大黄菌的个人博客

天下武功，无勤不破

算法一篇通——动态规划

2017-12-18 | 算法笔记 | 阅读次数:

上周四去参加了第一次校招实习考试。虽然原本只是和同学凑个热闹，但结果还是着实有点打击。算法题一点思路都没有，前端知识也基本忘干净了。回到学校反思了一下，感觉从成都回来后学习状态确实很差。

另外，感觉自己把自己就预先划分到保研的身份去了。之前一直认为机器学习和前端不可兼得，自己划死了高度。实际上，很多业界人士在前端、机器学习以及其他方向都有很高水平。前端是我的兴趣，机器学习是之后可能深入研究的领域，但是如果抱着只能二选一的心态去学习，只能说明自己不够勤奋。之后的学习目标首先是为读研打好稳固基石，然后也要涉及多方面的知识。

回到正题。这次实习考试的第一题在当时没有思路，出来后同学讨论说要用到动态规划思想。之前有听过几次这个词，但是没有去了解，恰逢这个机会（以及为之后的美赛做准备），查阅了很多资料。在此总结一下我对动态规划的了解，以及用几个例子来说明，希望能尽可能地把动态规划给弄通。

概念理解

动态规划（Dynamic programming）是一种在数学、计算机科学和经济学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法，常用于求解**最优化问题（Optimization problem）**。动态规划常常适用于有**重叠子问题**和**最优子结构性质**的问题，动态规划方法所耗时间往往远少于朴素解法。

当我们遇到一个大问题时，总是习惯把问题的规模变小，这样便于分析讨论。这个规模变小后的问题和原来的问题是同质的，除了规模变小，其它的都是一样的，本质上它还是同一个问题，我们就称其为原问题的**子问题**。

动态规划的核心是**状态**和**状态转移方程**：

- **状态**：描述该问题的子问题的解，即根据子问题来定义状态。
- **状态转移方程**：状态和状态之间的关系式。大部分情况下，某个状态只与它前面出现的状态有关，而独立于后面的状态（**无后效性**）。

能使用动态规划思想解决的问题都有最优子结构性质和重叠子问题：

- **最优子结构（Optimal substructure）**性质：如果问题的最优解所包含的子问题的解也是最优的，而这些子问题可以独立求解，我们就称该问题具有最优子结构性质。其包含"**全局最优解包含局部最优解**"的思想。
- 重叠子问题：指在用**递归**算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题，有些子问题会被重复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次，然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果，从而获得较高的效率。

解题思路

动态规划的解题思路如下：

1. 将原问题分解为子问题；
2. 确定状态：状态不是随便定义的，一般定义完就要找到状态转移方程；
3. 确定一些初始状态（边界状态）的值；
4. 确定状态转移方程。

如果问题看起来是个动态规划问题，但是无法定义出状态，那么试着将问题规约到一个已知的 DP 问题。

例题

讲了这么多，让我们一起做几道题目来练练手！

菜鸟级

这里我选了 LeetCode 的第 70 题 Climbing Stairs。说来惭愧，我第一次做这个题的时候半天没做出来，还先跳过去了。

题目是这样的：假设你在爬梯子，需要 n 步爬到顶。每一次只能爬 1 或 2 格，爬到顶一共有多少种不同的方法?

一步步沿着前面提到的解题思路来解题：

1. 将原问题分解为子问题：这里的子问题即“爬到 i 格共有多少种不同的方法（i < n）”；
2. 确定状态：我们通常用一个函数表达式来表示状态。这里我们可以用 `d(i)` 来表示“爬到 i 格共有的不同的方法数”；
3. 确定一些初始状态（边界状态）的值： `d(1) = 1` 、 `d(2) = 2` ；
4. 确定状态转移方程：当我们得知 `d(i-2)` 时，再往上爬一次 2 格即可到达 i 格；当我们得知 `d(i-1)` 时，再往上爬一次 1 格即可到达 i 格。因此有 `d(i) = d(i - 1) + d(i - 2)` 。没有重叠的解法，因为最后一步要么爬 1 格，要么爬 2 格，我们将这两种自然分开了。

