动态规划 $(Dynamic\ Programming)$

20数媒技欧芃芃

内容

（一）什么是动态规划(dp)

（二）线性dp

（三）背包dp

（四）总结

（五）题目链接

（一）什么是dp

> 将一个庞大的问题分解成小问题
> 解决每一个小问题
> 通过小问题之间的关联，逐步解决大问题

来看例题1：

> 有两种木块，一种长度为 1，一种长度为 2。
> 求组成一个长度为 n 的长条木块，有多少种组合方案。

初步思路

- 设 $f(n)$ 表示长度为 n 时的组合方案数
  易知 $f(1) = 1, f(2) = 2$
- 发现每个状态之间是有**关联**的
  长度为 n 的组合方案数，是长度为 $n-1$ 和 $n-2$ 的组合方案数之和。
- 即 $f(n) = f(n-1) + f(n-2)$
- ~~众所周知，在数据范围较大的情况下，朴素的暴力递归斐波那契数列容易翻车~~

最终思路

- 用 $dp[i]$ 来表示长度为 i 的组合方案数目。
  那么有 $dp[i] = dp[i-1] + dp[i-2]$，其中 $i > 2$
- **初始化**：$dp[1] = 1,\ dp[2] = 2$
- **目标值**：$dp[n]$
- 可以用**记忆化搜索**来优化

来看例题2：数字三角形

> 图给出了一个 n 层的数字三角形。
> 从顶部出发，每次只能向左下或右下走。
> 由此我们可以得到许多路径，求出路径上数字之和的最大值。

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

初步思路

- 我们从上往下尝试递推解决这个问题
  设 $f(i,j)$ 表示以坐标为 (i,j) 的点为结尾的最大路径和
- $f(i,j)$ **的更新，依赖于** $f(i-1,\ j-1)$ **和** $f(i-1,\ j)$ **这两个数据。**
- 即 $f(i,j) = max\{f(i-1,\ j-1),\ f(i-1,\ j)\} + num[i][j]$
- 处理完整个三角形后，取最底层的最大数值即可。
- ~~但是我很懒不想遍历最后一层取最大值~~，
  注意到路径**从上至下与从下至上处理是等价**的。方便起见，我们选择从下至上处理，这样答案直接被存储在顶端位置。

最终思路

- 用 $dp[i][j]$ 表示以点 (i,j) 为结尾的路径和
  那么有 $dp[i][j] = max(dp[i+1][j],\ dp[i+1][j+1]) + num[i][j]$
- **初始化**：$dp[n][i] = num[n][i]$，其中 $1 \le i \le n$
- **目标值**：$dp[1][1]$

代码时间：

```
int num[maxn][maxn];//存储数塔的原始数据
int dp[maxn][maxn];//存储最大路径和

void init() {//初始化函数
    for(int i = 1; i <= n; i++) {
        dp[n][i] = num[n][i];
    }
}

for(int i = n - 1; i > 0; i--) {//从下往上推
    for(int j = 1; j <= i; j++) {
        dp[i][j] = max(dp[i + 1][j], dp[i + 1][j + 1]) + num[i][j];
    }
}

printf("%d", dp[1][1]);//答案输出
```

不一定正确的总结：

- 动态规划应用**分治的思想**划分大问题，并找到每个子问题间的联系。
- 通过维护每个**必要的**子问题的最优解，**递推**得到大问题的最优解

（二）线性dp

> 线性dp是动态规划中的一类，指**状态之间有线性关系**的问题。

(1) 最长公共子序列 $(Longest\ Common\ Subsequence)$

> 一个序列 S
> 如果分别是**两个或多个**已知序列的子序列，且是所有符合此条件序列中**最长**的
> 则称 S 为已知序列的最长公共子序列 (LCS)

子串与**子序列**的区别：

- 子串要求在原字符串中是**连续**的
- 子序列只需要保持**相对顺序一致，并不要求连续。**

模板题面：

> 有长为 n 和 m 的两个字符串 S_1 和 S_2 ，求 S_1 和 S_2 的 LCS 长度。

尝试dp的思路：

- 对于每个字符串，我们**按位来看。**
- 用 $dp[i][j]$ 表示**遍历到 S_1 的第 i 位和 S_2 的第 j 位时 LCS 的长度。**
- 若 $S_1[i] = S_2[j]$，那么有 $dp[i][j] = dp[i-1][j-1]+1$
  否则有 $dp[i][j] = max(dp[i-1][j], dp[i][j-1])$
- **初始化**：$dp[i][0] = 0, \ dp[0][j] = 0$，其中 $0 \le i \le n, \ 0 \le j \le m$
- **目标值**：$dp[n][m]$
- 时间复杂度和空间复杂度均为 $O(n^2)$

