算法入门

前端精进（科班方向）：算法与数据结构入门

词汇表

中文	原文	中文	原文
递归	recursion	迭代	iteration
汉诺、河内	Hanoi	尾调用优化	tail call optimization
斐波那契	[意]Fibonacci		

版权声明

联系方式

如果你想要购买本课程
请微信联系 xiedaimala02 或 xiedaimala03

如果你发现有人盗用本课程
请微信联系 xiedaimala02 或 xiedaimala03

先来做一个很简单的题

求最大值

完善 JS 代码

```
const array = [23,99,17,28,84]
function max(array){
    请填充代码
}
max(array) // 99
```

你该如何思考?

算法题通用思考逻辑

- 一、普通人类思维
  - 将问题转化成现实生活中的事情
  - 人类是如何找出10个数字中最大的那一个的?
  - 将人类的思维过程用严谨的语言表达出来
  - 将其翻译成代码或伪代码
- 二、数学思维
  - 利用数学知识
  - 找到对应的数学公式/定理
  - 向解题一样解出来

一、普通人类思维

- 求最大值
  - ✓ 用眼睛从左到右扫描，发现是较大的就记下来
  - ✓ 扫描完毕，得到的数就是最大的了
  - ✓ 改写成代码

```
const array = [23,99,17,28,84]
function max(array){
  let result = array[0]
  for(let i = 1; i< array.length; i++){
    if(array[i] > result){
      result = array[i]
    }
  }
}
max(array) // 99
```

如何证明这个算法是对的?

- 证明
  - 比较难用数学方法证明
- 经验
  - 按照一般人的逻辑和经验，这个算法是对的
  - 无法给出反例，说明这个算法暂时是对的
  - 通过大量的测试，发现这个算法可以满足需求
- 总结
  - 大部分之后程序员只需要「能满足需求」的代码
  - 而不是「正确」的代码

二、数学思维

- 求最大值
  - ✓ 用公式表示这个过程（归纳法）
  - ✓ 把数字代入公式
  - ✓ 只要公式是对的，结果就是对的

$$\max(n_1,\ n_2,\ \ldots\ n_k)\begin{cases} n_1,\ (k\ =\ 1) \\ n_k\ >\ \max(n_1,\ n_2,\ \ldots\ n_{k-1})\ ?\ n_k\ :\ \max(n_1,\ n_2,\ \ldots\ n_{k-1}),\ (k\ =\ 2,3,\ \ldots) \end{cases}$$

```
const array = [23,99,17,28,84]
function max(array){
  if(array.length === 1){return array[0]}
  const otherMax = max(array.slice(1))
  return array[0] > otherMax ? array[0] : otherMax
}
max(array) // 99
```

```
const array = [23,99,17,28,84]
function max(array){
  if(array.length === 1){return array[0]}
  const otherMax = max(array.slice(1))
  return array[0] > otherMax ? array[0] : otherMax
}
max(array) // 99
```

改成新语法

```
const array = [23,99,17,28,84]
const maxOfTwo = (a,b) => a > b ? a : b
const max = ([first, ...others]) =>
  others.length < 1 ? first :
    maxOfTwo(first, max(others))

max(array) // 99
```

理解递归

```
const max = ([first, ...others]) =>
  others.length < 1 ? first :
    maxOfTwo(first, max(others))
```

max([23,99,17,28,84])
= m(23, max([99,17,28,84]))
= m(23, m(99, max([17,28,84])))
= m(23, m(99, m(17, max([28, 84]))))
= m(23, m(99, m(17, m(28, max([84])))))
= m(23, m(99, m(17, m(28, 84))))
= m(23, m(99, m(17, 84)))
= m(23, m(99, 84))
= m(23, 99)
= 99

