算法设计与分析第二次作业 - 动态规划

201628013229058 洪鑫

1 Largest Divisible Subset 1

问题描述:

> 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 $S_j\%S_i = 0$.

最优子结构

- 首先观察到，用较小数对较大数取余总为零，比如$3\%5 = 0$。这样考虑一对数是否满足条件只需要验证较大数能否被较小数整除即可。
- 给定的数组可能是乱序的，如果数组的元素按照从小到大的顺序排列有助于问题的解决。
- 考虑子结构$OPT(k)$表示到第k个数，最大可整除的子集合中元素的个数，其中序号为k的数是该子集合的最大值。
- 假如现已知$OPT(0) \sim OPT(k)$，再来一个数，则可以写出递归表达式:

$$OPT(k+1) = max \begin{cases} 1 \\ max\{OPT(i) + 1 \mid 0 \le i \le k,\ S_{k+1}\%S_i = 0\} \end{cases}$$

- 问题的解为 $max(OPT)$

伪代码

```
LargestDivisibleSubset(nums):
    n = size of nums;
    if n = 0 or 1
        return nums;
    sort nums from small to large;
    for i = 0 to n
        pre[i] = -1;    // record last divisible number's index
        opt[i] = 0;
    for i = 1 to n
        for j = 0 to i - 1
            if nums[i] % nums[j] == 0 and opt[i] < opt[j] + 1
```

```
                opt[i] = opt[j] + 1;
                pre[i] = j;
    /* backtrack to get the subset */
    m = index of opt's maximum;
    while m != -1
        add nums[m] to result;
        m = pre[m];
    return result;
```

正确性证明 （循环不变式）

- 初始化：如果集合大小为0或1，返回正确结果。第一次迭代（$i = 1$）前，循环不变式成立，即到第k个数，最大可整除的子集合中元素的个数为$opt[k]$，其中序号为k的数是该子集合的最大值。此时，$k = 0$，显然成立。
- 保持：第k次对i进行迭代，假设得到的$opt[1..k]$和$pre[1..k]$是正确的，第$k + 1$次迭代，从集合的第0个元素开始，如果遇到一个数，可以被其整除，并且与以该元素为最大值的可整除集合组合，得到的新集合长度比现有的集合长度要大，那么更新$opt[k + 1]$，并将该数的序号记录到$pre[k + 1]$里。这样遍历到序号为k元素时，得到的以序号为k的元素为最大值的可整除集合一定是最大的。保持了循环不变式的性质。
- 终止：第n次迭代后，循环结束。我们得到满足循环不变式性质的opt[1..n]和pre[1..n]。最后，根据opt和pre的定义，得到最优子集的序号，组成新的集合。因此算法正确。

算法复杂度

该算法的时间复杂度为$O(n^2)$。因为有n个子问题，每个子问题进行了i次求余和比较（其中i为子问题的序号），总时间复杂度为$O(\frac{n(n-1)}{2})$。另外，排序的时间复杂度为$O(nlog(n))$，求最大值和回溯的时间复杂度均为$O(n)$。

2 Partition 3

问题描述:

> 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.

最优子结构

- 对于字符串中任意一个回文子串s_{ij}，s_{0j}的最小分割方式为以下两种中的最优：
    - 现有s_{0j}的最小分割方式
    - 在$s_{0(i-1)}$基础上，在s_{i-1}和s_i之间进行分割。
- 用$OPT[j+1]$表示s_{0j}的最小分割数，初始化$OPT[j]=j-1$，即最差情况是分割每个字符。
- 递归表达式为：

$$OPT(j)=min\begin{cases}OPT(j)\\ min\{OPT(i)+1 \mid i\le j,\ s_{ij}\ is\ palindrome\}\end{cases}$$

- 另一个重要观察：如果s_{ij}不是回文串，那么$s_{(i-1)(j+1)}$也不是回文串。

伪代码

```
MinCut(s):
    n = length of s
    if n == 0 return 0
    for i = 0 to n
        OPT[i] = i-1;
    for i = 0 to n-1
        /* Consider odd length */
        j = 0;
        while i-j >= 0 and i+j <= n-1 and s[i-j] == s[i+j]
            OPT[i+j+1] = min(OPT[i+j+1], OPT[i-j]+1);
            j++;
        /* Consider even length */
        j = 0;
        while i-j-1 >=0 and i+j <= n-1 and s[i-j-1] ==s[i+j]
            OPT[i+j+1] = min(OPT[i+j+1], OPT[i-j-1]+1);
            j++;
    return opt(n);
```

正确性证明（循环不变式）

- 初始化：在第一次迭代前，如果s长度为0，返回0。如果长度大于0，第一次迭代前，opt[0]没有实际意义，不好证明其正确性。但我们知道第一次迭代后应该有opt[1]=0，即只有一个字符时的最小划分为0。而分析不难发现，opt[1]=min(opt[1], opt[0]+1)=0，所以第一次迭代前循环不变式成立。
- 保持：第k次迭代前，假设循环不变式成立，即opt[1..k-1]表示前$1..k-1$个字符的最小分割数。第k次循环，已经考虑了所有包含第k个字符的回文串的划分是否会减小现有的分割方式，并取最小分割数。注意，这些步骤实在前k次循环中共同完成的，不是仅在第k次循环中就全部完成。所以迭代后opt[k]的值满足循环不变式。
- 终止：第n次迭代后，循环终止，opt[n]表示前n个字符的最小分割数，即s的最小分割数。综上所述，算法正确。

