Assignment #6: Recursion and DP

Updated 2201 GMT+8 Oct 29, 2024

2024 fall, Complied by <mark>汤伟杰，信息管理系</mark>

说明：

1）请把每个题目解题思路（可选），源码 Python, 或者 C++（已经在 Codeforces/Openjudge 上 AC），截图（包含 Accepted），填写到下面作业模版中（推荐使用 typora https://typoraio.cn ，或者用 word）。AC 或者没有 AC，都请标上每个题目大致花费时间。

3）提交时候先提交 pdf 文件，再把 md 或者 doc 文件上传到右侧“作业评论”。Canvas 需要有同学清晰头像、提交文件有 pdf、"作业评论"区有上传的 md 或者 doc 附件。

4）如果不能在截止前提交作业，请写明原因。

1. 题目

sy119: 汉诺塔

recursion, https://sunnywhy.com/sfbj/4/3/119

思路：

思路如代码中的注释。第一次看这个题是在《数据结构与算法》里面，只看代码确实很绕而且看不懂；之后查找多方解释，发现最好的理解方法就是：定义的函数只需要在心里明确这个函数是用来做什么的，而不需要考虑具体如何实现，然后不断调用这个用途的函数，最终结果会自动递归出来。

这道题由于柱子的名称给定了，为了保持这类题代码的一致性，就把 A,B,C 的名称用默认参数标记，后面直接用 format。

代码：

```
#题目确定的柱的名称，这里选择使用默认参数，后续调用的时候就只需要把 n 一个参数传入即可
#默认参数必须放在非默认的后方！！
def move(n,a='A',b='B',c='C'):
    #base case 是 n==0
    if n>=1:
        #先把 n-1 个盘子    从 a 经过 c 放到 b 上
        move(n-1,f'{a}',f'{c}',f'{b}')
        #然后把 a 的剩余的一个盘子(第 n 个盘子)放到 c 上
        move_one(a,c)
        #最后把 n-1 个盘子   从 b 经过 a 放到 c 上
        move(n-1,f'{b}',f'{a}',f'{c}')

def move_one(a,c):
    print(f'{a}->{c}')

n=int(input())
```

```
print(2**n-1)
move(n)
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

```
#题目确定的柱的名称，直接使用默认参数
def move(n,a='A',b='B',c='C'):
    #base case是n==0
    if n>=1:
        #先把n-1个盘子从a经过c放到b上
        move(n-1,f'{a}',f'{c}',f'{b}')
        #然后把a的剩余的一个盘子(第n个盘子)放到c上
        move_one(a,c)
        #最后把n-1个盘子从b经过a放到c上
        move(n-1,f'{b}',f'{a}',f'{c}')

def move_one(a,c):
    print(f'{a}->{c}')

n=int(input())
print(2**n-1)
move(n)
```

测试输入 | 提交结果 | 历史提交

完美通过 查看题解

100% 数据通过测试

运行时长: 0 ms

sy132: 全排列 I

recursion, https://sunnywhy.com/sfbj/4/3/132

思路：

（第一次看这题：？？？这题和递归有啥关系阿？？？）先看了同学代码，用 pythontutor 没看懂，自己手动模拟出错，问 gpt 也看不懂，隔了一夜找知乎，发现了一个良心文章 LeetCode 31：递归、回溯、八皇后、全排列一篇文章全讲清楚 - 知乎，之后结

合 gpt 提供的代码才算稍微有点理解。这题我自己想到的是用 permutations 函数或者用以前做过的排列的方法实现，递归方法自己真的真的想不出来。

代码：

```
##使用 permutations                    ##
##from itertools import permutations ##
##n=int(input())                     ##
##a=[i for i in range(1,n+1)]        ##
##perms=sorted(permutations(a))      ##
##for i in perms:                    ##
##    print(*i)                      ##

