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 $S_j\%S_i = 0$.

(a). 首先将集合 S 中的数字以数组形式保存，并排序。定义数组$dp[0...n-1]$，其中$dp[i]$表示前（i+1）个数构成的能互相整除的子集合大小；数组$preidx[0...n-1]$，其中$preidx[i]$表示上一个能整除$s[i]$的数字的索引。递归表达式如下：

$$dp[i] = \begin{cases} 0 \quad if\ i\ == \ 0 \\ \max\limits_{0\leqslant j<i} \{dp[j]\ +\ 1, s[i]\%s[j] == 0\}\ otherwise \end{cases}$$

(b). 伪代码如下：

```
lagestDivSubsets(s):     //s[0,1,...,n-1]表示集合中的数字
    sort(s) //对s进行排序(升序排序)
    new array dp[0...n-1]
    new array preidx[0...n-1],initize this array with -1
    dp[0] = 0
    maxsize, maxidx = 0, -1
    for i = 1 to n-1 do
        for j = 0 to i-1 do
            if s[i] % s[j] == 0 and dp[i] < dp[j] + 1 then
                dp[i] = dp[j] + 1
                preidx[i] = j
            end if
        end for
        if dp[i] > maxsize then
            maxsize = dp[i]
            preidx = i
        end if
    end for
    new set subsets //保存互相能整除的整数集合
    while maxidx != -1:
        subsets.add(s[maxidx])
        maxidx = preidx[maxidx]
    return maxsize
```

(c). 正确性证明：

(d). 时间复杂度：$O(n^2)$；空间复杂度：$O(n)$

2 Money robbing

A robber is planning to rob houses along a street. 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?

1、

(a). $m[1\cdots n]$表示这条街的 n 个屋子，每个屋子的财务，其中$m[i]$表示第 i 个屋子中的财物数目；

$yes[1\cdots n]$，其中$yes[i]$用来记录抢劫第 i 个屋子，从前 i 个屋子获得的最大收益；

$no[1\cdots n]$，其中$no[i]$用来记录不抢劫第 i 个屋子，从前 i 个屋子获得的最大收益。

所以，得到递归表达式：

$$yes[i] = no[i-1] + m[i]$$
$$no[i] = \max\{no[i-1], yes[i-1]\}$$

最终结果为：$\max\{no[n], yes[n]\}$。观察递归表达式发现，$yes[i]$和$no[i]$的更新都只依赖于$yes[i-1]$和$no[i-1]$，所以我们只需要记录下$yes[i-1]$和$no[i-1]$，而无需使用一个数组进行记录。对变量进行重新定义，yes 表示对于前 i 个屋子，抢劫第 i 个屋子可以获得的最大收益；no 表示对于前 i 个屋子，不抢劫第 i 个屋子可以获得的最大收益；同时两个变量的更新规则如下：

$$yes_{new} = no_{old} + m[i]$$
$$no_{new} = \max\{no_{old}, yes_{old}\}$$

(b).

```
rob(m): //m[0,1,...,n-1]表示各间屋子的财物数目
    yes = 0;
    no = 0;
    for i = 0 to n-1 do
        tmp = yes;
        yes = no + m[i];
        no = max{no, tmp};
    end for
    return max{yes,no};
```

(c). 考虑前 n 间屋子，yes_{old}表示抢劫第 n-1 间屋子可以从前 n-1 间屋子获得的最大收益，no_{old}表示不抢劫第 n-1 间屋子可以从前 n-1 间屋子获得的最大收益。如果抢劫第 n 间

屋子，那么最大收益一定是$no_{old} + m[n]$，因为此时第 n-1 间屋子不能被抢劫，如果能获得更大收益，那么no_{old}一定有更大值；如果不抢劫第 n 间屋子，那么最大收益与从前 n-1 间屋子获得的最大收益相同，所以$no_{new} = \max\{yes_{old}, no_{old}\}$。

(d). 时间复杂度：$O(n)$，n是房屋的数目；空间复杂度：$O(1)$。

2、对于房屋环绕的情况，我们可以分两种情况考虑。抢劫最后一间房子和不抢劫最后一间房子。不抢劫最后一间屋子的收益$nolast = rob(m[0{:}n-2])$；抢劫最后一间屋子的收益$last = rob(m[1{:}n-1])$。最终最大收益为$\max\{nolast, last\}$。

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.

