习题课-第4周

目录

- 第一次上机题目
- 第一次书面作业
- 第3周Quiz题目
- 基础算法设计：分治、动态规划、贪心法

上机题目：多项式加法

- 输入：多项式的系数与幂次
- 输出：两个多项式相加的结果，按幂次降序输出，系数为0的项不输出

样例输入

```
2
-1 17 2 20 5 9 -7 7 10 4 22 2 -15 0 16 5 0 -1
2 19 7 7 3 17 4 4 15 10 -10 5 13 2 -7 0 8 -8
-1 17 2 23 22 2 6 8 -4 7 -18 0 1 5 21 4 0 -1
12 7 -7 5 3 17 23 4 15 10 -10 5 13 5 2 19 9 -7
```

– 主要困难：输出中包含幂次相同的项
  - 如果使用字典存储“幂次-系数”对，要判断幂次是否已经在字典中
  - 如果使用链表存储多项式数据，要首先按幂次排序后再相加。
  - 一个链表遍历结束后，将另一链表的剩余元素插入结果中时，也需要去重

上机题目：Number Sequence

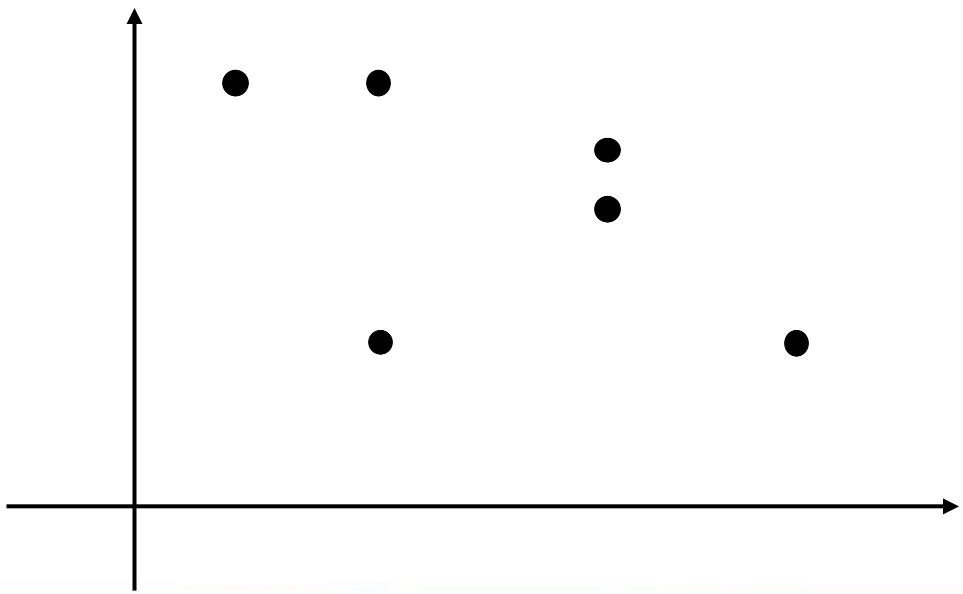
- 题目构造了这样的字符串：
  ''.join([‘1’, ‘12’, ‘123’, ‘1234’, ‘12435’, ……])
- 给定数字k，求该字符串的第k位是多少
  - 首先，确定第k位在哪一个子串中；然后返回对应下标元素即可
  - 问题在于，如何高效的找出这个子串
    - 进行数学推导：考虑数字位数的影响，计算第10，100，1000……个子串对应在原字符串中的位置，然后计算子串累计长度
    - 模拟原字符串？数据规模是2^31-1，一定会超出内存限制。
      - 2^10 ~ 10^3 ~ 1K，2^20 ~ 10^6 ~ 1M，2^30 ~ 10^9 ~ 1G
    - 只需要模拟原字符串的长度即可，内存中只需要维护一个变化的子串。累计长度+当前子串的长度 ≥ 目标长度时，模拟停止。
      ‘1’ >> ‘12’ >> ‘123’ >> ‘1234’ >> ‘12435’,

上机题目：Monkey’s Pride

- X-Y平面上的整数格点上有一些猴子，不同猴子的坐标都不同。坐标位于(x, y)的猴子为了成为国王，必须保证没有其他猴子位于(x_0, y_0)，使得$x_0 \ge x$且$y_0 \ge y$。找出所有潜在的国王。

