

2025/10/28 递归

Updated 2025-10-28 14:35 GMT+8

Compiled by Hongfei Yan (2024 Fall)

Log:

2025/9/29: 因为数算很依赖递归，所以把这个递归课件更新&过一遍。

2025/9/22: 建议每位同学创建自己的课程学习仓库，例如：<https://github.com/twj-ink/2024fall>

<https://github.com/twj-ink/2024fall>

2024fall

[Github入门教程，适合新手学习（非常详细）_github教程-CSDN博客](#)

这是github教程。不得不说寒假时候为了搞这个 Github 网站，专门下载了 Git Bash，纯命令行的操作很炫酷也很神奇，因为一次操作不慎把本地文件和没更新的仓库文件同步了，使得花费了半天时间更新的本地文件全部清零...所以文件要备份！！！ 设置图标也是花了一晚上才弄好...善用搜索！！！

这里是twj的计概总结。

I. 简介

2024fall 为这一学期计概的课程课件、我提交的作业、每日选做以及额外补充的题目代码，全部汇总于文件夹 python_work 中。在 python_work 中，codeforces problems, openjudge problems, sunnywhy_problems, 基础, cheat sheet 都是在学习前期放入的代码 没有经过题型的具体分类整理，可能检索和查看起来比较麻烦；算法 部分则是后期经过具体分类后存放的代码，查看起来更加直观方便，同时这一部分也是自己认为的精华。
Final_cheat_sheet 是期末的大总结，内部包含了较为全面的知识，同时也有其余文件夹中未包含的题目代码(这些是在做总结的时候联想起来的同类题目，直接 把代码附在了总结里面)。

II. 学习经验

本人在进入大学前为0基础。虽然高中时候有过python课，但是我的印象里面只有 int``input 这两个词，甚至在当时连意思都不知道。所以在开学时考虑到自身能力问题 选择了入门简单、配置简单、环境简单的python课，并且在学长的推荐下，成功选到了闫宏飞老师的班。期末成绩AK。

回想我的学习过程，最主要的是两部分：第一是刚开始的语法学习。我使用的资源是闫老师推荐的 <https://www.runoob.com/python3/python3-tutorial.html>，很详细且有例子，当时是在军训期间自学的语法，建议是边学边在pycharm或者其他IDE下自己跟着敲代码，毕竟熟能生巧。

第二是善用ChatGPT，不管是学习初期的语法问题，还是后期的代码纠错，它都是很好的助手。它极有可能是这辈子能遇到的知识水平最高的最有耐心的最懂你的朋友或老师了。简答题目可以用它来直接生成代码，但是有一定难度的题目它生成的代码有可能是错误的，自己也认真读一下它的代码。

此外，闫老师的课件内容的质量也是相当高，题目分类的针对性也很强，课件里面的题目值得去写一写，我在 python_work/算法/Dynamic_Programming 中把课件里面的dp例题几乎都写了一遍，收获很大，比如总结了几类整数划分的问题。

2024/11/02: 在 三 增加了 yield生成器

2024/11/01: 增加了 3.2 函数参数是可变、不可变对象

2024/10/24: 晴问 复活了。<https://sunnywhy.com> 是零基础同学的福音。“算法基础与在线实践”，有点难了。

2024/10/24 部分内容取自, https://github.com/GMyhf/2023fall-cs101/blob/main/recursion_questions.md

一、Recap

1 解决输入数据太多

1.1 处理不定行输入的常用方法

在 Python 中，处理多行输入有以下几种常见方式：

使用 `try...except` 捕获输入结束（如 `EOFError`）

利用 `sys.stdin` 逐行读取

通过 `sys.stdin.read()` 一次性读取所有输入



【高中与大学的计算机基础
课程衔接】

+ 新建对话

历史对话记录

Python多行输入

函数内外变量作用域

函数内外变量作用域

T25353排队超时

学习课程方法

Typora与Obs

编程环境配置

Python环境配置

编程核心与LeetCode

知识库查询

知识库概述

DeepSeek-V3 v

Python多行输入

对话记录

宏飞

处理不定行输入的常用方法 在 Python 中，处理多行输入有以下几种常见方式：使用 try...except 捕获输入结束（如 EOFError）利用 sys.stdin 逐行读取 通过 sys.stdin.read() 一次性读取所有输入



assistant

DeepSeek-V3

在 Python 中，处理不定行输入（即不知道具体行数的输入）有以下几种常用方法：

使用 try-except 捕获 EOFError

```
while True:  
    try:  
        line = input()  
        # 处理每一行的逻辑  
    except EOFError:  
        break
```

这种方法适用于手动输入时使用 Ctrl+D (Unix/Linux) 或 Ctrl+Z+Enter (Windows) 结束输入的场景 [来源 1] [来源 4] [来源 11]。

利用 sys.stdin 逐行读取

```
import sys  
for line in sys.stdin:  
    # 处理每一行的逻辑 (包含换行符)  
    line = line.strip() # 通常需要去除首尾空白
```

通过 sys.stdin.read() 一次性读取所有输入

```
import sys  
data = sys.stdin.read() # 整个输入作为字符串  
lines = data.splitlines() # 按行分割  
for line in lines:  
    # 处理每一行的逻辑
```

这种方法特别适合处理大规模输入，效率较高 [来源 2] [来源 8]。

列表存储法

```
lines = []  
while True:  
    try:  
        lines.append(input())  
    except EOFError:  
        break  
# 后续处理lines列表
```

这种方法将所有输入存储在列表中便于后续处理 [来源 4]。

注意事项：

Windows 和 Unix/Linux 系统的EOF输入方式不同

字符串处理时注意换行符和首尾空格

对于算法竞赛等场景，推荐使用sys.stdin.read()方法提高IO效率 [来源 2] [来源 8] [来源 16]

参考来源：

2020fall_cs101.ope... [来源1](#) [来源2](#) [来源3](#) [来源4](#) [来源6](#) [来源9](#)

Book_my_flight_20... [来源5](#) [来源7](#) 2020fall_Codeforce... [来源8](#) [来源16](#) [来源18](#)

20241203_knowledg... [来源10](#) 算法笔记.胡凡.pdf [来源11](#) [来源17](#) 20241112_Segment... [来源12](#)

2024fall_LeetCode... [来源13](#) [来源15](#) Python编程：从入门... [来源14](#)



学习技巧

课程知识库

课程广场

闫宏飞

开始新对话

使用Shift+Enter/Ctrl+Enter换行，Enter发送消息

知识库 3/3



提供的信息仅供参考，请仔细甄别核实

版权所有2025© 北京大学计算中心

1.2 缓存原理`sys.stdin.read`

原理很简单，吃饭时候一口一口吃，肯定比一粒一粒吃要快。

合理使用buffer是编程小技巧，但是有的题目不用的话，Python提交基本就是超时，如：CF1425A. Arena of Greed.

在编程中（例如 Codeforces 上），使用缓冲输入输出（I/O）可以显著提升程序性能，特别是当数据量非常大时。Python 默认情况下使用缓冲 I/O，但通过合理地控制输入输出，可以进一步减少因频繁的 I/O 操作导致的时间开销，避免超时。

输入优化：

1. `sys.stdin.read()` 代替 `input()`：

- `input()` 每次读取一行，而 `sys.stdin.read()` 会一次性读取所有输入，可以在数据量大时显著提升速度。
- 例如：

```
1 import sys
2 input = sys.stdin.read
3 data = input().split() # 读入所有数据并分割为列表
```

2. 批量读取：

- 对于较大的输入数据，通过一次性读取并处理比逐行读取更高效。

输出优化：

1. `sys.stdout.write()` 代替 `print()`：

- `print()` 函数默认会在每次调用后刷新输出缓冲区，而 `sys.stdout.write()` 允许在最后一次性输出所有结果。
- 例如：

```
1 import sys
2 sys.stdout.write('\n'.join(map(str, results)) + '\n')
```

- 这种方式将所有输出内容一次性写入缓冲区，在最后统一输出，避免了多次刷新。

2. 批量输出：

- 像处理输入一样，通过将所有输出内容收集后一次性输出可以极大地优化程序性能。

示例：

假设你要处理多个测试用例，下面是一个优化输入输出的 Python 代码示例：

```
1 import sys
```

```
3  input = sys.stdin.read
4  output = sys.stdout.write
5
6  def solve():
7      data = input().split()
8      n = int(data[0])
9      results = []
10     for i in range(1, n + 1):
11         # 假设是简单的加法运算
12         results.append(str(int(data[2*i - 1]) + int(data[2*i])))
13     output("\n".join(results) + "\n")
14
15 solve()
```

在这段代码中，我们一次性读取所有数据，并通过 `sys.stdout.write` 批量输出，减少了 I/O 操作的次数，能有效避免超时。

1.3 iter 与 next 配合

`iter` 与 `next` 配合，顺序操作一次性读入数据，还挺方便。



HOME TO

MAIN ACMMSGU

Status filter i

Contest sta

#

[31725748](#)[33358270](#)[33662018](#)[33668227](#)[33663725](#)[31294776](#)[32843867](#)[33664867](#)[12387193](#)[14899190](#)[32215915](#)[32216018](#)[33904412](#)[20788731](#)[31363889](#)

By OmarAboutaleb78, contest: Codeforces Round 321 (Div. 2), problem: (C) Kefa and Park, Accepted, #, Copy

```

import sys
from collections import deque

def solve():
    it = iter(sys.stdin.read().strip().split())
    n = int(next(it)); m = int(next(it))
    cats = [0] + [int(next(it)) for _ in range(n)]
    g = [[] for _ in range(n+1)]
    for _ in range(n-1):
        x = int(next(it)); y = int(next(it))
        g[x].append(y)
        g[y].append(x)
    visited = [False] * (n+1)
    q = deque()
    start_consec = 1 if cats[1] == 1 else 0
    q.append((1, start_consec))
    visited[1] = True
    ans = 0
    while q:
        node, consec = q.popleft()
        if consec > m:
            continue
        is_leaf = True
        for v in g[node]:
            if not visited[v]:
                is_leaf = False
                visited[v] = True
                new_consec = consec + 1 if cats[v] == 1 else 0
                q.append((v, new_consec))
        if is_leaf and node != 1:
            ans += 1
    print(ans)

if __name__ == "__main__":
    solve()

```

如果想查看某个题目的测试数据（往年可以看到其他人提交的代码，2024年10月被屏蔽了。2025年9月解封了），替换链接中数字和最后一个字母，例如查看580C可以访问
<http://codeforces.com/problemset/status/580/problem/C>

Codeforces题目是英文，洛谷提供中文。方法是拿到CF题号，如：1749C，直接编辑这个link的相应题号位置，

<https://www.luogu.com.cn/problem/CF1749C>

2 编程题目

示例：03248: 最大公约数

math, <http://cs101.openjudge.cn/practice/03248>

给定两个正整数，求它们的最大公约数。

输入

有多组数据，每行为两个正整数，且不超过int可以表示的范围。

输出

行对应输出最大公约数。

样例输入

```
1 | 4 8
2 | 8 6
3 | 200 300
```

样例输出

```
1 | 4
2 | 2
3 | 100
```

提示

用math.gcd

```
1 | from math import gcd
2 |
3 | while True:
4 |     try:
5 |         a, b = input().split()
6 |         print(gcd(int(a), int(b)))
7 |     except EOFError:
8 |         break
```

自己实现gcd

```
1 | def gcd(a, b):
2 |     while b:
3 |         a, b = b, a % b
4 |     return a
5 |
6 | def main():
7 |     import sys
8 |     input = sys.stdin.read
9 |     data = input().strip().split('\n')
10 |
11    for line in data:
12        a, b = map(int, line.split())
13        print(gcd(a, b))
14
```

```
15 | if __name__ == "__main__":
16 |     main()
```

练习E02804:词典

<http://cs101.openjudge.cn/pctbook/E02804/>

练习M12029: 水淹七军

<http://cs101.openjudge.cn/pctbook/M12029/>

二、递归

1 What Is Recursion?

<https://runestone.academy/ns/books/published/pythonds3/Recursion/WhatIsRecursion.html?mode=browsing>

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

递归是一种解决问题的方法，它涉及将一个问题分解成越来越小的子问题，直到得到一个足够小的问题，可以轻易地解决。递归涉及到一个函数调用自身。虽然表面上看起来可能没什么特别之处，但递归使我们能够编写出优雅的解决方案，来解决那些可能非常难以编程的问题。

示例：Calculating the Sum of a Vector of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a vector of numbers such as: [1,3,5,7,9]. An iterative function that computes the sum is shown in [ActiveCode 4.3.1](#). The function uses an accumulator variable (`the_sum`) to compute a running total of all the numbers in the list by starting with and adding each number in the list.

我们将从一个简单的、你已经知道如何在不使用递归的情况下解决的问题开始我们的探讨。假设你想计算一个数字向量（如 [1, 3, 5, 7, 9]）的总和。一个使用迭代方法计算总和的函数如 [ActiveCode 4.3.1](#) 所示。该函数使用一个累加器变量 (`the_sum`) 通过从初始值开始并加上列表中的每个数字来计算列表中所有数字的运行总和。

```
1 def list_sum(num_list):
2     the_sum = 0
3     for i in num_list:
4         the_sum = the_sum + i
5     return the_sum
6
7 print(list_sum([1, 3, 5, 7, 9]))
```

Activity: 4.3.1 Iterative Summation

Such an expression looks like this: (((1 + 3) + 5) + 7) + 9)

We can also parenthesize the expression the other way around, (1 + (3 + (5 + (7 + 9))))

Notice that the innermost set of parentheses, , is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

```
total = (1 + (3 + (5 + (7 + 9))))  
total = (1 + (3 + (5 + 16)))  
total = (1 + (3 + 21))  
total = (1 + 24)  
total = 25
```

How can we take this idea and turn it into a Python program? First, let's restate the sum problem in terms of Python lists. We might say the sum of the list `num_list` is the sum of the first element of the list (`num_list[0]`) and the sum of the numbers in the rest of the list (`num_list[1:]`). To state it in a functional form:

我们如何将这个想法转化为一个 Python 程序呢？首先，让我们用 Python 列表来重新表述求和问题。我们可以这样描述：列表 `num_list` 的总和等于列表的第一个元素 (`num_list[0]`) 与列表剩余部分 (`num_list[1:]`) 中所有数字的总和。以函数形式表示为：

$$\text{list_sum}(\text{num_list}) = \text{first}(\text{num_list}) + \text{list_sum}(\text{rest}(\text{num_list}))$$

In this equation `first(num_list)` returns the first element of the list and `rest(num_list)` returns a list of everything but the first element. This is easily expressed in Python as shown in [ActiveCode 4.3.2](#).

```
1 #Example of summing a list using recursion.  
2  
3 def list_sum(numList):  
4     if len(numList) == 1:  
5         return numList[0]  
6     else:  
7         return numList[0] + list_sum(numList[1:]) #function makes a recursive call  
to itself.  
8  
9 print(list_sum([1, 3, 5, 7, 9]))
```

Activity: 4.3.2 Recursive Summation

There are a few key ideas while using vector to look at. First, on line 4 we are checking to see if the vector is one element long. This check is crucial and is our escape clause from the function. The sum of a vector of length 1 is trivial; it is just the number in the vector. Second, on line 7 our function calls itself! This is the reason that we call the `vectsum` algorithm recursive. A recursive function is a function that calls itself.

在这段代码中有几个关键点需要注意。首先，在第4行，我们检查列表是否只有一个元素。这个检查是至关重要的，是我们从函数中退出的条件。长度为1的列表的总和是显而易见的；它就是列表中的那个数字。其次，在第7行，我们的函数调用了自身！这就是为什么我们将 `list_sum` 算法称为递归的原因。递归函数是指调用自身的函数。

Figure 1 shows the series of **recursive calls** that are needed to sum the list. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.

