动态规划

起源

动态规划的起源可以追溯到20世纪50年代。当时，美国的数学家理查德·贝尔曼（Richard Bellman）在研究最优控制问题时，提出了动态规划算法的基本思想。他在1953年发表了一篇名为《The Theory of Dynamic Programming》的论文，正式将动态规划算法提出并命名。当时，贝尔曼是在处理一类叫做“马尔可夫决策过程”的问题。这类问题涉及到多个阶段的决策，每个阶段的决策都会影响到下一个阶段的状态和决策，而最终的目标是寻找一种最优的决策方案，使得总体的收益或成本最小化。由于这类问题的状态空间和决策空间非常大，传统的枚举和递归算法很难处理。因此，贝尔曼提出了动态规划算法，通过将问题分解成多个子问题，并且利用子问题的最优解来推导出原问题的最优解，从而解决了这类问题的求解难题。

产生

动态规划是一种解决多阶段决策问题的最优化方法，其主要应用于以下两类问题：

1.最优子结构问题：最优子结构问题指的是一个问题的最优解包含其子问题的最优解，因此可以通过组合子问题的最优解来得到全局最优解。例如，最短路径问题、背包问题、最长公共子序列问题等都是最优子结构问题，可以使用动态规划来解决。

2.无后效性问题：无后效性问题指的是当前的状态只与之前的状态有关，与未来的状态无关。换句话说，当前状态的计算不受未来状态的影响。例如，编辑距离问题、斐波那契数列问题等都是无后效性问题，可以使用动态规划来解决。

动态规划主要应用于最优子结构问题和无后效性问题，可以通过分解问题成多个子问题，并且记录下每个子问题的最优解，最终通过组合这些子问题的最优解来得到全局最优解。

优点

动态规划是一种非常强大的算法思想，具有以下几个优点：

1.可以求解多阶段决策问题的最优解：动态规划算法可以用来解决多阶段决策问题的最优化问题。这类问题在实际应用中非常广泛，例如最短路径问题、背包问题、编辑距离问题、最长公共子序列问题等等。

2.可以避免重复计算：动态规划算法通过记录每个子问题的最优解，避免了重复计算，提高了算法的效率。这也是动态规划算法的一个重要优点。

3.可以分析问题的最优解结构：动态规划算法可以分析问题的最优解结构，将问题分解成多个子问题，并且记录下每个子问题的最优解，最终通过组合这些

子问题的最优解来得到全局最优解。这种分析方法可以用来解决其他类型的问题，不仅局限于动态规划算法。

研究动态规划算法可以帮助我们解决复杂问题的最优化问题，并且可以避免重复计算，提高算法效率。同时，动态规划算法也可以帮助我们分析问题的最优解结构，这种分析方法可以用来解决其他类型的问题。因此，动态规划算法是计算机科学和应用数学领域非常重要的一个研究方向。

使用场景

我们可以在以下情况使用动态规划：

1.重叠子问题：问题可以分解为较小的子问题，并且这些子问题的解决方案可以被多次重用。动态规划通过将子问题的解存储在表中，避免了多次计算相同子问题的时间开销。

2.最优子结构：问题的最优解包含其子问题的最优解。这意味着我们可以通过组合子问题的最优解来构建原始问题的最优解。

建模

动态规划是一种通过将原问题分解成多个子问题来解决复杂问题的方法。在数学建模中，动态规划通常用于解决优化问题，特别是涉及到多个决策的问题。动态规划可以用来解决一些著名的优化问题，如最长公共子序列问题、背包问题、最大子段和问题等。

动态规划的数学建模通常需要以下步骤：

1.定义问题：定义要解决的问题，包括输入和输出。例如，要解决的问题可能是如何找到最大的收益、最小的成本、最长的路径等。

2.确定状态：确定问题中的状态变量，并定义如何表示状态。状态变量是问题中发生变化的变量。例如，如果问题涉及一个背包，那么状态变量可能是背包中放入的物品数量和重量。

