02 暴力递归：当贪心失效了怎么办？

你好，我是卢誉声。

上一课我们学习了贪心算法，提出了硬币找零的问题，发现了贪心算法的局限性。与此同时，我还提出了一个重要概念，那就是局部最优与整体最优的概念，即最优化问题。今天，我们就从最优化问题开始聊起，引出学习动态规划时的另一重要概念：递归。

我们之前说过，贪心算法是求解整体最优的真正思路源头，这是为什么我们要在这门课程的一开始从贪心算法讲起。现在，你应该已经意识到贪心算法是有局限性的，它只能在局部最优的思想下工作，**那么当贪心算法失效了怎么办？**

接下来我们就带着这个问题，开始学习今天的内容：递归！看看它能否更进一步地解决我们遇到的棘手问题，从整体最优的角度来解决算法问题。

从最优化问题到递归

贪心算法失效的很大一个原因在于它明显的局限性：它几乎只考虑局部最优解。所谓局部最优，就是只考虑当前的最大利益，既不向前多看一步，也不向后多看一步，导致每次都只用当前阶段的最优解。

因此在绝大多数情况下，贪心算法不能得到整体最优解，但它的解是最优解的一个很好近似。同时，也是所有讨论最优化问题的核心基础。

既然无法通过贪心算法达到整体最优，我们就得换一个思路了：我们得从整体最优的层面上解决这个难缠的算法问题。那么从何说起呢？我认为你应该先理解最优化问题的本质，然后再把这个思考扩展到递归问题上。话不多说，我们这就开始吧！

最优化问题的本质

所谓最优化问题，就是指在某些约束条件下，决定可选择的变量应该取何值，使所选定的目标函数达到最优的问题。

从数学意义上说，最优化方法是一种求极值的方法，即在一组约束为等式或不等式的条件下，使系统的目标函数达到极值，即最大值或最小值。

如果只是从概念上来看最优化问题真的是玄而又玄，所以在上一课中我用了硬币找零的例子，引出了最优化的概念，以便你理解。

在数学里一切都是函数，现在我们先把这个问题用函数形式来表示。为了易于理解，下面我们不会使用向量。

我们假定需要给出y元硬币，硬币面额是5元和3元，求出需要的最少硬币数量。所谓的最少硬币数量就是5元硬币和3元硬币的总数，假定5元硬币数量为x_{0}，3元硬币数量为x_{1}，那么用函数表示就是：

$$f(x_{0}, x_{1})=x_{0}+x_{1}$$

这就是所谓的“目标函数”。

但是这个函数现在是没有任何限制的，我们希望对此进行约束，使得5元硬币和3元硬币的面值综合为y。为此我们需要给出一个约束：

$$5x_{0}+3x_{1}=y$$

这个时候我们的问题就变成了，当满足这个约束条件的时候，求解函数中的变量x_{0}和x_{1}，使得目标函数$f(x_{0}, x_{1})$的取值最小。如果用数学的描述方法来说的话，就是下面这样：

$$\arg\min_{(x_0,x_1)\in S} (x_0+x_1)$$

这个就是我们常见的$argmin$表示方式。它的意思是：当(x_{0}, x_{1})属于S这个集合的时候，希望知道$x_{0} + x_{1}$的最小值是多少。其中S集合的条件就是上面的约束。

所以最优化问题在我们生活中是非常普遍的，只不过大多数问题可能都像硬币找零问题这样看起来普普通通，概念其实是不难理解的。

回到硬币找零这个问题上。由于(x_{0}, x_{1})都是离散的值，因此所有满足上述约束的(x_{0}, x_{1})组合，就是我们最终所求的集合！而这个最优化问题的本质就是：从所有满足条件的组合(x_{0},x_{1})中找出一个组合，使得$x_{0}+x_{1}$的值最小。

所以，你会发现在这种离散型的最优化问题中，本质就是从所有满足条件的组合（能够凑出y元）中选择出使得我们的目标函数（所有硬币数量之和）最小的那个组合。而这个所谓满足条件的组合不就是$argmin$公式中的那个集合S吗？

因此，这种离散型的最优化问题就是去所有满足条件的组合里找出最优解的组合。我曾多次提到的**局部最优**就是在一定条件下的最优解，而**整体最优**就是我们真正希望得到的最优解。

那么我们的视角就转到另一边了：如何去找到这个最优解呢？

枚举与递归：最优组合的求解策略

如果想得到最优组合，那么最简单直接的方法肯定就是**枚举**。枚举就是直接求出所有满足条件的组合，然后看看这些组合是否能得到最大值或者最小值。

在硬币找零问题中，假设现在需要给出25元的硬币，有两种组合，分别是(5, 0)和(2, 5)，也就是5个5元硬币，或者2个5元硬币加上5个3元硬币，那么硬币数量最小的组合肯定就是(5, 0)。