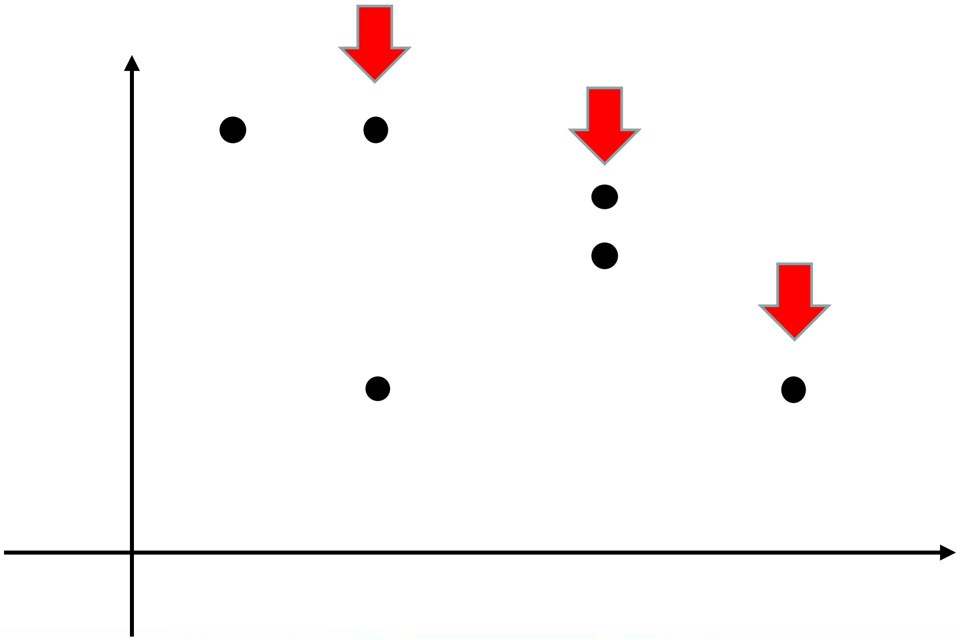
上机题目：Monkey’s Pride

- X-Y平面上的整数格点上有一些猴子，不同猴子的坐标都不同。坐标位于(x, y)的猴子为了成为国王，必须保证没有其他猴子位于(x_0, y_0)，使得$x_0 \geq x$且$y_0 \geq y$。找出所有潜在的国王。
  - 理解题目：哪些是国王？



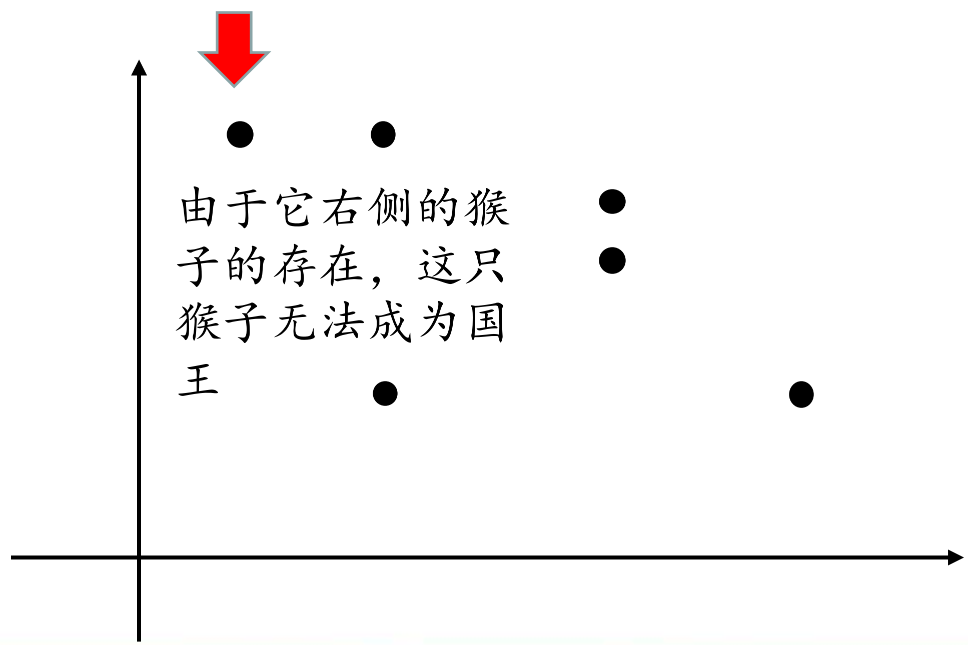
上机题目：Monkey’s Pride

- X-Y平面上的整数格点上有一些猴子，不同猴子的坐标都不同。坐标位于(x, y)的猴子为了成为国王，必须保证没有其他猴子位于(x_0, y_0)，使得$x_0 \geq x$且$y_0 \geq y$。找出所有潜在的国王。
  - 理解题目：哪些是国王？



