20231121-Week11 递归

Updated 2057 GMT+8 Nov 21 2023

2020 fall, Complied by Hongfei Yan

递归是dfs, dp的基础。需要朝向base case进行递归。

https://runestone.academy/ns/books/published/cppds/Recursion/WhatIsRecursion.html

What Is Recursion?

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.

5.3. 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].

```
#Example of summing a list using recurison.

def listsum(numList):
    if len(numList) == 1:
        return numList[0]
    else:
        return numList[0] + listsum(numList[1:]) #function makes a recursive call to itself.

print(listsum([1, 3, 5, 7, 9]))
```

Activity: 5.3.4 Recursion Summation Python (lst_recsumpy)

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.

Figure 1 shows the series of **recursive calls** that are needed to sum the vector [1,3,5,7,9]. 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.

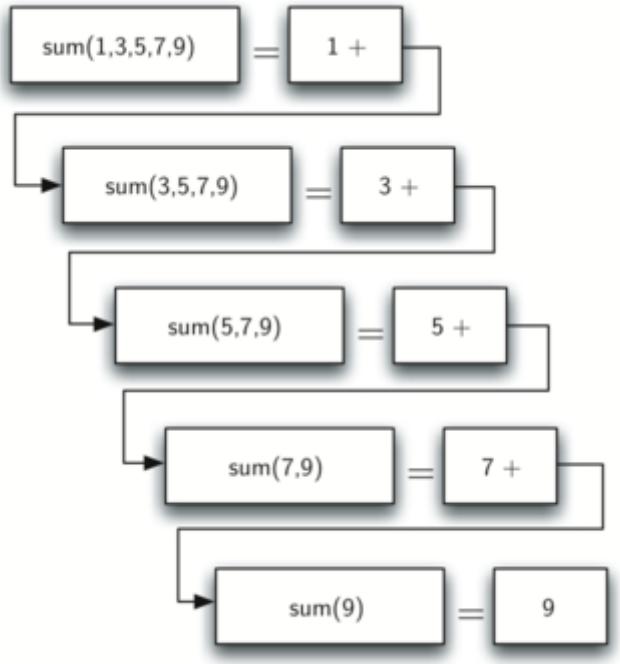


Figure 1: Series of Recursive Calls Adding a List of Numbers

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 `vectsum` works its way backward through the series of calls. When `vectsum` returns from the topmost problem, we have the solution to the whole problem.

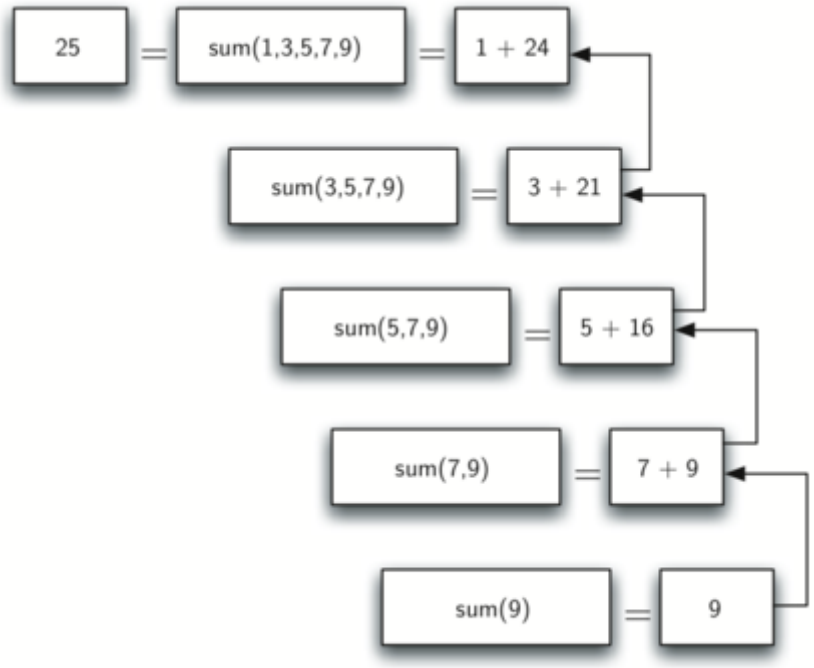


Figure2: Series of Recursive Returns from Adding a List of Numbers

1. The Three Laws of Recursion

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

> 1. A recursive algorithm must have a **base case**.
> 2. A recursive algorithm must change its state and move toward the base case.
> 3. A recursive algorithm must call itself, recursively.

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.

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.

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.

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).

https://runestone.academy/ns/books/published/cppds/Recursion/TheThreeLawsofRecursion.html

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 * … 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)

2. 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.

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.

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.
> 2. Convert the single digit-number to a string using a lookup.
> 3. Concatenate the single-digit strings together to form the final result.

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.

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<base, where 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.