所以最简单的方法就是找出所有满足条件的组合，也就是上面两个组合，然后去看这些组合中的最优解。

枚举本身很简单，就是把所有组合都遍历一遍即可。可现在问题就是，**如何得到这些组合呢？**

这就需要我们通过一些策略来生成所有满足条件的组合。而**递归**正是得到这些组合的方法。在解决问题前，我们先回顾一下递归问题的本质。

递归与问题表达

我们可以看出，其实最优化问题使用递归来处理是非常清晰的，递归是搜索组合的一种非常直观的思路。

当我在稍后的课程里讨论动态规划时，你就会发现所有问题都需要被描述成递归的形式来讨论。

所以我们有必要先巩固一下递归的概念。首先是在数学中我们怎么去用递归描述一个问题，然后是如何用递归描述最优化问题的解法。

从斐波那契数列说起

严格来说，斐波那契数列问题不是最优化问题，但它能很好地展示递归的概念。我们先来看一下斐波那契数列的问题描述。

问题：斐波那契数通常用 F(n) 表示，形成的序列称为斐波那契数列。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和：

\[F(n)=\\left\\{\\begin{array}{c}- 0,n=0\\\\\\- 1,n=1\\\\\\- F(n-1)+F(n-2),n>1-\\\\end{array}\\\\right.\]

```
示例 1：

输入：2
输出：1
解释：F(2) = F(1) + F(0) = 1 + 0 = 1。


示例 2：

输入：3
输出：2
解释：F(3) = F(2) + F(1) = 1 + 1 = 2.
```

很多人在解算法面试问题的时候有一种倾向性，那就是使用迭代而非递归来求解问题。我先不说这样的倾向性正确与否，那么我们就按照这个偏好来解一下（即斐波那契数列的循环解法）。

Java实现：

```java
int fibonacci(int n) {
    int[] resolution = {0, 1}; // 解的数组
    if(n < 2) { return resolution[n]; }

    int i = 1;
    int fib1 = 0, fib2 = 1, fib = 0;
    while(i < n) {
        fib = fib1 + fib2;
        fib1 = fib2;
        fib2 = fib;
        i++;
    }

    return fib; // 输出答案
}
```

C++实现：

```cpp
int Fibonacci(int n) {
    std::vector<int> resolution = {0, 1}; // 解的数组
    if(n < 2) { return resolution[n]; }

    int i = 1;
    int fib1 = 0, fib2 = 1, fib = 0;
    while(i < n) {
        fib = fib1 + fib2;
        fib1 = fib2;
        fib2 = fib;
```

```
        i++;
    }

    return fib; // 输出答案
}
```

嗯，这样的解法固然没错，但是它几乎脱离了题设的数学表达形式。在这道题目中，出题者“刻意”地写出了求解斐波那契数列的函数表达式，这其中有没有什么别的含义或原因呢？

当然有了，这个函数表达式很好地反应出了计算机科学中常见的算法形式：递归。 下面，就让我们来看看斐波那契数列与递归之间的关系。

使用递归求解斐波那契数列

事实上，斐波那契数列的数学形式就是递归的，我在这里直接贴出其递归形式的算法代码，你就能很清楚地看出这一点。

```
int Fibonacci(int n) {
  if (0 == n || 1 == n) { return n; }
  if(n > 1) { return Fibonacci(n - 1) + Fibonacci(n - 2); }

  return 0; // 如果输入n有误，则返回默认值
}
```

