T1

思路与证明：

假设街上有n栋房子，从头到尾按照1-n排序，有一个数组a[i]存储了n栋房子的存款量， DP[i]代表抢到第i栋房子时所能抢到的最多钱。对于每一栋房子而言有两个选项，抢或者不抢，如果不抢，所能抢到的最多钱不变，DP[i] = DP[i-1]，如果抢，那么第i-1栋就不能抢，抢到的最多钱为DP[i] = DP[i-2] + a[i]，因此DP[i] = max(DP[i-1]，DP[i-2] + a[i])

第二问，如果把房子安排为一个圈，那么第n栋房子和第1栋房子也不能同时抢。所以只需要计算从1-n-1栋房子和2-n栋房子分别能抢到的最多的钱，选其中较大者即可，这样保证了第1栋和第n栋至少有一栋是不抢的。体现在代码中，前者在更新完DP矩阵后，再令DP[n] = DP[n-1]，后者初始化DP[1] = 0，再从DP[2]开始更新。

伪代码：

def moneyRob(n):

if n == 1:

return 1

dp = new Array(n+1)

dp[1] = a[1]

dp[0] = 0

for i from 2 to n:

dp[i] = max(dp[i-1]，dp[i-2] + a[i])

return dp[n]

DP方程：DP[i] = max(DP[i-1]，DP[i-2] + a[i])

时间复杂度：只需要更新一维的DP矩阵，时间复杂度O(n)

T2

思路与证明：丑数是指质因子只有2,3,5的数字，暴力算法比较简单，从1开始对于每个数字进行检查是否是丑数，如果是丑数就给计数器加一，直到找到第n个。判断某个数是否是丑数，让这个数依次去除以2,3,5直到余数不为0，如果最后的余数为1则是丑数，如果是其他质数则不是。

动态规划的算法维护了三个指针p2，p3，p5，分别表示可能产生下一个丑数的位置，每次在ugly[p2]\*2，ugly[p3]\*3，ugly[p5]\*5中寻找最小值，作为新的丑数，并更新指针。证明这个算法的正确性有几点：证明ugly数组是递增的，每个元素都是丑数，并且不会有遗漏的丑数。每个元素都是丑数是显然的，因为每个元素都是由其他丑数乘2,3,5得到的；递增也是显然的，在每次找到最小值后，会把最小值对应的指针增加，下一次找到的最小值一定会增大；按照这种方法寻找也不会出现遗漏某个丑数的情况，每个丑数都会被考虑到，对于某个丑数w，拆分成w=2^x \* 3^y \* 5^z，这个数一定会在数组内出现，想要定位该数字的位置也很简单：从ugly[0]开始，找x次\*2操作，y次\*3操作，z次\*5操作，比如：对于丑数10=2^1 \* 5^1，先从ugly[0]开始，找到一次\*2操作，这次操作对应的是ugly[1]，再从ugly[1]开始，找哪一次min计算选择了ugly[1]\*5，发现是在ugly[8]的时候选择了这个数，因此ugly[8] = 10

伪代码：

Brute-force：

def isUgly(num):

if num <= 0:

return False

for p in [2, 3, 5]:

while num % p == 0:

num //= p

return num == 1

def nthUglyNumber(n):

count = 0

num = 1

while count < n:

if isUgly(num):

count += 1

num += 1

return num – 1

Dynamic：

def nthUglyNumber(n):

ugly = [0] \* n

ugly[0] = 1

p2 = p3 = p5 = 0

for i in range(1, n):

next\_ugly = min(ugly[p2] \* 2, ugly[p3] \* 3, ugly[p5] \* 5)

ugly[i] = next\_ugly

if next\_ugly == ugly[p2] \* 2:

p2 += 1

if next\_ugly == ugly[p3] \* 3:

p3 += 1

if next\_ugly == ugly[p5] \* 5:

p5 += 1

return ugly[n - 1]

DP方程：DP[i] = min(DP[p2]\*2，DP[p3]\*3，DP[p5]\*5)

时间复杂度：暴力算法需要从1开始搜索，

动态规划的算法只需要更新DP矩阵，并且每个元素的更新只需要常数次数的运算，复杂度为O(n)

T3

思路与证明：对于一个包含1-n的二叉搜索树，假设以i为根（1≤i≤n），i的左子树有i-1个元素，右子树有n-i个元素，左右子树的元素也是相邻的一串数字，因此将一个大问题分成了几个子问题。DP[i] 初始化DP[0] = 1.