(a). 定义数组$dp[0\cdots n-1, 0\cdots n-1]$，其中$dp[i][j]$表示对于子字符串$s[i\cdots j]$，$i \leqslant j$需要划分的次数。

$$dp[i][j] = \begin{cases} 0 & if\ i == j \\ 0 & if\ s[i\cdots j]\text{是回文字符串} \\ \min\limits_{i \leqslant k < j} \{dp[i][k] + dp[k+1][j] + 1\} & otherwise \end{cases}$$

(b).

```
partition(s):   //s[0,1,...,n-1]表示字符串s
    new array dp[0...n-1,0...n-1]
    for i = 0 to n-1 do
        dp[i][i] = 0
    end for
    for len_ = 1 to n-1 do
        for i = 0 to n - len_ do
            j = i + len_ - 1
            if s[i...j] is  Palindrome string then
                dp[i][j] = 0
                continue
            end if
            for k = i to j - 1 do
                dp[i][j] = min{dp[i][j], dp[i][k]+dp[k+1][j]+1}
            end for
        end for
    end for
    return dp[0][n-1]
```

(c). 对于长度为 1 的字符串和回文字符串，显然划分次数为 0；对于其他字符串$s[i\cdots j], i < j$，已知其所有前缀字符串$s[i\cdots k]$和后缀字符串$s[k+1\cdots j]$的最小划分次数分别为$dp[i][k]$和$dp[k+1][j]$，那么如果$s[i\cdots j], i < j$有更小的划分方式$dp[i][j]$，那么它的前缀或后缀也一定有更小的划分方式。

(d). 时间复杂度：$O(n^3)$；空间复杂度：$O(n^2)$。

4 Decoding

A message containing letters from A-Z is being encoded to numbers using the following mapping:

$$\begin{aligned}&\mathrm{A}:1\\&\mathrm{B}:2\\&\cdots\\&\mathrm{Z}:26\end{aligned}$$

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.

(a). 定义数组$dp[0\cdots n]$，其中$dp[i]$表示字符串$s[0\cdots i-1]$的解码方法数目，递归表达式如下所示：

$$dp[i]=\begin{cases}1 & if\ i==0\\ 1 & if\ i==1\ \ and\ \ s[0]!='0'\ \ else\ \ 0\\ (dp[i-1]\ \ if\ \ s[i-1]!='0'\ \ else\ \ 0)+(dp[i-2]\ \ if\ \ '10'\leqslant s[i-2,i-1]\leqslant '26'\ \ else\ \ 0)\end{cases}$$

(b). 伪代码如下：

```
decodeWays(encodeStr):   //encodeStr[0,1,...,n-1]表示编码字符串
    new array dp[0...n] //dp[i]表示字符串encodeStr[0...i-1]的解码方式
    dp[0]=1 //dp[0]表示空字符串的解码方式数目
    dp[1]=1 if s[0] != '0' else 0   //dp[1]表示长度为1的字符串的解码方式数目
    for i = 2 to n do
        if encodeStr[i-1] != '0' then
            dp[i] += dp[i-1]
        endif
        if '10'<=encodeStr[i-2,i-1]<='26' then
            dp[i] += dp[i-2]
        endif
    endfor
    return dp[n]
```

(c). 正确性证明：

(d). 时间复杂度：$O(n)$，空间复杂度：$O(n)$

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.

