(j >= a && k>= b) dp[i][j][k] = max(dp[i][j][k],dp[i-1][j-a][k-b] +1);
                }
            }

        }
        return dp[len][m][n];
    }
};


//优化后
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        int len = strs.size();
        vector<vector<int>>dp(m+1,vector<int>(n+1));
        for(int i = 1;i <=len ;i++)
        {
            int a = 0,b = 0;
```

```
            for(auto s : strs[i-1])
            {
                if(s == '0') a++;
                else b++;
            }
            for(int j = m;j >= a;j--)
            {
                for(int k = n; k>= b;k--)
                {
                   dp[j][k] = max(dp[j][k],dp[j-a][k-b] +1);
                }
            }

        }
        return dp[m][n];
    }
};
```