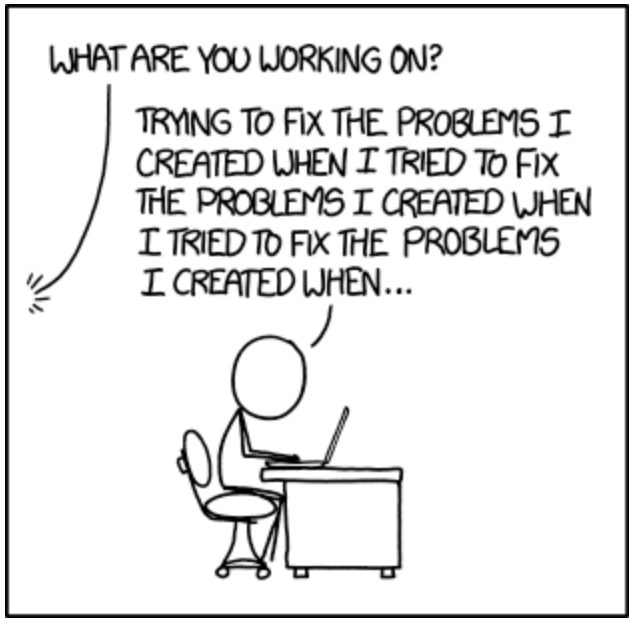
Recursion递归



https://imgs.xkcd.com/comics/fixing_problems.png

递归

- 递归是一种重复
- 递归是函数调用自己来实现的重复
- 每一次的递归调用是基于一组不同的，往往更小的数据的
- 递归是一种divide and conquer，是解决问题的top-down方法
    - 它把问题分解为更小的问题，不断重复这个过程，直到可以很容易得到结果

递归的4个基础

- Base Case：至少存在一种情况，不再进一步递归就可以直接得到结果
- Make Progress：每一次递归都是向着base case前进
- Always Believe：始终相信递归是可行的
- Compound Interest Rule：不要在不同的递归调用中做相同的计算

阶乘

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

- 注意到：

$$5! = 5 \times 4!$$

$$4! = 4 \times 3! \ldots$$

- 于是就可以递归计算（定义：$1! = 0! = 1$）：

$$2! = 2 \times 1! = 2 \times 1 = 2$$

$$3! = 3 \times 2! = 3 \times 2 = 6$$

递归算法

- 显然阶乘的计算可以被表达为一个分段函数：

$$n! = f(n) = \begin{cases} 1 & ; n = 0 \\ n \cdot f(n-1) & ; n > 0 \end{cases}$$