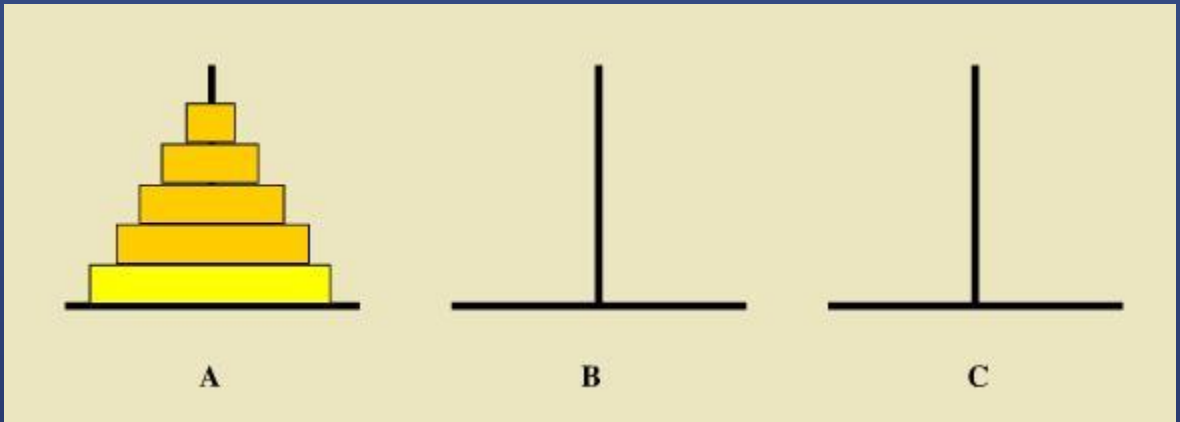
通过代入法，知道递归就是先递进，再回归。

如何证明这个算法是对的?

- 证明
  - ✓ 首先证明公式是对的（较难）
  - ✓ 然后证明代码和公式等价（简单）
- 总结
  - ✓ 数学方法更容易通过形式化证明保证代码的正确性
  - ✓ 但数学方法效率不一定高（但可以优化）
  - ✓ 数学方法往往不够直观，普通人没有什么数学知识
  - ✓ 一般不能对变量进行二次赋值，因为数学里没有