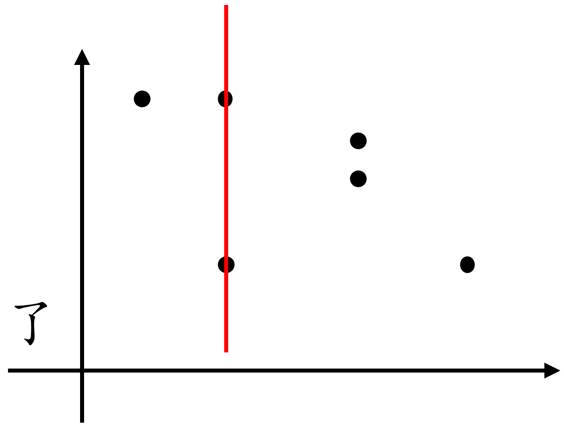
上机题目：Monkey’s Pride

- X-Y平面上的整数格点上有一些猴子，不同猴子的坐标都不同。坐标位于(x, y)的猴子为了成为国王，必须保证没有其他猴子位于(x_0, y_0)，使得$x_0 \geq x$且$y_0 \geq y$。找出所有潜在的国王。
  - 理解题目：哪些是国王？



上机题目：Monkey’s Pride

- 二维的问题解决起来比较困难
- 先在一维上来考虑：给定$x=x_0$，这条线上的猴子是否可能成为国王？
  - 只有y最大的猴子可能成为国王
  - 为了成为国王，这只猴子并不关心左侧的猴子
    - 左侧的猴子的x值更小
  - 右侧如果有猴子y值≥自身，自己就无法成为国王了
  - 只需要关注，右侧的点y值最大是多少



- 以x值为key，y值为value，使用字典存储猴子的位置
  - x值相同的点中，只需要存储y最大的
- 依照x值从大到小遍历，y值大于当前最大值的猴子即为国王

书面作业

- 顺序表与链表中增删改查操作的复杂度
  - 增删：向线性表中添加元素、删除线性表中的元素
  - 改查：修改、访问线性表中的元素
    - array[i] = value、value = array[i]
    - 顺序表中时间复杂度同为O(1)
    - 链表中时间复杂度同为O(n)

书面作业

- 关于增加、删除元素的复杂度，基本的结论是：
- 顺序表在任意位置插入或删除元素
  - 最好情况是在表尾操作，复杂度为O(1)；
  - 最坏情况是在表头操作，复杂度为O(n)；
  - 假设各位置上操作的概率均等，则平均复杂度为O(n)。
- 单链表在任意位置插入或删除元素
  - 最好情况是在表头操作，复杂度为O(1)；
  - 最坏情况是在表尾操作，复杂度为O(n)；
  - 假设各位置上操作的概率均等，则平均复杂度为O(n)。

书面作业

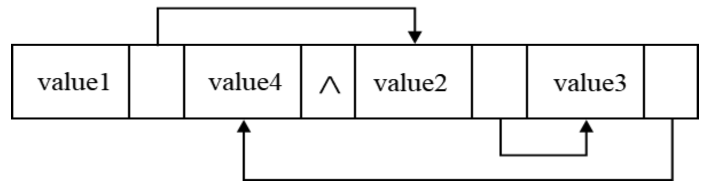
- 考虑以下场景中，增加/删除元素的时间复杂性
- 动态扩容的顺序表实现
  - 在表尾插入元素的平均复杂度为O(1)（均摊计算）
  - 在表头插入删除元素的复杂度为O(n)
  - 在任意位置插入元素的平均复杂度为O(n)
- 双向链表中，删除指定元素p
  - 时间复杂度为O(1)，只需要调整前后结点的指针即可
  - 省去了由表头开始的定位过程
  - 比较：链表中任意位置上的元素删除平均复杂度为O(n)
- 在单链表中，删除指定元素p
  - 并非O(1)，删除操作需要定位其前驱结点