3.确定转移方程：定义问题中的状态之间的关系，并将它们表示为递归方程或迭代方程。转移方程描述了如何从一个状态转移到另一个状态，并使用递归或迭代方法计算问题的解决方案。例如，如果问题涉及最长公共子序列，那么转移方程可能是这样的：如果字符相同，则最长公共子序列的长度为前一个字符的最长公共子序列长度加 1，否则，最长公共子序列的长度为前一个字符的最长公共子序列长度和当前字符的最长公共子序列长度的最大值。

4. 确定边界条件：定义问题的初始状态和结束状态，并将它们表示为递归方程或迭代方程的特殊情况。边界条件通常是问题的简单情况，比如空背包或一个字符的最长公共子序列。

5.计算最优解：根据定义的递归或迭代方程，计算出问题的最优解。这可以通过动态规划的自底向上和自顶向下两种方法来实现。

数学思想

递归：动态规划通常使用递归的方法将原问题分解成多个子问题，然后通过求解子问题来求解原问题。

最优子结构：动态规划问题通常具有最优子结构性质，即全局最优解可以通过局部最优解的组合来得到。

重叠子问题：许多动态规划问题具有重叠子问题性质，即在递归过程中会重复求解相同的子问题。通过存储已经求解过的子问题的解，可以避免重复计算，提高计算效率。

数学归纳法：动态规划中的递推公式可以看作是数学归纳法的一种应用，即假设已经求出了前一步的解，然后通过递推公式求出当前步的解，从而得到问题的最优解。

线性规划：一些动态规划问题可以转化为线性规划问题，进而使用线性规划算法求解。

状态转移方程：动态规划中的核心就是状态转移方程，它描述了问题的最优解与其子问题的最优解之间的关系。通过构建状态转移方程，可以将问题的规模不断缩小，最终求得问题的最优解。

优化理论：动态规划常常用来解决优化问题，因此与优化理论相关的思想也常常被应用在动态规划中，比如最大化或最小化目标函数、限制条件的约束等等。

概率论和统计学：有些动态规划问题涉及到概率和随机变量的计算，比如马尔可夫决策过程和随机游走问题。在这些问题中，概率论和统计学的知识往往能够提供关键的帮助。

图论：一些动态规划问题可以看作是图论问题的变种，比如最长路问题、最短路问题、最大流问题等等。因此，图论中的知识也可以被应用在动态规划中，比如动态规划中的状态可以看作是图中的节点，状态转移方程可以看作是图中的边。

计算机科学：动态规划是计算机科学中的一个重要分支，因此计算机科学中的知识对于动态规划的应用非常重要。比如，算法设计、数据结构、计算复杂度等等，都与动态规划密切相关。

马尔可夫决策过程

马尔可夫决策过程（Markov Decision Process，MDP）是一个用于描述序列决策问题的数学模型，是强化学习中常用的一种方法。MDP 具有以下四个主要组成部分：

1.状态（States）：一个状态集合，用于表示决策者所处的环境。

2. 动作（Actions）：一个动作集合，表示决策者可以采取的行动。

3.转移概率（Transition Probabilities）：描述在给定状态和动作下，决策者达到下一个状态的概率。通常表示为 P(s'|s, a)，其中 s 是当前状态，a 是采取的动作，s'是下一个状态。

4.奖励函数（Reward Function）： 表示决策者在采取某个动作并从一个状态转移到另一个状态时获得的奖励。通常表示为 R(s, a, s')。

MDP 的基本思想是将决策问题看作是一系列状态和动作的序列，每个状态都

有一定的概率转移到其他状态，并且对于每个状态和动作组合，都有一个对应的奖励值。强化学习就是在 MDP 中寻找最优的策略，使得长期奖励最大化。

在 MDP 中，动态规划是求解最优策略的一种重要方法，其基本思想是通过递推的方式求解当前状态的最优策略，从而得到整个决策序列的最优策略。通常使用的动态规划算法有价值迭代（通过迭代执行策略评估和策略改进来找到最优策略）和策略迭代（通过逐步更新状态值函数，最终找到最优策略）。

强化学习也是通过学习当前状态和奖励值之间的关系，来寻找最优的决策策略。但是，强化学习的一个重要特点是它不需要事先知道环境模型，即不需要事先知道状态转移概率和奖励函数。相反，它通过试错学习的方式，不断更新决策策略，直到找到最优策略。