递归形式的求解几乎就是简单的把题设中的函数表达式照搬过来，因此我们说从数学意义上讲，递归更直观，且易于理解。

使用递归求解硬币问题

你可以看出，理解递归并不难，现在我们要把这种思路套用到求解硬币的问题上来。话不多说，我在这里直接贴出使用递归求解硬币问题的代码实现。

Java实现：

```
void getMinCountsHelper(int total, int[] values, ArrayList<Integer> currentCounts, Arra
    if (0 == total) { // 如果余额为0，说明当前组合成立，将组合加入到待选数组中
        combinations.add(new ArrayList<Integer>(currentCounts));
        return;
    }

    int valueLength = values.length;
    for (int i = 0;  i < valueLength; i ++) { // 遍历所有面值
        int currentValue = values[i];
        if (currentValue > total) { // 如果面值大于当前总额，直接跳过
            continue;
        }
```

```
        // 否则在当前面值数量组合上的对应位置加1
        ArrayList<Integer> newCounts = new ArrayList<Integer>(currentCounts);
        newCounts.set(i, newCounts.get(i)+1);
        int rest = total - currentValue;

        getMinCountsHelper(rest, values, newCounts, combinations); // 求解剩余额度所需硬
    }
}

int getMinimumHelper(ArrayList<ArrayList<Integer>> combinations) {
    // 如果没有可用组合，返回-1
    if (0 == combinations.size()) { return -1; }

    int minCount = Integer.MAX_VALUE;
    for (ArrayList<Integer> counts : combinations) {
        int total = 0; // 求当前组合的硬币总数
        for (int count : counts) { total += count; }

        // 保留最小的
        if (total < minCount) { minCount = total; }
    }

    return minCount;
}

int getMinCountOfCoins() {
    int[] values = { 5, 3 }; // 硬币面值的数组
    int total = 11; // 总值

    ArrayList<Integer> initialCounts = new ArrayList<>(Collections.nCopies(values.lengt

    ArrayList<ArrayList<Integer>> coinCombinations = new ArrayList<>(); // 存储所有组合
    getMinCountsHelper(total, values, initialCounts, coinCombinations); // 求解所有组合

    return getMinimumHelper(coinCombinations); // 输出答案
}
```

C++实现：

```
void GetMinCountsHelper(int total, const std::vector<int>& values, std::vector<int> cur
    if (!total) { // 如果余额为0，说明当前组合成立，将组合加入到待选数组中
        combinations.push_back(currentCounts);
        return;
    }

    int valueLength = values.size();
    for (int i = 0;  i < valueLength; i ++) { // 遍历所有面值
        int currentValue = values[i];
        if (currentValue > total) { // 如果面值大于当前总额，直接跳过
            continue;
        }

        // 否则在当前面值数量组合上的对应位置加1
```

```
        std::vector<int> newCounts = currentCounts;
        newCounts[i] ++;
        int rest = total - currentValue;

        GetMinCountsHelper(rest, values, newCounts, combinations); // 求解剩余额度所需硬
    }
}

int GetMinimumHelper(const std::vector<std::vector<int>>& combinations) {
    // 如果没有可用组合，返回-1
    if (!combinations.size()) { return -1; }

    int minCount = INT_MAX;
    for (const std::vector<int>& counts : combinations) {
        int total = 0; // 求当前组合的硬币总数
        for (int count : counts) { total += count; }

        // 保留最小的
        if (total < minCount) { minCount = total; }
    }

    return minCount;
}

int GetMinCountOfCoins() {
    std::vector<int> values = { 5, 3 }; // 硬币面值的数组
    int total = 11; // 总值

    std::vector<int> initialCounts(values.size(), 0); // 初始值(0,0)
    std::vector<std::vector<int>> coinCombinations; // 存储所有组合
    GetMinCountsHelper(total, values, initialCounts, coinCombinations); // 求解所有组合

    return GetMinimumHelper(coinCombinations); // 输出答案
}
```

你从代码里可以看出，这里的操作被明确分成了两步：

1. 求解所有满足条件的组合；
2. 从组合中选出总和最小的组合。如果找不到满足条件的组合那么就返回-1。

我们也可以将这两步合并成一步来解决，就像下面这段代码。

Java实现：

```
int getMinCountsHelper(int total, int[] values) {
    // 如果余额为0，说明当前组合成立，将组合加入到待选数组中
    if (0 == total) { return 0; }

    int valueLength = values.length;
    int minCount = Integer.MAX_VALUE;
    for (int i = 0;  i < valueLength; i ++) { // 遍历所有面值
        int currentValue = values[i];
```

```
        // 如果当前面值大于硬币总额，那么跳过
        if (currentValue > total) { continue; }

        int rest = total - currentValue; // 使用当前面值，得到剩余硬币总额
        int restCount = getMinCountsHelper(rest, values);

        // 如果返回-1，说明组合不可信，跳过
        if (restCount == -1) { continue; }

        int totalCount = 1 + restCount; // 保留最小总额
        if (totalCount < minCount) { minCount = totalCount; }
    }

    // 如果没有可用组合，返回-1
    if (minCount == Integer.MAX_VALUE) { return -1; }

    return minCount; // 返回最小硬币数量
}

int getMinCountOfCoinsAdvance() {
    int[] values = { 3, 5 }; // 硬币面值的数组
    int total = 11; // 总值

    return getMinCountsHelper(total, values); // 输出答案
}
```