书面作业

- 在单链表中，删除指定元素p
  - 并非O(1)，删除操作需要定位其前驱结点
- 通常，在使用链表的场景中，增加/删除的位置都是确定了的，不需要重新定位所需的结点。因此，链表适合用于插入删除频繁的场景。
  - 利用循环链表来解决约瑟夫问题
- 判断题第8题：如果变量 p 指向双向链表中的一个结点，那么删除这个结点 p 只需要 O(1)的时间复杂度。（正确）

书面作业

- 判断题第10题：尽管各节点在内存上连续，但该数据结构是链式存储的。（正确）



- 回忆：逻辑结构与存储结构
  - 逻辑结构（线性结构）：具有唯一的首元素、尾元素；除首尾两个元素外，每个元素具有唯一的前驱与后继。
  - 使用不同存储结构，可以实现相同的逻辑结构
    - 所谓“实现逻辑结构”，即维护元素间的前驱后继关系
    - 顺序存储：利用元素内存地址的连续性，来维护前驱与后继关系
    - 链式存储：为表中的元素添加指针域，来维护前驱与后继关系

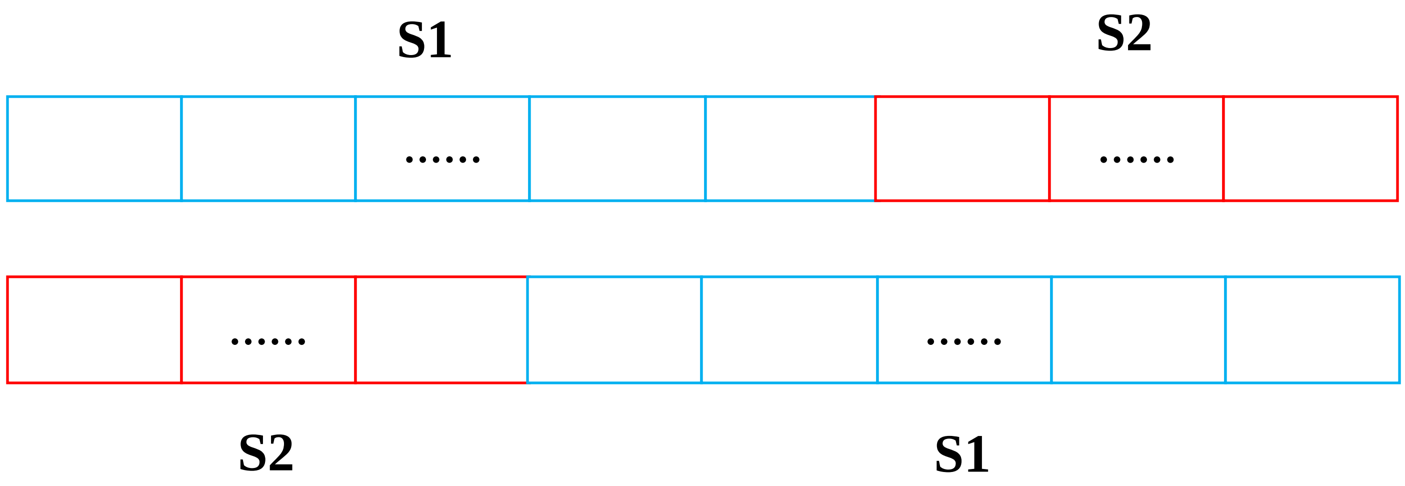
书面作业

- 关于书面作业的其他问题
  - 判断题：如果判为错误请给出解释
  - 算法设计题：简要地叙述算法设计思路（必须）；使用流程图、伪代码等工具更好地展示算法过程（可选）
  - 时间复杂度的分析：区分O(n)与O(m+n)
    - 取决于算法的输入规模需要用几个变量来描述
  - 提交格式：请检查提交后的图片是否被旋转，并尽量提交PDF文件。
  - 请手写作业内容，不要完全使用文档编辑软件编写作业内容
    - DeepSeek / CSDN 都可以帮助你完成作业
    - 确保你完全理解你提交的内容