因此，可以说动态规划是强化学习的一种基础方法，而强化学习则是一种更加通用、更加灵活的决策方法，它可以应用于更广泛的决策问题中，比如无模型强化学习（Model-free Reinforcement Learning）等。

马尔可夫决策过程提供了一种描述强化学习问题的框架，动态规划是一种求解 MDP 的方法，而强化学习算法则是在不完全了解环境模型的情况下学习最优策略的方法。

编程实例

Max Sum Plus Plus

Time Limit: 2000/1000 MS (Java/Others) Memory Limit: 65536/32768 K (Java/Others)
Total Submission(s): 58441 Accepted Submission(s): 21433

Problem Description

Now I think you have got an AC in Ignatius.L's "Max Sum" problem. To be a brave ACMer, we always challenge ourselves to more difficult problems. Now you are faced with a more difficult problem.

Given a consecutive number sequence $S_1, S_2, S_3, S_4 \dots S_x, \dots S_n$ ($1 \le x \le n \le 1{,}000{,}000$, $-32768 \le S_x \le 32767$). We define a function sum(i, j) = $S_i + \dots + S_j$ ($1 \le i \le j \le n$).

Now given an integer m ($m > 0$), your task is to find m pairs of i and j which make sum(i_1, j_1) + sum(i_2, j_2) + sum(i_3, j_3) + ... + sum(i_m, j_m) maximal ($i_x \le i_y \le j_x$ or $i_x \le j_y \le j_x$ is not allowed).

But I'm lazy, I don't want to write a special-judge module, so you don't have to output m pairs of i and j, just output the maximal summation of sum(i_x, j_x)($1 \le x \le m$) instead. ^_^

Input

Each test case will begin with two integers m and n, followed by n integers $S_1, S_2, S_3 \dots S_n$.
Process to the end of file.

Output

Output the maximal summation described above in one line.

Sample Input

```
1 3 1 2 3
2 6 -1 4 -2 3 -2 3
```

Sample Output

```
6
8
```

题目连接：https://acm.hdu.edu.cn/showproblem.php?pid=1024

分析：

a) 确定决策：给一个长为 n 的整数序列，从中取出 m 段使得这 m 段的和最大。

b) 确定状态：dp[i][j]是以第 j 个数结尾取 i 段数得到的最大和。

c) 状态转移方程：对于 dp[i][j]，有两种情况，情况 1：第 j'个数单独为第 i 段即 dp[i][j] = max(dp[li- 1][k],k<j) + a[j]；情况 2：第 j 个数融入最后一段，即 dp[i][j] = dp[i][j – 1] + a[j];综上：dp[i][j] = max(dp[i][j - 1],dp[i - 1][k]) + a[j]; k <= j- 1 。

代码实现：

```
#include <iostream>
#include <cmath>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <string>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <iomanip>
#include <functional>
#define quick ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
using namespace std;
typedef pair<int, int> PII;
typedef long long ll;
#define INF 0x3f3f3f3f
const int N = 1e6 + 10;

long long dp[N], pr[N], a[N];

int main()
{
    int n, m;
    while (scanf("%d%d", &m, &n) != EOF)
    {
        for (int i = 1; i <= n; i++)
        {
            scanf("%lld", &a[i]);
            dp[i] = pr[i] = 0;
        }
        ll ans;
        for (int i = 1; i <= m; i++)
        {
            ans = -INF;
            for (int j = i; j <= n; j++)
            {
                // dp[i][j] = max(dp[i][j - 1],dp[i - 1][k]) + a[j]; k <=
j- 1
```

```
                // 用 pre[j - 1]记录上一次的值则 dp[i][j] = max(dp[i][j - 1],pre[j]) + a[j];
                dp[j] = max(dp[j - 1], pr[j - 1]) + a[j];
                pr[j - 1] = ans;
                ans = max(ans, dp[j]);
            }
        }
        printf("%lld\n", ans);
    }
    return 0;
}
```

Bone Collector

Time Limit: 2000/1000 MS (Java/Others) Memory Limit: 32768/32768 K (Java/Others)
Total Submission(s): 140000 Accepted Submission(s): 55322

Problem Description

Many years ago , in Teddy's hometown there was a man who was called "Bone Collector". This man like to collect varies of bones , such as dog's , cow's , also he went to the grave ...
The bone collector had a big bag with a volume of V ,and along his trip of collecting there are a lot of bones , obviously , different bone has different value and different volume, now given the each bone's value along his trip , can you calculate out the maximum of the total value the bone collector can get ?