C++实现：

```
int GetMinCountsHelper(int total, const std::vector<int>& values) {
    // 如果余额为0，说明当前组合成立，将组合加入到待选数组中
    if (!total) { return 0; }

    int valueLength = values.size();
    int minCount = INT_MAX;
    for (int i = 0;  i < valueLength; i ++) { // 遍历所有面值
        int currentValue = values[i];

        // 如果当前面值大于硬币总额，那么跳过
        if (currentValue > total) { continue; }

        int rest = total - currentValue; // 使用当前面值，得到剩余硬币总额
        int restCount = GetMinCountsHelper(rest, values);

        // 如果返回-1，说明组合不可信，跳过
        if (restCount == -1) { continue; }

        int totalCount = 1 + restCount; // 保留最小总额
        if (totalCount < minCount) { minCount = totalCount; }
    }

    // 如果没有可用组合，返回-1
    if (minCount == INT_MAX) { return -1; }

    return minCount; // 返回最小硬币数量
```

```
}

int GetMinCountOfCoinsAdvance() {
    std::vector<int> values = { 5, 3 }; // 硬币面值的数组
    int total = 11; // 总值

    return GetMinCountsHelper(total, values); // 输出答案
}
```

在这段代码中，每一次递归返回的值，都是后续组合之和的最小值。它不再存储所有的组合，直到回退到递归的顶层。

这样可以极大节省存储空间，这是处理递归问题的通用方法。一般来说，你都应该用这种算法处理方式来解递归问题。

深入理解递归

在了解了递归的概念、问题的描述方式和解决问题方法后，我想让你来思考这样一个问题：**为什么递归能帮助我们解决最优化问题?**

堆栈与递归的状态存储

在计算机中，实现递归必须建立在堆栈的基础上，这是因为每次递归调用的时候我们都需要把当前函数调用中的局部变量保存在某个特定的地方，等到函数返回的时候再把这些局部变量取出来。

而用于保存这些局部变量的地方也就是堆栈了。

因此，你可以看到递归可以不断保存当前求解状态并进入下一层次的求解，并在得到后续阶段的解之后，将当前求解状态恢复并与后续求解结果进行合并。

在硬币找零问题中，我们可以放心的在函数中用循环不断遍历，找出当前面值硬币的可能数量。而无需用其它方法来存储当前或之前的数据。

得益于递归，我们通过堆栈实现了状态存储，这样的代码看起来简单、清晰明了。在本节课稍后的内容中，在我讲到递归树的求解组合空间时，你会更清晰地认识到堆栈和状态存储带来的价值!

递归与回溯

在求解最优化问题的时候，我们经常会用到**回溯**这个策略。

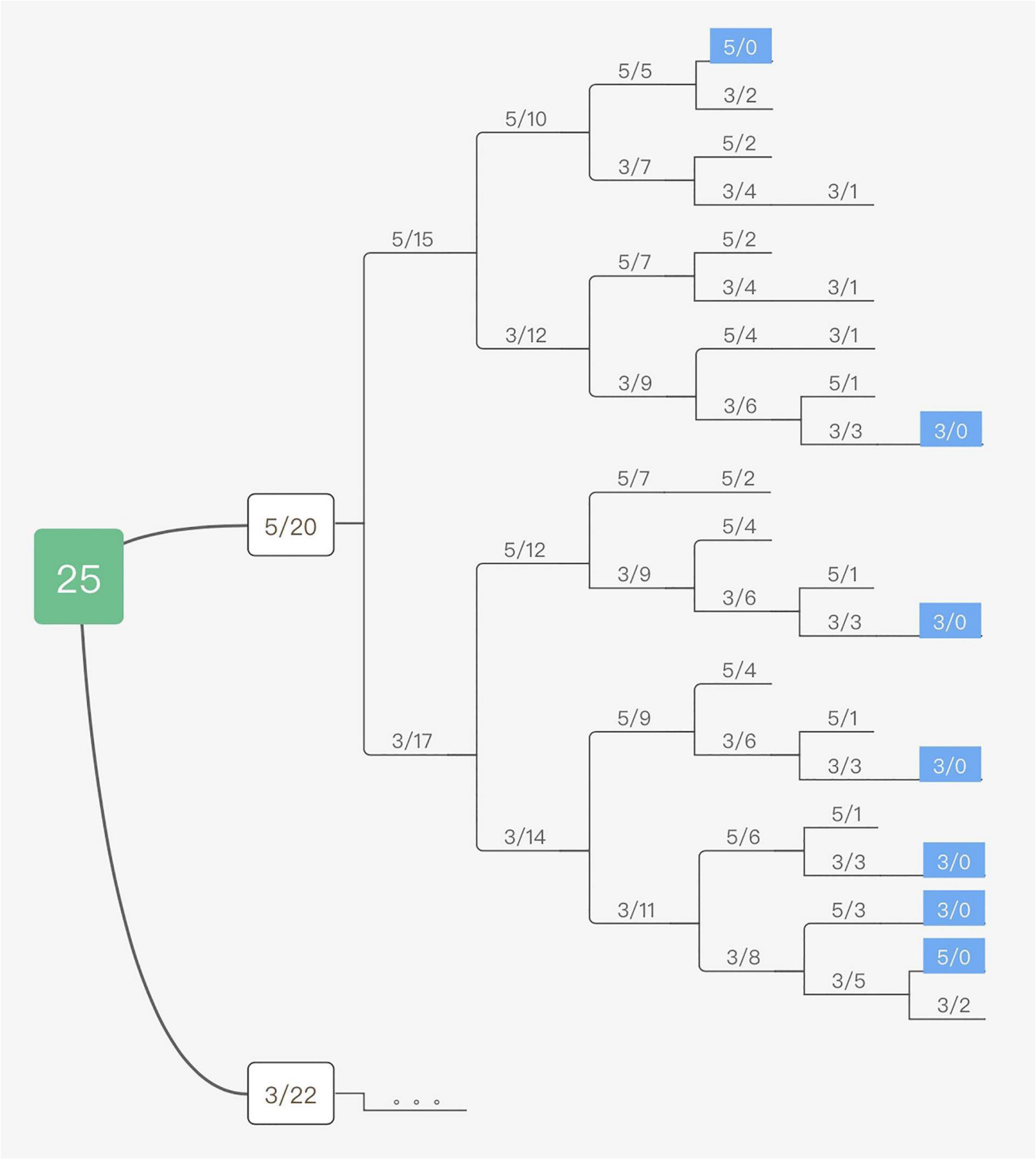
上一课中，我们已经提到过回溯的思想。在硬币找零这个问题里，具体说就是如果遇到已经无法求解的组合，那么我们就往回退一步，修改上一个面值的硬币数量，然后再尝试新的组合。