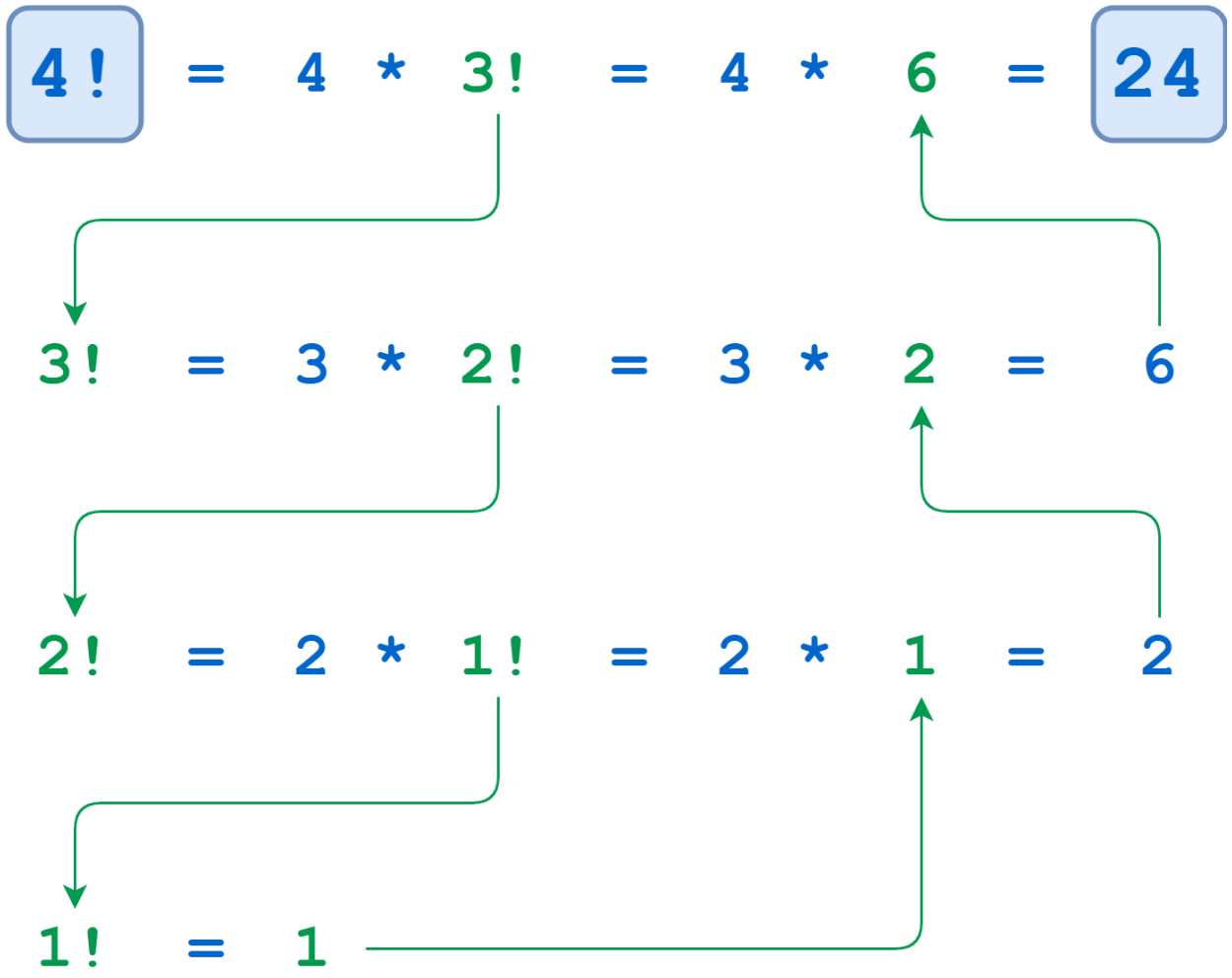
- `n=0` 就是这里的base case；而 `n>0` 时所做的就是make progress
- 在 `n>0` 的时候用到了 `f(n-1)`，而此时正在定义 `f(n)`，这就是always believe
- 只要识别出计算中存在递归，就可以用编程语言来实现

递归的阶乘

```
int factorial(int n) {
    if ( n==0 ) {
        return 1;
    }
    return n*factorial(n-1);
}
```

递归

- 递归的执行可以分为“递进”和“回归”两个阶段
- 递归总是从要解决的最大的状态开始，如n!的n
- 在递进的阶段，问题被不断地分解、缩小，直到遇到base case；这个阶段，并没有做实际的计算
- 在回归的阶段从已知结果的base case，带着结果做实际的计算，沿着递的路径，反方向回到原始状态
- 对于 `factorial(5)`
  - 递进时的分解是：
    `5-->4-->3-->2-->1-->0`
  - 回归时的计算是：
    `1*2*3*4*5`



https://files.realpython.com/media/jsturtz-factorial-example.496c01139673.png

斐波那契数列

`0、1、1、2、3、5、8、13、21、34...`

$$f(n) = \begin{cases} n & ;n = 0 \ or \ n = 1 \\ f(n-1) + f(n-2) & ;n > 1 \end{cases}$$

- 这个分段函数显然是可以用递归计算的

```c
int fib(int n) {
    if ( n==0 || n==1 ) {
        return n;
    }
    return fib(n-1)+fib(n-2);
}
```

- 不妨加入 `printf()` 来看执行情况

线性递归 vs 树状递归

- 阶乘是典型的线性递归，因为每一次递进只调用一次自己
- 斐波那契数列是典型的树状递归，因为每一次递进要调用两次自己
- 树状递归通常存在大量的重复计算
- “记忆”是消除重复计算的主要手段

`time` 是一个Unix程序，可以看程序运行所花费的时间

```
int fib(int n) {
    static int fs[100] = {0,1};
    if ( n>1 && n<100 && fs[n] ==0 ) {
        fs[n] = fib(n-1)+fib(n-2);
    }
    return fs[n];
}
```