Input

The first line contain a integer T , the number of cases.
Followed by T cases , each case three lines , the first line contain two integer N , V, (N <= 1000 , V <= 1000)representing the number of bones and the volume of his bag. And the second line contain N integers representing the value of each bone. The third line contain N integers representing the volume of each bone.

Output

One integer per line representing the maximum of the total value (this number will be less than 2^{31}).

题目链接：https://acm.hdu.edu.cn/showproblem.php?pid=2602
分析：
a）确定决策：给定一个背包，它能容纳一定体积的骨头，现在有一些骨头和它们的体积和价值，我们需要选择其中一些骨头放入袋子中，使得袋子中骨头的总价值最大。
b）确定状态：dp[i][j]表示在前 i 个骨头中选择体积不超过 j 的骨头能得到的最大价值。其中 i 表示前 i 个骨头，j 表示袋子的容量。
c）状态转移方程：对于每个骨头，有两种情况，放入和不放入。如果我们选择放入第 i 个骨头，那么它的价值将会被累加到当前的总价值中，同时袋子的容量将会减少 v[i]。因此，如果我们选择放入第 i 个骨头，那么当前的最大价值为 dp[i-1][j-v[i]]+w[i]，其中 dp[i-1][j-v[i]]表示在前 i-1 个骨头中选择体积不超过 j-v[i]的骨头能得到的最大价值，w[i]表示选择第 i 个骨头的价值。如果我们不选择放入第 i 个骨头，那么当前的最大价值为 dp[i-1][j]，即在前 i-1 个骨头中选择体积不超过 j 的骨头能得到的最大价值。综上，得

到状态转移方程：dp[i][j] = max(dp[i - 1][j], (dp[i - 1][j - v[i]] + w[i]));

代码实现 1：

```
#include <iostream>
#include <cmath>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <string>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <iomanip>
#include <functional>
#define quick ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
using namespace std;
typedef pair<int, int> PII;
typedef long long ll;
#define INF 0x3f3f3f3f
const int MAX = 1e3 + 10;

int T, N, V;
ll dp[MAX][MAX], v[MAX], w[MAX];

int main()
{
    quick;
    cin >> T;
    while (T--)
    {
        cin >> N >> V;
        for (int i = 1; i <= N; i++)
        {
            cin >> w[i];
        }
        for (int i = 1; i <= N; i++)
        {
            cin >> v[i];
        }
        memset(dp, 0, sizeof(dp));
        for (int i = 1; i <= N; i++)
        {
            for (int j = 0; j <= V; j++)
            {
```

```
                if (j >= v[i])
                    dp[i][j] = max(dp[i - 1][j], (dp[i - 1][j - v[i]] +
w[i]));
                else
                    dp[i][j] = dp[i - 1][j];
            }
        }
        cout << dp[N][V] << endl;
    }
    return 0;
}
```

代码实现 2:

```
#include <iostream>
#include <cmath>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <string>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <iomanip>
#include <functional>
#define quick ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
using namespace std;
typedef pair<int, int> PII;
typedef long long ll;
#define INF 0x3f3f3f3f
const int MAX = 1e3 + 5;

int T, N, V;
ll dp[MAX], v[MAX], w[MAX];

int main()
{
    quick;
    cin >> T;
    while (T--)
    {
        cin >> N >> V;
        for (int i = 1; i <= N; i++)
        {
```

```
            cin >> w[i];
        }
        for (int i = 1; i <= N; i++)
        {
            cin >> v[i];
        }
        memset(dp, 0, sizeof(dp));
        for (int i = 1; i <= N; i++)
        {
            for (int j = V; j >= v[i]; j--)
            {
                dp[j] = max(dp[j], (dp[j - v[i]] + w[i]));
            }
        }
        cout << dp[V] << endl;
    }
    return 0;
}
```