复杂度

该算法的时间复杂为$O(n^2)$。一共有n个子问题，第i个子问题以第i个字符为中心，向外扩展，分别计算子串长度为奇数和偶数的情况，最多计算$(n+1-|n-2i|) \leq n+1$种可能。

该算法的空间复杂度为$O(n)$。

3 Decoding 4

问题描述:

> A message containing letters from A-Z is being encoded to numbers using the following mapping:
>
> $$A:1$$
> $$B:2$$
> $$\cdots$$
> $$Z:26$$
>
> 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.

最优子结构

- 如果已知前长度为k和$k-1$编码消息的解码可能性$POS[k]$和$POS[k-1]$,
- 对于长度为k的编码消息，如果再来一个数字，情况如下:
    - 该数字可以独立解码，也可以和前一个数字组合
    - 该数字只能独立解码
    - 该数字必须和前一个数字组合才能解码
    - 该数字既不能独立解码，与前一个数字组合也不能解码
- 写成递归表达式:

$$POS[k+1] = \begin{cases} POS[k-1] & if\ (k+1)_{th}\ \text{digit must decode with}\ k_{th}\ \text{digit} \\ POS[k] & if\ (k+1)_{th}\ \text{digit must decode independently} \\ POS[k]+POS[k-1] & if\ (k+1)_{th}\ \text{digit can decode with}\ k_{th}\ \text{digit or independently} \\ 0 & \text{message can't be decoded} \end{cases}$$

伪代码

```
DecodeWays(s):
    n = length of s;
    if n == 0 or s[0] == '0'    /* case 4 */
        return 0;
    /* possibility of front and front of front */
    pos_f = pos_fof = 1;
    for i = 1 to n-1
        tmp = pos_f;
        if s[i] == '0'
            if s[i-1] == '1' or s[i-1] == '2':
                pos_f = pos_fof;    /* case 1 */
            else
                return 0;           /* case 4 */
        else if s[i-1] == '1' or (s[i-1] == '2' and s[i] <= '6')
            pos_f += pos_fof;       /* case 3 */
        /* case 2 doesn't change pos_f */
        pos_fof = tmp
    return pos_f
```

正确性证明（循环不变式）

- 初始化：第一次迭代前。当s的长度为0时，解码可能情况数为0。当长度k为1时，如果s = ‘0’，那么无法解码，否则，不满足for循环条件，直接返回1。当长度大于1时，如果第一个字符不是0，pos_f满足定义，即单独第一个元素只能有一种解码方式。
- 保持：假设第k次迭代前pos_f满足定义，即前k个字符的解码方式有pos_f种，前$k-1$个字符的解码方式有pos_fof种。那么对应四种情况，根据最优子结构的分析，易知该次迭代后，如果可以解码，则pos_f中存储了前$k+1$个字符的解码情况数，pos_fof存储了前k个字符的解码方式，保持了循环不变式的性质。
- 终止：如果迭代过程中，没有发现不可解码的情况，第$n-1$次迭代后，循环终止。pos_f中存储了前n个字符的解码情况数，即s的解码情况数。不可解码时，算法返回0。综上所述，该算法最终得到正确结果，算法正确。

时间复杂度

一共$n-1$个子问题，每个子问题只存在一种情况，所以该算法的复杂度为$O(n)$。

空间复杂度为$O(1)$。

4 Maximum length

问题描述：

> Given a sequence of n real numbers $a_1, \ldots, a_n$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly increasing sequence.

基本思路

该题的思路和第一题基本一致。*opt_n*[*i*]表示以*nums*[*i*]为最大值的递增子序列。每一次迭代，从左向右扫描，如果遇到与之组合，能得到比目前更长的递增子序列，则更新*opt_n*[*i*]。

最后输出*opt_n*这个数组里的最大值即为题目的解。

算法实现（*Python*）：

```python
def max_length(nums):
    n = len(nums)
    if n == 0 or n == 1:
        return n

    opt_n = [1] * n     # opt_n[i]: max length of subset, nums[i] as maximum
    pre = [-1] * n      # record the index of previous number

    for i in range(1, n):
        for j in range(0, i):
            if nums[i] > nums[j]:
                opt_n[i] = max(opt_n[i], opt_n[j] + 1)
                pre[i] = j

    # backtrack to get the max length subset
    subset = []
    max_i = opt_n.index(max(opt_n))
    while max_i != -1:
        subset.append(nums[max_i])
        max_i = pre[max_i]
    subset.reverse()

    return subset

# Test
x = max_length([1, 2, 7, 5, 6])
print x
```