图1展示了求和列表所需的递归调用序列。你可以将这一系列调用视为一系列简化过程。**每次进行递归调用时，我们都在解决一个更小的问题，直到问题不能再被简化为止。**

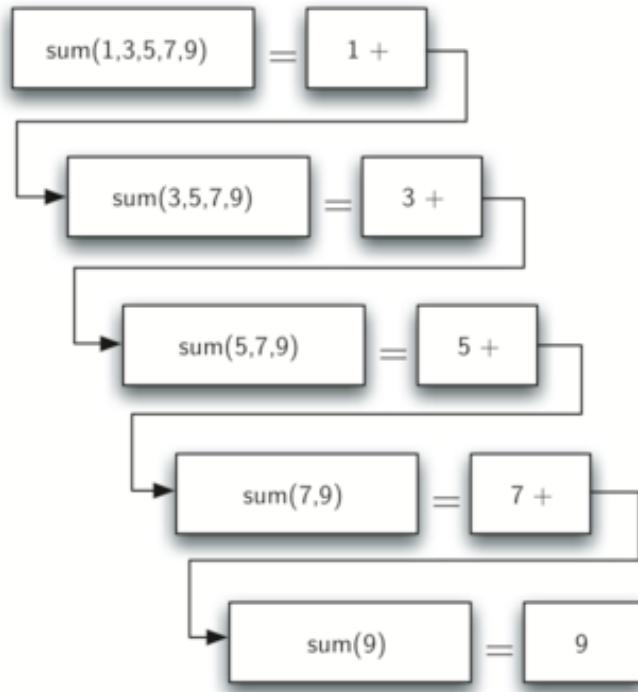
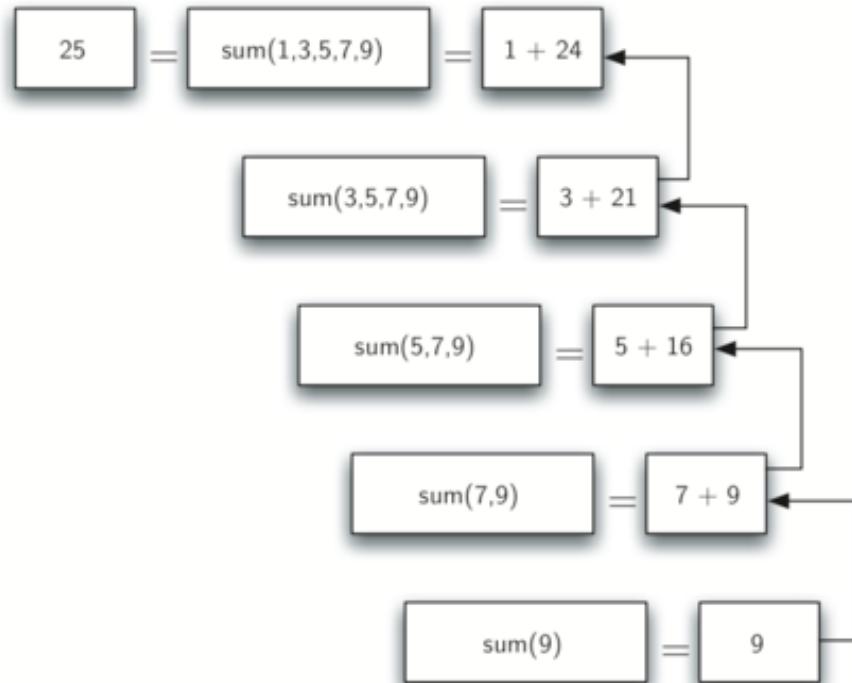


Figure 1: Series of Recursive Calls Adding a List of Numbers

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. Figure 2 shows the additions that are performed as `listsum` works its way backward through the series of calls. When `listsum` returns from the topmost problem, we have the solution to the whole problem.

当我们到达问题不能再被简化的地步时，我们开始将每个小问题的解逐步组合起来，直到最初的整个问题被解决。图2展示了随着 `listsum` 从最顶层的问题逐步回溯通过一系列调用时所进行的加法运算。当 `listsum` 从最顶层的问题返回时，我们就得到了整个问题的解。



2. Three Laws of Recursion递归三法则

<https://runestone.academy/ns/books/published/pythonds3/Recursion/TheThreeLawsofRecursion.html?mode=browsing>

Like the robots of Asimov, all recursive algorithms must obey three important laws:

1. A recursive algorithm must have a **base case**. 递归算法必须有一个基准情形。
2. A recursive algorithm must change its state and move toward the base case. 递归算法必须改变其状态并朝着基准情形前进。
3. A recursive algorithm must call itself, recursively. 递归算法必须调用自身，即进行递归调用。

Let's look at each one of these laws in more detail and see how it was used in the `vectsum` algorithm. First, a base case is the condition that allows the algorithm to stop recursing. A base case is typically a problem that is small enough to solve directly. In the `vectsum` algorithm the base case is a list of length 1.

基准情形是允许算法停止递归的条件。基准情形通常是一个足够小可以直接解决的问题。在 `vectsum` 算法中，基准情形是一个长度为1的列表。

To obey the second law, we must arrange for a change of state that moves the algorithm toward the base case. A change of state means that some data that the algorithm is using is modified. Usually the data that represents our problem gets smaller in some way. In the `vectsum` algorithm our primary data structure is a vector, so we must focus our state-changing efforts on the vector. Since the base case is a list of length 1, a natural progression toward the base case is to shorten the vector.

为了遵守第二条法则，我们必须安排状态的变化，使算法朝着基准情形前进。**状态变化意味着算法使用的一些数据被修改了。通常代表我们问题的数据会以某种方式变小。** 在 `vectsum` 算法中，我们的主要数据结构是一个向量，所以我们必须将状态变化的重点放在向量上。由于基准情形是一个长度为1的列表，因此一个自然的朝向基准情形的进展就是缩短向量。

The final law is that the algorithm must call itself. This is the very definition of recursion. Recursion is a confusing concept to many beginning programmers. As a novice programmer, you have learned that functions are good because you can take a large problem and break it up into smaller problems. The smaller problems can be solved by writing a function to solve each problem. When we talk about recursion it may seem that we are talking ourselves in circles. We have a problem to solve with a function, but that function solves the problem by calling itself! But the logic is not circular at all; the logic of recursion is an elegant expression of solving a problem by breaking it down into smaller and easier problems.

最后一条法则是算法必须调用自身。这就是递归的定义。递归对许多初学者来说是一个令人困惑的概念。作为一名新手程序员，你已经了解到函数的好处在于可以将一个大问题分解成更小的问题。这些问题可以通过编写函数来分别解决。当我们谈论递归时，似乎我们在绕圈子说话。我们有一个需要通过函数解决的问题，但该函数通过调用自身来解决问题！但实际上逻辑并不循环；**递归的逻辑是一种优雅的表达方式，它通过将问题分解成更小、更简单的问题来解决问题。**

It is important to note that regardless of whether or not a recursive function implements these three rules, it may still take an unrealistic amount of time to compute (and thus not be particularly useful).

Self Check

Q-2: Why is a base case needed in a recursive function?

- A. If a recursive function didn't have a base case, then the function would end too early.
- B. If a recursive function didn't have a base case, then the function would return an undesired outcome.
- C. If a recursive function didn't have a base case, then the function would continue to make recursive calls creating an infinite loop.
- D. If a recursive function didn't have a base case, then the function would not be able to ever make recursive calls in the first place.

Check Me

Compare me

✓ Correct! a base case is needed to end the continuous recursive calls, so that the program doesn't get stuck in a never ending loop.

Activity: 5.4.2 Multiple Choice (question_recsimp_1)

Q-3: How many recursive calls are made when computing the sum of the vector {2,4,6,8,10}?

- A. 6
- B. 5
- C. 4
- D. 3

Check Me

Compare me

✓ the first recursive call passes the vector {4,6,8,10}, the second {6,8,10} and so on until [10].

Activity: 5.4.3 Multiple Choice (question_recsimp_2)

Q-4: Suppose you are going to write a recursive function to calculate the factorial of a number. `fact(n)` returns $n * n-1 * n-2 * \dots$ Where the factorial of zero is defined to be 1. What would be the most appropriate base case?

- A. $n == 0$
- B. $n == 1$
- C. $n >= 0$
- D. $n <= 1$

Check Me

Compare me

✓ Good, this is the most efficient, and even keeps your program from crashing if you try to compute the factorial of a negative number.

Activity: 5.4.4 Multiple Choice (question_recsimp_3)

示例：Converting an Integer to a String in Any Base

Suppose you want to convert an integer to a string in some base between binary and hexadecimal. For example, convert the integer 10 to its string representation in decimal as "10", or to its string representation in binary as "1010". While there are many algorithms to solve this problem, including the algorithm discussed in the stack section, the recursive formulation of the problem is very elegant.

假设你想将一个整数转换为二进制到十六进制之间的某个进制的字符串表示。例如，将整数10转换为其他十进制字符串表示 "10"，或将其转换为二进制字符串表示 "1010"。虽然有很多算法可以解决这个问题，包括在栈部分讨论的算法，但递归方法的表述非常优雅。

Let's look at a concrete example using base 10 and the number 769. Suppose we have a sequence of characters corresponding to the first 10 digits, like `convString = "0123456789"`. It is easy to convert a number less than 10 to its string equivalent by looking it up in the sequence. For example, if the number is 9, then the string is `convString[9]` or "9". If we can arrange to break up the number 769 into three single-digit numbers, 7, 6, and 9, then converting it to a string is simple. A number less than 10 sounds like a good base case.

让我们来看一个具体的例子，使用十进制和数字769。假设我们有一个对应于前10个数字的字符序列，如 `convert_string = "0123456789"`。通过查找这个序列，很容易将小于10的数字转换为其字符串等价形式。例如，如果数字是9，那么字符串就是 `convert_string[9]` 或 "9"。如果我们能够将数字769分解成三个单个数字7、6和9，那么将其转换为字符串就很简单了。小于10的数字听起来像是一个好的基准情形。

Knowing what our base is suggests that the overall algorithm will involve three components:

1. Reduce the original number to a series of single-digit numbers. 将原始数字减少为一系列单个数字。
2. Convert the single digit-number to a string using a lookup. 使用查找表将单个数字转换为字符串。
3. Concatenate the single-digit strings together to form the final result. 将单个数字的字符串连接起来形成最终结果。

The next step is to figure out how to change state and make progress toward the base case. Since we are working with an integer, let's consider what mathematical operations might reduce a number. The most likely candidates are division and subtraction. While subtraction might work, it is unclear what we should subtract from what. Integer division with remainders gives us a clear direction. Let's look at what happens if we divide a number by the base we are trying to convert to.

下一步是弄清楚如何改变状态并朝着基准情形前进。由于我们在处理一个整数，让我们考虑哪些数学运算可以减少一个数字。最有可能的候选者是除法和减法。虽然减法可能有效，但我们不清楚应该从什么中减去什么。带余数的整数除法则为我们指明了明确的方向。让我们看看当我们试图将一个数字除以目标进制时会发生什么。

Using integer division to divide 769 by 10, we get 76 with a remainder of 9. This gives us two good results. First, the remainder is a number less than our base that can be converted to a string immediately by lookup. Second, we get a number that is smaller than our original and moves us toward the base case of having a single number less than our base. Now our job is to convert 76 to its string representation. Again we will use integer division plus remainder to get results of 7 and 6 respectively. Finally, we have reduced the problem to converting 7, which we can do easily since it satisfies the base case condition of $n < \text{base}$, where $\text{base}=10$. The series of operations we have just performed is illustrated in Figure 3. Notice that the numbers we want to remember are in the remainder boxes along the right side of the diagram.

使用整数除法将769除以10，我们得到商76和余数9。这给了我们两个很好的结果。首先，余数是一个小于我们基数的数字，可以通过查找立即转换为字符串。其次，我们得到了一个比原数小的数字，并且它使我们朝着只有一个小于基数的数字的基准情形前进。现在我们的任务是将76转换为其字符串表示。我们再次使用整数除法加上余数，分别得到7和6。最后，我们将问题简化为转换7，这很容易做到，因为它满足基准情形条件。我们刚刚执行的一系列操作如图3所示。注意，我们需要记住的数字在图表右侧的余数框中。

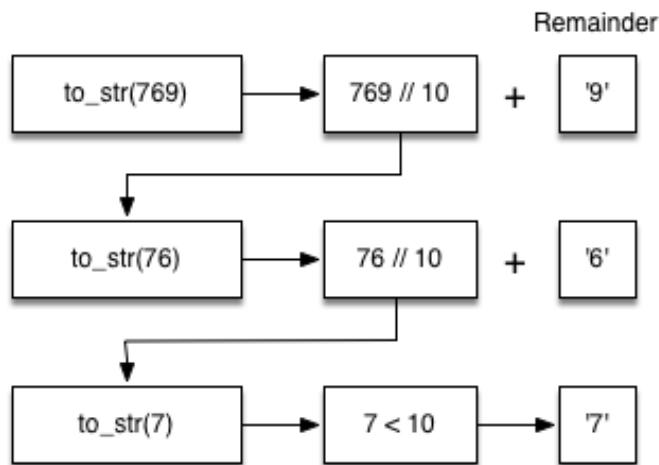


Figure 3: Converting an Integer to a String in Base 10

Activity: 4.5.1 shows the Python code that implements the algorithm outlined above for any base between 2 and 16.

```
1 def to_str(n, base):
2     # 定义用于转换的字符序列
3     convert_string = "0123456789ABCDEF"
4
5     # 基准情形：如果 n 小于基数，则直接返回对应的字符
6     if n < base:
7         return convert_string[n]
8     else:
9         # 递归调用：先处理商，再处理余数
10        # 通过延迟连接操作，确保结果的顺序是正确的
11        return to_str(n // base, base) + convert_string[n % base]
12
13
14 # 示例
15 print(to_str(10, 2))  # 输出: "1010"
16 print(to_str(255, 16)) # 输出: "FF"
```

Activity: 4.5.1 Recursively Converting from Integer to String

Notice that in line 6 we check for the base case where `n` is less than the base we are converting to. When we detect the base case, we stop recursing and simply return the string from the `convertString` sequence. In line 11 we satisfy both the second and third laws—by making the recursive call and by reducing the problem size—using division.

请注意，在第6行我们检查了基准情形，即当 `n` 小于我们要转换的基数时。当我们检测到基准情形时，我们停止递归，并直接从 `convert_string` 序列中返回相应的字符串。在第11行，我们通过进行递归调用并使用除法来减小问题规模，从而满足了第二条和第三条法则。

Let's trace the algorithm again; this time we will convert the number 10 to its base 2 string representation ("1010").

让我们再次跟踪 ActiveCode 4.5.1 中显示的算法；这次我们将把数字10转换为其二进制字符串表示 ("1010") 。

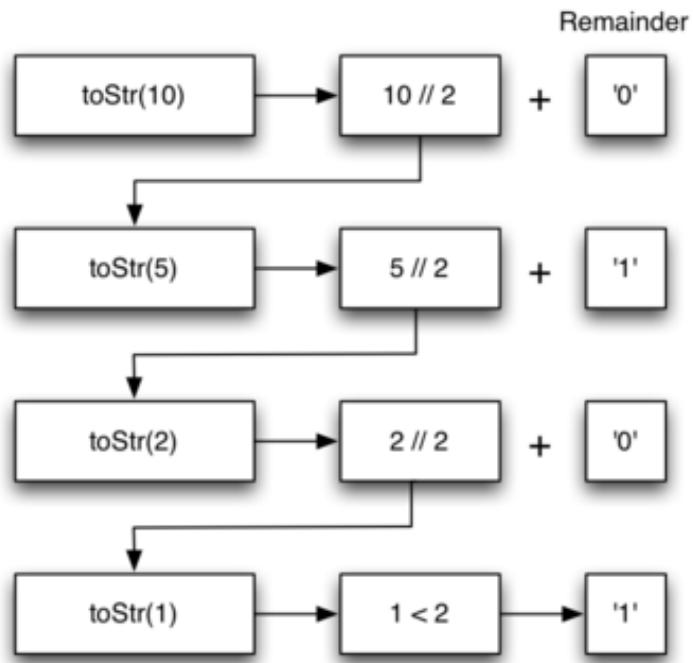


Figure 4: Converting the Number 10 to its Base 2 String Representation

[Figure 4](#) shows that we get the results we are looking for, but it looks like the digits are in the wrong order. The algorithm works correctly because we make the recursive call first on line 8, then we add the string representation of the remainder. If we reversed returning the `convertString` lookup and returning the `toStr` call, the resulting string would be backward! But **by delaying the concatenation operation until after the recursive call has returned, we get the result in the proper order.** This should remind you of our discussion of stacks back in the previous chapter.

图4 显示我们得到了预期的结果，但看起来数字的顺序是反的。算法之所以能正确工作，是因为我们在第6行首先进行了递归调用，然后才添加余数的字符串表示。如果我们先返回 `convert_string` 查找的结果，再返回 `to_str` 调用的结果，最终得到的字符串将会是反向的！但是，通过**将连接操作延迟到递归调用返回之后进行，我们得到了正确的顺序。**这应该让你想起我们在前一章中关于栈的讨论。

Self Check

Q-3: Is the process of stepping through a recursive function similar to the construct of a stack or a queue?

- A. A stack, because a recursive function will complete the final function call before any of the other function calls, similar to how a stack has the Last-in-First-out principle.
- B. A queue, because a recursive function will complete its intial function call before any of the other function calls, similar to how a queue has the First-in-First-out principle.

Check Me

Compare me

✓ Correct! a recursive function will complete the final function call first, because the rest of the calls are waiting for the results of the calls they made.

Activity: 5.5.3 Multiple Choice (RecursiveQ1)

3. 栈和递归的关系

3.1 Stack Frames: Implementing Recursion

<https://runestone.academy/ns/books/published/pythonds3/Recursion/StackFramesImplementingRecursion.html?mode=browsing>

Suppose that instead of concatenating the result of the recursive call to `to_str` with the string from `convertString`, we modified our algorithm to push the strings onto a stack instead of making the recursive call. The code for this modified algorithm is shown.

假设我们不是将递归调用 `to_str` 的结果与 `convertString` 中的字符串进行连接，而是修改算法，将字符串压入栈中，而不是进行递归调用。这个修改后的算法代码如下所示。

- 使用栈替代递归：通过将字符串压入栈中，我们可以避免递归调用，并在最后从栈中弹出字符串以获得正确的顺序。
- 栈的后进先出（LIFO）特性：这确保了我们在处理完所有子问题后，能够以正确的顺序组合结果。

```
1 rStack = []
2
3 def to_str(n,base):
4     convertString = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
5     while n > 0:
6         if n < base:
7             rStack.append(convertString[n]) #adds string n to the stack.
```

```

8     else:
9         rStack.append(convertString[n % base]) #adds string n modulo base to
the stack.
10        n = n // base
11        res = ""
12        while rStack:
13            #combines all the items in the stack to make the full string.
14            res = res + str(rStack.pop())
15        return res
16
17 print(to_str(1453,16))

```

Activity: 4.6.1 Converting an Integer to a String Using a Stack

Each time we make a call to `to_str`, we push a character on the stack. Returning to the previous example we can see that after the fourth call to `tostr` the stack would look like Figure 5. Notice that now we can simply pop the characters off the stack and concatenate them into the final result, "1010".