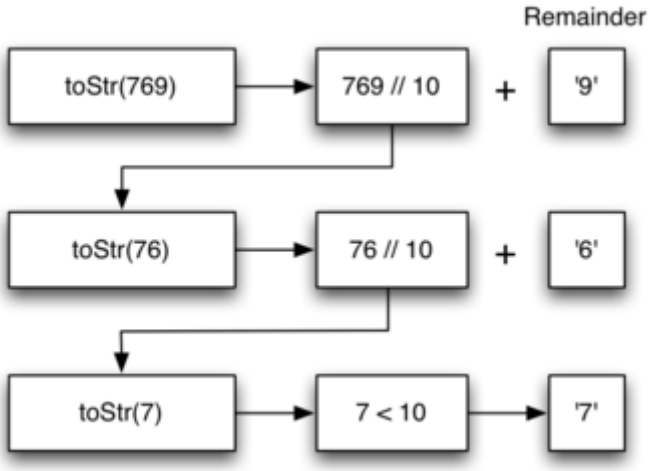


Figure 3: Converting an Integer to a String in Base 10

ActiveCode 1 shows the C++ and Python code that implements the algorithm outlined above for any base between 2 and 16.

```
#Recursive example of converting an int to str.

def toStr(n,base):
   convertString = "0123456789ABCDEF"
   if n < base:
      return convertString[n]
   else:
      return toStr(n//base,base) + convertString[n%base] #function makes a recursive call to i


print(toStr(1453,16))
```

Notice that in line 5 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 8 we satisfy both the second and third laws–by making the recursive call and by reducing the problem size–using division.

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

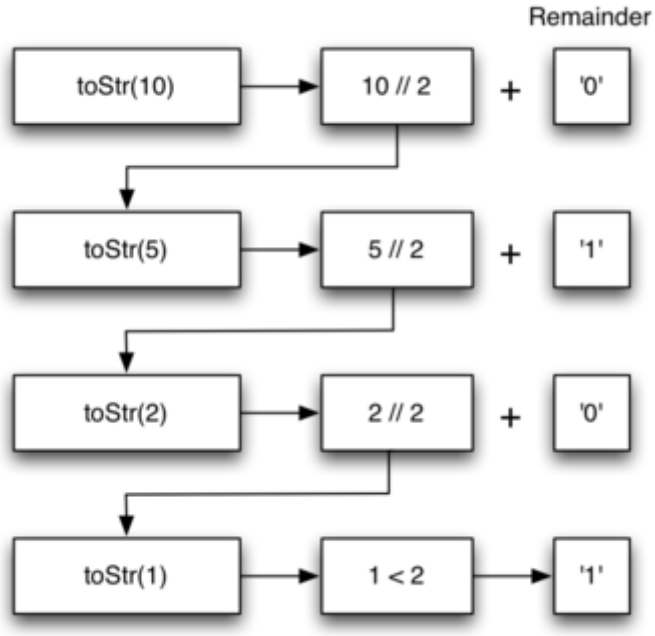


Figure 4: Converting the Number 10 to its Base 2 String Representation

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.

image-20231121113514094

3. Stack Frames: Implementing Recursion

Suppose that instead of concatenating the result of the recursive call to `toStr` 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.

```
rStack = []

def toStr(n,base):
    convertString = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            rStack.append(convertString[n]) #adds string n to the stack.
        else:
            rStack.append(convertString[n % base]) #adds string n modulo base to the stack.
```

```
            n = n // base
        res = ""
        while rStack:
            #combines all the items in the stack to make the full string.
            res = res + str(rStack.pop())
        return res

print(toStr(1453,16))
```

Each time we make a call to `toStr`, 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"`.

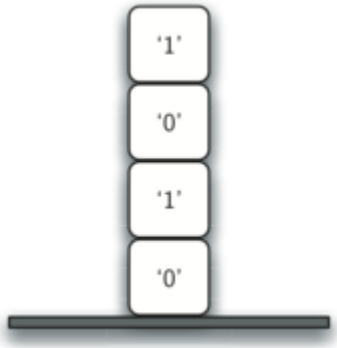


Figure 5: Strings Placed on the Stack During Conversion

The previous example gives us some insight into how C++ 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.

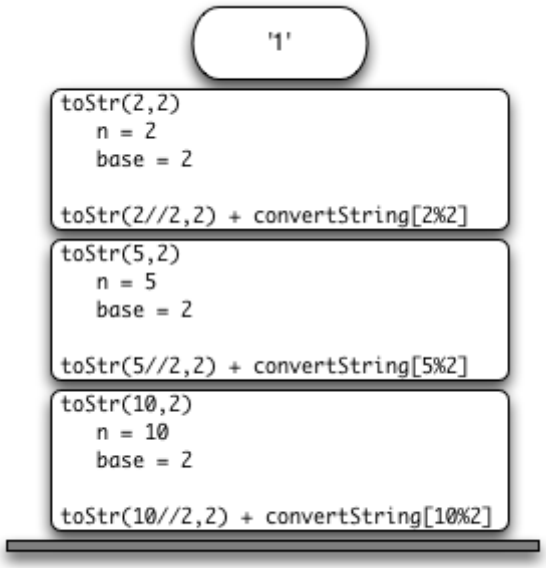


Figure 6: Call Stack Generated from `toStr(10,2)`

Notice that the call to `toStr(2//2,2)` leaves a return value of `"1"` on the stack. This return value is then used in place of the function call (`toStr(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 C++ 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.