(a). 定义$dp[0 \cdots n-1]$，$dp[i]$表示能从第 i 块石头跳跃的最长距离-1；

定义一个 map(映射) jumpUnits，它的 key 类型是 int，表示在第 i 块石头上；它的 value 类型是 set<int>，表示在第 i 块石头上可以跳跃的 units。

递归表达式如下：

$$dp[i] = \begin{cases} 0 \ if \ i == 0 \\ \max\limits_{0 \leqslant j < i} \{dp[j], frog\text{可以从第}j\text{块石头跳到第}i\text{块石头}\} \ \ otherwise \end{cases}$$

$$jumpUnits[i] = \begin{cases} 0 & if \ i == 0 \\ \bigcup\limits_{frog\text{可以从}j\text{跳到}i} & jumpUnits[j] \quad otherwise \end{cases}$$

(b). 伪代码如下：

```
frogJump(stones):   //stones[0...n-1]表示每块石头的坐标
    new array dp[0...n-1]   //dp[i]表示从第i块石头可以跳跃最远units-1
    new map<int,set> jumpUnits  //jumpUnits[i]表示从第i块石头可以有哪些跳法
    dp[0] = 0
    k = 0   // 最初在第0块石头上
    for i = 1 to n-1 do
        while dp[k] + 1 < stones[i] - stones[k] do
            k += 1
        end while
        for j = k to i-1 do
            dist = stones[i] - stones[j]
            if (dist-1) in jumpUnits[j] or dist in jumpUnits[j] or (dist+1) in jumpUnits[j+1] then
                jumpUnits[i].add(dist)
                dp[i] = max(dp[i],dist)
            end if
        end for
    end for
    return jumpUnits[n-1].size!=0
```

Python 代码如下：

```
from collections import defaultdict


class Solution(object):
    def canCross(self, stones):
        """
        :type stones: List[int]
        :rtype: bool
        """
        n = len(stones)
        dp = [0 for _ in range(n)]
        jumpUnits = defaultdict(set)
        jumpUnits[0].add(0)
        k = 0
        for i in range(1, n):
            while dp[k] + 1 < stones[i] - stones[k]:
                k += 1
            for j in range(k, i):
                dist = stones[i] - stones[j]
                if (dist - 1) in jumpUnits[j] or dist in jumpUnits[j] or (dist + 1) in jumpUnits[j]:
                    jumpUnits[i].add(dist)
                    dp[i] = max(dp[i], dist)
        return len(jumpUnits[n - 1]) != 0


if __name__ == '__main__':
    s = Solution()
    stones = [0, 1, 3, 6, 10, 13, 15, 18]
    print s.canCross(stones)
```

(c). 正确性证明：

(d). 时间复杂度：$O(n^2)$，空间复杂度：$O(n^2)$

6 Maximum profit of transactions

You have an array for which the i-th element is the price of 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 before you buy again).

定义数组$forward[0...n-1]$，其中$forward[i]$表示前(i+1)天进行交易可以获得的最大收益；定义数组$backward[0...n-1]$，其中$backwark[i]$表示在第 i 天到第 n 天进行交易可以获得的最大收益。递归表达式如下：

更新$forward$的同时，更新目前为止的最低价格$lowest$，首先更新$lowest = \min\{lowest, prices[i]\}$，

$$再更新forward[i] = \begin{cases} 0 \ \ if \ \ i == 0 \\ \max\{forward[i-1], prices[i] - lowest\} \ \ otherwise \end{cases}$$

更新$backward$的同时更新第i天后的最高价格$highest = \{highest, prices[i]\}$,

$$再更新backward[i] = \begin{cases} 0 \ \ if \ \ i == n-1 \\ \max\{backward[i+1], highest - prices[i]\} \ \ otherwise \end{cases}$$

最终，最大收益为$\max\{forward[i] + backwark[i], 0 \leqslant i \leqslant n-1\}$。

伪代码如下所示：

```
maxProfit(prices):  //prices[0,1,...,n-1]为价格数组，prices[i]表示第i天的价格
    new array forward[0...n-1]  //forward[i]表示在第1天到第i天进行交易可以获得的最大收益
    new array backward[0...n-1] //backward[i]表示在第i天到第n-1天进行交易可以获得的最大收益
    //首先求解forward
    lowest = prices[0]
    forward[0] = 0
    for i = 1 to n-1 do
        lowest = min{lowest,prices[i]}
        forward[i] = max{forward[i-1],prices[i]-lowest}
    end for
    //求解backward
    highest = prices[n-1]
    backward[n-1] = 0
    for i = n-2 to 0 do
        highest = max{highest,prices[i]}
        backward[i] = max{backward[i+1],highest-highest}
    end for
    //返回最大收益
    return max{forward[i]+backward[i] for i = 0 to n-1}
```

