对动态规划（Dynamic Programming）的理解：从穷举开始

21 Jan 2015

动态规划（Dynamic Programming，以下简称dp）是算法设计学习中的一道槛，适用范围广，但不易掌握。

笔者也是一直不能很好地掌握dp的法门，于是这个寒假我系统地按着LRJ的《算法竞赛入门经典》来学习算法，对dp有了一个比过往都更系统 \ 更深入的理解，并在这里写出来与大家分享。

笔者着重描述的是从穷举到dp的算法演进，并从中获取dp解法的思路，并给出多种思考的角度，务求解决的是LRJ提出的一种现象：

> “每次碰到新题自己都想不出来，但一看题解就懂”的尴尬情况。

DP与穷举

大多讲dp的文章都是以0-1背包为基础来讲解的，笔者想换个花样，以另一道题“划分硬币”（UVa-562 Dividing coins）来讲述。

> 现在有一袋硬币，里面最多有100个硬币，面值区间为[1, 500]，要分给两个人，并使得他们所获得的金钱总额之差最小，并给出这个最小差值。

这种问题笔者称之为**二叉树选择问题**。 假设袋中有N个硬币，我们从最原始的枚举法来一步步优化。

我们让其中一个人先挑硬币，挑剩的就是给另外一个人的。 第一个人对于每一个硬币，都有“选”和“不选”两种选择。 我们很容易穷举他全部的情况——全部硬币的任意大小的子集，共2^N种，并选取其中两人差值最小的解。

具体作法可以用递归法，二进制法等。 在这里给出笔者的递归解法的代码，因为它与后面的优化紧密关联。

```
int solve_by_brute_force(vector<int> &v, int cur, int limit, int
 sofar, int sum) {
    if (cur == limit) { // the border of recursion
```

```
        int other = sum - sofar;
        return sofar>other? sofar-other : other-sofar;
    }

    // choose or not the current coin
    int ans1 = solve_by_brute_force(v, cur+1, limit, sofar+v[cur
], sum);
    int ans2 = solve_by_brute_force(v, cur+1, limit, sofar, sum
);
    return ans1 < ans2? ans1 : ans2;
}

int main(){
    int t;
    cin >> t;
    for (int nc= 0; nc<t; nc++) {
        int m;
        cin >> m;
        vector<int> v(m);
        int tot = 0;
        for (int i= 0; i<m; i++) {
            cin >> v[i];
            tot += v[i];
        }
        int res = solve_by_brute_force(v, 0, m, 0, tot);
        cout << res << endl;
    }
    return 0;
}
```

$O(2^N)$的指数复杂度始终是撑不过去的，我们需要优化，但从哪里开始优化呢?

我们以测试样例“2 3 5”来看看，下图是样例的穷举**解答树**，左树为不选当前硬币，右树为选入，叶结点为第一个人的硬币总额。

```
           root
          /    \
 2       /      \
        /\      /\
 3     /  \    /  \
      /\  /\  /\  /\
 5   /  \/  \/  \/  \
    0   5        5  2+3+5
```

可以看到，叶结点中有两个相同的值，思考一下，如果再多加一个硬币（解答树变成4层），相同的叶结点会引起无谓的重复计算，造成浪费。

我们从1到N逐个做硬币的选择决策

- **以树的结构来记录当前硬币的累计金钱（中间结果）是低效的，因为它允许存在值相同的结点。** 我们思考，那么我们何不换一种方式来记录中间结果呢？我们何不采用数组的方式？假设一袋硬币存在 `int Coins[]` 数组中，中间结果的范围是可求的，区间为 `[0, Coins[0]+Coins[1]+..+Coins[cur]]` ，或者干脆是全部硬币的总和 $[0, SUM]$。

- **以树分叉的方式来做决策也是可以改变的。**改变中间结果的记录方式后，我们可以这样来递推： 假设有一个二维数组 `bool mat[N+1][SUM+1]` ，第一维表示当前已选择到第几个硬币，第二维表示中间结果，比如 `mat[3][10]` 表示在“2, 3, 5”样例中，我们选择到第3个硬币（最后一个），并且中间结果（正好是最后的叶结点）为10的情况，如果为 `true` ，则这种情况成立，是存在的，否则不存在，比如 `mat[3][9]` ，你是拼不出和为9的情况的。 这时递推公式变为 `mat[cur][i] |= mat[cur-1][i-Coins[cur]]` ，我们只要在每个硬币（每层）把 `i` 在区间$[0, SUM]$枚举一遍就可以了。