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.

4. 计算机原理：虚拟地址空间

三大计算机原理之一，@Book_my_flight_v0.3.md

计算机的基础架构自从 20 世纪 40 年代起就已经形成规范，包括处理器、存储指令和数据的内存、输入和输出设备。它通常叫作冯·诺依曼架构，以约翰·冯·诺依曼（德語：John Von Neumann，1903 年12 月 28 日－1957 年 2 月 8 日）的名字来命名，他在 1946 年发表的论文里描述了这一架构。论文的开头句，用现在的专门术语来说就是，CPU提 供算法和控制，而 RAM 和磁盘则是记忆存储，键盘、鼠标和显示器与操作人员交互。其中需要重点理解的是与存储相关的进程的虚拟地址空间。

在《深入理解计算机系统》[8]第一章中讲到了进程的虚拟地址空间。虚拟存储器是一个抽象概念，它为每个进程提供了一个假象，好像每个进程都在独占地使用主存。每个进程看到的存储器都是一致的，称之为虚拟地址空间。如图1-15所示的是 Linux 进程的虚拟地址空间（其他 Unix 系统的设计与此类似）。在 Linux 中，最上面的四分之一的地址空间是预留给操作系统中的代码和数据的，这对所有进程都一样。底部的四分之三的地址空间用来存放用户进程定义的代码和数据。请注意，图中的地址是从下往上增大的。