Python 代码如下所示：

```python
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        n = len(prices)
        if n < 1:
            return 0
        forward = [0 for _ in range(n)]
        backward = [0 for _ in range(n)]

        lowest = prices[0]
        for i in range(1, n):
            lowest = min(lowest, prices[i])
            forward[i] = max(forward[i - 1], prices[i] - lowest)
        highest = prices[n - 1]
        for i in range(n - 2, -1, -1):
            highest = max(highest, prices[i])
            backward[i] = max(backward[i + 1], highest - prices[i])

        return max([forward[i] + backward[i] for i in range(n)])


if __name__ == '__main__':
    s = Solution()
    prices = [1, 4, 5, 7, 6, 3, 2, 9]
    print s.maxProfit(prices)
```

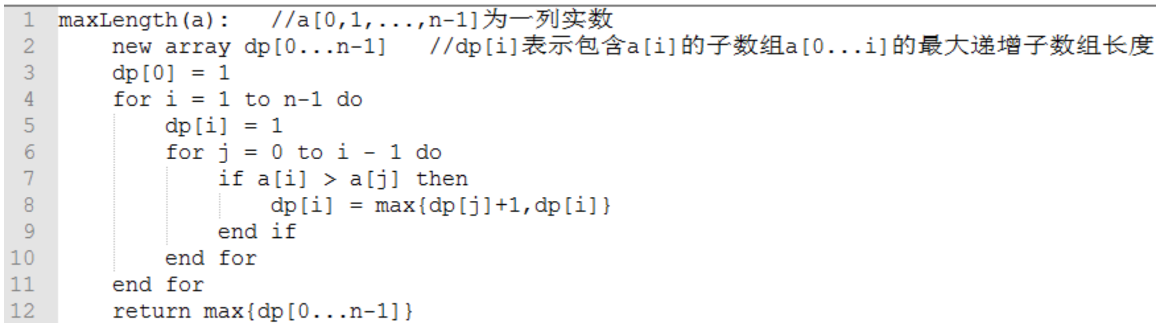
7 Maximum length

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.

递归表达式：

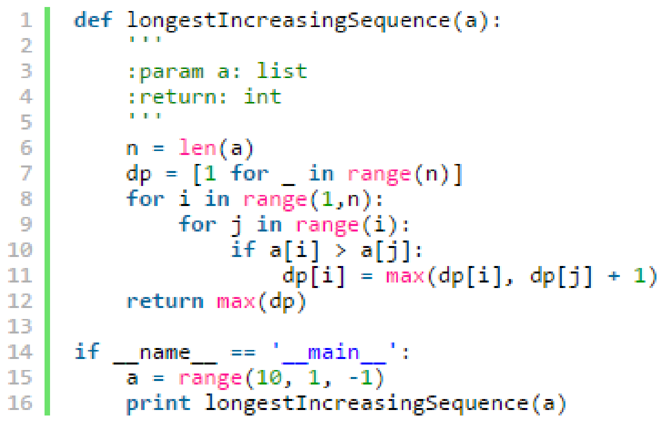
$$dp[i] = \begin{cases} 1 & if\ i == 0 \\ \max\limits_{0 \leqslant j < i} \{dp[j] + 1, a[i] > a[j]\} \end{cases}$$

伪代码如下：



```
maxLength(a):    //a[0,1,...,n-1]为一列实数
    new array dp[0...n-1]    //dp[i]表示包含a[i]的子数组a[0...i]的最大递增子数组长度
    dp[0] = 1
    for i = 1 to n-1 do
        dp[i] = 1
        for j = 0 to i - 1 do
            if a[i] > a[j] then
                dp[i] = max{dp[j]+1,dp[i]}
            end if
        end for
    end for
    return max{dp[0...n-1]}
```

Python 代码如下：



```python
def longestIncreasingSequence(a):
    '''
    :param a: list
    :return: int
    '''
    n = len(a)
    dp = [1 for _ in range(n)]
    for i in range(1,n):
        for j in range(i):
            if a[i] > a[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

if __name__ == '__main__':
    a = range(10, 1, -1)
    print longestIncreasingSequence(a)
```