使用递归，我们写出如下代码：

```
public int climbStairs(int n) {
    if(n == 1)
        return 1;
    if(n == 2)
        return 2;
    return climbStairs(n-1) + climbStairs(n-2);
}
```

Submit 就可以看到红红的 Time Limit Exceeded。计算时间超了，这是因为对于每一个 i，由于递归调用，我们反复求解相同的子问题，使得所作的工作量爆炸性增长。想要节省这些计算，我们可以采用拿空间换时间的方法，用一个大小为 n 的数组来记录子问题的计算结果：

```
public int climbStairs(int n) {
    int[] arr = new int[n+1];
    for(int i = 1; i < n+1; i++)
        fill(i, arr);
    return arr[n];
}

private void fill(int n, int[] arr) {
    if(n == 1)
```

```
            arr[n] = 1;
        else if(n == 2)
            arr[n] = 2;
        else
            arr[n] = arr[n-1] + arr[n-2];
    }
```

AC，Run Time 5ms。

也可以用递推法来解决，本质是一个 fibonacci。这里借 Discuss 里的解法一用：

```
public int climbStairs(int n) {
    // base cases
    if(n <= 0) return 0;
    if(n == 1) return 1;
    if(n == 2) return 2;

    int one_step_before = 2;
    int two_steps_before = 1;
    int all_ways = 0;

    for(int i=2; i<n; i++){
        all_ways = one_step_before + two_steps_before;
        two_steps_before = one_step_before;
        one_step_before = all_ways;
    }
    return all_ways;
}
```

普通级

稍微加点难度，来试一下 LeetCode 的第 198 题 House Robber。

题目：你是一个超高校级的小偷，唯一能阻止你的是你同一夜不能偷相邻的两家，否则警报装置会响。给一个全是非负整数的数组来代表每家有的钱，求你在不惊动警报的基础上今晚的最大收获。

还是一步步沿着动态规划的解题思路来解题：

1. 将原问题分解为子问题：假设数组长度为 n，则这里的子问题即“有 i 家可偷时最大收获（i < n）”；
2. 确定状态：用 `g(i)` 代表第 i 家有的钱，用 `d(i)` 来表示“有 i 家可偷时最大收获”；
3. 确定一些初始状态（边界状态）的值： `d(1) = g(1)` 、 `d(2) = max(g(1), g(2))` ；

4. 确定状态转移方程：可以想到，到第 i（i > 2）间屋子时，可能之前偷了第 i–1 间，那这间就不能偷；如果没偷第 i–1 间，那这间可以偷。于是有 `d(i) = max(d(i-2)+g(i), d(i-1))` 。

用数组来记录偷过第 i 间屋子时最大收获，代码如下：

```java
public int rob(int[] nums) {
    int n = nums.length;
    int[] d = new int[n];
    if(n == 0)
        return 0;
    for(int i = 0; i < n; i++)
        fill(d, nums, i);
    return d[n-1];
}

private void fill(int[] d, int[] nums, int i) {
    if(i == 0)
        d[i] = nums[0];
    else if(i == 1)
        d[i] = Math.max(nums[0], nums[1]);
    else
        d[i] = Math.max(d[i-2]+nums[i], d[i-1]);
}
```

Run Time 接近 0，非常理想。

也有非常巧妙的递推方案，空间复杂度 O(1)：

```java
public int rob(int[] num) {
    int prevNo = 0;
    int prevYes = 0;
    for (int n : num) {
        int temp = prevNo;
        prevNo = Math.max(prevNo, prevYes);
        prevYes = n + temp;
    }
    return Math.max(prevNo, prevYes);
}
```

挑战者级

来试试 LeetCode 的第 646 题 Maximum Length of Pair Chain。这是一道难度为 medium 的题，建议先跳过这一小节，看完“扩展”中的“最长非降子序列（LIS）”再回来，有助于解决和理解这道