每次我们调用 `to_str` 时，都会将一个字符压入栈中。回到前面的例子，我们可以看到，在第四次调用 `to_str` 后，栈的状态如图4.5所示。注意到现在我们可以简单地从栈中弹出字符并将它们连接成最终结果 "1010"。

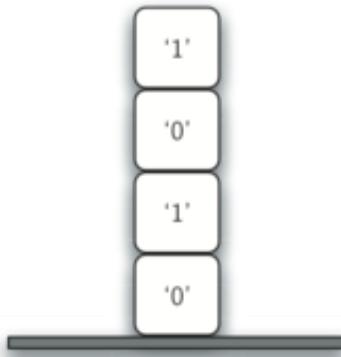


Figure 5: Strings Placed on the Stack During Conversion

The previous example gives us some insight into how Python implements a recursive function call. When a function is called in Python, a **stack frame** is allocated to handle the local variables of the function. When the function returns, the return value is left on top of the stack for the calling function to access. Figure 6 illustrates the call stack after the return statement on line 4.

前面的例子让我们对 Python 如何实现递归函数调用有了一些了解。当在 Python 中调用一个函数时，会分配一个栈帧来处理该函数的局部变量。当函数返回时，返回值会被留在栈顶，供调用函数访问。图6展示了第4行的返回语句后的调用栈。

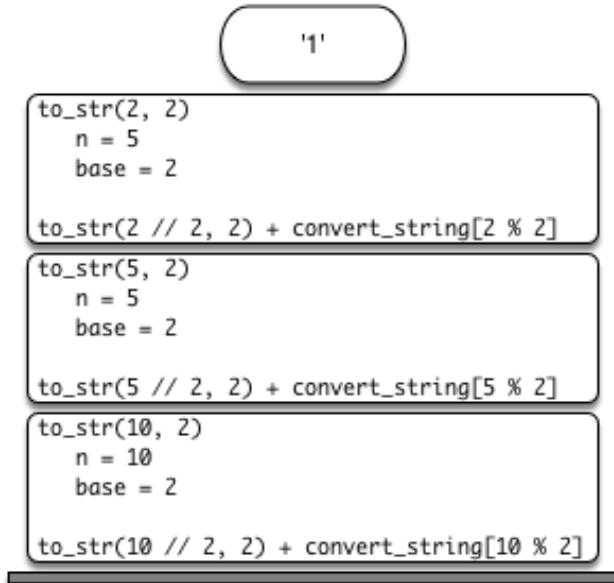


Figure 6: Call Stack Generated from `to_str(10, 2)`

Notice that the call to `to_tr(2//2,2)` leaves a return value of "1" on the stack. This return value is then used in place of the function call (`to_str(1,2)`) in the expression `"1" + convertString[2%2]`, which will leave the string "10" on the top of the stack. In this way, the Python call stack takes the place of the stack we used explicitly in [Listing 4](#). In our list summing example, you can think of the return value on the stack taking the place of an accumulator variable.

请注意，定义的 `to_str(2 // 2, 2)` 调用会在栈上留下返回值 "1"。这个返回值随后在表达式 `"1" + convert_string[2 % 2]` 中代替了函数调用 `to_str(1, 2)`，这将在栈顶留下字符串 "10"。通过这种方式，Python 的调用栈替代了我们在 ActiveCode 4.6.1 中显式使用的栈。在我们的列表求和示例中，你可以认为栈上的返回值替代了一个累加器变量的作用。

The stack frames also provide a scope for the variables used by the function. Even though we are calling the same function over and over, each call creates a new scope for the variables that are local to the function.

栈帧还为函数使用的变量提供了作用域。即使我们反复调用同一个函数，每次调用都会为函数的局部变量创建一个新的作用域。

3.2 函数参数是可变、不可变对象

参数是可变，还是不可变对象，是不一样的，前者在被调用者栈帧中，后者在调用者栈帧中。

在 Python 中，参数传递的方式取决于参数的类型：

1. 不可变对象（如数字、字符串、元组）：在函数调用时，传递的是值的副本。因此，对不可变对象的修改不会影响调用者。
2. 可变对象（如列表、字典）：在函数调用时，传递的是对象的引用。因此，对可变对象的修改会影响到调用者。

不可变对象

对于不可变对象，每次函数调用时传递的是值的副本，因此在被调用者中对参数的修改不会影响调用者。

示例

```
1 def modify_value(x):
2     x = x + 1
3     print(f"Inside modify_value, x = {x}")
4
5 x = 5
6 print(f"Before modify_value, x = {x}")
7 modify_value(x)
8 print(f"After modify_value, x = {x}")
```

输出

```
1 Before modify_value, x = 5
2 Inside modify_value, x = 6
3 After modify_value, x = 5
```

可变对象

对于可变对象，每次函数调用时传递的是对象的引用，因此在被调用者中对参数的修改会影响到调用者。

示例

```
1 def modify_list(lst):
2     lst.append(1)
3     print(f"Inside modify_list, lst = {lst}")
4
5 lst = [2, 3]
6 print(f"Before modify_list, lst = {lst}")
7 modify_list(lst)
8 print(f"After modify_list, lst = {lst}")
```

输出

```
1 Before modify_list, lst = [2, 3]
2 Inside modify_list, lst = [2, 3, 1]
3 After modify_list, lst = [2, 3, 1]
```

递归调用中的回溯

在递归调用中，如果传递的参数是一个可变对象（如列表），在被调用者中对这个对象的修改会影响到调用者。为了保持状态的一致性和正确性，通常需要在递归调用返回后恢复这个对象的状态，这就是**回溯的过程**。

示例

假设我们有一个递归函数 `dfs`，它接受一个列表 `col_selected` 作为参数，并在调用前对 `col_selected` 进行修改。为了确保每次递归调用返回后 `col_selected` 的状态正确恢复，我们需要在递归调用后进行回溯。

```
1 def dfs(col_selected, j, n):
2     col_selected.append(j) # 选择列 j
3     print(f"Inside dfs, col_selected = {col_selected}")
4
5     # 递归调用
6     for next_j in range(j + 1, n):
7         dfs(col_selected, next_j, n)
8
9     col_selected.pop() # 回溯，恢复 col_selected 的状态
10
11 # 初始调用
12 n = 4
13 dfs([], 0, n)
```

输出

```
1 Inside dfs, col_selected = [0]
2 Inside dfs, col_selected = [0, 1]
3 Inside dfs, col_selected = [0, 1, 2]
4 Inside dfs, col_selected = [0, 1, 2, 3]
5 Inside dfs, col_selected = [0, 1, 3]
6 Inside dfs, col_selected = [0, 2]
7 Inside dfs, col_selected = [0, 2, 3]
8 Inside dfs, col_selected = [0, 3]
```

通过在递归调用前进行修改，在递归调用后恢复（即回溯），可以确保每次递归调用返回后，对象的状态是正确的，从而保证算法的正确性。

传递的参数如果是可变对象（如列表），它在内存中是共享的。虽然每个函数调用都有自己的栈帧，但栈帧中存储的只是指向可变对象的引用，而不是对象本身。因此，对可变对象的修改会影响到所有引用该对象的地方。

示例图式

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

Print output (drag lower right corner to resize)

```
Inside dfs, col_selected = [0]
Inside dfs, col_selected = [0, 1]
Inside dfs, col_selected = [0, 1, 2]
Inside dfs, col_selected = [0, 1, 2, 3]
Inside dfs, col_selected = [0, 1, 3]
```

Ads keep this site free
for ad contents. [Make ad-free](#)

Frames Objects

Global frame

dfs	function	dfs(col_selected, j, n)
n	list	[0 1 2 3]

dfs

col_selected	j	0
	n	4
	next_j	1

dfs

col_selected	j	1
	n	4
	next_j	3

dfs

col_selected	j	3
	n	4

可变对象只有一份
被调用者栈帧

4 递归进阶三部曲

递归进阶三部曲：斐波那契数列、汉诺塔，全排列。递归是dfs, dp的基础。

《算法笔记》第8章

深度优先搜索(DFS)

设想我们现在以第一视角身处一个巨大的迷宫当中，没有上帝视角，没有通信设施，更没有热血动漫里的奇迹，有的只是四周长得一样的墙壁。于是，我们只能自己想办法走出去。如果迷失了内心，随便乱走，那么很可能被四周完全相同的景色绕晕在其中，这时只能放弃所谓的侥幸，而去采取下面这种看上去很盲目但实际上会很有效的方法。

以当前所在位置为起点，沿着一条路向前走，当碰到岔道口时，选择其中一个岔路前进如果选择的这个岔路前方是一条死路，就退回到这个岔道口，选择另一个岔路前进。如果岔路中存在新的岔道口，那么仍然按上面的方法枚举新岔道口的每一条岔路。这样，只要迷宫存在出口，那么这个方法一定能够找到它。可能有读者会问，如果在第一个岔道口处选择了一条没有出路的分支，而这个分支比较深，并且路上多次出现新的岔道口，那么当发现这个分支是个死分支之后，如何退回到最初的这个岔道口？其实方法很简单，只要让右手始终贴着右边的墙壁一路往前走，那么自动会执行上面这个走法，并且最终一定能找到出口。图 8-1 即为使用这个方法走一个简单迷宫的示例。

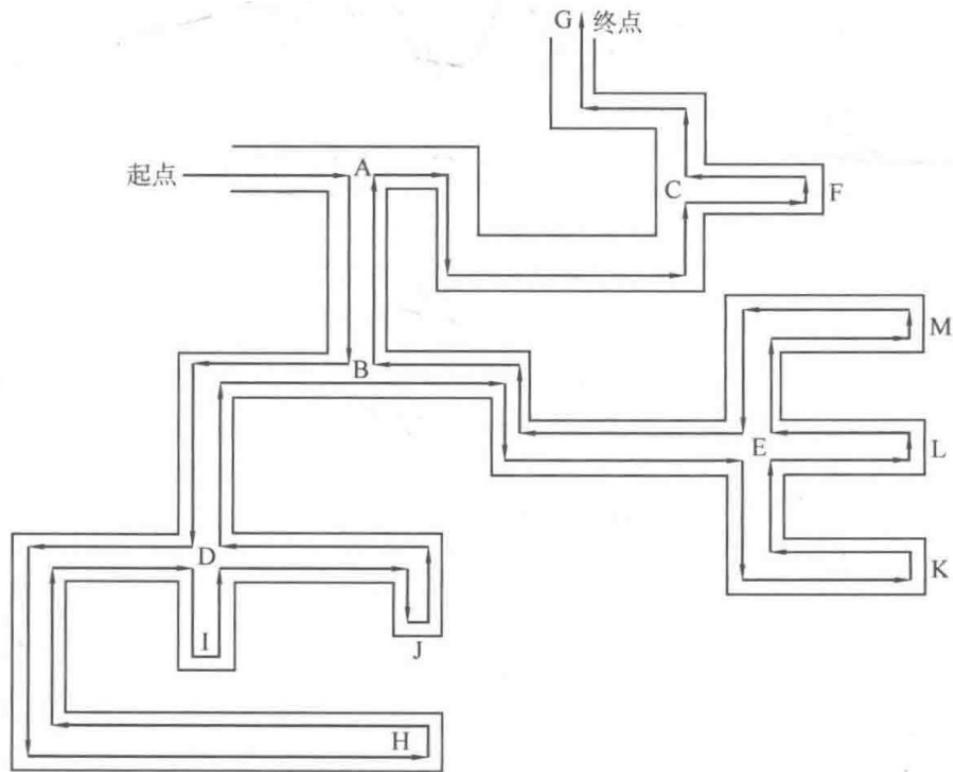


图 8-1 DFS 迷宫示意图

从图 8-1 可知，从起点开始前进，当碰到岔道口时，总是选择其中一条岔路前进（例如图中总是先选择最右手边的岔路），在岔路上如果又遇到新的岔道口，仍然选择新岔道口的其中一条岔路前进，直到碰到死胡同才回退到最近的岔道口选择另一条岔路。也就是说，当碰到岔道口时，总是以“深度”作为前进的关键词，不碰到死胡同就不回头，因此把这种搜索的方式称为深度优先搜索(Depth First Search, DFS)。

从迷宫的例子还应该注意到，深度优先搜索会走遍所有路径，并且每次走到死胡同就代表一条完整路径的形成。这就是说，深度优先搜索是一种枚举所有完整路径以遍历所有情况的搜索方法。

深度优先搜索 (DFS)可以使用栈来实现。但是实现起来却并不轻松，有没有既容易理解又容易实现的方法呢？有的——递归。现在从 DFS 的角度来看当初求解 Fibonacci 数列的过程。

回顾一下 Fibonacci 数列的定义： $F(0) = 1, F(1) = 1, F(n) = F(n - 1) + F(n - 2)$ ($n \geq 2$)。可以从这个定义中挖掘到，每当将 $F(n)$ 分为两部分 $F(n-1)$ 与 $F(n-2)$ 时，就可以把 $F(n)$ 看作迷宫的岔道口，由它可以到达两个新的关键结点 $F(n-1)$ 与 $F(n-2)$ 。而之后计算 $F(n-1)$ 时，又可以把 $F(n-1)$ 当作在岔道口 $F(n)$ 之下的岔道口。

既然有岔道口，那么一定有死胡同。很容易想象，当访问到 $F(0)$ 和 $F(1)$ 时，就无法再向下递归下去，因此 $F(0)$ 和 $F(1)$ 就是死胡同。这样说来，**递归中的递归式就是岔道口，而递归边界就是死胡同**，这样就可以把如何用递归实现深度优先搜索的过程理解得很清楚。为了使上面的过程更清晰，可以直接来分析递归图（见图 4-3）：可以在递归图中看到，只要 $n > 1$ ， $F(n)$ 就有两个分支，即把 $F(n)$ 当作岔道口；而当 n 为 1 或 0 时， $F(1)$ 与 $F(0)$ 就是迷宫的死胡同，在此处程序就需要返回结果。这样当遍历完所有路径（从顶端的 $F(4)$ 到底层的所有 $F(1)$ 与 $F(0)$ ）后，就可以得到 $F(4)$ 的值。

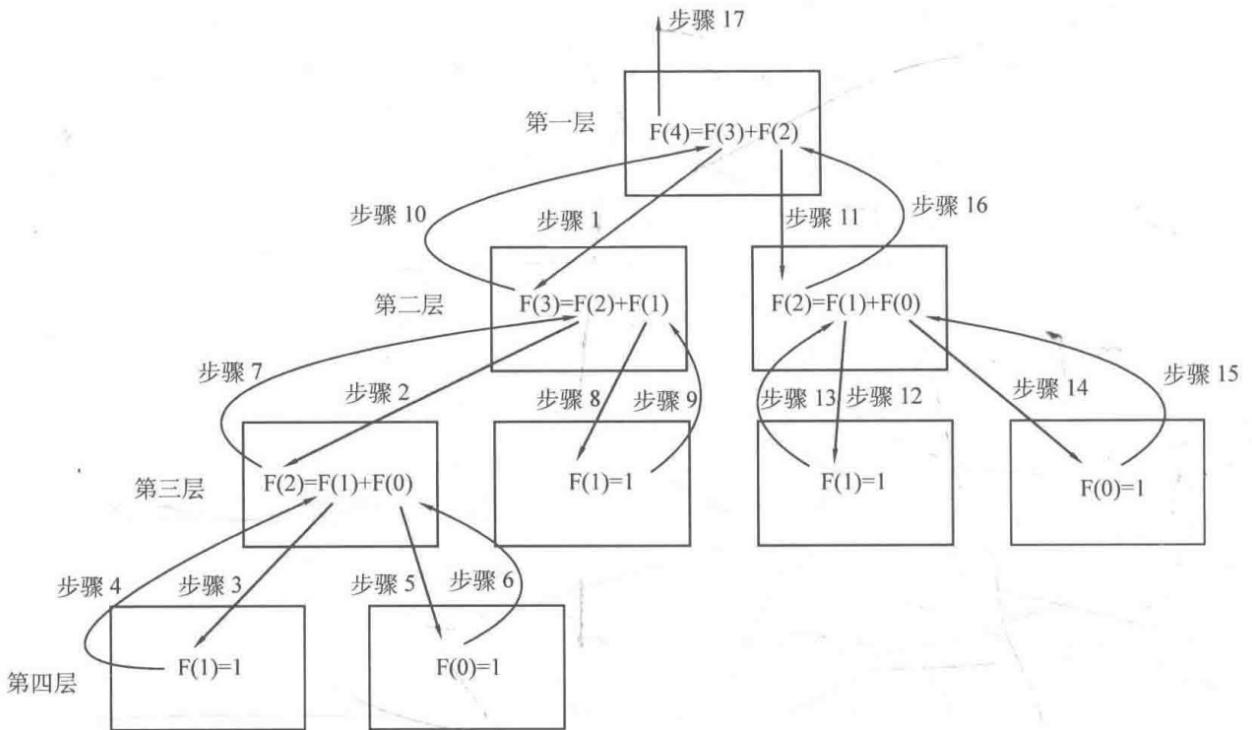


图 4-3 斐波那契数列递归求解示意图

因此，使用递归可以很好地实现深度优先搜索。这个说法并不是说深度优先搜索就是递归，只能说递归是深度优先搜索的一种实现方式，因为使用非递归也是可以实现 DFS 的思想的，但是一般情况下会比递归麻烦。不过，使用递归时，系统会调用一个叫系统栈的东西来存放递归中每一层的状态，因此使用递归来实现 DFS 的本质其实还是栈。

4.1 递归序曲示例：sy115：斐波拉契数列 简单

<https://sunnywhy.com/sfbj/4/3/115>

给定正整数n，求斐波那契数列的第n项F(n)。

令表示斐波那契数列的第n项，它的定义是：

当n=1时， $F(n)=1$ ；

当n=2时， $F(n)=1$ ；

当n>2时， $F(n) = F(n-1) + F(n-2)$ 。

大数据版：[斐波拉契数列-大数据版](#)

输入描述

一个正整数n ($1 \leq n \leq 25$)。

输出描述

斐波那契数列的第n项F(n)。

样例1

输入

```
1 | 1
```

输出

```
1 | 1
```

样例2

输入

```
1 | 3
```

输出

```
1 | 2
```

样例3

输入

```
1 | 5
```

输出

```
1 | 5
```

```
1 def fibonacci(n):
2     if n == 1 or n == 2:
3         return 1
4     else:
5         return fibonacci(n-1) + fibonacci(n-2)
6
7 n = int(input())
8 print(fibonacci(n))
```

4.2 递归三部曲：汉诺塔

<https://runestone.academy/ns/books/published/pythonds3/Recursion/TowerofHanoi.html?mode=browsing>

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three

poles to another, with two important constraints. They could only move one disk at a time, and they could never place a larger disk on top of a smaller one. The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.