Euclid算法

1. 设a, b为两个自然数，欲求a, b的最大公约数
2. 若b为0，则a就是a, b的最大公约数，计算结束
3. 否，则令a为b，而b为原来的a%b，重复步骤2

$$gcd(x,y) = \begin{cases} x & ;y = 0 \\ gcd(y,\ x \mod y) & ;y > 0 \end{cases}$$

```
int gcd(int x, int y) {
    if ( y==0 ) {
        return x;
    }
    return gcd(y, x%y);
}
```

尾递归

- 当递进时，不对递进调用返回的值再做任何计算就直接返回的是尾递归：
  - `return gcd(y, x%y);`
- 注意不是因为在函数的尾部做递归
- 阶乘和斐波那契数列都不是尾递归：
  - `return n*factorial(n-1);`
  - `return fib(n-1)+fib(n-2);`
- 尾递归是伪递归，因为它是在递进的时候做了计算，而回归的时候没有做计算

尾递归优化

- 所有的尾递归都可以在代码形式上机械地被优化为非递归的形态
  - 整个函数用`while (1)`包起来
  - base case改成`if ... break`
  - 递进改成对相应的变量赋值

```
int gcd(int x, int y) {
    if ( y==0 ) {
        return x;
    }
    return gcd(y, x%y);
}
```

- 变成

```
int gcd(int x, int y) {
    while ( 1 ) {
        if ( y==0 ) {
            break;
        }
        int t = x%y;
        x=y;
        y=t;
    }
    return x;
}
```

杨辉（Pascal）三角

- 杨辉三角，是二项式系数在三角形中的一种几何排列，中国南宋数学家杨辉1261年所著的《详解九章算法》一书中出现。在欧洲，帕斯卡（1623----1662）在1654年发现这一规律，所以这个表又叫做帕斯卡三角形。帕斯卡的发现比杨辉要迟393年，比贾宪迟600年
- 每行端点与结尾的数为1
- 每个数字等于上一行的左右两个数字之和
- 第n行的m个数可表示为 C(n-1，m-1)，即为从n-1个不同元素中取m-1个元素的组合数
- $(a+b)^n$的展开式中的各项系数依次对应杨辉三角的第(n+1)行中的每一项

