计算机算法设计与分析-作业2(DP)

- Author: hrwhipser
- https://github.com/hrwhisper/algorithm_course/

说明

- 采用python 3.5.2编写了所有的代码
- 在作业提交期限截止后，所有的代码可以在如下网址找到:
  - https://github.com/hrwhisper/algorithm_course

1. Largest Divisible Subset

> Given a set of distinct positive integers,find the largest subset such that every pair (S_i , S_j) of elements in this subset satisfies: S_i % S_j = 0 or S_j % S_i = 0.

问题分析（最优子结构及**DP**表达式）

该题目给定了一个正整数的集合，要求求最大的子集使得在子集中任意的元素均有 S_i% S_j = 0 或 S_j%S_i = 0.

若我们先对原集合排序，那么对于一个元素X，有X % Y == 0 （Y为它之前的元素），那么有X % Z == 0 （Z为Y之前的元素且有Y % Z == 0）。因此，该问题具有最优子结构。

设dp[i]为到达i为止的最大的可整除集合大小，我们可以写出dp表达式如下（有些类似于LIS算法）：

dp[i] = max(dp[j] + 1) ，j < i \ \&\&\ num[i] \% num[j] == 0

最后，要求出最大的子集合，只需要对dp数组进行回溯查找即可。当然也可以用另个一个数组记录更新的下标，可以更快的进行回溯。

代码

```python
def largest_divisible_subset(nums):
    if len(nums) <= 1: return nums
    nums.sort()
    n = len(nums)
    dp = [1] * n
    update_from = [-1] * n
    max_len, max_index = 1, 0
    for i in range(1, n):
        for j in range(i - 1, -1, -1):
            if nums[i] % nums[j] == 0 and dp[j] + 1 > dp[i]:
                dp[i] = dp[j] + 1
                update_from[i] = j

        if dp[i] > max_len:
            max_len = dp[i]
            max_index = i

    ans = []
    while max_index != -1:
        ans.append(nums[max_index])
        max_index = update_from[max_index]
    return ans
```

正确性证明

在问题分析中，已经给出，若对集合排序，则有如下成立：
$$
X \% Y = 0\ \&\& \ Y \% Z = 0 => X \% Z = 0 ， 其中Y < X ， Z < Y
$$

由于Z能被Y整除，说明Y有Z这个因子，而Y能被X整除，说明X有Y这个因子，而Y有Z这个因子，所以Z能被X整除。

由于上述的最优子结构正确，因此我们的递推表达式通过枚举小于X的所有元素进行更新也同样正确。

时间复杂度分析

对于dp的过程，最坏情况下为$O(n^2)$，而回溯过程为$O(n)$，因此总复杂度为$O(n^2)$

2. Money robbing

> A robber is planning to rob houses along astreet. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
>
> 1. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.
> 2. What if all houses are arranged in a circle?

问题分析（最优子结构及DP表达式）

我们设dp[i]表示到第i个房子能抢到的最大值。

对于一个房子，若选择抢, 则上一个房子不能抢，若选择不抢，则保留为到i-1个房子的最大值。

因此有如下递推表达式：

$$
dp[i] = max(dp[i-1],dp[i-2] + nums[i])
$$

若为圆形，说明第一个房子和最后的房子不能同时抢，此时可以设两个dp，dp1为为抢了第一个房子，dp2为不抢第一个房子，然后按照上面的式子更新, 最后结果为max(dp1[n - 2], dp2[n - 1])。或者是，分别计算 [0,n-2]和[1,n-1] 能抢到的最大值，然后取max.