汉诺塔谜题是由法国数学家埃杜阿德·卢卡斯于1883年发明的。他受到一个传说的启发，这个传说讲述了一个印度寺庙中的年轻祭司被赋予了这个谜题。在时间的开端，祭司们得到了三根柱子和64个金盘，每个盘子都比它下面的一个稍微小一点。他们的任务是将这64个盘子从一根柱子移动到另一根柱子，但有两个重要的限制：每次只能移动一个盘子，并且不能将较大的盘子放在较小的盘子上面。祭司们非常高效地工作，日以继夜，每秒移动一个盘子。根据传说，当他们完成任务时，寺庙会化为尘土，世界也将消失。

Although the legend is interesting, you need not worry about the world ending any time soon. The number of moves required to correctly move a tower of 64 disks is $2^{64} - 1 = 18,446,744,073,709,551,615$. At a rate of one move per second, that is 584,942,417,355 years! Clearly there is more to this puzzle than meets the eye.

虽然这个传说很有趣，但你不必担心世界会在短时间内终结。正确移动64个盘子所需的步数是 $2^{64} - 1$ 。如果以每秒移动一次的速度来计算，那需要 5.85×10^{11} 年！显然，这个谜题背后有更多的东西值得探索。

Figure 1 shows an example of a configuration of disks in the middle of a move from the first peg to the third. Notice that, as the rules specify, the disks on each peg are stacked so that smaller disks are always on top of the larger disks. If you have not tried to solve this puzzle before, you should try it now. You do not need fancy disks and poles—a pile of books or pieces of paper will work.

图1展示了在从第一根柱子移动到第三根柱子的过程中，盘子配置的一个例子。请注意，正如规则所指定的那样，每根柱子上的盘子都是按照较小的盘子始终在较大盘子之上堆叠的方式。如果你以前没有尝试过解决这个谜题，你现在应该试试。你不需要精美的盘子和柱子——一堆书或几张纸就可以使用。

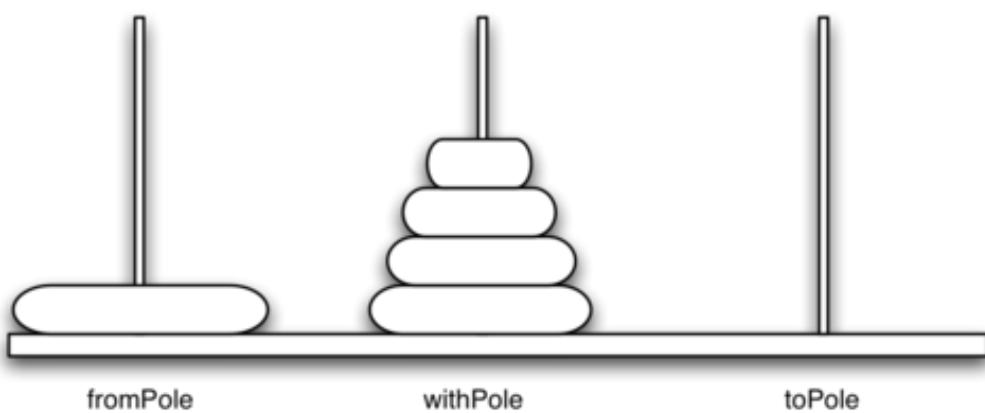


Figure 1: An Example Arrangement of Disks for the Tower of Hanoi

How do we go about solving this problem recursively? How would you go about solving this problem at all? What is our base case? Let's think about this problem from the bottom up. Suppose you have a tower of five disks, originally on peg one. If you already knew how to move a tower of four disks to peg two, you could then easily move the bottom disk to peg three, and then move the tower of four from peg two to peg three. But what if you do not know how to move a tower of height four? Suppose that you knew how to move a tower of height three to peg three; then it would be easy to move the fourth disk to peg two and move the three from peg three on top of it. But what if you do not know how to

move a tower of three? How about moving a tower of two disks to peg two and then moving the third disk to peg three, and then moving the tower of height two on top of it? But what if you still do not know how to do this? Surely you would agree that moving a single disk to peg three is easy enough, trivial you might even say. This sounds like a base case in the making.

我们如何递归地解决这个问题？你将如何着手解决这个问题？我们的基准情形是什么？让我们从底部开始思考这个问题。假设你有一个五层的塔，最初在柱子1上。如果你已经知道如何将四层的塔移动到柱子2上，那么你可以轻松地将最底层的盘子移动到柱子3上，然后再将四层的塔从柱子2移动到柱子3上。但如果你不知道如何移动四层的塔怎么办？假设你知道如何将三层的塔移动到柱子3上；那么很容易将第四层的盘子移动到柱子2上，并将三层的塔移到它上面。但如果你不知道如何移动三层的塔呢？如果将两层的塔移动到柱子2上，然后将第三层的盘子移动到柱子3上，再将两层的塔移到它上面呢？但如果你还是不知道怎么做呢？你肯定会同意，将一个单独的盘子移动到柱子3上是足够简单的，甚至可以说是微不足道的。这听起来像是一个基准情形。

Here is a high-level outline of how to move a tower from the starting pole, to the goal pole, using an intermediate pole:

1. Move a tower of height-1 to an intermediate pole, using the final pole.
2. Move the remaining disk to the final pole.
3. Move the tower of height-1 from the intermediate pole to the final pole using the original pole.

下面是一个高层次的概述，说明如何使用中间柱子将高度为 n 的塔从起始柱子移动到目标柱子：

1. 通过目标柱子将高度为 n-1 的塔从起始柱子移动到中间柱子。
2. 将剩余的一个盘子从起始柱子移动到最终柱子。
3. 通过起始柱子将高度为 n-1 的塔从中间柱子移动到目标柱子。

As long as we always obey the rule that the larger disks remain on the bottom of the stack, we can use the three steps above recursively, treating any larger disks as though they were not even there. The only thing missing from the outline above is the identification of a base case. The simplest Tower of Hanoi problem is a tower of one disk. In this case, we need move only a single disk to its final destination. A tower of one disk will be our base case. In addition, the steps outlined above move us toward the base case by reducing the height of the tower in steps 1 and 3. Listing 1 shows the Python code to solve the Tower of Hanoi puzzle.

只要我们始终遵守较大的盘子保持在栈底的规则，我们可以使用上述三个步骤进行递归处理，就好像较大的盘子不存在一样。上述概述中唯一缺少的是基准情形的识别。最简单的汉诺塔问题是只有一个盘子的塔。在这种情况下，我们只需要将单个盘子移动到它的最终目的地。一个盘子的塔将作为我们的基准情形。此外，上述步骤通过在第1步和第3步中减少塔的高度，使我们逐步接近基准情形。清单1展示了用Python代码解决汉诺塔谜题的方法。

Listing 1

```
1 def moveTower(height,fromPole, toPole, withPole):  
2     if height >= 1:  
3         moveTower(height-1,fromPole,withPole,toPole) #Recursive call  
4         moveDisk(fromPole,toPole)  
5         moveTower(height-1,withPole,toPole,fromPole) #Recursive call
```

Notice that the code in Listing 1 is almost identical to the English description. The key to the simplicity of the algorithm is that we make two different recursive calls, one on line 3 and a second on line 5. On line 3 we move all but the bottom disk on the initial tower to an intermediate pole. The next line simply moves the bottom disk to its final resting place. Then on line 5 we move the tower from the intermediate pole to the top of the largest disk. The base case is detected when the tower height is 0; in this case there is nothing to do, so the `moveTower` function simply returns. The important thing to remember about handling the base case this way is that simply returning from `moveTower` is what finally allows the `moveDisk` function to be called.

请注意，清单1中的代码几乎与英文描述完全相同。算法简单性的关键在于我们进行了两次不同的递归调用，一次在第3行，另一次在第5行。在第3行，我们将初始塔上除了最底层盘子以外的所有盘子移动到中间柱子上。下一行则简单地将最底层的盘子移动到它的最终位置。然后在第5行，我们将中间柱子上的塔移动到最大盘子的顶部。当塔的高度为0时，检测到基准情形；在这种情况下，没有什么需要做的，所以 `moveTower` 函数直接返回。以这种方式处理基准情形的重要之处在于，仅仅是返回 `moveTower` 函数就最终允许了 `moveDisk` 函数被调用。

The function `moveDisk`, shown in Listing 2, is very simple. All it does is print out that it is moving a disk from one pole to another. If you type in and run the `moveTower` program you can see that it gives you a very efficient solution to the puzzle.

清单2中显示的 `moveDisk` 函数非常简单。它所做的只是打印出从一根柱子移动一个盘子到另一根柱子的信息。如果你输入并运行 `moveTower` 程序，你会看到它为你提供了谜题的一个非常高效的解决方案。

Listing 2

```
1 def moveDisk(fp,tp):
2     print("moving disk from",fp,"to",tp)
```

The program in ActiveCode 1 provides the entire solution for three disks.

```
1 #Simulation of the tower of hanoi.
2
3 def moveTower(height,fromPole, toPole, withPole):
4     if height >= 1:
5         moveTower(height-1,fromPole,withPole,toPole) #Recursive call
6         moveDisk(fromPole,toPole)
7         moveTower(height-1,withPole,toPole,fromPole) #Recursive call
8
9 def moveDisk(fp,tp):
10    print("moving disk from",fp,"to",tp)
11
12
13 moveTower(3,"A","B","C")
```

Activity: 5.10.2 Solving Tower of Hanoi Recursively Python (hanoipy)

Now that you have seen the code for both `moveTower` and `moveDisk`, you may be wondering why we do not have a data structure that explicitly keeps track of what disks are on what poles. Here is a hint: if you were going to explicitly keep track of the disks, you would probably use three `Stack` objects, one for each pole. The answer is that Python provides the stacks that we need implicitly through the call stack.

现在你已经看到了 `moveTower` 和 `moveDisk` 的代码，你可能会想知道为什么我们没有一个显式的数据结构来跟踪每个柱子上有哪些盘子。这里有一个提示：如果你打算显式地跟踪这些盘子，你可能会使用三个 `Stack` 对象，每个柱子一个。答案是 Python 通过调用栈隐式地提供了我们需要的栈。

示例: 04147汉诺塔问题(Tower of Hanoi)

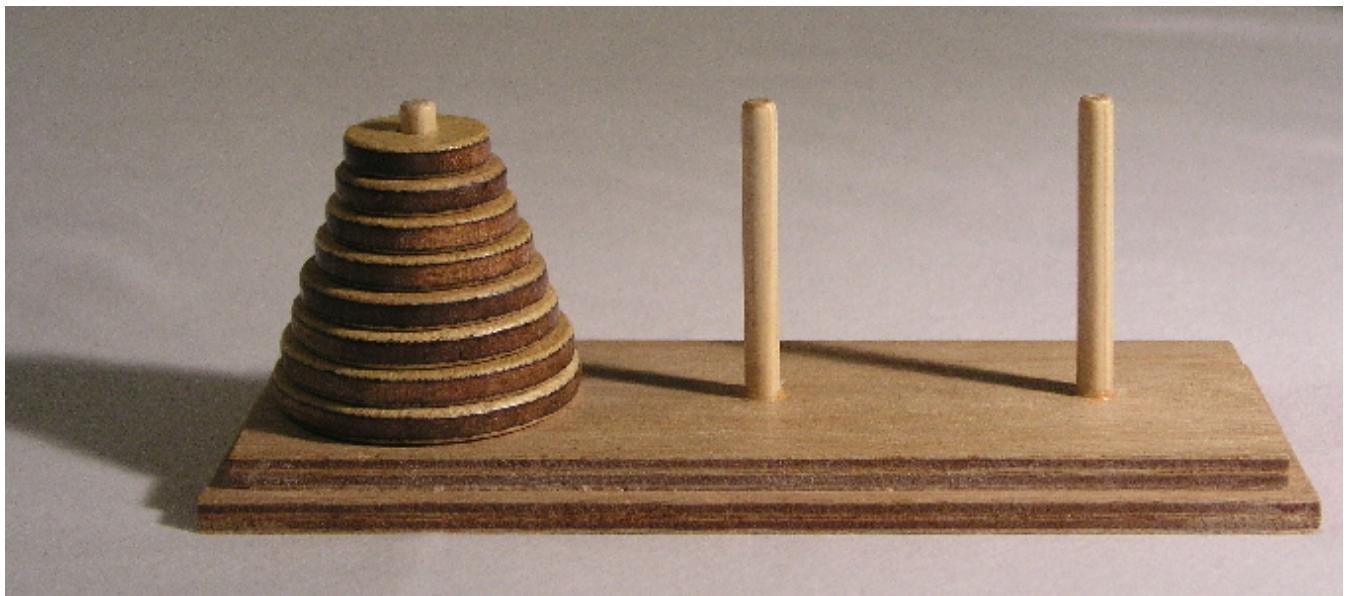
<http://cs101.openjudge.cn/practice/04147>

一、汉诺塔问题

有三根杆子A, B, C。A杆上有N个(N>1)穿孔圆盘，盘的尺寸由下到上依次变小。要求按下列规则将所有圆盘移至C杆：每次只能移动一个圆盘；大盘不能叠在小盘上面。提示：可将圆盘临时置于B杆，也可将从A杆移出的圆盘重新移回A杆，但都必须遵循上述两条规则。

问：如何移？最少要移动多少次？

汉诺塔示意图如下：



三个盘的移动：



二、故事由来

法国数学家爱德华·卢卡斯曾编写过一个印度的古老传说：在世界中心贝拿勒斯（在印度北部）的圣庙里，一块黄铜板上插着三根宝石针。印度教的主神梵天在创造世界的时候，在其中一根针上从下到上地穿好了由大到小的64片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：一次只移动一片，不管在哪根针上，小片必须在大片上面。僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。

不管这个传说的可信度有多大，如果考虑一下把64片金片，由一根针上移到另一根针上，并且始终保持上小下大的顺序。这需要多少次移动呢？这里需要递归的方法。假设有 n 片，移动次数是 $f(n)$ 。显然 $f(1)=1, f(2)=3, f(3)=7$ ，且 $f(k+1)=2*f(k)+1$ 。此后不难证明 $f(n)=2^n-1$ 。 $n=64$ 时，假如每秒钟一次，共需多长时间呢？一个平年365天有31536000秒，闰年366天有31622400秒，平均每年31556952秒，计算一下： 18446744073709551615 秒。这表明移完这些金片需要5845.54亿年以上，而地球存在至今不过45亿年，太阳系的预期寿命据说也就是数百亿年。真的过了5845.54亿年，不说太阳系和银河系，至少地球上的一切生命，连同梵塔、庙宇等，都早已经灰飞烟灭。

三、解法

解法的基本思想是递归。假设有A、B、C三个塔，A塔有 N 块盘，目标是把这些盘全部移到C塔。那么先把A塔顶部的 $N-1$ 块盘移动到B塔，再把A塔剩下的大盘移到C，最后把B塔的 $N-1$ 块盘移到C。每次移动多于一块盘时，则再次使用上述算法来移动。

输入

输入为一个整数后面跟三个单字符字符串。

整数为盘子的数目，后三个字符表示三个杆子的编号。

输出

输出每一步移动盘子的记录。一次移动一行。

每次移动的记录为例如 $3:a\rightarrow b$ 的形式，即把编号为3的盘子从a杆移至b杆。

我们约定圆盘从小到大编号为 $1, 2, \dots, n$ 。即最上面那个最小的圆盘编号为1，最下面最大的圆盘编号为 n 。

样例输入

```
1 | 3 a b c
```

样例输出

```
1 1:a->c
2 2:a->b
3 1:c->b
4 3:a->c
5 1:b->a
6 2:b->c
7 1:a->c
```

提示

可参考如下网址：

<https://www.mathsisfun.com/games/towerofhanoi.html>

<http://blog.csdn.net/geekwangminli/article/details/7981570>

<http://www.cnblogs.com/yanlingyin/archive/2011/11/14/2247594.html>

```
1 # https://blog.csdn.net/geekwangminli/article/details/7981570
2
3 # 将编号为numDisk的盘子从init杆移至desti杆
4 def moveOne(numDisk : int, init : str, desti : str):
5     print("{}:{}->{}".format(numDisk, init, desti))
6
7 #将numDisks个盘子从init杆借助temp杆移至desti杆
8 def move(numDisks : int, init : str, temp : str, desti : str):
9     if numDisks == 1:
10         moveOne(1, init, desti)
11     else:
12         # 首先将上面的 (numDisk-1) 个盘子从init杆借助desti杆移至temp杆
13         move(numDisks-1, init, desti, temp)
14
15         # 然后将编号为numDisks的盘子从init杆移至desti杆
16         moveOne(numDisks, init, desti)
17
18         # 最后将上面的 (numDisks-1) 个盘子从temp杆借助init杆移至desti杆
19         move(numDisks-1, temp, init, desti)
20
21 n, a, b, c = input().split()
22 move(int(n), a, b, c)
```

练习: 01958 Strange Towers of Hanoi (选做)

<http://cs101.openjudge.cn/practice/01958/>

Charlie Darkbrown sits in another one of those boring Computer Science lessons: At the moment the teacher just explains the standard Tower of Hanoi problem, which bores Charlie to death!