#使用 next_permutation 的交换元素方法
from math import factorial

n=int(input())
a=[i for i in range(1,n+1)]
def next_perm(a,n):
    k=-1
    for i in range(n-1,0,-1):
        if a[i-1]<a[i]:
            k=i-1
            break
    if k==-1:
        return a.reverse()
    l=-1
    for i in range(n-1,0,-1):
        if a[i]>a[k]:
            l=i
            break
    a[l],a[k]=a[k],a[l]
    a[k+1:]=reversed(a[k+1:])
    return a

print(*a)
for i in range(factorial(n)-1):
    print(*next_perm(a,n))

#gpt 提供的递归框架和大佬代码的重新融合
n=int(input())
chosen=[False]*(n+1) #n+1 其中第一个位置不使用，变成以 1 为 base 的索引
ans=[0]*(n+1)
def dfs(k):                  #从 k=1 开始，对第一层（第一个位置）开始递归
    if k>n:                  #这两行是退出条件
        print(' '.join(map(str,ans[1:])))
    for i in range(1,n+1):   #每个位置都遍历所有可能选择 for choice in
all_choices
        if chosen[i]:        #这两行是防止使用同一元素
            continue
        chosen[i]=True       #这两行是在 record(choice)
        ans[k]=i
        dfs(k+1)             #这一行是在递归，开始第二层（第二个位置）的遍历
all_choices
```

```
        chosen[i]=False       #这一行是在 roll_back(chioce)，条件是上面递归的 dfs 遇到了 k>n 而结束
                              #从而回溯到上一层的 k，再接着对上一层没遍历完的 i 继续遍历
dfs(1)
#这里 rollback 的逻辑是因为当前位置选定的 i 已经进行了对后面位置所有可能的数字排列，因此回溯时先把这个选定的 i 给标成 False 显示可用，再遍历到下一个 i 时在这个位置的用过的数字还可以在其他位置使用
```

代码运行截图 ==（至少包含有"Accepted"）==

```
32  n=int(input())
33  chosen=[False]*(n+1)
34  ans=[0]*(n+1)
35  def dfs(k):
36      if k>n:
37          print(' '.join(map(str,ans[1:])))
38      for i in range(1,n+1):
39          if chosen[i]:
40              continue
41          chosen[i]=True
42          ans[k]=i
43          dfs(k+1)
44          chosen[i]=False
45  dfs(1)
46
```

测试输入 | 提交结果 | 历史提交

完美通过 查看题解

100% 数据通过测试

运行时长: 0 ms

02945: 拦截导弹

dp, http://cs101.openjudge.cn/2024fallroutine/02945

思路：

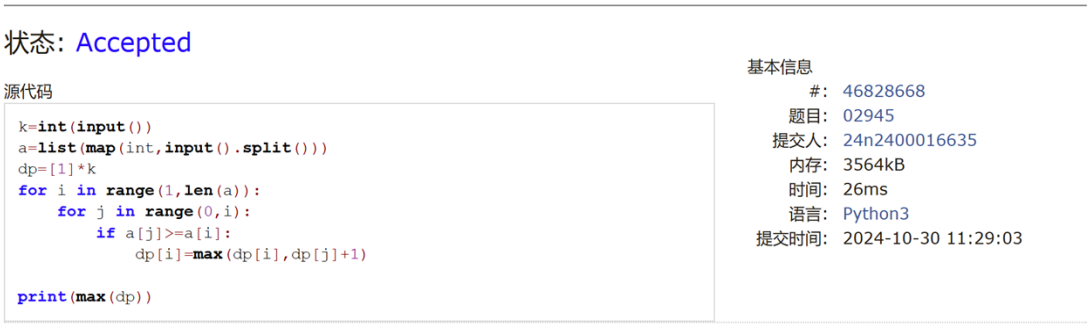
参考了最长上升子序列的思路，这个就是最长下降子序列了，关键是dp[i]=max(dp[i],dp[j]+1)。其中 j 是对=从 0 到 i-1 的遍历。初始化是 dp 的每一个值都是 1，而不是 0。

代码：