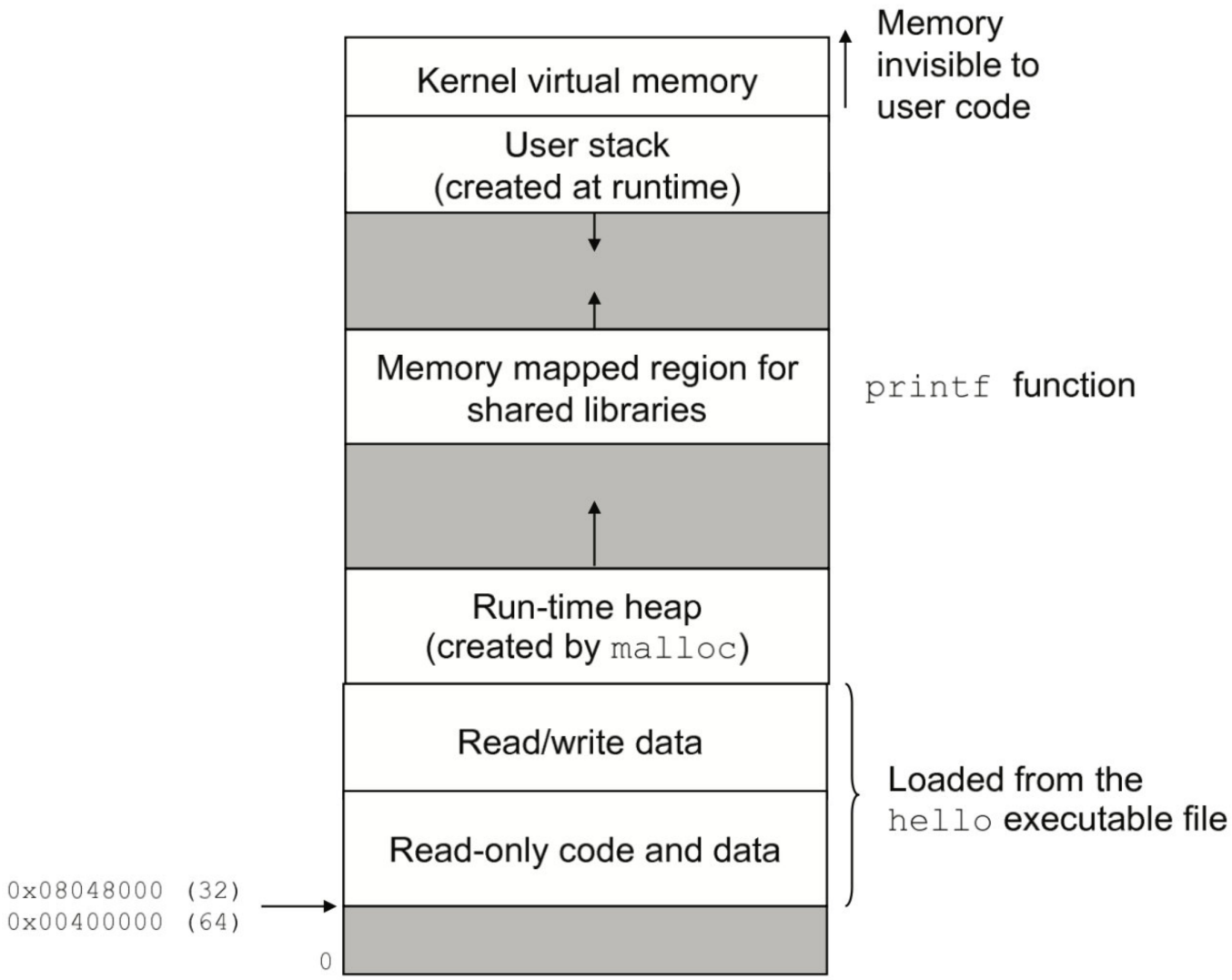


图1-15 进程的虚拟地址空间（Process virtual address space）（注：图片来源为 Randal Bryant[8]，2015年3月）

每个进程看到的虚拟地址空间由准确定义的区（area）构成，每个区都有专门的功能。简单看下每一个区，从最低的地址开始，逐步向上研究。

- 程序代码和数据（code and data）。代码是从同一固定地址开始，紧接着的是和全局变量相对应的数据区。代码和数据区是由可执行目标文件直接初始化的，示例中就是可执行文件hello。
- 堆（heap）。紧随代码和数据区之后的是运行时堆（Run-time heap）。代码和数据区是在进程一旦开始运行时就被指定了大小的，与此不同，作为调用像 malloc 和 free 这样的 C 标准库函数的结果，堆可以在运行时动态地扩展和收缩。
- 共享库（shared libraries）。在地址空间的中间附近是一块用来存放像标准库和数学库这样共享库的代码和数据的区域。共享库的概念非常强大。
- 栈（stack）。位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。和堆一样，用户栈（User stack）在程序执行期间可以动态地扩展和收缩。特别地，每次我们调用一个函数时，栈就会增长。每次我们从函数返回时，栈就会收缩。
- 内核虚拟存储器（kernal virtal memory）。内核是操作系统总是驻留在存储器中的部分。地址空间顶部是为内核预留的。应用程序不允许读写这个区域的内容或者直接调用内核代码定义的函数。

虚拟存储器的运作需要硬件和操作系统软件间的精密复杂的互相合作，包括对处理器生成的每个地址的硬件翻译。基本思想是把一个进程虚拟存储器的内容存储在磁盘上，然后用主存作为磁盘的高速缓存。

5. Tower of Hanoi

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.

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.

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.

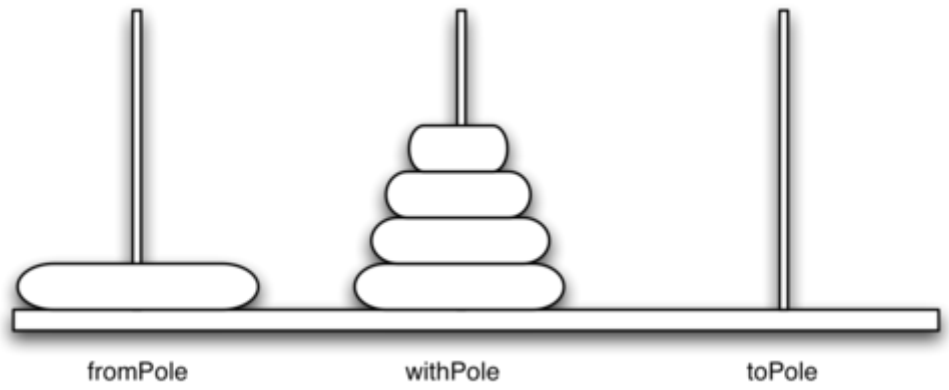