题。

题目：给定 n 对数，每一对中前一个数总小于后一个数。现在，我们定义当且仅当 `b < c` 时 `(c, d)` 可以跟在 `(a, b)` 后，来形成一条链。给定一系列对，求出*用上述方法形成的链条的最大长度*。不需要用完所有的给定对，而且可以以任意顺序选取。

可以看到，这题和 LIS 问题比较相似，但又有一些不同。由于可以以任意顺序选取，而非 LIS 问题中选取最长非降子序列的顺序固定，因此需要对一系列数对进行一个从小到大的排序。

状态转移方程为：

```
d(i) = max{1, d(j)+1}，其中 j < i，pairs[j][1] < pairs[i][0]
```

想要求 `d(i)` ，就把 i 前面的各个链中，最后一个数对的最大值不大于 `pairs[i][0]` 的序列长度加 1，然后取出最大的长度即为 `d(i)` 。当然，有可能 i 前面的各个链中最后一个数对的最大值都大于 `pairs[i][0]` ，那么 `d(i)=1` ，即它自身成为一个长度为 1 的子序列。

代码如下：

```
public int findLongestChain(int[][] pairs) {
    Arrays.sort(pairs, (a, b) -> (a[1] - b[1]));
    int len = pairs.length;
    int[] arr = new int[len];
    int l = 1;
    for(int i = 0; i < len; i++) {
        arr[i] = 1;
        for(int j = 0; j < i; j++)
            if(pairs[j][1] < pairs[i][0] && arr[j] + 1 > arr[i])
                arr[i] = arr[j] + 1;
        if(arr[i] > l)
            l = arr[i];
    }
    return l;
}
```

时间复杂度是 O(n^2)，不是最佳解法。由于根据数对的最大值排好了序，因此可以直接用下列方法来完成：

```
public int findLongestChain(int[][] pairs) {
```

```
    Arrays.sort(pairs, (a,b) -> a[1] - b[1]);
    int sum = 0, n = pairs.length, i = -1;
    while (++i < n) {
        sum++;
        int curEnd = pairs[i][1];
        while (i+1 < n && pairs[i+1][0] <= curEnd) i++;
    }
    return sum;
}
```

具体实现方法总结

经过以上三个例题的锻炼，一般难度的动态规划问题应该都能解决了。总结一下，动态规划思想具体实现有以下两种方法：

1. 可以用**带备忘的自顶向下法（top-down with memoization）**的方法计算状态转移方程。此方法仍然按自然的递归形式编写过程，但是用一个数组或者散列表来存储每个子问题的解，当需要时先检查是否已经保存过此解并取用。

2. 还可以采用**递推法**自底向上地计算状态转移方程。递推的关键是边界和计算顺序，将子问题按照规模从小到大进行求解，当求解某个子问题时，其所依赖的更小的子问题都已求解完毕。在多数情况下，递推法的时间复杂度是：状态总数 *每个状态的决策个数* 决策时间。如果不同状态的决策个数不同，需具体问题具体分析。注意递归和递推的区别：一个自顶向下，一个自底向上。

扩展

最长非降子序列（LIS）

给定一个序列 `A[1]`、`A[2]`、`...`、`A[n]`，求其最长非降子序列（LIS，longest increasing subsequence）的长度。这是讲动态规划时基本都会讲到的一个问题。

其最小子问题即求 `A[1]`、`A[2]`、`...`、`A[i]` 的最长非降子序列的长度，其中 `i < N`；而状态则定义有 `d(i)` 表示前 i 个数中以 `A[i]` 结尾的最长非降子序列。

当要考虑初始状态（边界状态）的值时，最好是以一个实际输入为例。假定要求的序列是： `5, 3, 4, 8, 6, 7`，则有：

- 前 1 个数的 LIS 长度 `d(1) = 1`；
- 前 2 个数的 LIS 长度 `d(2) = 1`（序列：3；3 前面没有比 3 小的）；
- 前 3 个数的 LIS 长度 `d(3) = 2`（序列：3，4；4 前面有个比它小的 3，所以 `d(3)=d(2)+1`）；
- 前 4 个数的 LIS 长度 `d(1) = 1`（序列：3，4，8；8 前面比它小的有 3 个数，所以 `d(4) = max{d(1),d(2),d(3)}+1 = 3`）；