两种方法孰优孰劣

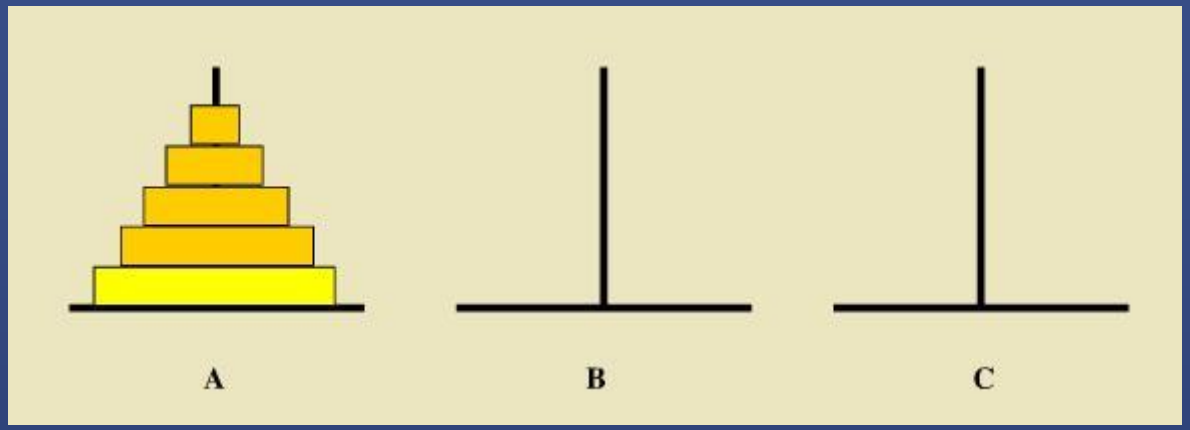
目前还没有定论



汉诺塔问题

简单算法

题干



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。

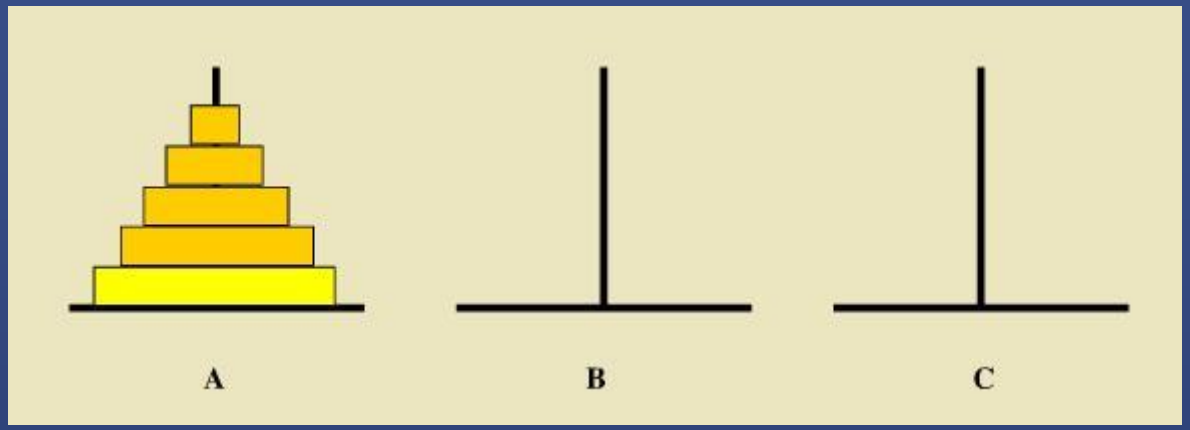
要求按下列规则将所有圆盘移至 C 杆：

- 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？

题干



怎么把 A 上面的 N 个移到 C?

这还不简单，先把 A 上面的 N-1 个移到 B，然后把最大的移到 C，最后把 B 上面的 N - 1 个移到 C。

那怎么把 A 上面的 N-1 个移到 B？ 把 B 上面的 N-1 个移到 C

这还不简单，先把 A 上面的 N - 2 个移到 C，然后把最大的移到 B，最后把 C 上面的 N - 2 个移到 B?

那怎么把 A 上面的 N - 2 个移到 C? ……

普通人类思维想不通，试试数学思维

汉诺塔问题

- 简化
  - 把 A 顶部的盘移到 B，记为 AB
  - AB + AC 表示先 AB，然后 AC
  - h(n, A, B, C) 表示 n 盘在 A ，想去 C ， B 无用。
  - h(1, A, B, C) = AC
  - h(2, A, B, C) = h(1, A,C,B) + AC + h(1,B,A,C)
  - h(2, A, B, C) = AB + AC + BC
  - h(3, A, B, C) = h(2, A,C,B) + AC + h(2,B,A,C)
  - h(n, A, B, C) = h(n-1, A, C, B) + AC + h(n-1, B, A, C)
- 归纳
  - 发现 n 的问题，总是可以化成两个 n - 1 的问题

完全归纳法（数学归纳法）

- 步骤
  - 证明当 n = 1 时命题成立
  - 证明如果 n = k 时命题成立，那么 n = k + 1 时命题也成立（k = 1, 2, 3 ...）
  - 证毕
- 变形
  - 不一定要从 1 开始
  - 不一定每次都要加 1，比如加 2 以证明对所有奇数成立
  - 很可以反着来，从大数开始，每次减一

不完全归纳法

- 步骤
  - 观察前 N 步
  - 得出一个经验性的结论
- 特点
  - 可能是错的，比如观察 2, 4, 6, __ 10 后认为应该填入 8
  - 人类一般使用这种方法认知世界

公式

$$h(n,\ A,\ B,\ C)\begin{cases} AC,\ (n=1) \\ h(n-1,A,C,B)\ +\ AC\ +\ h(n-1,B,A,C) \end{cases}$$

- 转为代码

```
h = (n, from, cache, to) =>
  n === 1 ? `${from}${to}` :
     h(n-1, from, to, cache) + ',' + `${from}${to},` +
h(n-1, cache, from, to)
```