第3周-Quiz题目

1.设S1, S2为串，请给出使S1+S2=S2+S1成立的所有可能的情况（其中 + 为连接运算）；

- 记len(S1)=m, len(S2)=n，不失一般性，假设m>n
- 容易观察到，S2是S1的公共前后缀。但这并非充分条件。



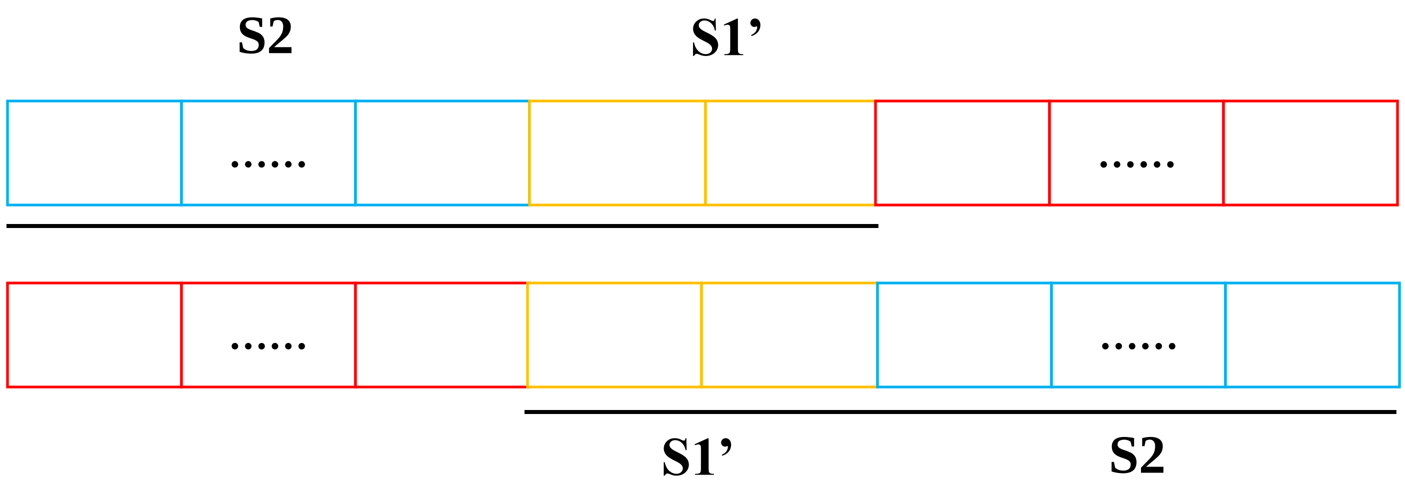
第3周-Quiz题目

1.设S1, S2为串，请给出使S1+S2=S2+S1成立的所有可能的<u>情况</u>（其中 + 为连接运算）；

– 如何考虑中间的部分S1’？

– 观察到，S1 = S1’+S2 = S2 + S1’ >> 一种递归形式

- 长度：len(S1’)=m-n，len(S2)=n



第3周-Quiz题目

1.设S1, S2为串，请给出使S1+S2=S2+S1成立的所有可能的情况（其中 + 为连接运算）；

- 如何考虑中间的部分S1’?
- 观察到，S1 = S1’+S2 = S2 + S1’ >> 一种递归形式
  - 长度：len(S1’)=m-n，len(S2)=n
- 不断进行这个过程，会以什么方式终止?
  - 如果你记得如何使用辗转相除来求最大公约数
  - 考虑这个过程中字符串的长度会如何变化
  - 最终：两个字符串相等，长度为m, n的最小公倍数
- 结论：存在一个长为d的字符串（d=gcd(m, n)），使得S1, S2都是该字符串的若干次重复。
  - 平凡情况：S1, S2的任意一个为空字符串

第3周-Quiz题目

2. 对 t=“ababbaaaaba”, p = “aab” 进行KMP 快速模式匹配，请补全模式串改进后的next数组，并画出匹配过程的图示。

下标i	0	1	2
模式串P_i	a	a	b
P[0: i]最大前后缀长度k	-1	0	1
P_k与P_i是否相等		=	≠
改进后的Next[i]	-1	-1	1

目标串T	a	b	a	b	b	a	a	a	b	a
第1次匹配	a	a								
第2次匹配			a	a						
第3次匹配					a	a	b			
第4次匹配						a	a	b		
第5次匹配							a	a	b	

第3周-Quiz题目

- 问题：必须要把无意义的Next[0]设置为-1吗？
  - 理论上不必须，修改相应的算法实现即可。
  - 在一些实现中，Next[i]的定义甚至也不同：Next[i]被定义为P[0: i+1]的最大前后缀长度，而非我们定义的P[0: i]。
  - 尽管各种定义之间是显然等价的，但从统一的角度出发，尽量使用课程中的定义方式。
  - 算法本身与定义方式无关：每一趟的匹配结果，匹配的总趟数，等等。
- 其他常见问题
  - 公共前后缀的定义：同时是**真前缀**和后缀，不包括字符串自身。
    - 字符串‘a’的最长前后缀长度为：0
    - 字符串‘aaaa’的最长前后缀长度为：3