递归这种形式，正是赋予了回溯这种可以回退一步的能力：它通过堆栈保存了上一步的当前状态。

因此，如果想要用回溯的策略来解决问题，那么递归应该是你的首选方法。所以说，回溯在最优化问题中有多么重要，递归也就有多么重要。

树形结构与深度优先搜索

为了理解递归，我在这里用合适的结构来描述递归的求解过程。这种结构正是计算机数据结构中的树。如下图所示：

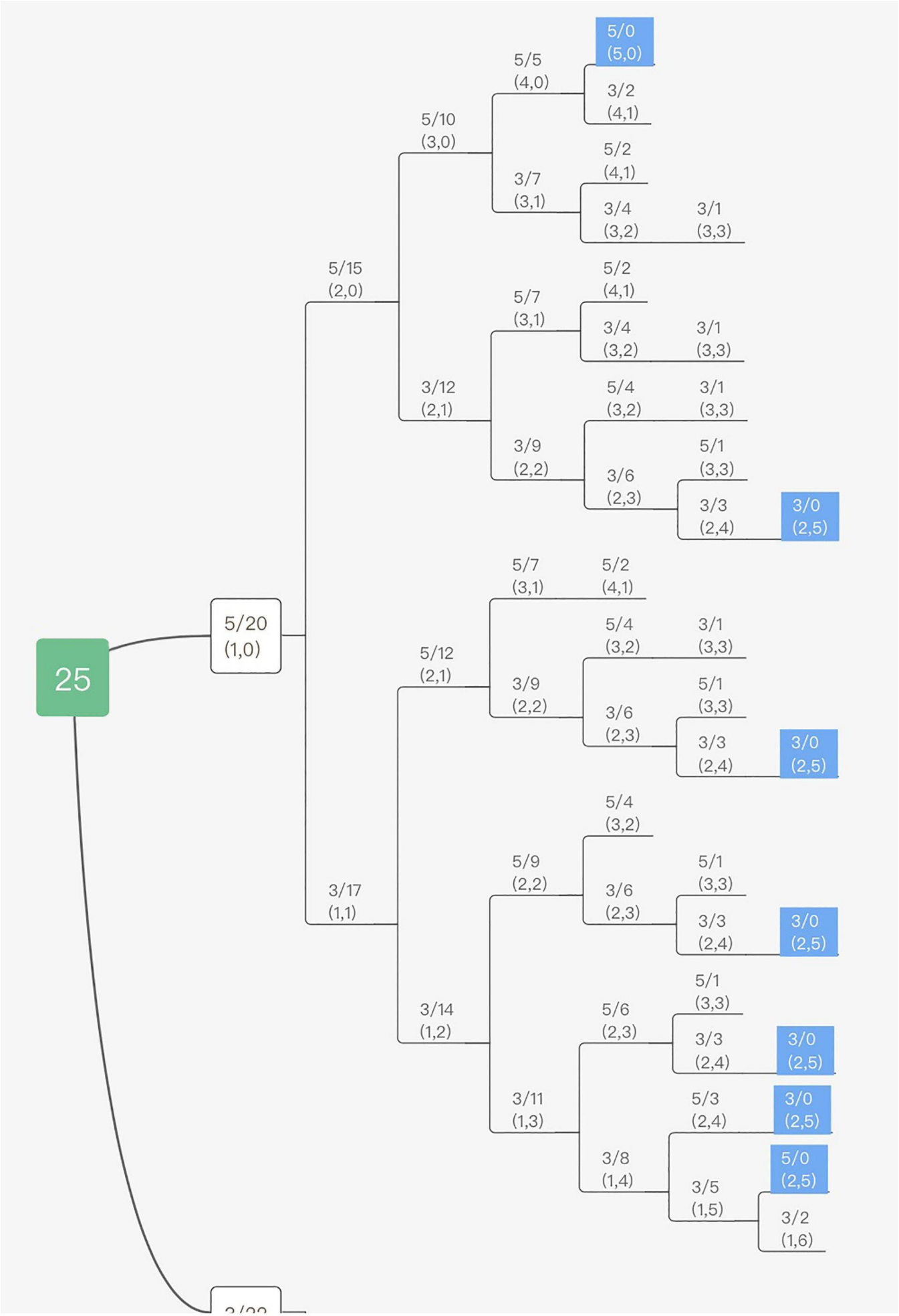


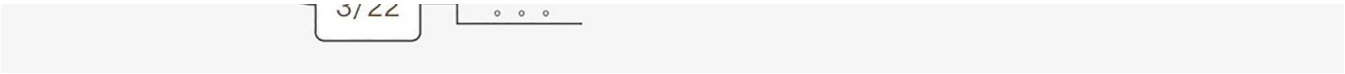
你可以从中看到形象的递归求解过程，每个节点的 /（斜线）左边表示当前节点使用的硬币面值，右边表示使用面值后的余额。图中的蓝色节点就表示我们目前得到的解。

递归的过程的确就是一个树形结构，而递归也就是一个深度优先搜索的过程，先找到下一步的解，然后再回退，如此往复。

所以我们可以这样理解递归：作为一个算法解决方案，它采用了深度优先搜索的策略，去搜索所有可能的组合，并得到最优解的最优化问题。

如果在每个节点上加上当前这个节点求得的组合结果，就可以用递归树表示**求解的组合空间**：


25
5/20
(1,0)
5/15
(2,0)
5/10
(3,0)
5/5
(4,0)
5/0
(5,0)
3/2
(4,1)
3/7
(3,1)
5/2
(4,1)
3/4
(3,2)
3/1
(3,3)
3/12
(2,1)
5/7
(3,1)
5/2
(4,1)
3/4
(3,2)
3/1
(3,3)
3/9
(2,2)
5/4
(3,2)
3/1
(3,3)
3/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
3/17
(1,1)
5/12
(2,1)
5/7
(3,1)
5/2
(4,1)
3/9
(2,2)
5/4
(3,2)
3/1