先整个模板康康:

```
int dp[maxn][maxn];

void init(int n, int m) {//初始化
    for(int i = 0; i < n; i++) {
        dp[i][0] = 0;
    }
    for(int i = 0; i < m; i++) {
        dp[0][i] = 0;
    }
}
void LCS(string s1, string s2) {
    int n = s1.length(), m = s2.length();//两个字符串从下标为1开始存

    init(n, m);

    for(int i = 1; i < n; i++) {//每一位遍历
        for(int j = 1; j < m; j++) {
            if(s1[i] == s2[j]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            }
            else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
}
```

一些个空间上的优化：滚动数组

观察代码发现：

- dp数组是按行进行更新的。
- 更新第i行时，无须用到 $i-2$ 行及先前行的数据。
- **故 $i-2$ 行及之前的数据可以清除**

优化思路：

- 考虑用新的 $dp[i]$ 行的数据去**覆盖** $dp[i-2]$ 行的数据。
- 即可以用一个大小为 $2 \times m$ 的**滚动数组**去代替原先大小为 $n \times m$ 的dp数组。
- 将新数组的两层分别命名为cur和pre
  通过cur**层和**pre**层的不断交换**实现原先dp数组的功能。
- 此时空间复杂度优化为 $O(n)$

优化后代码：

```
int dp[2][maxn];

void init(int n) {//初始化
    for(int i = 0; i < n; i++) {
        dp[0][i] = 0;
    }
    dp[1][0] = 0;//cur层只需要初始化一个数据即可
}
void LCS_better(string s1, string s2) {
    int n = s1.length(), m = s2.length();//两个字符串从下标为1开始存

    init(m);

    int pre = 0, cur = 1;
    for(int i = 1; i < n; i++) {
        for(int j = 1; j < m; j++) {
            if(s1[i] == s2[j]) {
                dp[cur][j] = dp[pre][j - 1] + 1;
            }
            else {
                dp[cur][j] = max(dp[pre][j], dp[cur][j - 1]);
            }
        }
        //实现滚动
        //也可以直接调用swap()函数，但是要注意最后数据到底存在pre层还是cur层
        pre = (pre + 1) % 2;
        cur = (cur + 1) % 2;
    }
}
```

(2) 最长上升子序列 $(Longest\ Increasing\ Subsequence)$

> 在一个给定的数值序列中，找到一个子序列
> 使得这个子序列元素的**数值严格递增**
> 其中**最长**的序列称为最长上升子序列 **(LIS)**

值得注意的是，LIS 不一定唯一。

模板题面：

> 有长度为 n 的序列 S，求其 LIS 长度。

先来朴素的dp思路：

- 依然选择按位来看。
- 要求 LIS，我们需要知道当前阶段**最后一个元素值**，同时为了求出答案，我们需要**维护当前 LIS 的长度**。
- 用 $dp[i]$ 来表示以 $s[i]$ 为结尾的 LIS 长度。
  那么有 $dp[i] = max(dp[j]) + 1$，其中 $0 < j < i$，且 $s[j] < s[i]$
- **初始化**：$dp[1] = 1$
- **目标值**：$max(dp[i])$，其中 $1 \leq i \leq n$
- 时间复杂度：$O(n^2)$，空间复杂度：$O(n)$

转化为代码：

```cpp
int dp[maxn];

void LIS(int s[]) {//从下标为1开始存储数据
    dp[1] = 1;//初始化

    for(int i = 2; i <= n; i++) {//n为序列长度，可以从第二位开始处理
        for(int j = 1; j < i; j++) {
            if(s[i] > s[j]) {//更新条件
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }
}

//获取答案
int ans = 0;
for(int i = 1; i <= n; i++) {
    ans = max(ans, dp[i]);
}
```

惯例优化：贪心 + 二分

思路：

1. 显然对于一个 LIS，其**结尾元素越小**，越有可能在后面接上其他元素，就**越有可能变得越长。**
2. 那么我们可以采用**维护长度为 i 的上升子序列的结尾元素**这一**贪心**策略。
3. 假设我们目前已经得到的 LIS 长度为 len。那么对于往后遍历的每个 $s[i]$：
    - 若 $s[i] > dp[len]$，那么 $s[i]$ 可以直接接在当前 LIS 后面，并且 len++。
    - 否则，我们就对应更新 dp 数组，在 dp 数组中寻找第一个不小于 $s[i]$ 的数据 $dp[j]$，将其更新为 $s[i]$。
4. 那么如何**快速**在具有**单调性**的序列中**查找**这个位置？
   ——当然是**二分**啦。