Figure 4: The standard (three) Towers of Hanoi.

The teacher points to the blackboard (Fig. 4) and says: "So here is the problem:

- There are three towers: A, B and C.
- There are n disks. The number n is constant while working the puzzle.

- All disks are different in size.
- The disks are initially stacked on tower A increasing in size from the top to the bottom.
- The goal of the puzzle is to transfer all of the disks from tower A to tower C.
- One disk at a time can be moved from the top of a tower either to an empty tower or to a tower with a larger disk on the top.

So your task is to write a program that calculates the smallest number of disk moves necessary to move all the disks from tower A to C."

Charlie: "This is incredibly boring—everybody knows that this can be solved using a simple recursion. I deny to code something as simple as this!"

The teacher sighs: "Well, Charlie, let's think about something for you to do: For you there is a **fourth tower D**. Calculate the smallest number of disk moves to move all the disks from tower A to tower D using all four towers."

Charlie looks irritated: "Urgh. . . Well, I don't know an optimal algorithm for four towers. . . "

Problem

So the real problem is that problem solving does not belong to the things Charlie is good at. Actually, the only thing Charlie is really good at is "sitting next to someone who can do the job". And now guess what — exactly! It is you who is sitting next to Charlie, and he is already glaring at you.

Luckily, you know that the following algorithm works for $n \leq 12$: At first $k \geq 1$ disks on tower A are fixed and the remaining $n-k$ disks are moved from tower A to tower B using the algorithm for four towers. Then the remaining k disks from tower A are moved to tower D using the algorithm for three towers. At last the $n - k$ disks from tower B are moved to tower D again using the algorithm for four towers (and thereby not moving any of the k disks already on tower D). Do this for all $k \in \{1, \dots, n\}$ and find the k with the minimal number of moves.

So for $n = 3$ and $k = 2$ you would first move 1 (3-2) disk from tower A to tower B using the algorithm for four towers (one move). Then you would move the remaining two disks from tower A to tower D using the algorithm for three towers (three moves). And the last step would be to move the disk from tower B to tower D using again the algorithm for four towers (another move). Thus the solution for $n = 3$ and $k = 2$ is 5 moves. To be sure that this really is the best solution for $n = 3$ you need to check the other possible values 1 and 3 for k . (But, by the way, 5 is optimal. . .)

输入

There is no input.

输出

For each n ($1 \leq n \leq 12$) print a single line containing the minimum number of moves to solve the problem for four towers and n disks.

样例输入

```
1 | No input.
```

样例输出

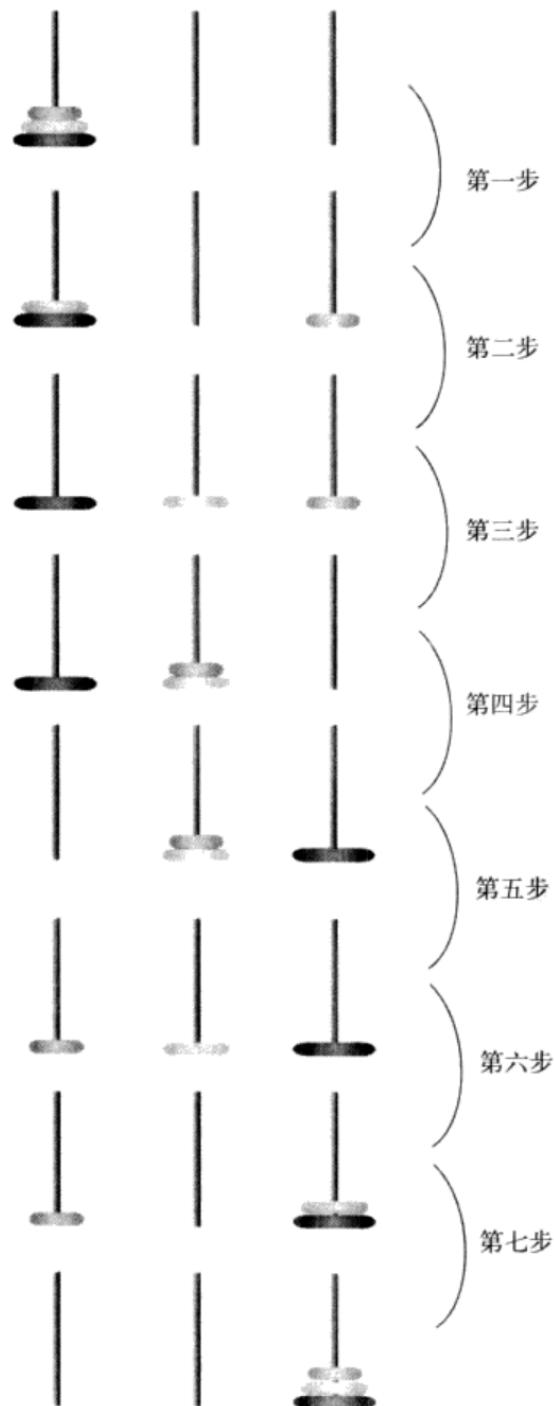
```
1 | REFER TO OUTPUT.
```

来源

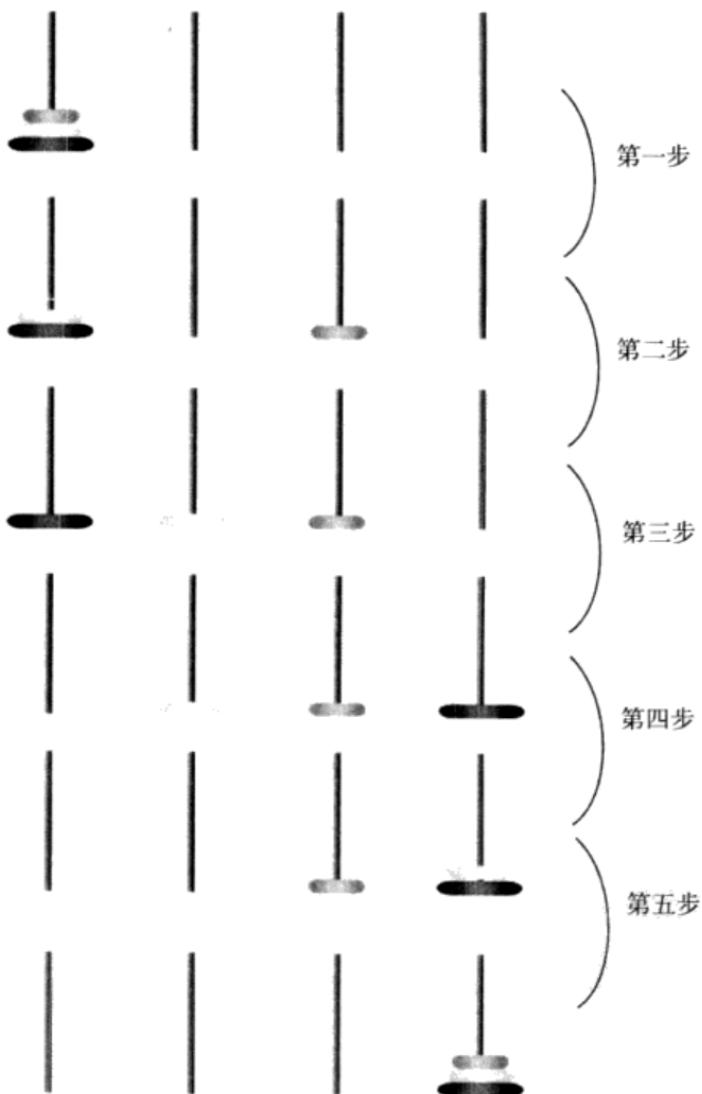
TUD Programming Contest 2002, Darmstadt, Germany

《短码之美》2007年，184页

汉诺塔，大家知道吗？汉诺塔由3根柱子、大小不同的空心圆盘组成。所有圆盘最初都放在最左边的柱子上。圆盘的摆放规则是上面的圆盘必须小于下面的圆盘。把这些圆盘一个一个都移动到最右边的柱子上，如果圆盘的个数是n，大家都知道一般需要移动 $(2^n - 1)$ 次。比如，n=3的时候，



的确是用了 $2^3 - 1 = 7$ 次完成了移动。那么，这次的问题不是基本的汉诺塔，而是把柱子的根数增加1根。如果柱子增加到 4 根，原来需要移动 7 次完成，现在只需要 5 次就可以了。

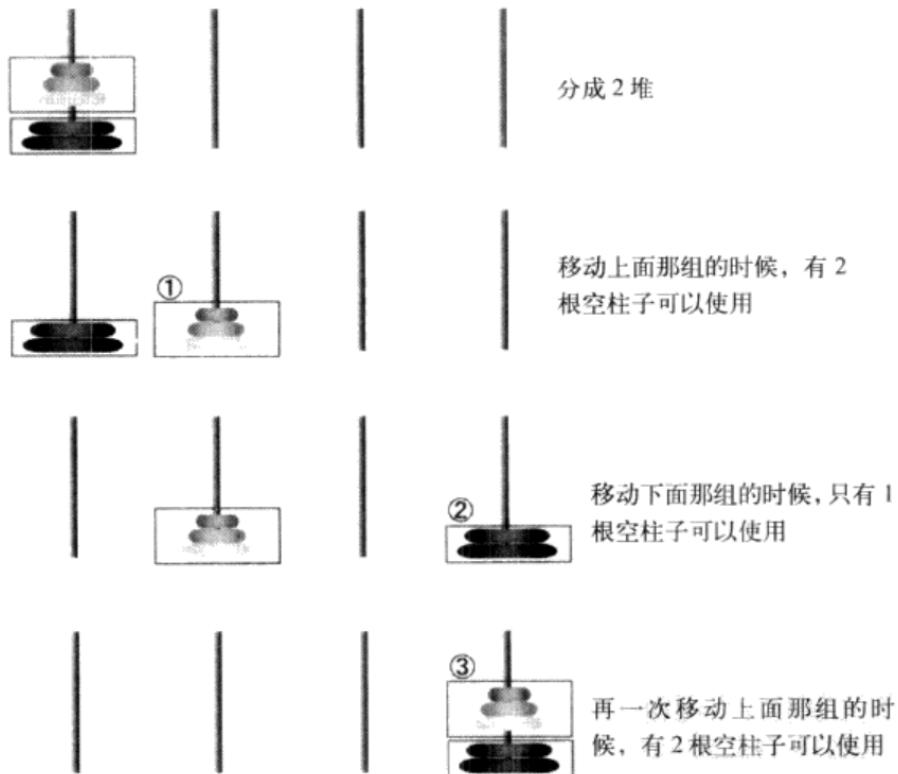


如果增加圆盘个数，就应该能省下更多的步数，但是这个规则还不是很清楚。题面要求编写程序计算 4 根柱子的时候，1~12 个盘子所需的最小移动次数。

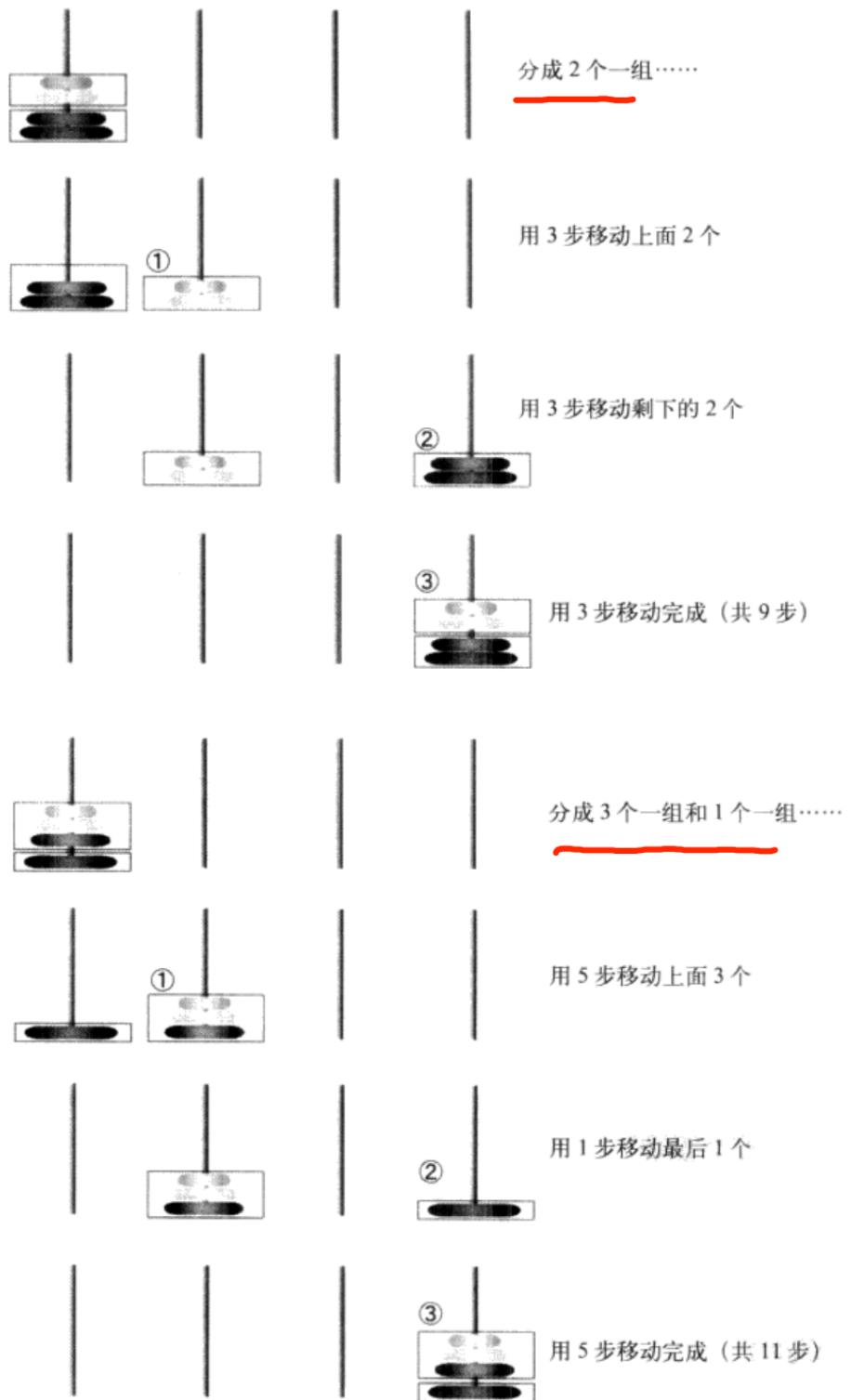
有 4 根柱子的时候，可以利用 2 根空的柱子移动圆盘，圆盘数 n 是 1、2、3 的时候只需顺序移动，所以各需要 1、3、5 次移动。4 个圆盘以上：

- (1) 首先移动其中的几个盘子；
- (2) 把剩余的圆盘移动到指定的位置；
- (3) 把 (1) 的圆盘移动到 (2) 的上面。

这个时候，(1) 和 (3) 可以有 2 根空柱子可以使用，所以可以互换，但是 (2) 的时候只有一根空柱子。也就是说移动所需的步数与一般汉诺塔 (3 根柱子) 相同。



具体地用 4个圆盘来考虑一下，如下图所示。4个圆盘的时候，①可移动2个圆盘(3步)，②可移动2个圆盘(3步)，③再移动2个圆盘(3步)，总共最少需要 9步。如果①移动3个的时候，则需要 5步，②只移动一个需要 1步，③再移动3个需要 5步，总共需要 11步，不是最小的移动步数。但是，①只移动1个的话需要 1步，②只移动3个需7步，③再移动1个需要1步，总共需要 9步，这才是最小步数。**在什么情况下移动步数最小不太容易看出来**，所以要像这样把 n 个圆盘分成 k 个和 $(n-k)$ 个来检查移动步数，找出最小移动步数的移法。



圆盘个数增加后需要增加移动步数，如果每次都计算将是很庞大的计算量，所以需要使用DP(Dynamic Programming, 动态规划法)求解。

```

1 d = [0] * 15
2 f = [float('inf')] * 15
3
4 d[1] = 1
5 for i in range(2, 13):
6     d[i] = d[i - 1] * 2 + 1
7

```

```

8 f[1] = 1
9 for i in range(2, 13):
10    for j in range(1, i):
11        f[i] = min(f[i], f[i - j] * 2 + d[j])
12
13 for i in range(1, 13):
14     print(f[i])