```
k=int(input())
a=list(map(int,input().split()))
dp=[1]*k
for i in range(1,len(a)):
    for j in range(0,i):
        if a[j]>=a[i]:
            dp[i]=max(dp[i],dp[j]+1)

print(max(dp))
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



23421: 小偷背包

dp, http://cs101.openjudge.cn/practice/23421

思路：

参考了算法图解的思路，先设置一个 n*b 的 dp 表格，然后使用书上的公式计算。不过麻烦的是第一行的初始化，要单独讨论 i==0 时，把第一行物品填进去，然后才能进行下面的计算。此外，还有剩余重量的正负判断，我采用了

```
dp[i][j]=max(dp[i-1][j], goods[i][0]+(dp[i-1][j-goods[i][1]] if j-
goods[i][1]>=0 else 0))
```

的方式来直接讨论，而没有额外分类。然后最后的输出的最后一个格子。

代码：

```
n,b=map(int,input().split())
v=list(map(int,input().split()))
w=list(map(int,input().split()))
goods=[]
for i in range(n):
    goods.append((v[i],w[i]))
dp=[[0]*b for _ in range(n)]
for i in range(n):
```

```
    for j in range(b):
        if i==0 and goods[0][1]<=j+1:              #初始化第一行
            dp[0][j]=max(dp[0][j],goods[0][0])
        else:
            if goods[i][1]<=j+1:
                dp[i][j]=max(dp[i-1][j],goods[i][0]+ \
                             (dp[i-1][j-goods[i][1]] if j-goods[i][1]>=0 else
0))
            else:
                dp[i][j]=dp[i-1][j]
#print(dp)
print(dp[-1][-1])
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

02754: 八皇后

dfs and similar, http://cs101.openjudge.cn/practice/02754

思路：

由于每行只能放一个，那就可以看成是全排列的二维展开形式，但是输出上实际还是全排列。这道题与全排列不同的地方在于：①在函数中不能直接 print 了要先在函数外面设置一个 result 用于储存结果，然后在函数内部声明其为全局变量。②在 record(choice) 阶段，不仅仅是判断这个数字有没有用过了，还要判断其是否与其他数字在对角线上，因此此处仍然需要一个 for 循环来遍历已经确定位置的数字。

代码：

```
chosen=[False]*9
ans=[0]*9
result=[]
def dfs(k):
    global result
    if k>8:                                #退出条件
```

```
        result.append(ans[1:])
    for i in range(1,9):                #每一层的可能数字遍历
        if chosen[i]:                   #这两行的判定是防止数字重复
            continue
        f=False                         #这七行的判定是防止皇后在对角线
        for j in range(1,k):
            if abs(ans[j]-i)==abs(j-k):
                f=True
                break
        if f:
            continue
        chosen[i]=True                  #这两行是在 record(choice)
        ans[k]=i
        dfs(k+1)                        #这一行是在递归，递归出该层向下的所有可能结果
        chosen[i]=False                 #这一行是在（下一层的递归结束后）回溯，标记为未使用
dfs(1)                                  #调用函数，获得全局遍历 result，也就获得了 92 个排列
#print(result)
n=int(input())
for _ in range(n):
    t=int(input())
    print(''.join(map(str,result[t-1])))
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>