Figure 1: An Example Arrangement of Disks for the Tower of Hanoi

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.

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.

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.

Listing 1

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

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.

Listing 2

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

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

```
#Simulation of the tower of hanoi.

def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole) #Recursive call
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole) #Recursive call

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


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.

04147: 汉诺塔问题(Tower of Hanoi)

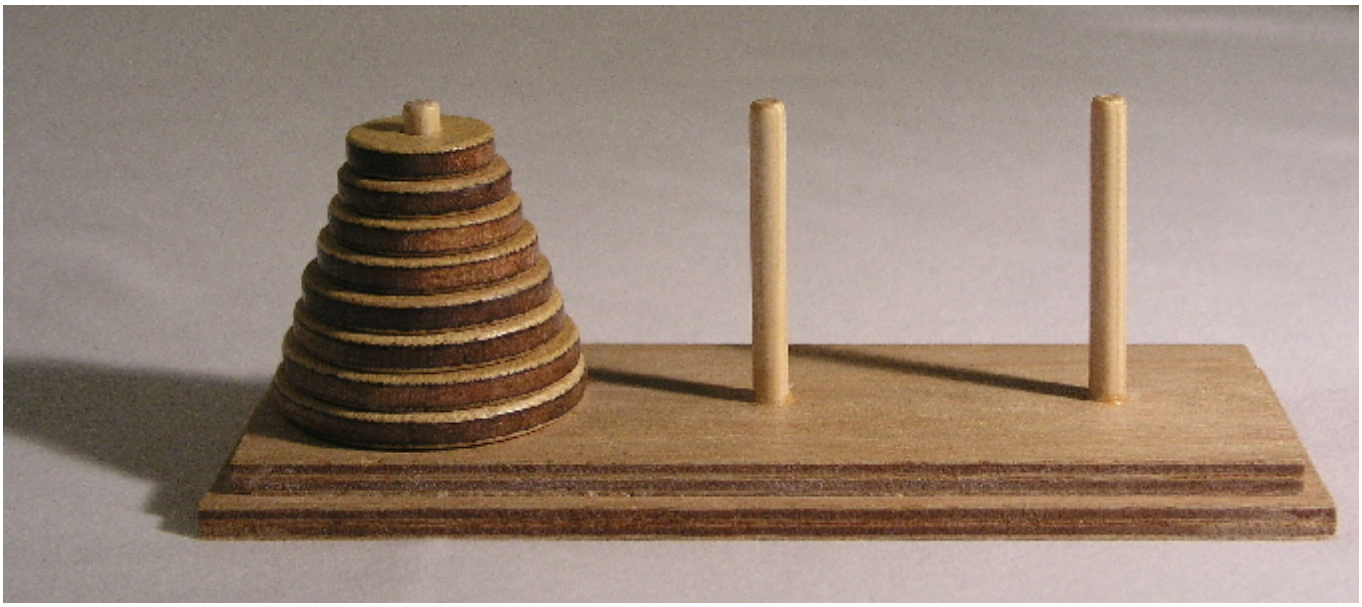
http://cs101.openjudge.cn/practice/04147

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->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, ...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