由此得到状态转移方程：

```
d(i) = max{1, d(j) + 1}，其中 i > j，A[i] >= A[j]
```

想要求 `d(i)`，就把 i 前面的各个子序列中，最后一个数不大于 `A[i]` 的序列长度加 1，然后取出最大的长度即为 `d(i)`。当然，有可能 i 前面的各个子序列中最后一个数都大于 `A[i]`，那么 `d(i)=1`，即它自身成为一个长度为 1 的子序列。

代码实现如下：

```
int lis(int[] A) {
    int n = A.length;
    int len = 1;
    int[] d = new int[n];
    for(int i = 0; i < n; i++) {
        d[i] = 1;
        for(int j = 0; j < i; j++)
            if(A[j] <= A[i])
                d[i] = Math.max(d[i], d[j] + 1);
        len = Math.max(d[i], len);
    }
    return len;
}
```

时间复杂度为 O(n^2)，不是最优解法。可以看看最长递增子序列 O(NlogN)算法，有点复杂，这里就不多谈了。

背包问题

0-1 背包问题是最广为人知的动态规划问题之一，拥有很多变形。

有 n 种物品，每种只有一个。第 i 种物品的体积为 V[i]，价值为 W[i]。选一些物品装到一个容量为 C 的背包，使得背包内物品在总体积不超过 C 的前提下价值尽量大。1 <= n <= 100，1 <= V[i] <= C <= 10000，1 <= W[i] <= 10^6。

将原问题分解为子问题后，状态还是比较好找的。我们可以用 `d(i, j)` 来表示前 i 个物品装到剩余体积为 j 的背包里能达到的最大价值。

对于第 i 个物品，可以装进或不装进背包。不装进背包，则背包中物品最大总价值为 `d(i-1, j)`；而如果装进背包，对于前 i-1 个物品的空间就只有 j-V[i] 了。

由此得到状态转移方程：

```
d(i, j) = max{d(i-1, j), d(i-1, j-V[i]) + W[i]}
```

得到状态转移方程后，代码也不难写出了。这里就不贴了，有兴趣可以自己试试。

Dijkstra 算法

Dijkstra 算法也是以动态规划为基础的。你可以到我《算法》笔记的相关章节对 Dijkstra 算法进行进一步了解：Algorithms-notes/笔记/4.4 最短路径 （Dijkstra 算法相关内容正在添加中）。

结语

动态规划有着很强的理论性和实践性，可以考验出算法能力，因此经常在各类算法竞赛、面试题中出现。想要完全掌握，光搞定这一篇博客的几个例题远远不够，只有多做经典题目，才能当再碰到动态规划相关题目的时候做到游刃有余。

参考资料

写作参考

- 动态规划：从新手到专家
- 教你彻底学会动态规划——入门篇 - CSDN博客
- 《算法竞赛入门经典（第 2 版）》第 9 章

学习参考

- 漫画：什么是动态规划？ – 掘金：可以借助这个漫画来理解动态规划，并了解动态规划在某些背包问题的特例上计算速度的局限性。
- 动态规划之背包问题（一）
- 什么是动态规划？动态规划的意义是什么？ – 知乎
- javascript背包问题详解 – 个人文章 – SegmentFault

姊妹篇

- 算法一篇通——贪心算法

> **本文作者：** Kyon Huang
>
> **本文链接：** http://kyonhuang.top/dynamic-programming/
>
> **版权声明：** 本博客所有文章除特别声明外，均采用 CC BY-NC-SA 4.0 许可协议。转载请注明出处！

算法 # 动态规划 # DP

花旗杯–新的终点，新的起点

Goodbye 2017, hello 2018