189A. Cut Ribbon

brute force, dp 1300 https://codeforces.com/problemset/problem/189/A

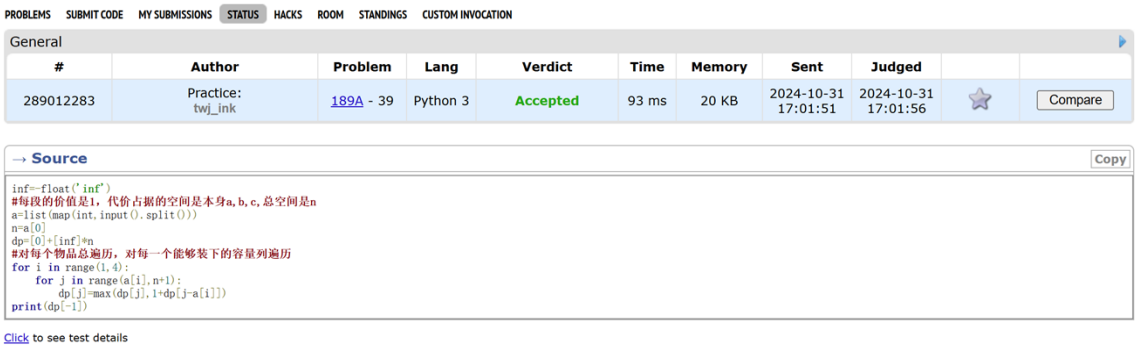
思路：

看答案才知道的思路，是完全背包问题。看了题解发现用 dp[i]=max(dp[i-a],max(dp[i-b],dp[i-c]))+1 更好理解，因为每次切割是增加一段而不是两段。

代码：

```
inf=-float('inf')
#每段的价值是 1，代价占据的空间是本身 a,b,c,总空间是 n
a=list(map(int,input().split()))
n=a[0]
dp=[0]+[inf]*n
#对每个物品总遍历，对每一个能够装下的容量列遍历
for i in range(1,4):
    for j in range(a[i],n+1):
        dp[j]=max(dp[j],1+dp[j-a[i]])
print(dp[-1])
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



2. 学习总结和收获

<mark>如果作业题目简单，有否额外练习题目，比如：OJ“计概 2024fall 每日选做”、CF、LeetCode、洛谷等网站题目。</mark>

本次作业全部都是自己看了答案（并理解了很久答案😓）写出来的，每日选做的难度也很大有些题一直卡着做不出来。

dp：

dp 的题目感觉有三个难点：找从小问题到大问题的递推策略、找递推式、dp 数组初始化的状态。而我目前都没有好的办法，感觉很难。作业题在第一次看时给我的感受是：这题跟 dp 有啥关系？？/这题跟递归有啥关系？？群里面大佬是怎么用时这么短做完的阿？？

作业里面的背包问题涉及了 01 背包和完全背包，二者有极其相似的公式：

假设有 n 个物品，每个物品的价值和占用空间分别是 ci、ai，背包总容量是 A，则两者均采用大层遍历每个物品，小层遍历**每个可容纳物品的容量值**（这样便省去了 i 和 j 的比较，保证 dp 不会越界）。

01 背包：初始化全为 0；j 采用逆向遍历

```
dp=[0]*(n+1)
for i in range(1,n+1):
	for j in range(A,ci-1,-1):
		dp[j]=max(dp[j],ci+dp[j-ai])
print(dp[-1])
```

完全背包：初始化除了 0 位置是 0，其余位置是-inf；j 采用正向遍历

```
dp=[0]+[-float('inf')]*n
for i in range(1,n+1):
	for j in range(ci,A):
		dp[j]=max(dp[j],ci+dp[j-ai])
print(dp[-1])
```

递归：

汉诺塔的函数反复调用的内部具体实现仍然感觉很深很绕，目前只能做到先**明确函数的作用**并不断调用直到自动递归出结果。

从 LeetCode 31：递归、回溯、八皇后、全排列一篇文章全讲清楚 - 知乎截取里面的一部分精华，对我这种初学者有很大帮助：

> 如果是初学者，即使是理解了递归的概念想要自己写出代码来也是一件不容易的事情。因为在我们看来记录所有节点的状态，遍历，再回到之前的状态，再继续搜索，这是一个非常复杂的过程。
> 好在我们并不需要自己记录这个过程，因为计算机在执行程序的时候会有一个称为方法栈的东西，记录我们执行的所有函数和方法的状态。比如我们在 A 程序中执行了 B 程序的代码，like this:

```
def A():
do_something()
B()
do_something()
```

> 我们在 A 当中的第二行执行了 B 这个函数，那么系统会为我们记录下我们在执行 B 的时候的位置是第二行，当 B 执行结束，还会回到调用的位置，继续往下执行。而递归呢，则是利用这一特性，让 A 来执行自己：