```
3 a b c
```

样例输出

```
1:a->c
2:a->b
1:c->b
3:a->c
1:b->a
2:b->c
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

来源：重庆科技学院 WJQ

```
# https://blog.csdn.net/geekwangminli/article/details/7981570

# 将编号为numdisk的盘子从init杆移至desti杆
def moveOne(numDisk : int, init : str, desti : str):
    print("{}:{}->{}".format(numDisk, init, desti))

#将numDisks个盘子从init杆借助temp杆移至desti杆
def move(numDisks : int, init : str, temp : str, desti : str):
    if numDisks == 1:
        moveOne(1, init, desti)
    else:
        # 首先将上面的（numDisk-1）个盘子从init杆借助desti杆移至temp杆
        move(numDisks-1, init, desti, temp)

        # 然后将编号为numDisks的盘子从init杆移至desti杆
        moveOne(numDisks, init, desti)

        # 最后将上面的（numDisks-1）个盘子从temp杆借助init杆移至desti杆
        move(numDisks-1, temp, init, desti)

n, a, b, c = input().split()
```

```
move(int(n), a, b, c)
```

01958: Strange Towers of Hanoi

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 <= 12: At first k >= 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 2 ∈{1, ... , 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 <= n <= 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

```
No input.
```

样例输出

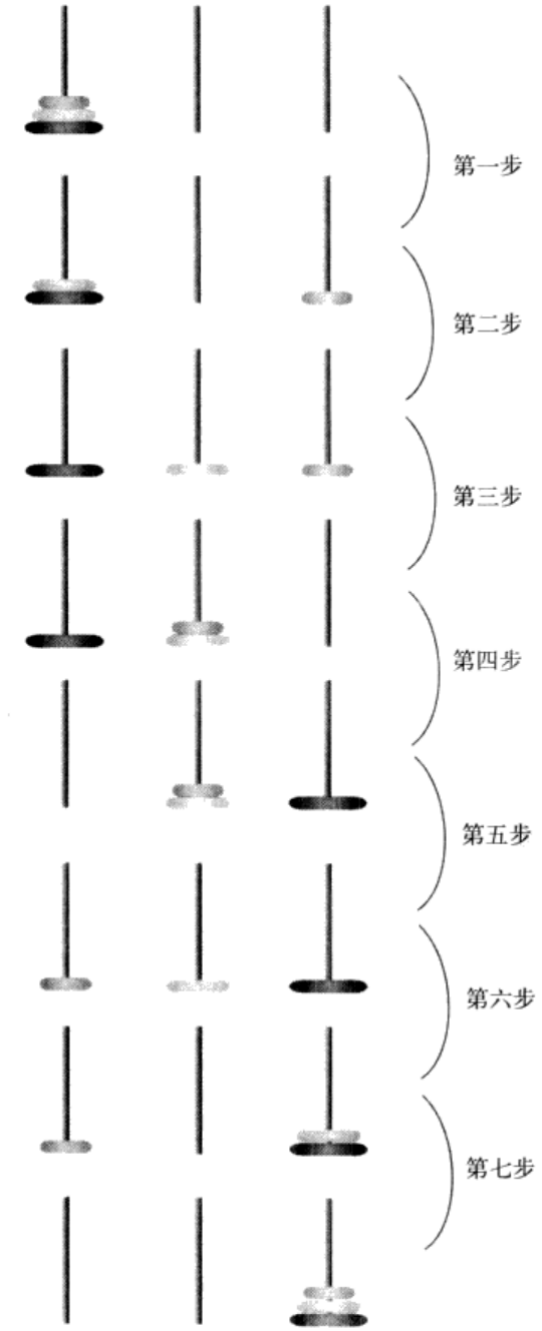
```
REFER TO OUTPUT.
```

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 (2^2^-1)次。比如，n=3的时候，

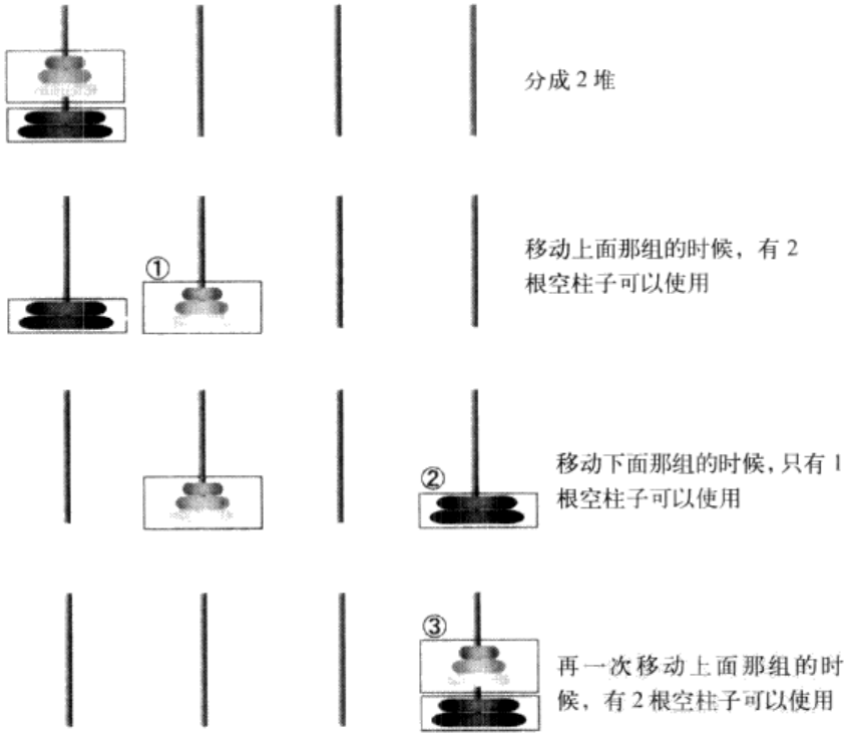


的确是用了 2^3^-1=7 次完成了移动。那么，这次的问题不是基本的汉诺塔，而是把柱子的根数增加1根。如果柱子增加到 4根，原来需要移动 7次完成，现在只需要 5次就可以了。