代码

- 所有的房子为直线

```python
def rob_no_circle(nums):
    if not nums: return 0
    n = len(nums)
    if n <= 2: return max(nums)
    dp = [0] * n
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])
    for i in range(2, n):
        dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])
    return dp[n - 1]
```

- 房子为环 - 方法1 双dp

```python
def rob_circle(nums):
    if not nums: return 0
    n = len(nums)
    if n <= 2: return max(nums)
    dp1 = [0] * n
    dp2 = [0] * n
    dp1[0] = dp1[1] = nums[0]
    dp2[1] = nums[1]
    for i in range(2, n):
        dp1[i] = max(dp1[i - 1], dp1[i - 2] + nums[i])
        dp2[i] = max(dp2[i - 1], dp2[i - 2] + nums[i])
    return max(dp1[n - 2], dp2[n - 1])
```

- 房子为环 - 方法2 直接调用为直线情况的code

```python
def rob_circle2(nums):
    if not nums: return 0
    if len(nums) <= 2: return max(nums)
    return max(rob_no_circle(nums[:-1]), rob_no_circle(nums[1:]))
```

正确性证明

该题正确性的证明在于对状态转移方程的正确性证明。

对于i不抢，显然只能为dp[i-2]，而抢显然为dp[i-2] + nums[i]，因此我们只需要取其最大值即可。

环形情况同理。

时间复杂度分析

- 对于直线的情况，由于只遍历了一次数组，因此复杂度为O(n)
- 对于环形的情况：
  - 方法1也只遍历了一次数组，复杂度为O(n)
  - 方法2遍历了两次，复杂度也为O(n)

3. Partition

> Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s.
> For example, given s = “aab", return 1 since the palindrome partitioning ["aa", "b"] could be produced using 1 cut.

问题分析（最优子结构及DP表达式）

该问题要求求解最小的划分数使得给定的字符串s每个子串均为回文串。

于是定义dp[i]为s[0....i]的最小切分次数，使得s[0.....i]的每个子串均为回文串。

于是有：
$$
s[i] = min(s[j-1] + 1)　其中 j < i 且s[j.....i]是回文串
$$

上述的可以简单的写出python代码如下：

```
def partition(s):
    n = len(s)
    dp = [0] + [0x7fffffff] * n
    for i in range(1, n):
        for j in range(i + 1):
            if is_palindrome(s[j:i+1]): # to check whether s[j....i] is palindrome or not.
                if j > 0:
                    dp[i] = min(dp[j - 1] + 1, dp[i])
                else:
                    dp[i] = 0
    return dp[n - 1] # dp[n-1] is the answer

def is_palindrome(s):
    return s==s[::-1]
```

在上述的解法中，我们枚举i为当前计算的位置，然后枚举j，看看i加入后，是否能和前面的构成了一个回文串，最后在查看j...i是否为回文串。 这样复杂度为O(n^3) （枚举i O(n) 枚举起始位置j O(n) 判断回文O(n)，所以为O(n^3)）

但是，我们还可以做得更好，我们枚举k为当前计算的位置，然后用双指针的思想，从k向两边扩散，判断是否回文（要分别计算长度为奇数和偶数的情况），并根据上述公式更新dp数组。这样，就可以将第一种解法的枚举j和判断回文合并起来，从而把复杂度降低为$O(n^2)$

代码

```
def partition(s):
    def helper(i, j):
        while j >= 0 and i < n:
            if s[i] != s[j]:
                break
            dp[i] = min(dp[i], dp[j - 1] + 1 if j > 0 else 0)
            i, j = i + 1, j - 1

    n = len(s)
    dp = [0] + [0x7fffffff] * n
    for k in range(1, n):
        helper(k, k)  # odd case
        helper(k, k - 1)  # even case

    return dp[n - 1]
```

正确性证明

对于新加的一个字母，若能和前面的组成回文串(即s[j...i]为回文串），则可以划分的次数显然为s[j-1] + 1，我们枚举j 显然能得到最小的解。因此算法正确。

时间复杂度分析

在问题分析中已经分析出，复杂度为$O(n^2)$

4. Decoding

> A message containing letters from A-Z isbeing encoded to numbers using the following mapping:
>
> A : 1
>
> B : 2
>
> …
>
> Z : 26
>
> Given an encoded message containing digits, determine the total number of ways to decode it.
>
> For example, given encoded message “12”, it could be decoded as “AB” (1 2) or “L”(12).
>
> The number of ways decoding “12” is 2.