这个方法的完整代码：

```
const int MAX_SUM = 100000;
bool arr[MAX_SUM];

int solve_by_dp(vector<int> &v, int sum) {
    fill(arr, arr+MAX_SUM, false);
    arr[0] = true;
    for (int i= 0; i<v.size(); i++) {
        for (int j= v[i]; j<=sum; j++) {
            if (arr[j-v[i]])
                arr[j] = true;
        }
    }
    int mid = sum/2;
    for (int i=0; i<=mid; i++) {
        int one = mid-i;
        if (arr[one]) {
            int other = sum - one;
            return other>one? other-one : one-other;
        }
    }
    return -1;  // never been here
}
```

这样复杂度变为\(O(NSUM)\)，*即*\(O(500N^2)\)，从指数降到多项式了。 而空间复杂度，考虑到可以当前硬币的中间结果，只与上一层有关，我们使用**滚动数组**的方法就可以避免开一个二维数组的耗费了。

其实中间数组的每个元素，就是dp中的“状态”。 **对dp的理解首先要从枚举开始，发现穷举过程中做无用功的地方，改变记录和递推的方式去优化它。** 关于格举的解答树和dp的状态之间的关系，LRJ有一个很好的总结。

> 状态及其转移类似于回溯法听解答树。解答树中的“层数”，也就是递归函数中的“当前填充位置” `cur` ，描述的是即将完成的决策序号，在动态规划中被称为“阶段”。

如何寻找DP的思路

在做题的时候，我有意识地注意对自己dp的思路寻找过程，而不满足于单纯地套模板。

我往往**先考虑最朴素的穷举思路，并努力发现隐藏其中的“最优子结构”，寻找子问题的递推关系。** 我相信这也是LRJ在书中按“穷举－分治－dp”的顺序编排章节的原因。

LCS问题（Longest Common Sequence）

比如一个典型的dp问题——LCS（Longest Common Sequence），求两个字符串的最长公共子串（不要求连续），比如这道题UVA - 10405。

假设两个字符串为a, b，最后结果为子串s，按枚举的思路，对a的每一个字符进行枚举，假设它作为公共子串的首字符，并在b中从左往右寻找与它相同的字符，假设 `a[i]==b[j]` ，这样就得出子问题——现在问题变成了求a, b匹配字符的后缀的LCS，即 `lcs(a+i+1, b+j+1)` 。 文字描述不仔细，请看看代码。

```
// example
// string a = “a1b2c3d4e”, b = “zz1yy2xx3ww4vv”;
// cout << lcs(a, b, 0, 0, 0);
int lcs(string &a, string &b, int ia, int ib, int cur) {
    int max = cur;
    for (int i= ia; i<a.length(); i++) {
        char c = a[i];
        int j;
        for (j= ib; j<b.length(); j++) {
            if (c == b[j])
                break;
        }
        if (j == b.length()) continue;
        int ans = lcs(a, b, i+1, j+1, cur+1);
        if (ans > max) max = ans;
    }
    return max;
}
```

上述的代码复杂度非常大，原因在于相同的 `lcs(a, b, i, j, cur)` 可能会被多次调用，比如 `lcs(a, b, 3, 4, 1)` 和 `lcs(a, b, 3, 4, 2)` （数据乱编的），它们的目的都是相同的——想求得 `lcs(a+i, b+j)` 。发现穷举过程中隐藏的子模式 \ 子问题之后，可以把中间结果先算出来，并且它们正好可以存在一个二维表中，代码如下。这就是LCS问题的思考过程了。

```
const int N = 100;
int mat[N][N];
int lcs_dp(string &a, string &b) {
    fill(&mat[0][0], &mat[0][0]+N*N, 0);
    for (int i= a.length()-1; i>=0; i--) {
        for (int j= b.length()-1; j>=0; j--) {
            int &cur = mat[i][j];
            if (a[i] == b[j])
                cur = 1 + mat[i+1][j+1];
            else
                cur = mat[i][j+1]>mat[i+1][j] ? mat[i][j+1] : ma
t[i+1][j];
        }
    }
    return mat[0][0];
}
```