总结一下：

- **状态**：$dp[i]$ 表示长度为 i 的上升子序列的最小结尾元素
- **转移方程与策略**：
  当 $s[i] > dp[len]$ 时，$dp[{+}{+}len] = s[i]$
  否则，$dp[j] = s[i]$，其中 $dp[j]$ 为数组中第一个不小于 $s[i]$ 的数
- **初始化**：$len = 1,\ dp[1] = s[1]$
- **目标值**：len，即数组长度
- 时间复杂度：$O(nlogn)$，空间复杂度：$O(n)$

惯例上代码：

```
int len;//LIS的长度
int s[maxn];//存储原始序列
int dp[maxn];

void init() {//初始化
    //序列第一位直接放进去就行
    len = 1;
    dp[len] = s[1];
}
void LIS() {//从下标为1开始存储数据
    init();

    for(int i = 2; i <= n; i++) {//n为所给序列长度
        if(s[i] > dp[len]) {//如果比当前LIS最后一位要大就直接接上去
            dp[++len] = s[i];
        }
        else {//否则就修改数据
            int pos = lower_bound(dp + 1, dp + len + 1, s[i]) - dp;
            dp[pos] = s[i];
        }
    }
}
```

一些叨叨逼逼：

- 关于 LCS 和 LIS 的优化方法还有更优的，但是对于代码的理解也比较头疼orz，学有余力的同学可以自行了解。
- 给出的代码**着重于求出长度**，至于 LCS 和 LIS 序列是什么还需要**另外维护**。放在今天的题里给大家自行思考。
- 虽然 LCS 和 LIS 看起来像是两个模型，但其实是可以**相互转换**的。同样放在题里给大家思考。

（三）**背包**dp

> 如何选择**最合适的物品**放置于**给定容量背包**中，使其**价值最大化。**

(1) 部分背包

题胚

> 给定一个最大负重为 m 的背包和 n 种物品。第 i 种物品重量为 w_i，价值为 v_i。
> **物品可按单位价值任意分割**
> 求背包中能放入物品的最大总价值。

基本思路：

- 贪心即可。

(2) 0/1背包

题胚

> 给定一个最大负重为 m 的背包和 n 种物品。第 i 种物品重量为 w_i，价值为 v_i。
> **每种物品仅有一件**
> 求背包中能放入物品的最大总价值。

Q: **可以用贪心解决** 0/1 **背包问题吗?**

A: **问就是不行。**

为什么不能用贪心去解决?

- 由于物品是不可分割的，所以**不能保证背包的容量被完美填充。**
- 那么去贪单个物品的单位最大价值是没有意义.
- 因为**剩余的背包容量会破坏单位背包容量的最大价值。**

例：背包容量 $m = 10$

编号	v	w	v/w
1	9	7	1.28
2	5	5	1.00
3	5	4	1.25

显然，若是用贪心去写，仅会选择1号物品。但正确的策略应该是放入2号和3号物品。

放入1号物品后，背包单位价值为0.9。而放入2号和3号物品后，背包单位价值为1.0

该种贪心策略仅能保证**所占容量取得单位最大价值**，但**剩余的空间有可能会破坏这个单位最大值。**

那就整个 dp 的思路：

- 我们遍历每一种物品，每种物品都有**取**或**不取**两种选择。
- 用 $dp[i][j]$ 表示**遍历到第 i 件物品，且背包容量为 j 时的最大总价值。**
- 如果**不取**第 i 件物品，那么有 $dp[i][j] \ = \ dp[i-1][j]$
  如果**取**第 i 件物品，那么有 $dp[i][j] \ = \ dp[i-1][j-w_i]+v_i$
- **初始化**：$dp[0][i]=0$，其中 $0 \le i \le m$
- **目标值**：$dp[n][m]$

喜闻乐见上代码：

```
int dp[maxn][maxm];

void init() {//初始化
  for(int i = 0; i <= m; i++) {
    dp[0][i] = 0;
  }
}

for(int i = 1; i <= n; i++) {//开始遍历每种物品
    for(int j = m; j >= 0; j--) {//更新每一种剩余承重的情况
        //如果放不下了肯定不能硬塞
        if(j < w[i]) {
            dp[i][j] = dp[i - 1][j];
        }
        //如果能放的下
        else {
            //就看看要不要放入这件物品
            dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]);
        }
    }
}

printf("%d", dp[n][m]);//输出答案
```