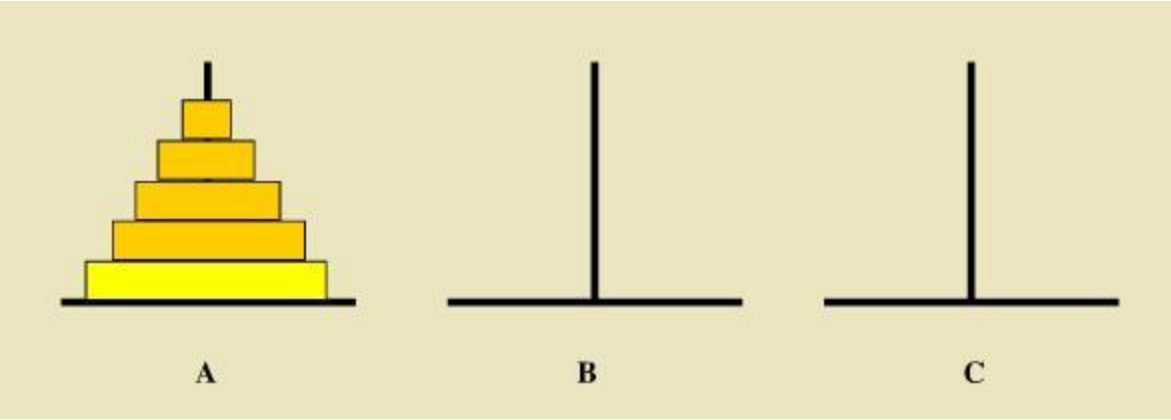
杨辉三角

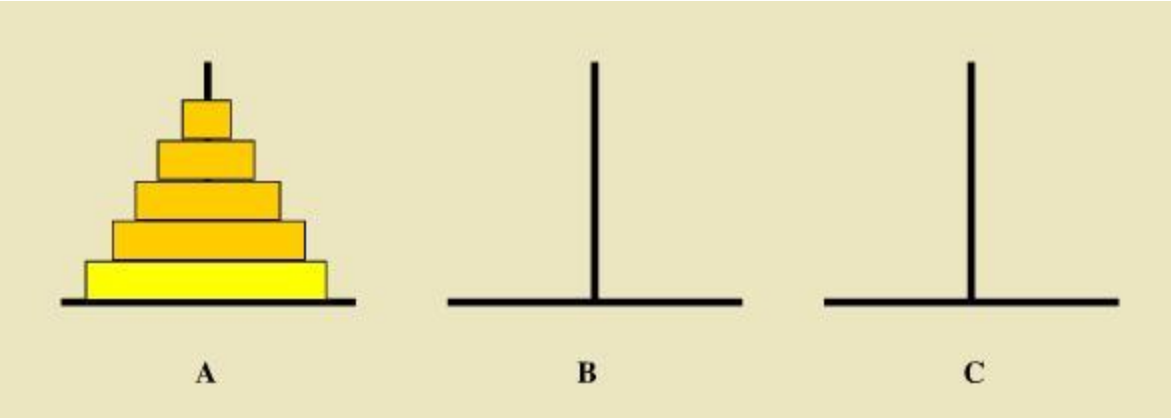
```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

- 令层数为n，则P(n)是一个数列（列表）
- 当n==1，P(n)=[1]
- 当n>1，P(n)是在P(n-1)上推算的
  - P(n)[0]=P(n)[n-1]=1
  - P(n)[i]=P(n-1)[i-1]+P(n-1)[i], i=1...n-2

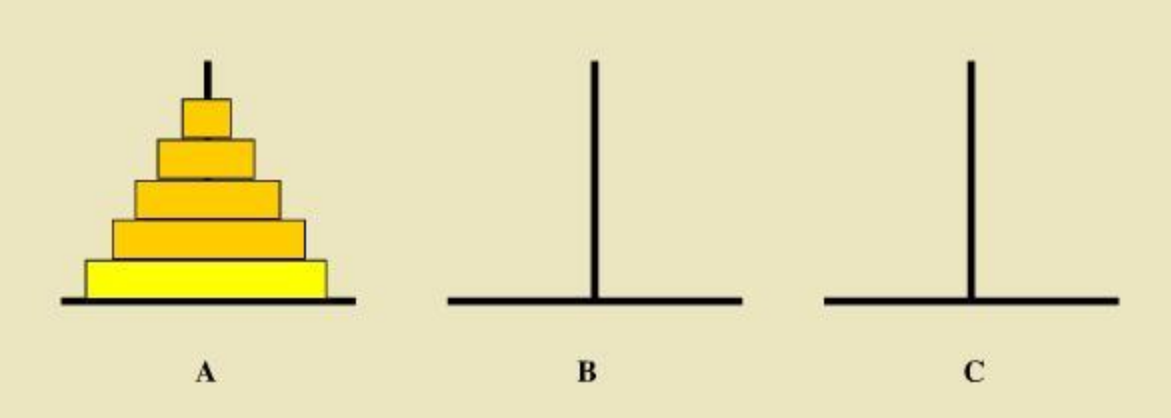
汉诺塔

- 汉诺塔问题是源于印度的一个古老传说。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘





- 要把n个盘从source移到到targer
- 定义另一个柱子是aux
- Base Case：没有盘要移动
- Progress:
  - 把 n-1 个盘从source移到aux
  - 把剩下的第n个盘从source移到target
  - 把原来的 n-1 个盘从aux移到target




```
void hanoi(int n, char source, char target, char aux) {
    if ( n>0 ) {
        hanoi(n-1, source, aux, target);     // 以target为辅助柱
        printf("move %d from %c to %c\n", n, source, target);
        hanoi(n-1, aux, target, source)
    }
}
hanoi(3, 'A', 'C', 'B')
```

牛顿迭代求x的平方根

- 首先猜测结果为$\frac{x}{2}$，f(n)表示第n次猜测

$$f(n) = \begin{cases} \frac{x}{2} & ;n = 1 \\ \frac{f(n-1)+\frac{x}{f(n-1)}}{2} & ;n > 1 \end{cases}$$

- 收敛条件为
  $f(n)^2 = x$
  或
  $|f(n)-f(n-1)| < \epsilon$
- 有base case（收敛条件），有progress，应该可以递归计算

牛顿迭代求x的平方根

$$f(n) = \begin{cases} \frac{x}{2} & ;n = 1 \\ \frac{f(n-1)+\frac{x}{f(n-1)}}{2} & ;n > 1 \end{cases}$$

$$f(x, g) = \begin{cases} g & ;|g^2 - x| < \epsilon \\ f(x, \frac{g+\frac{x}{g}}{2}) & ;otherwise \end{cases}$$