基础算法设计

分治、动态规划、贪心法

分治

- 分治（Divide and Conquer）：将问题分解为（更易解决的）形式相同的子问题，将子问题的解合并为原问题的解。
- 熟悉的例子：汉诺塔问题
  - Divide：先移动n-1个圆盘、再移动一个圆盘、再移动n-1个圆盘
  - Conquer：递归地解决n-1规模的汉诺塔问题
- 分治与递归：分治算法通常都是递归的，应用算法本身来解决分解出的子问题。通常包括三步
  - 分解：找出与原问题关联的若干个形式相同的子问题
  - 解决：递归地解决子问题（考虑递归边界）
  - 组合：把子问题的解组合成原问题的解

动态规划

- 动态规划（Dynamic Programming）方法通常用于求解最优化问题，在众多可行解中寻找一个最优解
  - 一个最优解 (an optimal solution) 而非唯一最优解 (the optimal solution)
- 分治法通常把原问题划分成互不相交的子问题，而动态规划方法适用于子问题相互重叠的情况。
  - 通常，在原问题上任意行动一步，剩余问题就构成形式相同的子问题
  - 重叠(相同)的子问题只需要求解一次，把结果保存起来即可
- 无需采用递归式的方法，只需要识别所有涉及到的子问题，并依次求解即可。
  - 通常，原问题就是规模最大的子问题；
  - 或者，原问题的最优解依赖于所有子问题的最优解。

贪心法

- 贪心法（Greedy Method）同样通常用于求解最优化问题。
- 同样通过解决子问题来解决原问题
  - 区别在于，动态规划方法必须考虑所有相关的子问题
  - 贪心法依据“贪心准则”，仅选择一个子问题加以解决，并通过这个子问题解决原问题。
  - 之所以称为贪心，是因为“贪心准则”每次都选择局部的最优解
- 显然，贪心法不保证总是得到最优解。但在许多问题上，合理设计的贪心法确实可以得到最优解。
  - Huffman树与Huffman编码
  - Dijkstra最短路径算法
  - 图的最小生成树算法

两个概念

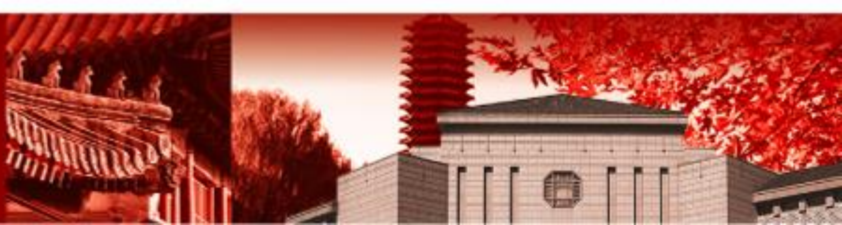
- 能否采用上述方法、应该使用何种方法？分析问题的性质：
- 性质1：最优子结构
  - 原问题的最优解是由其子问题的最优解组成的
  - 但是，可能无法直接知道是哪些子问题的解构成的
  - 为了使用动态规划、贪心法，问题需要满足这一性质
- 性质2：贪心选择性质
  - 在某种"贪心准则"下，贪心选择与被选择的子问题的最优解总是构成原问题的最优解。
  - 利用贪心准则，可以直接知道哪些子问题的解构成了原问题的解
  - 为了使用贪心法，问题还需要满足贪心选择性质
  - 尽管贪心法通常十分高效，但是并不容易说明贪心选择性质

找零问题

- 假设你为自动售货机编写找零程序，对于任意找零金额，程序应该总是使用尽可能少的纸币。
  - 可用的纸币面额为：1元、5元、10元、20元、50元、100元
  - 假设可用的纸币是无限的，只考虑整数的找零金额
- 例如，为了找零36元
  - 使用20元、10元、5元、1元各一张
  - 该问题的最优解为4张
- 应该选择何种方法解决该问题？

找零问题：最优子结构

- 考虑：问题是否具有最优子结构？能否通过解决子问题来解决原问题？
  - 原问题：对于任意金额 amount，求最优解需要的纸币数量 S(amount)
  - 任意行动一步：选择一个面值为 C_i 的纸币
  - 剩余的子问题：求 S(amount - C_i)
- 是否满足最优子结构？
  - 最优子结构：原问题的最优解包含子问题的最优解
  - 显然满足：在最优的解决方案中，除去C_i，剩下的一定是对 amount – C_i 问题的最优解。