好活重整：空间压缩

再次观察代码发现：

1. dp 数组也是按行进行更新的。且更新第i行时，无须用到 $i-2$ 行及先前行的数据。**故 $i-2$ 行及之前的数据可以清除**
2. 更新第 i 行时，$i-1$ 行的数据每个只会**用到一次**
3. 容量为 j 的数据更新仅依赖于**它本身和 $j-w[i]$ 的数据**
   即**下标大的数据更新依赖于下标小的数据。**

优化后代码：

```
int dp[maxn];

memset(dp, 0, sizeof(dp));//初始化

for(int i = 1; i <= n; i++) {//遍历每一种物品
    for(int j = m; j >= w[i]; j--) {//注意j一定要由大到小更新，且下界为w[i]
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}

printf("%d", dp[m]);//输出答案
```

Q：j **可以由小到大更新吗？**

A：**问就是不行。**

让我们假设：

- 背包容量 $m = 10$

编号	v	w
1	20	5

j 由大到小更新：

dp_j	0	1	2	3	4	5	6	7	8	9	10
(初始化)	0	0	0	0	0	0	0	0	0	0	0
i = 1	0	0	0	0	0	20	20	20	20	20	20

答案为 $dp[10] = 20$，正确。

j **由小到大更新：**

dp_j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
(初始化)	0	0	0	0	0	0	0	0	0	0	0
i = 1	0	0	0	0	0	20	20	20	20	20	40

答案 $dp[10] = 40$，**拿了两件物品1**，但每种物品**只有一件，与题意不符**

- 对于0/1背包模型，j 由小到大循环，则数据由小到大更新。
  但**下标大的数据更新依赖于下标小的数据，而下标小的数据有可能在本轮次已被更新，所以会出错。**
- 数据多次更新，转化为现实模型即**同种物品数量无限，可以任取。**

(3) 完全背包

题胚

> 给定一个最大负重为 m 的背包和 n 种物品。第 i 种物品重量为 w_i，价格为 v_i。
> **每种物品的数量无限**
> 求背包中能放入物品最大的价格之和。

美美改代码

```
int dp[maxn];

memset(dp, 0, sizeof(dp));//初始化

for(int i = 1; i <= n; i++) {//遍历每一种物品
    for(int j = w[i]; j <= m; j++) {//直接改为j由小到大更新即可
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}

printf("%d", dp[m]);//输出答案
```

(4) **其他**

- 多重背包：各物品**个数有限**且不一定相同
- 混合背包：各物品**或个数唯一**，**或个数无限**，**或个数有限**
- ……

(四) 总结

- dp 思路
- 无后效性原则
- 初始化问题

(1) dp 思路

- **状态**：明确数组下标和数组维护的数据含义
- **转移**：写出转移方程
- **决策**：用合理的顺序去更新每个状态
- **边界**：初始化问题

(2) **无后效性原则**

> 现在与未来无关
> 未来做的决定不会影响现在做的决定

不符合这个条件的话就**不能**用动态规划了。
可以考虑修改状态变量，或者换个决策，~~或者这题压根就不是dp~~

(3) **背包问题的初始化**

- 不同的问法需要不同的初始化：

    i. **恰好装满**背包时的最优解：$dp[0] = 0$，$dp[i] = -\infty$，其中 $0 < i \leq n$

    ii. 不要求装满时的最优解：$dp[i] = 0$，其中 $0 \leq i \leq n$

- 初始化 $dp[i]$ 的过程，可以理解为**更新在没有任何物品时，背包承重为 i 的答案。**

    i. 没有任何物品，可以理解为一个**重量和价值均为**0的特殊物品

    ii. 如果要求背包**恰好装满**
    那么承重为0的背包，可理解成**被该特殊物品恰好装满**
    其它容量的背包无解，初始化为$-\infty$。

    iii. 如果背包**不要求装满**
    那么任何容量的背包都有一个**合法解：装入该特殊物品**
    这个特殊物品的价值为0，所以全初始化为0。

（五）题目链接

- 不完全模板的LIS模板题
- 0/1背包模板题
- 完全背包模板题

~~另附一些我觉得挺有意思的:~~

- LCS与LIS转换
- Dilworth定理
- "逆推"
- 路径输出
- 等差数列

参考课件来自

- 17 计算机科学与技术 黄彪
- 18 软件工程 肖子萌

非常感谢师哥们！！！