```
double _sqrt_it(double x, double guess) {
    if (fabs(guess*guess-x)<1e-3) {
        return guess;
    return _sqrt_it(x, (guess+x/guess)/2);
}

double mysqrt(double x) {
    return _sqrt_it(x, x/2);
}
```

- 这是尾递归吗?

迭代算法vs递归算法

- 简单地可以认为用循环实现的是迭代算法
- 迭代算法总是从问题最小的状态开始，展开到最大的状态:
  - `n! = 1*2*3….*n`
- 递归算法则是从问题最大的状态开始，分解到最小的状态（base case）:
  - `n! = n* (n-1) * (n-2) …. *1`

递归 --> 迭代

- 绝大部分递归算法都可以被改造成迭代算法
- 机械的方法是，将结果作为一个参数，在递进的过程中计算并传递进下一轮递进，从而把递归改造成尾递归

```
int factorial(int n) {
    if ( n==0 ) {
        return 1;
    }
    return n*factorial(n-1);
}
```

```
int factorial_it(int n, int f) {
    if ( n==0 ) {
        return f;
    }
    return factorial_it(n-1, f*n);
}
printf("%d\n", factorial(5, 1));
```

计算幂

$$x^n = \begin{cases} 1 & ;n = 0 \\ x \cdot x^{n-1} & ;n > 0 \end{cases}$$

- 有base case，有progress，典型的递归

```
int power(int x, int n) {
    if ( n==0 ) {
        return 1;
    }
    return x * power(x, n-1);
}
```

机械改造成迭代

```
int power(int x, int n, int p) {
    if (n==0) {
        return p;
    }
    return power(x, n-1, p*x);
}
```

加速冪計算

$$x^n = \begin{cases} 1 & ;n = 0 \\ (x^{\frac{n}{2}})^2 & ;n > 0 \; and \; n \; is \; even \\ x \cdot x^{n-1} & ;n > 0 \; and \; n \; is \; odd \end{cases}$$

```
int power(int x, int n) {
    if (n==0) {
        return 1;
    } else if (n%2==1) {
        return x * power(x, n-1);
    } else {
        int t = power(x, n/2);
        return t*t;
    }
}
```

迭代版本的快速幂

- 任何一个数都可以写成若干个2的幂次的和

$$\because 13\text{的二进制是}1101\text{，即}13 = 8 + 4 + 1$$
$$\therefore x^{13} = x^8 \times x^4 \times x^1 = x^1 \times x^4 \times x^8$$

- 也就是说，在x的$2^0, 2^1, 2^2, 2^3$次幂中，x的$2^0, 2^2, 2^3$是要乘进结果里的，对应的是`1101`中的`1`，而

$$x^1 = x$$
$$x^2 = (x^1)^2$$
$$x^4 = (x^2)^2$$

- 所以，只要连续计算$(x^2)^2 \cdots)^2$，同时分解n的二进制，将其中对应`1`的幂次乘入结果

理解快速冪

```
int x=2;
int n=13;
printf("n\tx\tn%2\n");
while (n>0) {
    printf("%d\t%d\t%d\n', n, x, n%2);
    x *= x;
    n /= 2;
}
```

- 將 `n%2==1` 的行的 `x` 的值乘起來：$2 \times 16 \times 256 = 8192 = 2^{13}$

迭代版本的快速幂

```
int power(int x, int n) {
    int res = 1;          // 结果从1开始
    while (n) {
        if (n % 2) {      // 最后一位为1，就需要把当前的幂乘到结果中
            res *= x;
        }
        x *= x;           // 一直累乘
        n /= 2;           // 去掉最后一位
    }
    return res;
}
```

递归的Pros & Cons

- Pros
  - 一旦被优化，性能还不错
  - 代码更简短
  - 易于理解：递归代码更有描述性，比迭代代码更接近问题本身
- Cons
  - 递归深度受到线程运行栈大小限制不能太深
  - 占用内存较多
- 建议：用递归计算的思路思考问题，代码实现后，转成迭代实现

What we've learned today?

- 递归