(3,3)
3/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
3/14
(1,2)
5/9
(2,2)
5/4
(3,2)
3/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
3/11
(1,3)
5/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
3/8
(1,4)
5/3
(2,4)
3/0
(2,5)
3/5
(1,5)
5/0
(2,5)
3/2
(1,6)



通过穷举法从所有的解中得到最优解

从上图中我们可以发现，每个节点都存储了一个当前求解过程中的组合，和后续节点的组合合并到一起形成完整的答案。

而真正的求解组合，就是把所有余额为0的组合拿出来，经过去重之后得到的结果。

所以，你可以看到求解的组合就蕴含在这个递归的树形结算的节点空间中，这也就是为什么递归策略是行之有效的：我们可以通过穷举法从所有的解中得到最优解！

暴力递归的问题与优化

从上一课介绍的贪心算法，到我在这里跟你讲的暴力递归法，看起来硬币找零问题有了一个稳定且行之有效的解题思路。

但这就足够了吗？哈哈，显然不是。因为这样的穷举法效率实在低下，不仅如此，这样的代码可读性低且调试困难。我在这里给你具体分析一下。

性能问题

暴力递归的最后一个特点就是穷举（都叫暴力，你说是不是）。如果我们只使用朴素的递归思路解题，就需要通过递归来暴力穷举出所有的组合，而且我们穷举的不只是组合，还是所有可能得到目标组合的组成路径！

这个在上面的图中我们可以看到，同样是求解(2, 5)这个组合，图中有多少种路径？这还只是25元和两种面值的情况。如果求解的金额和面值数量增加，那么我们可以看到这个树会以非常难以置信的方式增长，那么带来的性能问题就是灾难性的。