找零问题：动态规划

- 考虑：原问题与哪些子问题有关
  - 先使用一个1元纸币，再解决 S(amount - 1)
  - 先使用一个5元纸币，再解决 S(amount - 5)
  - ……
- 并不确定先使用哪一种能够导致最优解
  - 但6种选择中至少有1种能够导致最优解
  - 原问题仅依赖于，S(amount - 1)， S(amount - 5)， S(amount - 10)， S(amount - 20)， S(amount - 50)， S(amount - 100)

$$S(amount) = 1 + \min\{S(amount - C_i)\}, \forall i$$

  - 对于x<0，定义S(x)为正无穷即可，代表错误的找零方案

找零问题：动态规划

- $S(amount) = 1 + \min\{S(amount - C_i)\}, \forall i$
- 想象一下，如果使用递归式解法：
  - 要求解 S(1000)，必须求解S(999), S(995), S(990), S(980), S(950), S(900)
  - 进而要求解S(999)，则必须求解S(998), S(994), S(989), ……
  - 递归出口在x=0或x<0
  - 大量的S(x) 会被重复计算，极度冗余
- 解决方法：记住已经求出的子问题的答案即可。
  - 设置列表dp，用dp[i]记录S(i)的答案
  - 递归求解时，首先查看这个“缓存”，只有在未求解过时才递归求解
  - 这种递归方法称为，带备忘录的 (memorized) 自上而下求解方法

找零问题：动态规划

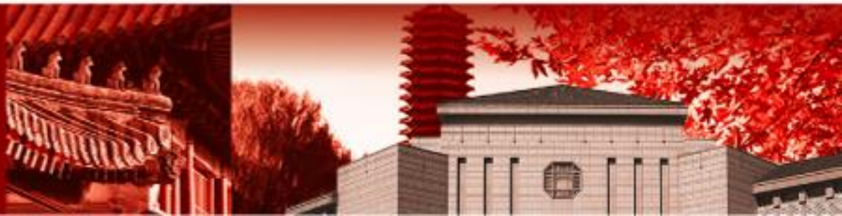
- $S(amount) = 1 + \min\{S(amount - C_i)\}, \forall i$
- 另一个视角：不需要递归式地求解，直接解决所有可能涉及到的子问题即可。
  - 对于S(amount)，直接求解所有的S(x)，$0 \leq x \leq amount$
- 分析子问题之间的依赖关系
  - S(amount) 仅依赖于 S(amount - 1), S(amount - 5), ……, S(amount - 100)
  - 顺序遍历所有的x，求解S(x)时能够保证所有依赖的子问题都已经求解
  - 顺序遍历并不总是保证这一点
    - 在这个例子里，动态规划是1维的，因为子问题仅由1个变量确定
    - 在多维动态规划里，需要考虑不同维度之间的相互依赖关系。
    - 大多数情况下可以采取顺序遍历，但也可能需要采取逆序遍历某一维度

找零问题：动态规划

- $S(amount) = 1 + \min\{S(amount - C_i)\}$，$\forall$ i 称为动态规划的状态转移方程，描述了子问题之间的依赖关系
- 实际上，可用的面值可以是任意数。但如果不包含最小单位 (1元)，算法应该识别可能存在的无法找零的情况。
- 时间复杂度为O(mn)，m为coins长度，n为找零金额

```
def change_coin(coins: list, amount: int):
    infinity = float('inf')
    dp = [infinity for i in range(amount + 1)]

    dp[0] = 0
    for i in range(1, amount + 1):
        for j in coins:
            if j <= i:
                dp[i] = min(dp[i], 1 + dp[i - j])
    if dp[amount] == infinity:
        return -1
    return dp[amount]
```

找零问题：贪心法

- 回到原问题：假设你为自动售货机编写找零程序，对于任意找零金额，程序应该总是使用尽可能少的纸币。
  - 可用的纸币面额为：1元、5元、10元、20元、50元、100元
  - 假设可用的纸币是无限的，只考虑整数的找零金额
- 就这个问题本身而言，人们其实通常会这么做
  - 首先使用尽可能多的100元面值的纸币
  - 然后使用尽可能多的50元面值的纸币
  - ……