问题分析（最优子结构及DP表达式）

对于一个编码后的串s，s的所有的字符出现在0~9之间。

要查看其解码方式有多少种可能，主要在于因为有的字符可以被拆分，如12可以算L也可以算AB，而这样的在10~26均是可能的。

设dp[i]为s[0...i]最多的解码方式，因此我们有：
$$
dp[i] += dp[i-1] (如果 s[i]!='0')
$$
$$
dp[i] += dp[i-2] (如果 i >= 2 \&\& 10<=int(s[i-1..i]) <= 26)
$$

代码

```
def decoding_ways(s):
    if not s: return 0
    n = len(s)
    dp = [0] * n
    dp[0] = 1 if s[0] != '0' else 0
    for i in range(1, n):
        if 10 <= int(s[i - 1:i + 1]) <= 26:
            dp[i] += dp[i - 2] if i >= 2 else 1
        if s[i] != '0':
            dp[i] += dp[i - 1]
    return dp[n - 1]
```

正确性证明

对于当前位置，若该位置不是‘0’，则dp[i] += dp[i-1] (为0说明是上一个的遗留)

若能和上一个组合的数字范围在[10,26]，那么说明解码的方式可以在加上dp[i-2]

因此，算法最优子结构和递推表达式均无误。

时间复杂度分析

上述的算法只遍历了一次数组，因此复杂度为O(n)

5. Frog Jump

> A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.
>
> If the frog’s last jump was k units, then its next jump must be either k −1,k, or k +1 units. Note that the frog can only jump in the forward direction.
>
> Given a list of stones’ positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

问题分析（最优子结构及DP表达式）

青蛙过河，上一次跳k长度，下一次只能跳k-1,k或者k+1。

因此对于到达了某一个点，我们可以查看其上一次是从哪个点跳过来的。

设dp[j][i] 从i可以到达j，因此，对于点 j，我们只需要查看可以从哪个地方跳转过来（这里假设为i），然后查看其跳跃的距离$step = stones[j] - stones[i]$, 则下一次的跳的距离为$step + 1, step, step - 1$ ，然后查看下一个点_id存不存在（用Hash），存在将dp[_id][j] 设置为可达 ,若$_id==n-1$，说明到达了对岸。这样复杂度为$O(n^2)$

代码

在具体的实现上，使用了类似邻接表的方式来加快速度。

```
def can_cross(stones):
    n = len(stones)
    val2id = {stone: i for i, stone in enumerate(stones)}
    dp = collections.defaultdict(lambda :collections.defaultdict(int))
    dp[1][0] = True
    for j in range(1, n):
        for i in dp[j]: # the same as dp[j].keys()
            step = stones[j] - stones[i]
            for k in [step + 1, step, step - 1]:
                _next = stones[j] + k
                if _next in val2id:
                    _id = val2id[_next]
                    if _id == n - 1:
                        return True
                    if _id != j:
                        dp[_id][j] = True
    return False
```

正确性证明

上述的算法，用可达矩阵dp[j][i]来标记从i可以到达j，对于任意的点j，我们均查看其可达矩阵中所有的能到达j的点，并计算上一次的步长 step ,然后枚举走 $step + 1, step, step - 1$ 在数组中的位置，并标记对应的可达矩阵。因此这样做不会丢失解。

时间复杂度分析

由于最坏的情况下，每个点距离为1，均为可达的，每次均要从可达矩阵中扫描所有0...j的点，因此复杂度为$O(n^2)$

6. Maximum profit of transactions

> You have an array for which the *i*-th element is the priceof a given stock on day *i*.
>
> Design an algorithm and implement it to find the maximum profit. You may complete at most two transactions.
>
> Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock beforeyou buy again).