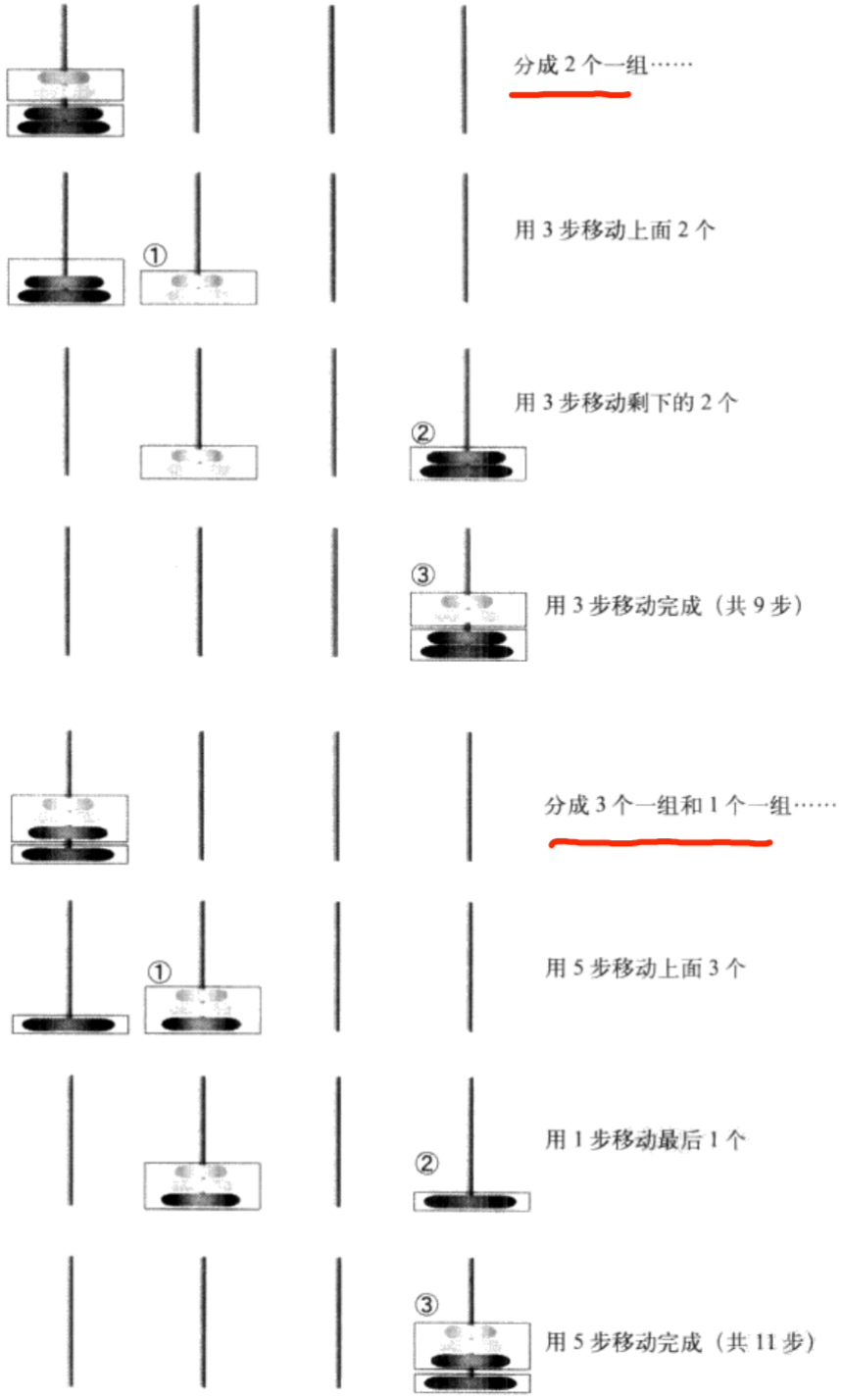
image-20231030194009343

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。

```
d = [0] * 15
f = [float('inf')] * 15

d[1] = 1
for i in range(2, 13):
    d[i] = d[i - 1] * 2 + 1

f[1] = 1
```

```
for i in range(2, 13):
    for j in range(1, i):
        f[i] = min(f[i], f[i - j] * 2 + d[j])

for i in range(1, 13):
    print(f[i])
```

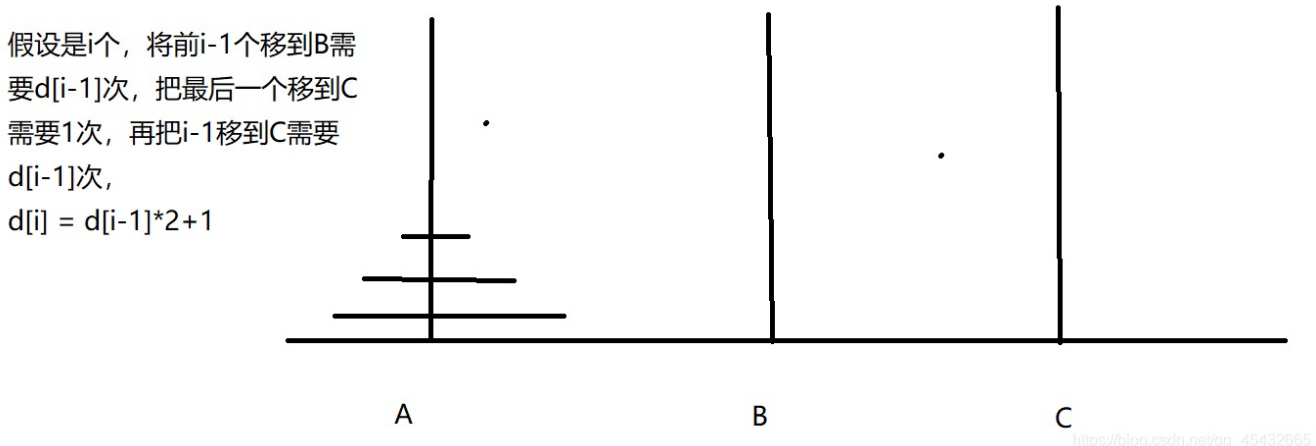
```
# 23n2300011072，蒋子轩
def hanoi_four_towers(n, source, target, auxiliary1, auxiliary2):
    if n == 0:
        return 0
    if n == 1:
        return 1
    min_moves = float('inf')
    for k in range(1, n):
        three_tower_moves = 2**(n-k)-1
        moves = hanoi_four_towers(k, source, auxiliary1, auxiliary2, target) +\
            three_tower_moves +\
            hanoi_four_towers(k, auxiliary1, target, source, auxiliary2)
        min_moves = min(min_moves, moves)
    return min_moves


for n in range(1, 13):
    print(hanoi_four_towers(n, 'A','D','B','C'))
```

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

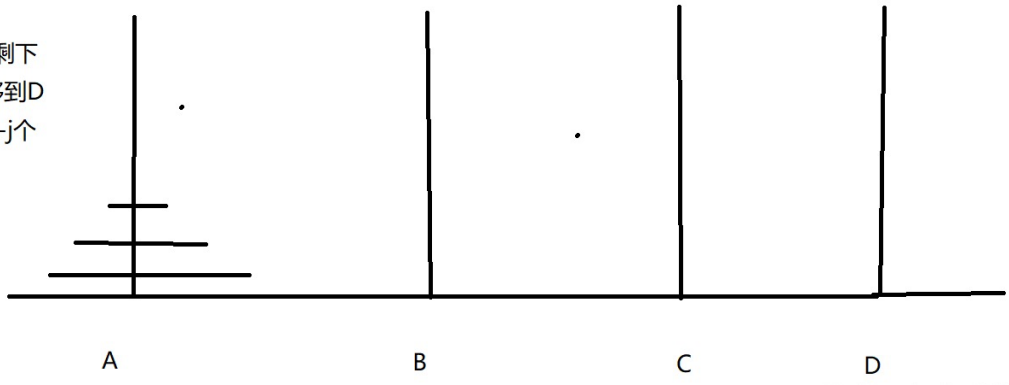
https://blog.csdn.net/qq_45432665/article/details/104825847

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



当有4塔时，也是一样的思路，f[1] = 1

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]



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.