找零问题：贪心法

- 很容易实现这一算法
  - 时间复杂度仅为O(n/k)，n为找零金额，k为最大可用的面值
    - 现实中k为固定值，复杂度可以认为是O(n)的
  - 与O(mn)的动态规划算法相比，十分高效

```
def coin_change(coins: list, amount: int):
    coins = sorted(coins, reverse=True)    # 降序排列
    total_amount = 0
    greedy_coin_index = 0
    while amount > 0:
        if coins[greedy_coin_index] <= amount:
            amount -= coins[greedy_coin_index]
            total_amount += 1
        elif greedy_coin_index + 1 < len(coins):
            greedy_coin_index += 1
        else:
            return -1
    return total_amount
```

找零问题：贪心法

- 相比于动态规划算法试图解决所有子问题，贪心算法只选择一个子问题：S(amount - C_{max}) 来解决。
- 但是，如何来说明贪心法的正确性？
  - 需要说明，在前述的贪心准则下，问题满足贪心选择性质
  - 贪心选择性质：贪心选择与被选择的子问题的最优解总是构成原问题的最优解。
- 实际上，找零问题是否满足这一性质，与具体可用的纸币面值有关。
  - 想象这样的纸币体系：1元，20元，50元，100元。如何找零60元？
  - 最优解为，20元 * 3张
  - 贪心解为，50元 * 1张，1元 *10张

找零问题：贪心法

- 贪心选择性质的证明
  - 说明给定纸币组合（1元，5元，10元，20元，50元，100元）下的找零问题满足这一性质
- 关键：说明任何找零金额的最优解都满足如下性质
  - 在任何最优解中，对于任意一个面值C_i，面值小于C_i的纸币覆盖的面值之和，严格小于C_i

找零问题：贪心法

- 贪心选择性质的证明
  - 说明给定纸币组合（1元，5元，10元，20元，50元，100元）下的找零问题满足这一性质
- 关键：说明任何找零金额的最优解都满足如下性质
  - 在任何最优解中，对于任意一个面值C_i，面值小于C_i的纸币覆盖的面值之和，严格小于C_i
  - 举例来理解这一点：找零82元问题的最优解为50+20+10+1+1
    - 若$C_i=50$，面值小于50的纸币覆盖的面值之和为 $20+10+1+1=32 < 50$
    - 若$C_i=20$，面值小于20的纸币覆盖的面值之和为 $10+1+1=12 < 20$
  - 如果这一点成立，我们就知道，在任何时刻，当前可用的最大面值的纸币一定包含在当前问题的全局最优解中
    - 因为较小金额的纸币无法覆盖这一金额

找零问题：贪心法

- 贪心选择性质的证明
  - 说明给定纸币组合（1元，5元，10元，20元，50元，100元）下的找零问题满足这一性质
- 关键：说明任何找零金额的最优解都满足如下性质
  - 在任何最优解中，对于任意一个面值C_i，面值小于C_i的纸币覆盖的面值之和，严格小于C_i
- 考虑：C_i=5
  - 在最优解中，1元纸币最多存在4张，总面值小于5元
  - 否则可以用1张5元纸币来替代，违背了解的最优性

找零问题：贪心法

- 贪心选择性质的证明
  - 说明给定纸币组合（1元，5元，10元，20元，50元，100元）下的找零问题满足这一性质
- 关键：说明任何找零金额的最优解都满足如下性质
  - 在任何最优解中，对于任意一个面值C_i，面值小于C_i的纸币覆盖的面值之和，严格小于C_i
- 考虑：$C_i=10$
  - 在最优解中，1元纸币最多存在4张，总面值小于5元
  - 在最优解中，5元纸币最多存在1张，否则可以用10元纸币替代
  - 二者总面值小于10元
- $C_i=20$也是类似的

找零问题：贪心法

- 贪心选择性质的证明
  - 说明给定纸币组合（1元，5元，10元，20元，50元，100元）下的找零问题满足这一性质
- 关键：说明任何找零金额的最优解都满足如下性质
  - 在任何最优解中，对于任意一个面值C_i，面值小于C_i的纸币覆盖的面值之和，严格小于C_i
- C_i=50则复杂一些：
  - 已知，1, 5, 10 三种纸币覆盖金额小于20元。如果加上20元纸币后金额达到了50元，那么有多少张20元？
    - 仅有2张20元：为了达到50元至少还需要一张10元，20+20+10=50
    - 至少3张20元：20+20+20=50+10
    - 总是可以用更少数量的纸币来实现，与最优解矛盾