```

```

1 # 23n2300011072, 蒋子轩
2 def hanoi_four_towers(n, source, target, auxiliary1, auxiliary2):
3     if n == 0:
4         return 0
5     if n == 1:
6         return 1
7     min_moves = float('inf')
8     for k in range(1, n):
9         three_tower_moves = 2***(n-k)-1
10        moves = hanoi_four_towers(k, source, auxiliary1, auxiliary2, target) + \
11            three_tower_moves +
12            hanoi_four_towers(k, auxiliary1, target, source, auxiliary2)
13        min_moves = min(min_moves, moves)
14    return min_moves
15
16
17 for n in range(1, 13):
18     print(hanoi_four_towers(n, 'A', 'D', 'B', 'C'))

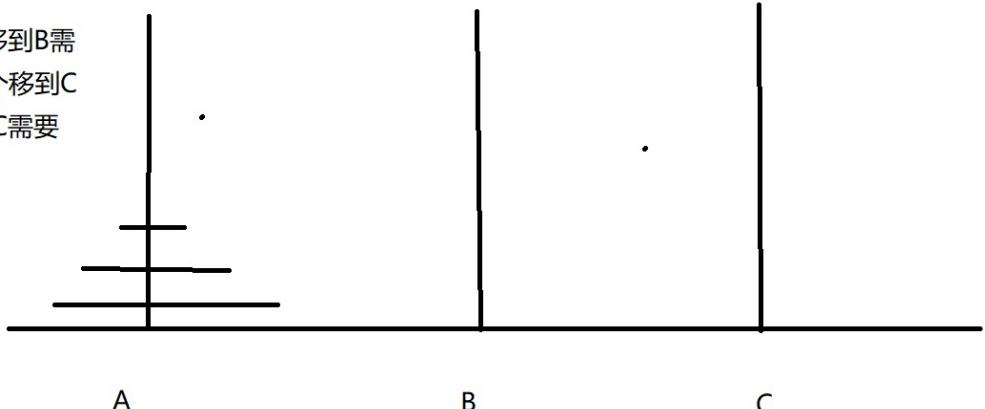
```

POJ - 1958 Strange Towers of Hanoi 汉诺塔递推问题（4塔）,

https://blog.csdn.net/qq_45432665/article/details/104825847

思路：我们先将3塔的情况递推出来，用d[i] 表示有i个盘的时候的最小移动次数， $d[1] = 1$

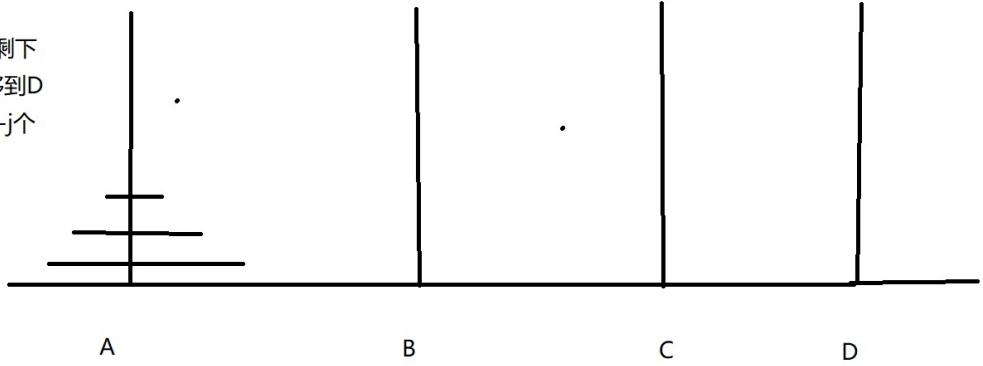
假设是i个，将前*i-1*个移到B需要
 $d[i-1]$ 次，把最后一个移到C
需要1次，再把*i-1*移到C需要
 $d[i-1]$ 次，
 $d[i] = d[i-1]*2+1$



当有4塔时，也是一样的思路， $f[1] = 1$

https://blog.csdn.net/qq_45432665

$f[i]$ 的前 $i - j$ 个移到B上面, 剩下的 j 个就是3塔问题, 把 j 个移到D上面就需要 $d[j]$ 次, 最后把 $i - j$ 个再移到D上需要 $f[i-j]$ 次,
 $f[i] = f[i-j]*2 + d[j]$



https://blog.csdn.net/qq_45432665

4 柱汉诺塔游戏是否已经解决了?

<https://www.zhihu.com/question/54353032>

4 柱汉诺塔游戏是否已经解决了?

- 三柱=起始柱+缓存柱+目标柱
- 四柱=起始柱+缓存柱+缓存柱的缓存柱+目标柱

也就是: $F[n] = F[x] + 2^{n-x} - 1 + F[x]$

那就naive了, 因为这样并没有最大化的利用缓存, 实际上

- 四柱=起始柱+主缓存柱+副缓存柱+目标柱

你要比较所有的缓存方案

$$F[n] = \min_{1 < x < n} 2 * F[x] + 2^{n-x} - 1$$

- 问题到这里可以说是已经解决了.

等等, 还没完呢, $O(n^2)$ 的时间复杂度太高了.

从上面的过程我们可以看到过程能写成:

$$a_n = a_{n-1} + 2^{f(x)}; a_1 = 1$$

$f(x)$ 是满足方程 $n - 1 < \frac{1}{2}x(x + 1)$ 的最小正整数解.

$$\text{于是 } f(x) = \left\lfloor \frac{\sqrt{8n-7}-1}{2} \right\rfloor = \lfloor \sqrt{2n} \rfloor - 1$$

此时使用递推式计算时间复杂度降低到了 $O(n)$.

- 能不能做得更好?

当然可以, Poole 在1994年解出了这个公式:

$$a_n = 1 + \sum 2^{f(x)} = 1 - 2^{t-2}(t^2 - 3t - 2n + 4), \text{ 其中 } t = \lfloor \sqrt{2n} \rfloor$$

```
In[1]:= Clear["`*"]
クリア
dp[n_] := dp[n] = Min[Table[2*dp[k] + 2^(n-k)-1, {k, 0, n-1}]];
dp[1] = 1;
dt[n_] := dt[n] = dt[n-1] + 2^(Round@Sqrt[2 n]-1);
dt[1] := 1;
```

4.3 递归三部曲：全排列

示例sy132: 全排列I 中等

<https://sunnywhy.com/sfbj/4/3/132>

给定一个正整数n，假设序列S=[1,2,3,...,n]，求S的全排列。

输入描述

一个正整数n ($1 \leq n \leq 8$)。

输出描述

每个全排列一行，输出所有全排列。

输出顺序为：两个全排列A和B，若满足前k-1项对应相同，但有 $A_k < B_k$ ，那么将全排列Ak优先输出（例如[1,2,3]比[1,3,2]优先输出）。

在输出时，全排列中的每个数之间用一个空格隔开，行末不允许有多余的空格。不允许出现相同的全排列。

样例1

输入

```
1 | 1
```

输出

```
1 | 1
```

样例2

输入

```
1 | 2
```

输出

```
1 | 1 2  
2 | 2 1
```

样例3

输入

```
1 | 3
```

输出

```
1 1 2 3  
2 1 3 2  
3 2 1 3  
4 2 3 1  
5 3 1 2  
6 3 2 1
```

```
1 maxn = 11  
2 hashTable = [False] * maxn # 当整数i已经在数组 P中时为 true  
3  
4 #@recviz  
5 def increasing_permutaions(n, prefix=[]):  
6     if len(prefix) == n: # 递归边界, 已经处理完排列的1~位  
7         return [prefix]  
8  
9     result = []  
10    for i in range(1, n + 1):  
11        if hashTable[i]:  
12            continue  
13  
14        hashTable[i] = True # 记i已在prefix中  
15        # 把i加入当前排列, 处理排列的后续号位  
16        result += increasing_permutaions(n, prefix + [i])  
17        hashTable[i] = False # 处理完为i的子问题, 还原状态  
18  
19    return result  
20  
21  
22 n = int(input())  
23 result = increasing_permutaions(n)  
24 for r in result:  
25     print(' '.join(map(str,r)))
```

5. 计算机原理：虚拟地址空间

自 20 世纪 40 年代以来，计算机的基础架构已逐渐形成标准，包括处理器、用于存储指令和数据的内存、以及输入输出设备。这一架构通常称为冯·诺依曼架构（Von Neumann Architecture），以数学家与计算机科学家约翰·冯·诺依曼（John von Neumann，1903 年 12 月 28 日 – 1957 年 2 月 8 日）的名字命名。他在 1946 年发表的论文中首次系统描述了这种架构。论文开篇用现代术语来解释，就是：**CPU** 负责算法和控制，**RAM** 与磁盘承担数据与指令存储，而键盘、鼠标、显示器等则与操作人员交互。

在这一架构中，与存储相关的进程的虚拟地址空间是需要重点理解的部分。

在《深入理解计算机系统》第一章中介绍到，**虚拟存储器**（Virtual Memory）是一种抽象机制，它为每个进程提供了一个假象——仿佛自己独占全部主存。实际上，所有进程都看到相同且连续的内存布局，这个抽象的内存视图称为**虚拟地址空间**。

如图 1-16 所示，是一个典型 Linux 进程的虚拟地址空间（其他 Unix 系统类似）。在 Linux 中，最高四分之一的地址空间保留给内核代码与数据，这对所有进程都一样；其余四分之三则分配给用户进程的代码与数据。需要注意的是，图中的内存地址是自下而上递增的。

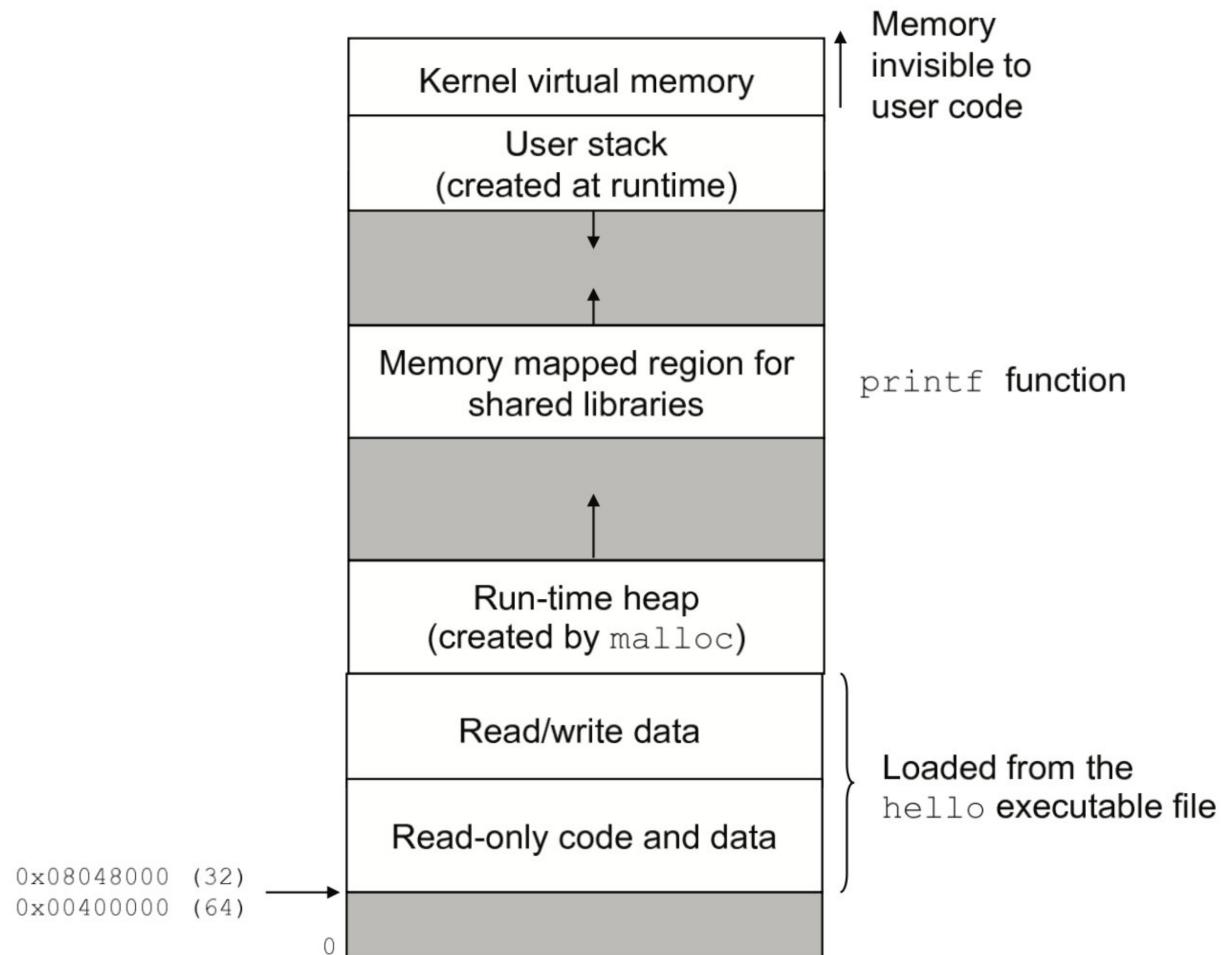


图1-16 进程的虚拟地址空间 (Process virtual address space) (注：图片来源为 Randal Bryant[8], 2015年3月)

每个进程的虚拟地址空间由一系列功能明确的区域（area）构成。按照地址从低到高，大致可以分为以下几个部分：

1. 程序代码与数据 (Code and Data)

程序代码从固定地址开始，紧接其后的数据区存放全局变量等。它们由可执行文件直接初始化，例如示例程序 `hello` 的可执行文件。

2. 堆 (Heap)

位于代码和数据区之后，是运行时堆（Run-time Heap）。与启动时大小固定的代码与数据区不同，堆的大小可在程序运行过程中动态变化，例如通过 C 标准库函数 `malloc` 和 `free` 来分配或释放内存。

3. 共享库 (Shared Libraries)

位于地址空间中部，用于存放共享库（如标准 C 库、数学库等）的代码与数据。这一机制允许多个进程

共享相同的库文件，从而节省内存并便于更新。

4. 栈 (Stack)

位于用户虚拟地址空间顶部，用于函数调用与局部变量存储。与堆一样，用户栈 (User Stack) 在程序执行时可动态扩展或收缩——函数调用时栈增长，函数返回时栈缩小。

5. 内核虚拟存储器 (Kernel Virtual Memory)

占据地址空间最顶端，存放内核常驻代码和数据。用户程序不能直接访问这一区域，也不能调用内核定义的函数。

虚拟存储器的实现依赖于硬件与操作系统的紧密协作，包括对处理器生成的每一个地址进行硬件级翻译。核心思想是：将进程的虚拟内存内容保存在磁盘上，并利用主存作为磁盘的高速缓存，从而在保证进程隔离的同时提高访问效率。

6. 编程题目

练习28717: 递归比较字符串大小

<http://wjjc.openjudge.cn/2024jgc4/002/>

程序填空，完成按奇异规则比较字符串大小的递归函数strCmp(a,b)，返回值为True或False，表示a是否小于b。

不可使用循环，只能使用递归。

字符串按奇异规则比较大小，就是逐个字符比较大小直到分出胜负。两个字符比较大小的规则是哪个字符的编码和字母'k'的编码的差的绝对值小，哪个字符就算小。这样两个不同字符可能也算一样大。

字符串很短，不用考虑效率问题。

```
1 def strCmp(a,b) :
2     if a == "" and b != "":
3         return True
4     elif a != "" and b == "":
5         return False
6     elif a == "" and b == "":
7         return False
8     else :
9         if abs(ord(a[0]) - ord('k')) < abs(ord(b[0]) - ord('k')):#abs是求绝对值的函数
10             return True
11 // 在此处补充你的代码
12 #填空
13
14
15 n = int(input())
16 for _ in range(n):
17     s1,s2 = input().split()
18     if strCmp(s1,s2):
19         print("YES")
20     else:
```

```
21 |     print("NO")
```

输入

第一行是整数n，表示接下来有n对字符串

接下来有n行，每行有用空格分隔的两个字符串，字符串由小写英文字母组成

输出

对没对字符串，如果第一个小于第二个，输出YES，否则输出NO

样例输入

```
1 | 4
2 | ebc eab
3 | ac acd
4 | kk ki
5 | abc abc
```

样例输出

```
1 | YES
2 | YES
3 | YES
4 | NO
```

来源

Guo Wei

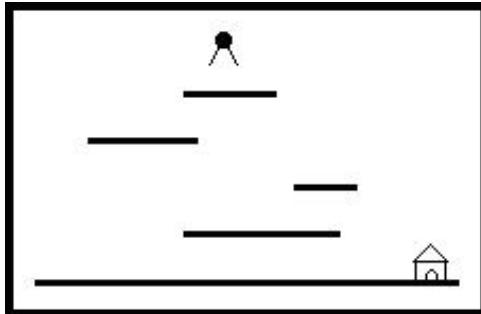
```
1 | def strCmp(a, b):
2 |     if a == "" and b != "":
3 |         return True
4 |     elif a != "" and b == "":
5 |         return False
6 |     elif a == "" and b == "":
7 |         return False
8 |     else:
9 |         if abs(ord(a[0]) - ord('k')) < abs(ord(b[0]) - ord('k')):
10 |             return True
11 |         elif abs(ord(a[0]) - ord('k')) > abs(ord(b[0]) - ord('k')):
12 |             return False
13 |         else:
14 |             return strCmp(a[1:], b[1:])
15 |
16 | if __name__ == "__main__":
17 |     n = int(input())
18 |     for _ in range(n):
19 |         s1, s2 = input().split()
20 |         if strCmp(s1, s2):
```

```
21         print("YES")
22     else:
23         print("NO")
```