```
def A():
do_something()
```

```
A()
do_something()
```

和刚才一样，系统同样会为我们记录执行 A 的位置。A 执行了 A，在我们看来是一样的，但是系统会储存两份方法状态，我们可以简单认为后面执行的 A 是 A1，那么当 A1 执行结束之后，又会回到 A 执行 A1 的位置。而用递归来实现搜索呢，只是在此基础上加上了一点循环而已。

```
def dfs():
for choice in all_choices():
record(choice)
dfs()
rollback(choice)
```

这就是一个简单的搜索的代码框架了，我们遍历每个节点的所有决策，然后记录下这个选择，进行往下递归。当我们执行完再次回到这个位置这一行代码的时候，我们已经遍历完了这个选择所有的情况，所以我们要撤销这个选择带来的一些影响，好方便枚举下一个选择。

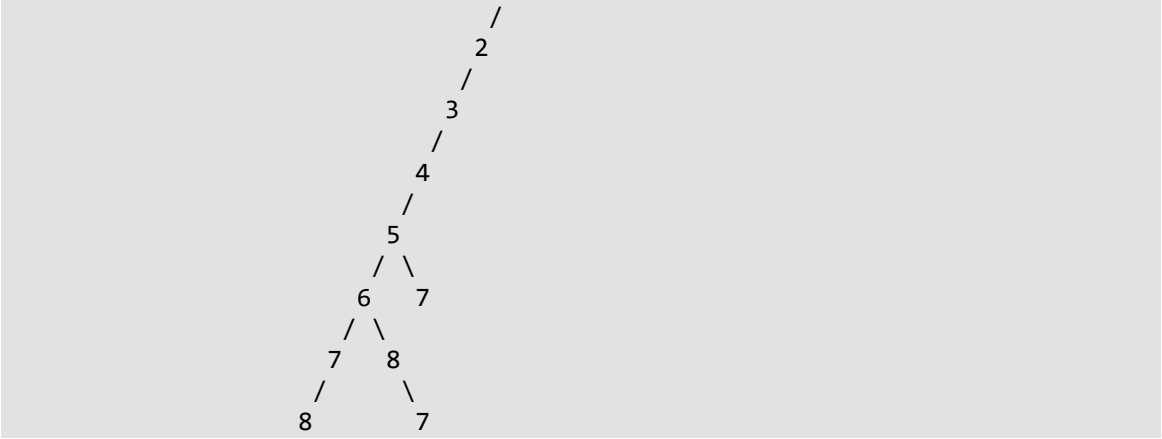
但是这样还没结束，还有一点小问题，就是我们只需要放置 8 个皇后，我们需要针对这点做限制，不然就会无穷无尽递归下去了。做限制的方法也很简单，本质上来说我们是要限制递归的深度，所以我们可以用一个变量来记录递归的深度，每次往下递归的时候就加上 1，如果递归深度达到了我们的要求，就不再往下递归了。

加上深度判断之后，我们就得到了经典的回溯的代码框架。几乎可以说无论什么样的回溯搜索问题，都脱离不了这个代码框架，只是具体执行和撤销的逻辑有所不同而已：

```
def dfs(depth):
①if depth >= 8:
return
②for choice in all_choices():
③  record(choice)
④  dfs(depth+1)
⑤  rollback(choice)
```

以及一个方便理解递归与回溯的图：

```
                                root
                                 /
                                1
```



刚开始每一个位置都是可以放 1 到 8 的，这也就是

②处 for 循环的目的。然后进行

③每一层的选定数字，并

④向下递归选定下一层的数字，直到

①边界退出，在退出后

⑤回溯，回溯后下面的层实际上都已经遍历完成了。

其中④的递归操作只需要无脑调用函数本身，③和⑤需要动脑筋构造代码实现，这样的模板感觉有助于理解，但是真正自己实现起来仍然感觉非常非常难！！！

总结：目前自己处于一道题的思路也想不出来的状态（），只能看别人的代码/看别人的思路才能尽量做出来，还需要练习和提高敏感度。

看了群里老师发的代码和同学的代码，发现有些写法简便但是仍然是递归和回溯的思维，有些又看不懂了...感觉月考可能会相当失败😔