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

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:

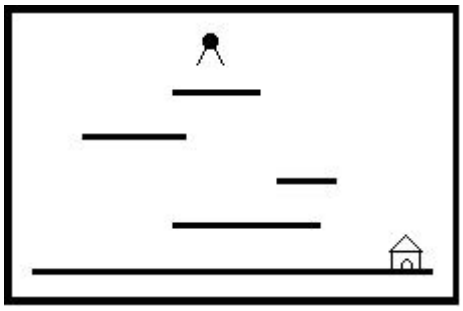
$a_n = 1 + \sum 2^{f(x)} = 1 - 2^{t-2}(t^2 - 3t - 2n + 4)$, 其中 $t = \lfloor \sqrt{2n} \rceil$

```
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;
                          最小 リストを作成
        dt[n_] := dt[n] = dt[n - 1] + 2^(Round@Sqrt[2 n] - 1); dt[1] := 1;
```

6.部分递归题目

01661: Help Jimmy

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 ≤ t ≤ 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 ≤ N ≤ 1000，-20000 ≤ X, X1[i], X2[i] ≤ 20000，0 < H[i] < Y ≤ 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
3 8 17 20
0 10 8
0 10 13
4 14 3
```

样例输出

23

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

```
# 查达闻 2300011813
from functools import lru_cache

@lru_cache
def dfs(x, y, z):
    for i in range(z+1, N+1):
        if y - MaxVal > p[i][2]:
            return 1 << 30
        elif p[i][0] <= x <= p[i][1]:
            left = x - p[i][0] + dfs(p[i][0], p[i][2], i)
            right = p[i][1] - x + dfs(p[i][1], p[i][2], i)
            return min(left,right)

    if y <= MaxVal:
        return 0
    else:
        return 1 << 30


for _ in range(int(input())):
    N, ini_x, ini_y, MaxVal = map(int, input().split())

    p = []      #platform
    p.append( [0, 0, 1 << 30] ) # 1<<30 大于 20000*2*1000
    for _ in range(N):
        p.append([int(x) for x in input().split()])
    p.sort(key = lambda x:-x[2])

    print(ini_y + dfs(ini_x, ini_y, 0))
```

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 x M (1 <= N <= 100; 1 <= M <= 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

```
10 12
W........WW.
.WWW.....WWW
....WW...WW.
.........WW.
.........W..
..W......W..
.W.W.....WW.
W.W.W.....W.
.W.W......W.
..W.......W.
```

样例输出

```
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.dfs
import sys
sys.setrecursionlimit(20000)
def dfs(x,y):
        #标记，避免再次访问
    field[x][y]='.'
    for k in range(8):
        nx,ny=x+dx[k],y+dy[k]
        #范围内且未访问的lake
        if 0<=nx<n and 0<=ny<m\
                and field[nx][ny]=='W':
            #继续搜索
            dfs(nx,ny)
n,m=map(int,input().split())
field=[list(input()) for _ in range(n)]
cnt=0
dx=[-1,-1,-1,0,0,1,1,1]
```

```
dy=[-1,0,1,-1,1,-1,0,1]
for i in range(n):
    for j in range(m):
        if field[i][j]=='W':
            dfs(i,j)
            cnt+=1
print(cnt)
```

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 <= n<= 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

```
2
6
r##bb#
###b##
#r##b#
#r##b#
#r####
######
4
####
#rrb
#rr#
##bb
```

样例输出

```
2 2
1 2
```

```
dire = [[-1,0], [1,0], [0,-1], [0,1]]

def dfs(x, y, c):
    m[x][y] = '#'
    for i in range(len(dire)):
        tx = x + dire[i][0]
        ty = y + dire[i][1]
        if m[tx][ty] == c:
            dfs(tx, ty, c)

for _ in range(int(input())):
    n = int(input())
    m = [[0 for _ in range(n+2)] for _ in range(n+2)]

    for i in range(1, n+1):
        m[i][1:-1] = input()

    r = 0 ; b=0
    for i in range(1, n+1):
        for j in range(1, n+1):
            if m[i][j] == 'r':
                dfs(i, j, 'r')
                r += 1
            if m[i][j] == 'b':
                dfs(i,j,'b')
                b += 1
    print(r, b)
```

02786: Pell数列

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

Pell数列a1, a2, a3, ...的定义是这样的，a1 = 1, a2 = 2, ... , an = 2 * an – 1 + an - 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 ≤ 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

```
2
1
8
```

样例输出

```
1
408
```

```
#2300011786 裘思远
from functools import lru_cache

@lru_cache(maxsize=None)
def series(n):
    if n>2:
        return (series(n-1)*2+series(n-2))%32767
    elif n==2:
        return 2
    else:
        return 1

n=int(input())
for _ in range(n):
    k=int(input())%150
    ans=series(k)
    print(ans)
```