伪代码：

function numUniqueBSTs(n):

if n == 0 or n == 1:

return 1

dp = new Array(n+1)

dp[0] = dp[1] = 1

for i from 2 to n:

dp[i] = 0

for j from 1 to i:

// 以j为根节点，左子树有j-1个节点，右子树有i-j个节点

dp[i] += dp[j-1] \* dp[i-j]

return dp[n]

DP方程：

时间复杂度：更新大小为O(n)的数组，但是数组中每个元素在进行更新时，都要进行O(n)次乘法运算，因此时间复杂度为O(n²)

T4

思路与证明：先分析题目，要求子集中所有的数对都能有整除关系，这个要求是比较严格的，比如3,6,9或者4,6,12这样的集合，虽然都有着不错的整除关系，但是6和9、4和6都没有整除关系，不满足。假如某个集合含有3个元素abc且满足题意，a整除b，b整除c，a整除c，其实我们发现，假如把这些元素升序排列，每一个元素都是前一个元素的x倍才可以。因此我们可以这样构造这个子集：把元素升序排列，假如我们在前面的部分元素中找到了一个子集，那么就根据目前子集中的最大值往后找，假如能够找到这个最大值的x倍，那么把这个元素加入子集。等价地，构造与集合等大的数组DP，将元素升序排列为数组a[n]，DP[i]表示以a[i]为最大值的满足条件的子集中最大子集的元素数。由于a[i]可能是很多数列的最大值（比如，12可以作为2,4的下一个，也可以作为3,6的下一个），因此更新之前要和自己比对一下，寻找最大数列。

伪代码：

def largestDivisibleSubset(nums):

if not nums or len(nums) == 0:

return 0

nums.sort()

n = len(nums)

dp = [1] \* n //初始化为1，代表只包含自己的集合

maxSubsetSize = 1

for i in range(1, n):

for j in range(i):

if nums[i] % nums[j] == 0:

dp[i] = max(dp[i], dp[j] + 1)

maxSubsetSize = max(maxSubsetSize, dp[i])

return maxSubsetSize

DP方程：if(a[i] % a[j] == 0) DP[i] = max(DP[i]，DP[j] + 1)

时间复杂度：将集合元素升序排列，需要O(nlogn)

更新DP矩阵，对于矩阵中每个元素，都要遍历一遍这个元素之前的所有元素，需要O(n²)

最后遍历一遍DP矩阵，找到最大值即为答案，需要O(n)

总共时间复杂度为O(n²)

T5

思路与证明：实际上还是一个0-1背包问题，不过这里不仅有放入背包和不放入背包的选择，还有扣除的选择，只需要在更新DP矩阵的时候多更新一个就好。DP[i][j]的值表示：在前i个数里面，能够达到j的计算方式有几种。在下面的伪代码中，默认这个target有可能是负数，因此dp矩阵的含义修改一下：dp[i][j]的值表示：在前i个数里面，能够达到j-total\_num的计算方式有几种

伪代码：

def findTargetSumWays(nums, target):

total\_sum = sum(nums)

if abs(target) > total\_sum:

return 0

n = len(nums)

dp = [[0] \* (2 \* total\_sum + 1) for \_ in range(n + 1)]

dp[0][total\_sum] = 1 // 初始化，以total\_sum为原点，左边为负数

for i in range(1, n + 1):

for j in range(nums[i - 1], 2 \* total\_sum + 1 - nums[i - 1]):

//这里j不能从0到2\*total\_sum遍历，保证j +1 nums[i-1]有意义

dp[i][j] += dp[i - 1][j - nums[i - 1]]

dp[i][j] += dp[i - 1][j + nums[i - 1]]

return dp[n][target + total\_sum]

DP方程：DP[i][j] = max(DP[i][j]，DP[i-1][j-a[i]] + DP[i-1][j+a[i]])

时间复杂度：需要更新二维矩阵，复杂度为O(n×sum)