如果你仔细观察一下，就会发现这个树会随着总额的增加呈现指数形式的增长。对于这种事情，我们难以接受。

因此，递归只是让问题可以求解，但是如果数据规模过大的时候暴力递归会引发极大的性能问题。

可读性与调试问题

虽然递归在数学意义上非常直观，但是如果问题过于复杂，一般是无法直接画出上面我画的那棵求解树的。

有画求解树的时候，我们可以想出我们的求解过程是怎么进行的，但如果求解树的分支极多，那么很多人就很难继续在脑海中模拟出整个求解过程了。

因此，一旦程序出现bug，当你想尝试去调试的时候，就会发现这样的代码几乎没有调试的可能性。这种问题在数据规模很大的情况下尤为明显。

那么针对性能低下、代码可读性降低和调试问题，我们有什么办法去解决吗？有，听我给你讲下面的内容。

优化暴力递归：剪枝与优化

你可以从前面的图中看到，这棵树中有很多分支是完全相同的：起码从理论上讲最终只有两个组合。但是这棵树到达同一种组合的路径却非常多，所以优化递归的思路其实就是如何减少搜索的分支数量。

分支数量减少了，递归效率也就高了。这就是所谓的**剪枝**优化。对于优化方法，这里我提供两种思路给你。

1. 参考贪心算法

第一种思路是仿照贪心算法，从整个搜索策略上来调整。也就是说，你要考虑这个问题的性质，即面值大的硬币用得足够多，那么这个组合的硬币总数肯定就最小。

所以在每一次递归时，我们不应该暴力地搜索所有的面值，而应该从面值最大的硬币着手，不断尝试大面值硬币的最大情况。

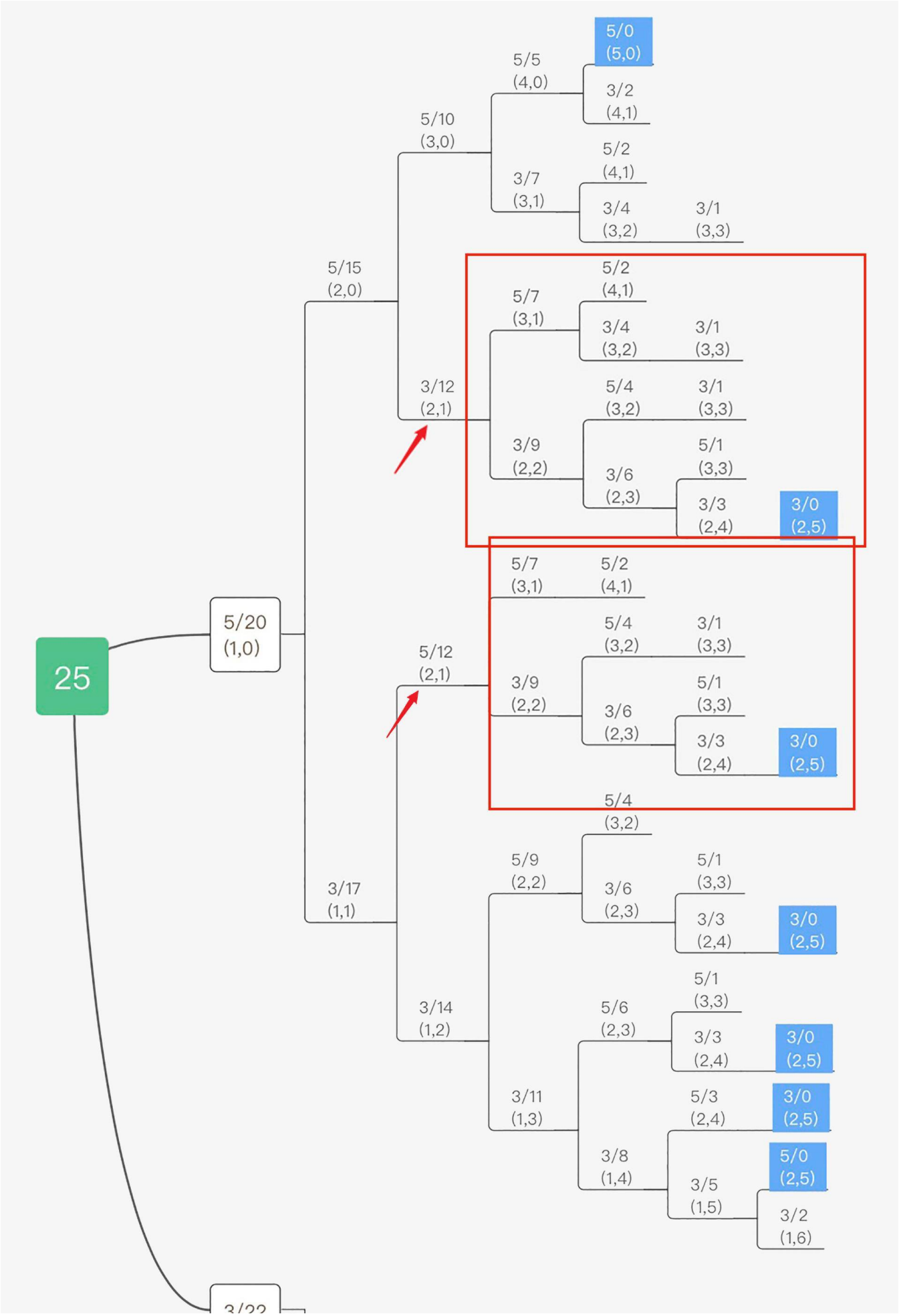
如果无法满足条件再减少一个，再递归搜索。最后的代码就跟我在上一课中写给你的回溯代码一样，即通过贪心这种思路结合递归实现一种组合搜索。