问题分析

该问题要求最多两次交易下能取得的最大值，我们可以先计算一次交易下能取得的最大值。

设dp[i]为第i天能取得的最大利润，我们维护一个到[0...i-1]的最小值min_price，因此有

$$
dp[i] = max(dp[i - 1], prices[i] - min\_price)
$$

这样，我们就求出了一次交易下能获取的最大值。

要求两次交易的最大值，我们可以逆序扫描数组，维护一个[i+1....n-1]的最大值max_price，并且维护一个到当前位置的最大的利润max_profit，则有：

$$
ans = max(ans, max\_profit + dp[i - 1])
$$

其中，$max\_profit = max(max\_profit, max\_price - prices[i])$

代码

```python
def max_profit(prices):
    if not prices or len(prices) < 2: return 0
    n = len(prices)
    min_price = prices[0]
    dp = [0] * n
    for i in range(1, n):
        dp[i] = max(dp[i - 1], prices[i] - min_price)
        min_price = min(prices[i], min_price)

    max_price = prices[n - 1]
    ans = dp[n - 1]
    max_profit = 0
    for i in range(n - 1, 0, -1):
        max_profit = max(max_profit, max_price - prices[i])
        max_price = max(max_price, prices[i])
        ans = max(ans, max_profit + dp[i - 1])
    return ans
```

时间复杂度分析

由于进行了两次线性扫描，因此复杂度还是O(n)

7. Maximum length

> Given a sequenceof n real numbers *a*1,…,*an*, determine asubsequence (not necessarily contiguous) of maximum length in which the valuesin the subsequence form a strictly increasing sequence.

方法1 $O(n^2)$ naive method

设dp[i]为以i结尾的最长上升子序列长度，则显然有

$$
dp[i] = max(dp[j] + 1),\ \ j < i \ \&\&\ \ nums[j]\ <\ nums[i]
$$

这样复杂度为$O(n^2)$

因此写出代码如下：

```
def longest_increasing_subsequence(nums):
    if not nums: return 0
    n = len(nums)
    dp = [1] * n
    update_from = [-1] * n

    lis_len = 1
    index = 0
    for i in range(1, n):
        for j in range(i - 1, -1, -1):
            if nums[j] < nums[i] and dp[i] < dp[j] + 1:
                dp[i] = dp[j] + 1
                update_from[i] = j
        if dp[i] > lis_len:
            lis_len, index = dp[i], i

    ans = []
    while index != -1:
        ans.append(nums[index])
        index = update_from[index]
    return lis_len, ans[::-1]
```

方法2 O(nlogn) method

我们可以做得比O(n^2)更好，这里仍然设dp[i]为以i结尾的最长上升子序列长度。

假设 $dp[x] = dp[y] $, 且$ nums[x] < nums[y] $， 那么对于后续的所有状态i来说（i>x,i>y）， 显然满足$nums[y] < nums[i] $ 的必然满足$ nums[x] < nums[i]$,反之不一定成立。因此只需要保留x的值即可。

这里引进辅助数组g，g[i]为LIS长度为i的结尾最小的值。显然 $ g(1) <= g(2) <=……g(n) $

因此，可以进行二分查找。我们查找nums[i]在g中，可以插入的位置，换句话说，就是找一个最小的下标k，使得 $ nums[i] <= g[k] $ ， 此时，dp[i] = k, 并且可以更新g[k] = nums[i]

```python
def binary_search(g, x, L, R):
    while L < R:
        mid = (L + R) >> 1
        if g[mid] < x:
            L = mid + 1
        else:
            R = mid
    return L


def longest_increasing_subsequence_nlogn(nums):
    if not nums: return 0
    n = len(nums)
    dp = [1] * n
    g = [0x7fffffff] * (n + 1)
    update_from = [-1] * (n + 1)
    indexs = [-1] * (n + 1)
    lis_len = 1
    index = 0
    for i in range(n):
        k = binary_search(g, nums[i], 1, n)
        g[k] = nums[i]
        dp[i] = k
        indexs[k] = i
        update_from[i] = indexs[k - 1]
        if dp[i] > lis_len:
            lis_len, index = dp[i], i

    ans = []
    while index != -1:
        ans.append(nums[index])
        index = update_from[index]
    return lis_len, ans[::-1]
```