第四讲
递归算法 - 2
Lecture 4
Recursive Algorithms - 2

明玉瑞 Yurui Ming

yrming@gmail.com

声明
Disclaimer

- 本讲义在准备过程中由于时间所限，所用材料来源并未规范标示引用来源。所引材料仅用于教学所用，作者无意侵犯原著者之知识版权，所引材料之知识版权均归原著者所有；若原著者介意之，请联系作者更正及删除。

  The time limit during the preparation of these slides incurs the situation that not all the sources of the used materials (texts or images) are properly referenced or clearly manifested. However, all materials in these slides are solely for teaching and the author is with no intention to infringe the copyright bestowed on the original authors or manufacturers. All credits go to corresponding IP holders. Please address the author for any concern for remedy including deletion.

动态规划
Dynamic Programming

- 除了前章所讲的分治方法之外，还有一类算法，即动态规划，也会用到递归。虽然动态规划与分治法一样，都是通过组合子问题的解来解决问题，但分治算法可以将问题划分为不相交的子问题，递归地解决这些子问题，然后合并解决原始问题的解决方案。相反，利用动态规划时，虽然也是分解为子问题，但这些子问题会有重叠，即子问题共享有相同的子子问题。在这种情况下，如果用分而治之，则会做许多不必要的工作，比如需要重复解决共同的子子问题。如果利用动态规划算法，则每个子子问题只需解决一次，然后将其答案保存在表中，从而避免每次解决每个子子问题时，需要重新计算答案的工作。
- 动态规划规划一般用来解决最优问题，与分治所解决的问题不一样的是，分治问题一般有唯一答案，而最优问题不一定有唯一解。可以存在多个满足最优条件的解。

动态规划
Dynamic Programming

- Besides the divide-and-conquer method discussed in the previous chapter, there exists another category of methods called dynamic programming that also use recursion intensively. Like the divide-and-conquer algorithms, they both solve problems by combining the solutions to subproblems. However, for divide-and-conquer, it can partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, when dynamic programming applies, usually the decomposed subproblems are overlapped. That is, subproblems potentially share sub-subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub-subproblems. But a dynamic-programming algorithm solves each sub-subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each sub-subproblem.

动态规划
Dynamic Programming

- 在开发动态规划算法时，我们遵循四个步骤的序列：
  1. 表征最优解的结构。
  2. 递归定义最优解的值。
  3. 计算最优解的值，通常采用自下而上的方式。
  4. 根据计算信息构建最佳解决方案。

  When developing a dynamic-programming algorithm, we follow a sequence of four steps:

  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution, typically in a bottom-up fashion.
  4. Construct an optimal solution from computed information.

动态规划
Dynamic Programming

- 我们以一个例子来看，动态规划中的子问题又包含公共子问题是什么意思。
- 假设我们在一个小岛上旅游购物，当地发行的纸币面值有1元，5元，11元。支付时，必须要求所用纸币数量最少。假设购物花了$n = 15$元，问用什么样的纸币组合才能满足支付要求。
- 根据生活中的经验，我们一般会采用一种贪心策略，即尽可能先用大面额的纸币，减少最后所用纸币的数量。但这里采用贪心策略的话，反而不能满足支付要求。比如，如果先用面值为11元的纸币，则剩余4元，需要用4张1元纸币；因此总共用了5张纸币。如果直接用5元纸币，则只需用3张即可。

动态规划
Dynamic Programming

- 假设，我们将凑出数字n所用纸币的总数记为$f(n)$，再次分析一下上述过程。如果我们先用面值为11的纸币，则接下来就面对$n = 15 - 11 = 4$的情况；如果先用面值为5的纸币，则接下来面对$n = 15 - 10 = 5$的情况。我们发现这些问题都有相同的形式：“给定n，凑出n所用的最少钞票是多少张？”
- 我们说，如果规划和预算关联，则执行规划时，必然和代价关联。那么，如果我们取了11，最后的代价（用掉的钞票总数）是多少呢？很明显，我们可以表述为：利用11来凑出15，付出的代价等于$f(4)$加上一张面值11的钞票。我们暂时不管$f(4)$如何求出来，以此类推，则可以知道：如果我们用5来凑出15，付出的代价等于$f(10)$加上一张面值5的钞票。
- 显而易见，求$n = 15$时的支付方案，就是各种方案中，代价最低的那一个。

动态规划
Dynamic Programming

- 考察上面两种规划方式：
  - 1. $f(n=15) \rightarrow f(n=4)+1$
  - 2. $f(n=15) \rightarrow f(n=10)+1$

- 假设现在发行的纸币面值还有6元的，则后面一种规划方式有可以分出下面两个子规划：
  - 2.1 $f(n=15) \rightarrow f(n=10)+1 \rightarrow f(n=5)+1+1$
  - 2.2 $f(n=15) \rightarrow f(n=10)+1 \rightarrow f(n=4)+1+1$

- 我们发现，每种规划路径都涉及递归调用，但在子问题1与2.2中，出现了公共的子问题$f(n=4)$。如果不考虑动态规划的特殊性，即子问题的$f(4)$的解不保存，则会出现重复计算的问题从而降低效率。