练习01661: Help Jimmy (选做)

dfs/dp, <http://cs101.openjudge.cn/practice/01661>

"Help Jimmy" 是在下图所示的场景上完成的游戏：



场景中包括多个长度和高度各不相同的平台。地面是最低的平台，高度为零，长度无限。

Jimmy老鼠在时刻0从高于所有平台的某处开始下落，它的下落速度始终为1米/秒。当Jimmy落到某个平台上时，游戏者选择让它向左还是向右跑，它跑动的速度也是1米/秒。当Jimmy跑到平台的边缘时，开始继续下落。Jimmy每次下落的高度不能超过MAX米，不然就会摔死，游戏也会结束。

设计一个程序，计算Jimmy到底地面时可能的最早时间。

输入

第一行是测试数据的组数t ($0 \leq t \leq 20$)。每组测试数据的第一行是四个整数N, X, Y, MAX, 用空格分隔。N是平台的数目（不包括地面），X和Y是Jimmy开始下落的位置的横竖坐标，MAX是一次下落的最大高度。接下来的N行每行描述一个平台，包括三个整数，X1[i], X2[i]和H[i]。H[i]表示平台的高度，X1[i]和X2[i]表示平台左右端点的横坐标。 $1 \leq N \leq 1000$, $-20000 \leq X, X1[i], X2[i] \leq 20000$, $0 < H[i] < Y \leq 20000$ ($i = 1..N$)。所有坐标的单位都是米。

Jimmy的大小和平台的厚度均忽略不计。如果Jimmy恰好落在某个平台的边缘，被视为落在平台上。所有的平台均不重叠或相连。测试数据保证问题一定有解。

输出

对输入的每组测试数据，输出一个整数，Jimmy到底地面时可能的最早时间。

样例输入

```
1 1
2 3 8 17 20
3 0 10 8
4 0 10 13
5 4 14 3
```

样例输出

来源：POJ Monthly--2004.05.15, CEOI 2000, POJ 1661, 程序设计实习2007

```

1 # 查达闻 2300011813
2 from functools import lru_cache
3
4 @lru_cache
5 def dfs(x, y, z):
6     for i in range(z+1, N+1):
7         if y - MaxVal > p[i][2]:
8             return 1 << 30
9         elif p[i][0] <= x <= p[i][1]:
10            left = x - p[i][0] + dfs(p[i][0], p[i][2], i)
11            right = p[i][1] - x + dfs(p[i][1], p[i][2], i)
12            return min(left,right)
13
14     if y <= MaxVal:
15         return 0
16     else:
17         return 1 << 30
18
19
20 for _ in range(int(input())):
21     N, ini_x, ini_y, MaxVal = map(int, input().split())
22
23     p = []      #platform
24     p.append([0, 0, 1 << 30]) # 1<<30 大于 20000*2*1000
25     for _ in range(N):
26         p.append([int(x) for x in input().split()])
27     p.sort(key = lambda x:-x[2])
28
29     print(ini_y + dfs(ini_x, ini_y, 0))

```

1 << 30

1,073,741,824

练习02386: Lake Counting

dfs similar, <http://cs101.openjudge.cn/practice/02386>

Due to recent rains, water has pooled in various places in Farmer John's field, which is represented by a rectangle of $N \times M$ ($1 \leq N \leq 100$; $1 \leq M \leq 100$) squares. Each square contains either water ('W') or dry land ('.'). Farmer John would like to figure out how many ponds have formed in his field. A pond is a connected set of squares with water in them, where a square is considered adjacent to all eight of its neighbors.

Given a diagram of Farmer John's field, determine how many ponds he has.

输入

* Line 1: Two space-separated integers: N and M

* Lines 2..N+1: M characters per line representing one row of Farmer John's field. Each character is either 'W' or '.'. The characters do not have spaces between them.

输出

* Line 1: The number of ponds in Farmer John's field.

样例输入

```
1 10 12
2 W.....WW.
3 .WWW.....WWW
4 ....WW....WW.
5 .....WW..
6 .....W..
7 ..W.....W..
8 ..W.W.....WW.
9 W.W.W.....W.
10 ..W.W.....W.
11 ..W.....W.
```

样例输出

```
1 | 3
```

提示

OUTPUT DETAILS:

There are three ponds: one in the upper left, one in the lower left, and one along the right side.

来源: USACO 2004 November

```
1 #1.dfs
2 import sys
3 sys.setrecursionlimit(20000)
4 def dfs(x,y):
5     #标记，避免再次访问
6     field[x][y]='.'
7     for k in range(8):
8         nx,ny=x+dx[k],y+dy[k]
9         #范围内且未访问的lake
10        if 0<=nx<n and 0<=ny<m \
11            and field[nx][ny]=='W':
12                #继续搜索
```

```

13     dfs(nx,ny)
14 n,m=map(int,input().split())
15 field=[list(input()) for _ in range(n)]
16 cnt=0
17 dx=[-1,-1,-1,0,0,1,1,1]
18 dy=[-1,0,1,-1,1,-1,0,1]
19 for i in range(n):
20     for j in range(m):
21         if field[i][j]=='W':
22             dfs(i,j)
23             cnt+=1
24 print(cnt)

```

练习05585: 晶矿的个数

matrices/dfs similar, <http://cs101.openjudge.cn/practice/05585>

在某个区域发现了一些晶矿，已经探明这些晶矿总共有分为两类，为红晶矿和黑晶矿。现在要统计该区域内红晶矿和黑晶矿的个数。假设可以用二维地图m[][]来描述该区域，若m[i][j]为#表示该地点是非晶矿地点，若m[i][j]为r表示该地点是红晶矿地点，若m[i][j]为b表示该地点是黑晶矿地点。一个晶矿是由相同类型的并且上下左右相通的晶矿点组成。现在给你该区域的地图，求红晶矿和黑晶矿的个数。

输入

第一行为k，表示有k组测试输入。

每组第一行为n，表示该区域由n*n个地点组成， $3 \leq n \leq 30$

接下来n行，每行n个字符，表示该地点的类型。

输出

对每组测试数据输出一行，每行两个数字分别是红晶矿和黑晶矿的个数，一个空格隔开。

样例输入

```

1 2
2 6
3 r##bb#
4 ###b##
5 #r##b#
6 #r##b#
7 #r#####
8 #####
9 4
10 ####
11 #rrb
12 #rr#
13 ##bb

```

样例输出

1	2 2
2	1 2

```
1 dire = [[-1,0], [1,0], [0,-1], [0,1]]
2
3 def dfs(x, y, c):
4     m[x][y] = '#'
5     for i in range(len(dire)):
6         tx = x + dire[i][0]
7         ty = y + dire[i][1]
8         if m[tx][ty] == c:
9             dfs(tx, ty, c)
10
11 for _ in range(int(input())):
12     n = int(input())
13     m = [[0 for _ in range(n+2)] for _ in range(n+2)]
14
15     for i in range(1, n+1):
16         m[i][1:-1] = input()
17
18     r = 0 ; b=0
19     for i in range(1, n+1):
20         for j in range(1, n+1):
21             if m[i][j] == 'r':
22                 dfs(i, j, 'r')
23                 r += 1
24             if m[i][j] == 'b':
25                 dfs(i,j,'b')
26                 b += 1
27     print(r, b)
```

练习02786: Pell数列

dp, <http://cs101.openjudge.cn/practice/02786/>

Pell数列 a_1, a_2, a_3, \dots 的定义是这样的， $a_1 = 1, a_2 = 2, \dots, a_n = 2 * a_{n-1} + a_{n-2}$ ($n > 2$)。给出一个正整数 k ，要求Pell数列的第 k 项模上32767是多少。

输入

第1行是测试数据的组数 n ，后面跟着 n 行输入。每组测试数据占1行，包括一个正整数 k ($1 \leq k < 1000000$)。

输出

n 行，每行输出对应一个输入。输出应是一个非负整数。

样例输入

```
1 | 2  
2 | 1  
3 | 8
```

样例输出

```
1 | 1  
2 | 408
```

```
1 #2300011786 裴思远  
2 from functools import lru_cache  
3  
4 @lru_cache(maxsize=None)  
5 def series(n):  
6     if n>2:  
7         return (series(n-1)*2+series(n-2))%32767  
8     elif n==2:  
9         return 2  
10    else:  
11        return 1  
12  
13 n=int(input())  
14 for _ in range(n):  
15     k=int(input())%150  
16     ans=series(k)  
17     print(ans)
```

练习02754: 八皇后

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

描述：会下国际象棋的人都很清楚：皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将8个皇后放在棋盘上（有 8×8 个方格），使它们谁也不能被吃掉！这就是著名的八皇后问题。

对于某个满足要求的8皇后的摆放方法，定义一个皇后串 a 与之对应，即 $a = b_1 b_2 \dots b_8$ ，其中 b_i 为相应摆法中第 i 行皇后所处的列数。已经知道8皇后问题一共有92组解（即92个不同的皇后串）。

给出一个数 b ，要求输出第 b 个串。串的比较是这样的：皇后串 x 置于皇后串 y 之前，当且仅当将 x 视为整数时比 y 小。

八皇后是一个古老的经典问题：如何在一张国际象棋的棋盘上，摆放8个皇后，使其任意两个皇后互相不受攻击。该问题由一位德国国际象棋排局家 Max Bezzel 于 1848年提出。严格来说，那个年代，还没有“德国”这个国家，彼时称作“普鲁士”。1850年，Franz Nauck 给出了第一个解，并将其扩展成了“n皇后”问题，即在一张 $n \times n$ 的棋盘上，如何摆放 n 个皇后，使其两两互不攻击。历史上，八皇后问题曾惊动过“数学王子”高斯(Gauss)，而且正是 Franz Nauck 写信找高斯请教的。

输入

第1行是测试数据的组数n，后面跟着n行输入。每组测试数据占1行，包括一个正整数b($1 \leq b \leq 92$)

输出

输出有n行，每行输出对应一个输入。输出应是一个正整数，是对于b的皇后串。

样例输入

1	2
2	1
3	92

样例输出

1	15863724
2	84136275

这里在记录解的时候，不能直接引用数组，否则最终解集中的解都是重复的，要进行拷贝，另外开辟出一个数组空间用解集记录。

八皇后思路：回溯算法通过尝试不同的选择，逐步构建解决方案，并在达到某个条件时进行回溯，以找到所有的解决方案。从第一行第一列开始放置皇后，然后在每一行的不同列都放置，如果与前面不冲突就继续，有冲突则回到上一行继续下一个可能性。

```
1 def solve_n_queens(n):
2     solutions = [] # 存储所有解决方案的列表
3     queens = [-1] * n # 存储每一行皇后所在的列数
4
5     def backtrack(row):
6         if row == n: # 找到一个合法解决方案
7             solutions.append(queens.copy())
8         else:
9             for col in range(n):
10                 if is_valid(row, col): # 检查当前位置是否合法
11                     queens[row] = col # 在当前行放置皇后
12                     backtrack(row + 1) # 递归处理下一行
13                     queens[row] = -1 # 回溯，撤销当前行的选择
14
15     def is_valid(row, col):
16         for r in range(row):
17             if queens[r] == col or abs(row - r) == abs(col - queens[r]):
18                 return False
19             return True
20
21     backtrack(0) # 从第一行开始回溯
22
```

```

23     return solutions
24
25
26 # 获取第 b 个皇后串
27 def get_queen_string(b):
28     solutions = solve_n_queens(8)
29     if b > len(solutions):
30         return None
31     queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
32     return queen_string
33
34
35 test_cases = int(input()) # 输入的测试数据组数
36 for _ in range(test_cases):
37     b = int(input()) # 输入的 b 值
38     queen_string = get_queen_string(b)
39     print(queen_string)

```

```

1 ans = []
2 def queen_dfs(A, cur=0):          #考虑放第cur行的皇后
3     if cur == len(A):             #如果已经放了n个皇后，一组新的解产生了
4         ans.append(''.join([str(x+1) for x in A])) #注意避免浅拷贝
5         return
6
7     for col in range(len(A)):      #将当前皇后逐一放置在不同的列，每列对应一组解
8         for row in range(cur):    #逐一判定，与前面的皇后是否冲突
9             #因为预先确定所有皇后一定不在同一行，所以只需要检查是否同列，或者在同一斜线上
10            if A[row] == col or abs(col - A[row]) == cur - row:
11                break
12            else:                  #若都不冲突
13                A[cur] = col        #放置新皇后，在cur行，col列
14                queen_dfs(A, cur+1) #对下一个皇后位置进行递归
15
16 queen_dfs([None]*8)
17 for _ in range(int(input())):
18     print(ans[int(input()) - 1])

```

回溯不是靠“撤销”，而是靠“重写”。

每次尝试新的列时，`A[cur]` 被覆盖，相当于“自动回溯”。

这就是为什么代码虽然没有显式回溯，却能正确运行的原因

The screenshot shows the Spyder IDE interface with the following components:

- Editor:** Displays the code for `recviz.py`, which includes imports for `recviz` and a function `queen_dfs` for solving the N-Queens problem.
- Variable Explorer:** Shows a table with columns: Name, Type, Size, and Value. It contains one entry: `ans` with type `list` and size `1`.
- IPython Console:** Shows the command `!pip install recviz` being run, followed by the output of the package installation process.
- Status Bar:** Shows the LSP Python: ready status, conda: base (Python 3.9.12) environment, Line 4, Col 8, UTF-8 encoding, LF file type, and 86% memory usage.

```

from recviz import recviz
ans = []
@recviz
def queen_dfs(A, cur=0):    #考虑放第cur行的皇后
    if cur == len(A):
        ans.append(''.join([str(x+1) for x in A])) #注意避免浅拷贝
        return
    for col in range(len(A)):    #将当前皇后逐一放置在不同的列，每列对应一组解
        for row in range(cur):   #逐一判定，与前面的皇后是否冲突
            if A[row] == col or abs(col - A[row]) == cur - row:
                break
        else:
            A[cur] = col      #放置新皇后，在cur行，col列
            queen_dfs(A, cur+1) #对下一个皇后位置进行递归
    queen_dfs([None]*8)
for _ in range(int(input())):
    print(ans[int(input()) - 1])

```

```

In [64]: !pip install recviz
Collecting recviz
  Downloading recviz-0.0.4-py3-none-any.whl (3.3 kB)
Installing collected packages: recviz
Successfully installed recviz-0.0.4

[notice] A new release of pip is available: 23.1.2 => 23.3.1
[notice] To update, run: pip install --upgrade pip
In [65]: runfile('/Users/hfyan/.spyder-py3/untitled53.py', wdir='/Users/hfyan/.spyder-py3')
-> queen_dfs([None, None, None, None, None, None, None, None], 1)
-> queen_dfs([0, None, None, None, None, None, None, None], 2)
-> queen_dfs([0, 2, None, None, None, None, None, None], 3)
-> queen_dfs([0, 2, 4, None, None, None, None, None], 4)
-> queen_dfs([0, 2, 4, 1, None, None, None, None], 5)
-> queen_dfs([0, 2, 4, 1, 3, None, None, None], 5)
-> queen_dfs([0, 2, 4, 1, 7, None, None, None], 5)
-> queen_dfs([0, 2, 4, 6, 7, None, None, None], 4)
-> queen_dfs([0, 2, 4, 6, 1, None, None, None], 5)
-> queen.dfs([0, 2, 4, 6, 1, 3, None, None], 6)
-> queen.dfs([0, 2, 4, 6, 1, 3, 5, None], 7)
-> None
-> None
-> queen_dfs([0, 2, 4, 6, 3, 3, 5, None], 5)
-> None
-> queen_dfs([0, 2, 4, 7, 3, 3, 5, None], 4)
-> queen_dfs([0, 2, 4, 7, 1, 3, 5, None], 5)
-> queen.dfs([0, 2, 4, 7, 1, 3, 5, None], 6)
-> queen.dfs([0, 2, 4, 7, 1, 3, 5, None], 7)
-> None
-> None
-> queen_dfs([0, 2, 5, 7, 3, 3, 5, None], 3)
-> queen.dfs([0, 2, 5, 1, 3, 3, 5, None], 4)
-> queen.dfs([0, 2, 5, 1, 6, 3, 5, None], 5)
-> queen.dfs([0, 2, 5, 1, 6, 4, 5, None], 6)
-> None
-> None
-> queen.dfs([0, 2, 5, 7, 3, 4, 5, None], 4)
-> queen.dfs([0, 2, 5, 7, 1, 4, 5, None], 5)
-> queen.dfs([0, 2, 5, 7, 1, 3, 5, None], 6)
-> queen.dfs([0, 2, 5, 7, 1, 4, 5, None], 6)
-> None
-> None

```

三、递归优化和可视化

1 递归程序优化两板斧

递归程序在处理大规模问题时经常会遇到两个主要问题：**递归深度限制** 和 **重复计算子问题**。这两个问题可以通过以下两种方法来解决：

增加递归深度限制：使用 `sys.setrecursionlimit` 来增加 Python 的递归深度限制。

缓存中间结果：使用 `functools.lru_cache` 或其他形式的 memoization（记忆化）来避免重复计算。

Python 默认的递归深度限制是 1000，对于某些问题来说可能不够。你可以通过 `sys.setrecursionlimit` 来增加这个限制。

```
1 import sys  
2 sys.setrecursionlimit(1 << 30) # 将递归深度限制设置为 2^30
```

使用 `functools.lru_cache` 可以缓存函数的返回值，从而避免重复计算相同的子问题。这对于递归算法尤其有用，可以显著提高性能。

```
1 from functools import lru_cache  
2  
3 @lru_cache(maxsize=None)  
4 def recursive_function(n):  
5     if n == 0:  
6         return 1  
7     elif n == 1:  
8         return 1  
9     else:  
10        return recursive_function(n - 1) + recursive_function(n - 2)
```