有些子问题也穷举思路中也许不明显，需要读者自己去寻找一个**划分点**，然后创造**重叠子问题**。本节问题的划分点是公共子串的第一个字符，再如“表达式链”问题（如UVA - 348）的划分点则是“最后一次运算”，都可以从穷举的思路中发现问题子模式。

经典的0-1背包问题也是如此，本来每个物品均是两种选择——放和不放，穷举法是用一种二叉树来表示，可以用本文第一节所说的方法来解决——记录并更新全部可能重量的数组。也可以用另一种递归思路，即从第1个物品开始，放和不放，影响的是背包的剩余容积，很容易可以构造出重叠子问题——在特定容积下，余下的物品可以放入的最大重量。

多阶段决策问题

还有一类dp问题，是多阶段决策中需要“分层”或者“分阶段”的dp问题。

比如找**零钱的方案数**(UVA - 357)问题。

> 有$50, $25, $10, $5 and $1不同面值的纸钞，现在给定一个需要找零的金额数$N，问一共有多少种不同的找零方法。

如果按照上节所说的穷举方法，我们很容易得到一个4叉树（有4种面值的纸钞），这棵树一直延伸发散，直到从根结点到叶结点的路径总和等于或大于N才停止，然后统计路径总和为N的路径数。

但这个穷举思路有问题，比如现在 `N=15` ，穷举树先伸出5再伸出10，和先伸出10再伸出5，都可以达到路径和为15，并算作两条路径，但其实它们是等价的。 这类问题需要加入一个“**先后顺序**”的概念，LRJ的原话是：

> 原来的状态转移太敌了，任何时候都允许使用任何一种物品，难以控制。为了消除这种混乱，需要让状态转移（也就是决策）有序化。

我们可以从每种硬币的个数上进行穷举，代码如下。

```
int coins[] = {1, 5, 10, 25, 50};

int count_ways(int n) {
    int i1, i5, i10, i25, i50;
    int cnt = 0;
    for(i1= 0; i1<=n; i1++) {
        for(i5= 0; i5*5<=n; i5++) {
            for(i10= 0; i10*10<=n; i10++) {
                for(i25= 0; i25*25<=n; i25++) {
                    for(i50= 0; i50*50<=n; i50++) {
                        int tmp = i1 + i5*5 + i10*10 + i25*25 +
i50*50;
                        if (tmp == n) cnt++;
                    }
                }
            }
        }
    }
    return cnt;
}
```

同样是穷举，不同之处在于第二种方案中有“顺序”\“层”\“阶段”的概念，各种硬币并不允许随时被加入最终的总和。 不像第一种方案，为了得到$15，可以按$5+$10和$10+$5多种顺序得到，第二种只能是$5+$10，不会重复。

反映在dp方法上，我们也得做出改变，我们要**依次记录不同阶段的中间结果——**用 `arr[N]` 表示 `N` 有多少种拼凑方法，只用$1硬币计算一遍，再加上$5硬币计算一遍，再加入……直到5种硬币都加入了，最终结果就出来了，而且结果不会重复。

```
const int N = 1000;
int arr[N];
```

```
int count_ways_dp(int n) {
    fill(arr, arr+N, 0);
    arr[0] = 1; // initialization
    for (int i= 0; i<NCOINS; i++) {
        for (int j = coins[i]; j<=n; j++) {
            arr[j] += arr[j-coins[i\\);
        }
    }
    return arr[n];
}
```

> “阶段”只是帮助我们思考的，在动态规划的状态描述中最好避免“阶段” \ “层”这样的术语。

用这种方式来描述“阶段”这种概念，笔者也略感到勉力，希望各位看官能反馈回来更好的见解。

最后的话

笔者的算法学习经历并不多，大一的时候读了半本《算法概论》，选了一门水水的算法课（作用几可忽略＝＝），课余参加过一些Codeforces的在线比赛，最认真的，就是这个大学最后的寒假闭关在家按着LRJ的《算法竞赛入门经典》苦练一番，收获颇丰，而其中以dp为最难，故才有此一文来总结这个寒假的算法学习。

不得不说，整个算法的思考过程对我来说很不容易。以在Quora看来的一句与算法（数学）学习相关的话跟诸君共勉。

> Most people simply don’t put in the time to make something intuitive.

参考资料

- 算法竞赛入门经典

Related Posts

调研：词性标注（POS Tagging）各主流实现 10 Apr 2017

生产与消费，共性与个性 09 Sep 2015

直男护肤二三事 25 Jul 2015

Comments

多说

Disqus