动态规划
Dynamic Programming

- 上面的整个分析过程有点类似刻画最优解的过程，经过上面的刻画，不难知最优解有形式

$$f(n) = \min\{f(n-1), f(n-5), f(n-11)\} + 1$$

- 现在我们再来回顾一下，可以看到，$f(n)$与$f(n-1)$、$f(n-5)$、$f(n-11)$均相关，这避免了在贪心算法中，仅看$f(n-11)$的短视行为。

- 如果我们用暴力枚举方法时，对$f(n)$，需要知道一个个具体的规划，再来评价每个规划的优劣；而用动态规划时，我们不需要一下子规划到底，只需要$f(n)$和直接子问题$f(n-1)$、$f(n-5)$、$f(n-11)$的关系即可。

动态规划
Dynamic Programming

- 对于利用动态规划算法中的最后一个步骤，即由计算信息构造出最优解，则可以通过回溯最优解的求解过程得到。
- 例如，对于上面例子，我们有如下计算过程的记录信息：

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$f(n)$	0	1	2	3	4	1	2	3	4	5	2	1	2	3	4	3

- 则当$f(15)=\min\{f(14),f(10),f(4)\}+1$，我们知道$\underset{n:f(15-n)}{\operatorname{argmin}}\{f(14),f(10),f(4)\}=5$，即先用一张面值为5的纸币；
- 当$f(10)=\min\{f(9),f(5)\}+1$，我们知道$\underset{n:f(10-n)}{\operatorname{argmin}}\{f(9),f(5)\}=5$，即再用一张面值为5的纸币；
- 当$f(5)=\min\{f(4),f(0)\}+1$，我们知道$\underset{n:f(5-n)}{\operatorname{argmin}}\{f(4),f(0)\}=5$，即最后用一张面值为5的纸币。

动态规划
Dynamic Programming

- 动态规划中有两个概念，对判别是否可以运用动态规划比较重要
- 首先是无后效性，其严格定义是如果给定某一阶段的状态，则在这一阶段以后过程的发展不受这阶段以前各段状态的影响。
- 例如，当计算$f(15)$时，需要且仅需要知道$f(14)$、 $f(10)$与$f(4)$的值即可，我们计算阶段也不用关心15是各种面值的纸币如何凑出来的。这是因为递归计算子问题的过程，并不会影响原先问题。这保证当归并子问题解决原问题时，不会因为子问题，或前阶段的处理，而影响原先问题，或后续阶段。
- 另外一个概念是最优子结构，即大问题的最优解可以由小问题的最优解推出，此性质保证求解过程以递归的形式实现。
- 因此，判断一个问题能否使用动态规划解决就看是否能将大问题拆成几个小问题，并且小问题满足无后效性、且符合最优子结构性质。

动态规划
Dynamic Programming

- 我们再举一例说明如何应用动态规划。
- 伐木公司收购原木后，需要将其截为一定的长度出售。不同长度的获利是不一样的，如下表所示：

长度	1	2	3	4	5	6	7	8	9	10
获利	1	5	8	9	10	17	17	20	24	30

- 此原木裁截问题，即问对给定长度为n的原木，怎么截获利才最大，其也是一个最优化问题。注意，当本身不用裁获利已最大时，即可以整根卖。

动态规划
Dynamic Programming

- 我们说明，对于长度为n的原木，共有2^{n-1}种截法。
  - 不截（或截为一段）共有1种方法，即C_{n-1}^{0}
  - 截为两段，即在$n-1$个位置中，任取一个位置，即C_{n-1}^{1}
  - 截为三段，即在$n-1$个位置中，任取两个位置，即C_{n-1}^{2}
  - …
  - 截为$n-1$段，即在$n-1$个位置中，任取$n-1$个位置，即C_{n-1}^{n-1}

- 因此，最后共有$C_{n-1}^{0}+C_{n-1}^{1}+\cdots+C_{n-1}^{n-2}+C_{n-1}^{n-1}=2^{n-1}$。所以，对此问题，要仔细分析，不然容易导致计算量过大。

动态规划
Dynamic Programming

- 注意到，当将长度为n的原木截为更细时，可以认为是在之前步骤上的细化。如截为3段时，可以认为是先截为2段，然后再截为3段。虽然情况更为复杂，但如此处理，为利用递归算法创造了条件，因此我们考虑如下解结构：

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1\}$$

其中，r_n表示长度为n的原木不同截法时的最大获利，p_n表示长度为n的原木整体售卖时的获利（符号p表示其不用再施加递归调用）。$r_i + r_{n-i}$表示初始地将原木解为两段，每段的潜在最大获利。

- 我们注意到，不论何种截法，总可以在上式中找到一个对应的初始开始截的阶段。也即，递归地对上式进行求解，一定会得到最后的最优解。因此通过上述方式，我们将求原问题n的解，转化为求两个规模更小的i与$n-i$问题的解。此分析过程也说明原木裁截问题存在最优子结构，并同时存在无后效性特点。

动态规划
Dynamic Programming

- 另外，注意到$r_i + r_{n-i}$的表示中，$r_i + r_{n-i}$与$r_{n-i} + r_i$的对称性，则可以形式上固定r_i不动（即不裁，根据前面符号说明，此时有$r_i \to p_i$），递归地求r_{n-i}。此时，我们进一步将解的形式写为：

$$r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\}$$

  在这个公式中，最优解由之前的包含两个相关子问题的解，变成了现在的只包含一个相关子问题的解。

- 由下面伪代码，不难求出解：CUT-ROD(p, n)

```
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```