示例：斐波那契数列

下面是一个具体的示例，展示了如何使用这两板斧来解决斐波那契数列的问题。

没有优化的递归实现

```
1 def fibonacci(n):
2     if n == 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         return fibonacci(n - 1) + fibonacci(n - 2)
8
9 # 测试
10 print(fibonacci(35)) # 这将非常慢，甚至可能导致栈溢出
```

使用 `sys.setrecursionlimit` 和 `lru_cache` 优化后的递归实现

```
1 import sys
2 from functools import lru_cache
3
4 # 设置递归深度限制
5 sys.setrecursionlimit(1 << 30)
6
7 # 使用 lru_cache 缓存中间结果
8 @lru_cache(maxsize=None)
9 def fibonacci(n):
10     if n == 0:
11         return 0
12     elif n == 1:
13         return 1
14     else:
15         return fibonacci(n - 1) + fibonacci(n - 2)
16
17 # 测试
18 print(fibonacci(35)) # 现在会非常快
```

详细解释

1. 增加递归深度限制：

```
1 import sys
2 sys.setrecursionlimit(1 << 30)
```

这行代码将递归深度限制设置为 2^{30} ，足够应对大多数递归问题。

2. 使用 `lru_cache` 缓存中间结果：

```

1  from functools import lru_cache
2
3  @lru_cache(maxsize=None)
4  def fibonacci(n):
5      if n == 0:
6          return 0
7      elif n == 1:
8          return 1
9      else:
10         return fibonacci(n - 1) + fibonacci(n - 2)

```

`@lru_cache(maxsize=None)` 装饰器会自动缓存 `fibonacci` 函数的结果，避免重复计算相同的子问题。`maxsize=None` 表示没有缓存大小限制。

注意事项

- **内存使用：**虽然 `lru_cache` 可以显著提高性能，但需要注意它会占用额外的内存来存储缓存结果。对于非常大的输入，可能会导致内存不足。
- **递归深度：**即使增加了递归深度限制，递归调用仍然有可能导致栈溢出。如果递归层数非常深，考虑使用迭代方法或其他非递归算法。

示例21760: 递归复习法

<http://wjjc.openjudge.cn/2024jgc5/002/>

据说，学渣复习期末考试，要用递归复习法，即当他复习知识点 A 的时候，他发现理解知识点 A 必须先理解知识点 B 和知识点 C，于是他先去学习知识点 B 和知识点 C，当他复习知识点 B 的时候，又发现理解知识点 B 必须先理解知识点 D 与知识点 E，又得先去复习知识点 D 和知识点 E。

现在学渣小明正在通过递归复习法复习知识点 n。对任意知识点 $1 \leq k \leq n$ ，他复习这个知识点本身需要 k 小时的时间。但是，小明对这些知识点非常不熟悉，以至于他对任意知识点 k , $3 \leq k \leq n$ ，都必须先复习知识点 $k-1$ 和 $k-2$ 才能复习知识点 k ；在复习知识点 $k-1$ 的时候，又得先复习知识点 $k-2$ 和 $k-3$ 才能复习知识点 $k-1$ ；以此类推……。注意，即使在复习知识点 $k-1$ 的时候他已经复习过了知识点 $k-2$ ，在复习知识点 k 之前他已经忘掉了知识点 $k-2$ ，因此他还是会再复习一遍知识点 $k-2$ ，并重复上述的递归过程完成新一轮 $k-2$ 的复习后，才会复习知识点 k 。

现在请问他一共需要多少个小时才能完成知识点 n 的复习？

输入

第一行是一个整数 m ，代表数据组数， $1 \leq m \leq 25$

之后 m 行，每行是一组数据，即一个整数 n ， $1 \leq n \leq 25$

输出

对每组数据，输出小明复习知识点 n 所需要的时间

样例输入

```
1 9  
2 1  
3 2  
4 3  
5 5  
6 7  
7 9  
8 15  
9 20  
10 25
```

样例输出

```
1 1  
2 2  
3 6  
4 23  
5 71  
6 200  
7 3786  
8 42164  
9 467833
```

提示

第一个输入n=1，需要复习一个小时。

第二个输入n=3，此时他需要先复习知识点1和知识点2，再复习知识点3，需要复习 $1+2+3=6$ 个小时。

第三个输入n=5，此时他为了复习知识点5，必须先复习知识点3与知识点4。之前已知复习知识点3需要6个小时。复习知识点4前需要再复习知识点3与知识点2，加上复习知识点4本身的时间，共需要 $2+6+4=12$ 个小时。因此，复习知识点5共需要 $6+12+5=23$ 小时。

来源

何昊高洁

```
1 def study_time(n, memo):  
2     if n == 1 or n == 2:  
3         return n  
4     if n in memo:  
5         return memo[n]  
6     memo[n] = n + study_time(n - 1, memo) + study_time(n - 2, memo)  
7     return memo[n]  
8  
9 if __name__ == "__main__":  
10    import sys  
11    input = sys.stdin.read  
12    data = input().split()  
13
```

```
14     m = int(data[0])
15     results = []
16     memo = {}
17
18     for i in range(1, m + 1):
19         n = int(data[i])
20         results.append(study_time(n, memo))
21
22     for result in results:
23         print(result)
```

利用 Python 的 `functools.lru_cache` 装饰器来自动处理缓存。这样可以简化代码，并且避免手动管理 `memo` 字典。

```
1 from functools import lru_cache
2 import sys
3
4 @lru_cache(maxsize=None)
5 def study_time(n):
6     if n == 1 or n == 2:
7         return n
8     return n + study_time(n - 1) + study_time(n - 2)
9
10 if __name__ == "__main__":
11     # 读取所有输入数据
12     input_data = sys.stdin.read().strip()
13     data = input_data.split()
14
15     m = int(data[0])
16     results = []
17
18     for i in range(1, m + 1):
19         n = int(data[i])
20         results.append(study_time(n))
21
22     for result in results:
23         print(result)
```

2 递归可视化

方法1：程序在 <http://pythontutor.com> 中运行，直接可视化。

示例：归并排序

递归程序运行过程，不容易理解。<https://pythontutor.com>，完美展示归并排序的递归过程。

方法2：在输出的调试信息前先输出一些和递归深度相关的数量的空格，可以看出递归的层级。

示例sy127：递归调试

<https://sunnywhy.com/sfbj/4/3/127>

斐波那契数列的定义：

- 1 令 $F(n)$ 表示斐波那契数列的第 n 项，则：
- 2 当 $n=1$ 时， $F(n)=1$ ；
- 3 当 $n=2$ 时， $F(n)=1$ ；
- 4 当 $n>2$ 时， $F(n)=F(n-1)+F(n-2)$ 。

下面是斐波那契数列问题的递归实现方式的伪代码：

```

1 F(n) {
2     输出调试信息;
3     if (n <= 2) {
4         return 1;
5     } else {
6         return F(n - 1) + F(n - 2);
7     }
8 }
```

递归代码的调试往往会很头疼，一个很重要的原因是在递归代码中输出的信息会因为多层而混在一起。但如果我们能在输出的调试信息前先输出一些和递归深度相关的数量的空格，就可以看出递归的层级，方便我们调试。例如当递归深度为1时先输出0个空格，递归深度为2时先输出4个空格，递归深度为3时先输出8个空格，以此类推，递归深度每多1，空格的个数就多4个）。

输入描述

一个正整数n ($2 \leq n \leq 12$) 。

输出描述

按题目描述的方式，每行输出调试信息，格式如下：

```
1 | [与递归深度相关的一堆空格]n=具体值
```

样例1

输入

```
1 | 1
```

输出

```
1 | n=1
```

样例2

输入

```
1 | 2
```

输出

```
1 | n=2
```

样例3

输入

```
1 | 3
```

输出

```
1 | n=3
2 |     n=2
3 |     n=1
```

样例4

输入

```
1 | 4
```

输出

```
1 | n=4
2 |     n=3
3 |         n=2
4 |         n=1
5 |     n=2
```

样例5

输入

```
1 | 5
```

输出

```
1 | n=5
2 |     n=4
3 |         n=3
4 |             n=2
5 |                 n=1
6 |             n=2
7 |         n=3
8 |             n=2
9 |                 n=1
```

```
1 | def F(n, depth=0):
2 |     depth += 1
3 |     blank = ' ' * 4 * (depth-1)
4 |     print(f'{blank}n={n}')
5 |     if n <= 2:
6 |         return 1
7 |     else:
8 |         return F(n-1, depth) + F(n-2, depth)
9 |
10| n = int(input())
11| if n == 1 or n == 2:
12|     print(f'n={n}')
13| else:
14|     F(n)
```

方法3：用recviz包

`recviz` 是一个用于 Python 的可视化递归调用的库。它可以帮助初学者更好的理解递归，实际开发中不会用这个库。

`recviz` 需要另外安装。

示例：dfs生成排列

```
1  from recviz import recviz
2
3
4  maxn = 11
5  hashTable = [False]*maxn # 当整数i已经在数组 P中时为 true
6
7  @recviz
8  def increasing_permutaions(n, prefix=[]):
9      if len(prefix) == n: # 递归边界，已经处理完排列的1~位
10         return [prefix]
11
12     result = []
13     for i in range(1, n+1):
14         if hashTable[i]:
15             continue
16
17         hashTable[i] = True # 记i已在prefix中
18         # 把i加入当前排列，处理排列的后续号位
19         result += increasing_permutaions(n, prefix+[i])
20         hashTable[i] = False # 处理完为i的子问题，还原状态
21
22     return result
23
24
25 n = int(input())
26 result = increasing_permutaions(n)
27 for r in result:
28     print(r)
```

The screenshot shows the Spyder IDE interface with the following details:

- Editor Tab:** The current file is `/Users/hfyan/untitled10.py`. The code implements a recursive function `increasing_permutations` to generate increasing permutations of a list of integers.
- Console Tab:** The console output shows the step-by-step execution of the function for `n=4`, displaying the state of the `hashTable` and the recursive calls.
- Help Panel:** A help panel titled "Usage" provides information on how to get help for objects in the editor or console.

```
from recviz import recviz

maxn = 11
hashTable = [False]*maxn # 当整数i已经在数组 P中时为 true

@recviz
def increasing_permutations(n, prefix=[]):
    if len(prefix) == n: # 递归边界, 已经处理完排列的1~位
        return [prefix]

    result = []
    for i in range(1, n+1):
        if hashTable[i]:
            continue

        hashTable[i] = True #标记已在prefix中
        # 把i加入当前排列, 处理排列的后缀号位
        result += increasing_permutations(n, prefix+[i])
        hashTable[i] = False #处理完为i的问题, 还原状态

    return result

n = int(input())
result = increasing_permutations(n)
for r in result:
    print(r)
```

```
> increasing_permutations(4)
-> increasing_permutations(4, [1])
-> increasing_permutations(4, [1, 2])
-> increasing_permutations(4, [1, 2, 3])
-> increasing_permutations(4, [1, 2, 3, 4])
<- [[1, 2, 3, 4]]
-> increasing_permutations(4, [1, 2, 4])
-> increasing_permutations(4, [1, 2, 4, 3])
<- [[1, 2, 4, 3]]
-> increasing_permutations(4, [1, 2, 3, 4])
-> increasing_permutations(4, [1, 3])
-> increasing_permutations(4, [1, 3, 2])
-> increasing_permutations(4, [1, 3, 2, 4])
<- [[1, 3, 2, 4]]
-> increasing_permutations(4, [1, 3, 4])
-> increasing_permutations(4, [1, 3, 4, 2])
<- [[1, 3, 4, 2]]
-> increasing_permutations(4, [1, 4])
-> increasing_permutations(4, [1, 4, 2])
-> increasing_permutations(4, [1, 4, 2, 3])
<- [[1, 4, 2, 3]]
-> increasing_permutations(4, [1, 4, 3])
-> increasing_permutations(4, [1, 4, 3, 2])
<- [[1, 4, 3, 2]]
-> increasing_permutations(4, [1, 4, 2])
-> increasing_permutations(4, [1, 2, 4])
-> increasing_permutations(4, [1, 2, 4, 3])
<- [[1, 2, 4, 3], [1, 4, 2, 3], [1, 4, 3, 2]]
-> increasing_permutations(4, [2])
-> increasing_permutations(4, [2, 1])
-> increasing_permutations(4, [2, 1, 3])
-> increasing_permutations(4, [2, 1, 3, 4])
<- [[2, 1, 3, 4]]
-> increasing_permutations(4, [2, 1, 4])
-> increasing_permutations(4, [2, 1, 4, 3])
<- [[2, 1, 4, 3]]
-> increasing_permutations(4, [2, 1, 3, 4])
-> increasing_permutations(4, [2, 1, 4, 3])
<- [[2, 1, 4, 3]]
```

LC46.全排列

backtracking, <https://leetcode.cn/problems/permutations/>

```
1 class Solution:
2     def permute(self, nums: List[int]) -> List[List[int]]:
3         n=len(nums)
4         res=[ ]
5         def backtrack(first=0):
6             if first==n:
7                 res.append(nums[:])
8             for i in range(first,n):
9                 nums[i],nums[first]=nums[first],nums[i]
10                backtrack(first+1)
11                nums[i],nums[first]=nums[first],nums[i]
12            backtrack()
13        return res
```

```

1  from typing import List
2  from recviz import recviz
3
4  class Solution:
5      def permute(self, nums: List[int]) -> List[List[int]]:
6          n=len(nums)
7          res=[]
8
9          @recviz
10         def backtrack(first=0):
11             if first==n:
12                 res.append(nums[:])
13                 print(res)
14             for i in range(first,n):
15                 nums[i],nums[first]=nums[first],nums[i]
16                 backtrack(first+1)
17                 nums[i],nums[first]=nums[first],nums[i]
18             backtrack()
19
20         return res
21
22     if __name__ == "__main__":
23         sol = Solution()
24         print(sol.permute([1,2,3]))

```

Run scratch_729

可视化递归

```

/usr/bin/python3 /Users/hfyan/Library/Application Support/JetBrains/PyCharm2024.1/scratches/scratch_729.py
-> backtrack()
-> backtrack(1)
-> backtrack(2)
-> backtrack(3)
[[1, 2, 3]]
-> None
-> None
-> backtrack(2)
-> backtrack(3)
[[1, 2, 3], [1, 3, 2]]
-> None
-> None
-> None
-> backtrack(1)
-> backtrack(2)
-> backtrack(3)
[[1, 2, 3], [1, 3, 2], [2, 1, 3]]
-> None

```

3 yield生成器

示例sy132: 全排列

<https://sunnywhy.com/sfbj/4/3/132>

给定一个正整数 n，假设序列 S = [1,2,3,...,n]，求 S 的全排列。

```
1 n = int(input())
2 l = []
3 for i in range(1,n+1):
4     l.append(f'{i}')
5
6 def arrange(l):
7     if len(l) == 1:
8         """
9             当列表中只有一个元素时，使用yield关键字返回这个元素。这里使用了生成器，而不是直接返回
10            (return) 值，
11            这意味着函数可以暂停执行并在需要时恢复，这对于处理大量数据或递归调用非常有用。
12            """
13         yield l[0]
14     else:
15         for i in range(len(l)):
16             new_l = l[:i] + l[i+1:]
17             for rest in arrange(new_l):
18                 yield l[i] + ' ' + rest
19
20 for ans in arrange(l):
21     print(ans)
22
```

`yield` 是 Python 中用于定义生成器函数的关键字。**生成器是一种特殊的迭代器，它允许你在函数内部逐步生成值，而不是一次性生成所有值并将它们存储在内存中**。当你在函数中使用 `yield` 语句时，这个函数就变成了一个生成器。当调用生成器函数时，它不会立即执行函数体内的代码，而是返回一个生成器对象。只有当这个生成器对象被迭代时，才会执行函数体内的代码，直到遇到 `yield` 语句，此时函数会暂停执行，并返回 `yield` 后面的表达式的值。当再次迭代生成器时，函数会从上次暂停的地方继续执行，直到遇到下一个 `yield` 语句，依此类推，直到函数执行完毕。

`yield` 与 `return` 的区别

- 执行时机：**当函数中使用 `return` 时，函数会立即终止执行，并返回一个值；而使用 `yield` 时，函数会生成一个生成器对象，该对象可以在需要时逐步产生值。
- 内存占用：**`return` 需要一次性计算并返回所有的值，如果这些值的数量很大，可能会消耗大量的内存。相比之下，`yield` 可以按需生成值，因此更加节省内存。
- 可迭代性：**使用 `return` 的函数只能返回一次值，而使用 `yield` 的生成器可以多次产生值，使得生成器可以用于迭代。
- 状态保持：**`yield` 使函数能够记住其上一次的状态，包括局部变量和执行的位置，因此当生成器再次被调用时，它可以从中断的地方继续执行。而 `return` 则不会保存任何状态信息，每次调用都是全新的开始。

使用 `yield` 的好处

- 节省资源：**由于生成器是惰性求值的，只有在需要的时候才计算下一个值，所以它可以有效地处

理大数据集，避免一次性加载所有数据到内存中。

- **简化代码**: 生成器提供了一种简单的方式来实现复杂的迭代模式，而不需要显式地管理迭代状态。
- **提高效率**: 对于需要连续处理大量数据的应用场景，生成器可以避免不必要的内存分配和垃圾回收，从而提高程序的运行效率。
- **易于使用**: 生成器可以像普通迭代器一样使用，可以很容易地集成到现有的代码中，如 for 循环等。

综上所述，`yield` 提供了一种强大的机制，用于处理那些需要逐步生成或处理大量数据的情况，同时保持代码的简洁性和高效性。