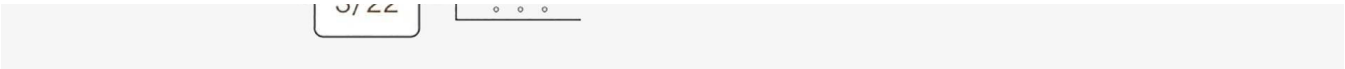
殊途同归啊！我们从递归的角度重新解释了这个算法问题，而且代码实现也是一样的。

2. 从解空间图解释

除了参考贪心算法的思想，我们还可以从解空间的角度来解释这个问题。

请你注意观察一下：在解空间的图中，只要是余额相同的情况下，后面的搜索路径是完全一致的！


25
5/20
(1,0)
5/15
(2,0)
5/10
(3,0)
5/5
(4,0)
5/0
(5,0)
3/2
(4,1)
3/7
(3,1)
5/2
(4,1)
3/4
(3,2)
3/1
(3,3)
3/12
(2,1)
5/7
(3,1)
5/2
(4,1)
3/4
(3,2)
3/1
(3,3)
3/9
(2,2)
5/4
(3,2)
3/1
(3,3)
3/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
5/12
(2,1)
5/7
(3,1)
5/2
(4,1)
3/9
(2,2)
5/4
(3,2)
3/1
(3,3)
3/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
3/17
(1,1)
5/9
(2,2)
5/4
(3,2)
3/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
3/14
(1,2)
5/6
(2,3)
5/1
(3,3)
3/3
(2,4)
3/0
(2,5)
3/11
(1,3)
5/3
(2,4)
3/0
(2,5)
3/8
(1,4)
3/5
(1,5)
5/0
(2,5)
3/2
(1,6)



我在图中圈出的两个部分就是重复的搜索路径。因为余额都是12元，所以后续的求解路径和结果完全相同。

这是一个重要线索，在这个硬币求解问题中，当余额相同的时候，最优解是确定的。那么你想想看，如果能够避免相同余额下的重复搜索过程，那么算法执行速度是不是可以加快了？

这就是我在上一课中提到过的**重叠子问题**。

你可以把求解12元的硬币数量理解成求解25元的硬币数量的一个子问题。在求解25元硬币过程中，会有很多种情况都要求解12元硬币的最优解。我们把这类会出现重复求解的子问题称之为重叠子问题。

显然，这就是我们可以优化的出发点。至于如何进行优化，则需要用比较多的篇幅讨论，在下一节课中，我会跟你细谈这个问题。

课程总结

今天我们学习了最优化问题的本质，即从所有满足条件的组合里找出最优解的组合。贪心算法只能解决**局部最优**问题，而我们的最终目标是解决**整体最优**问题（即最优解）。

自然地，**枚举**是获得最优解的理想方法。而**递归**可以帮助我们获得所有可能答案的组合。递归形式的求解几乎就是简单地把题设中的函数表达式照搬过来，它相较于迭代来说更直观，且易于理解。

但暴力递归有着十分明显的缺陷，存在性能低下、可读性低和调试困难等问题。为此，我们提出了剪枝与优化这两种方法：

1. 利用预设条件减少搜索路径，优化最优组合搜索方案（硬币的优化）；
2. 利用重叠子问题，避免重叠子问题的计算。

因此，在面试问题中，考虑贪心算法和递归是我们求解问题时思考的重要方向。很多面试问题已经可以使用这两种算法来解决了。

但在稍复杂的面试问题面前，我们还需要借助于更高级的手段：备忘录和动态规划。而重叠子问题是理解这些高级手段的基础，下节课我会具体来讲。

课后思考

今天我讲了递归求解最优解问题的思路，并强调了回溯的重要性。那如何通过编程，求出所有有效的括号组合呢？（设输入是有几组括号，输出是所有有效的括号组合）

欢迎留言和我分享你的答案，我会第一时间给你反馈。如果今天的内容对你有所启发，也欢迎把它分享给你身边的朋友，邀请他一起学习！

上一页 下一页