数学解法有什么问题

用斐波那契数列能很清楚地看出问题

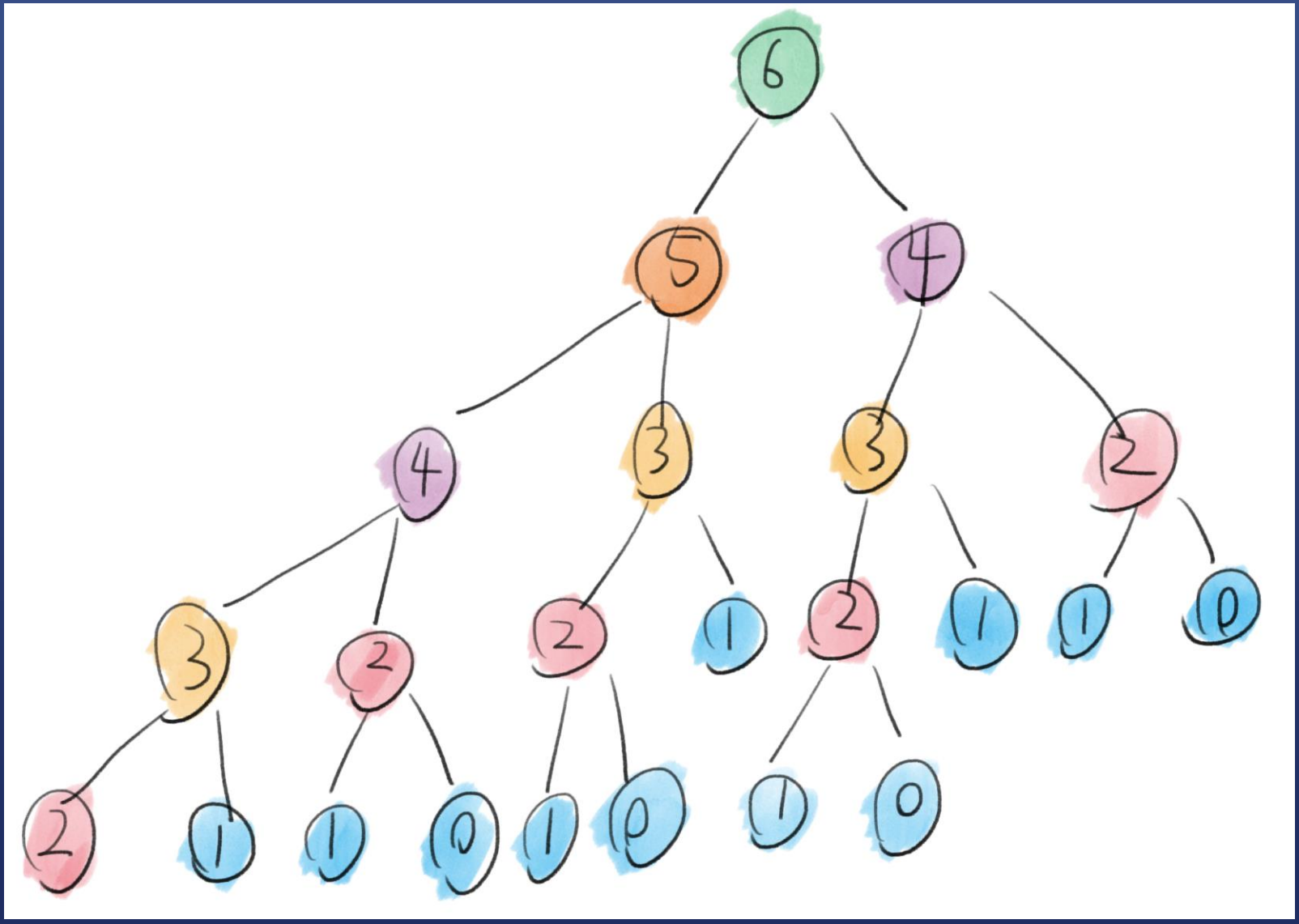
斐波那契数列

$$f(n)\begin{cases}0,\ n=0\\1,\ n=1\\f(n-1)+f(n-2),\ n\geqslant 2\end{cases}$$

- 代码

```
f = n =>
  n === 0 ? 0 :
  n === 1 ? 1 :
    f(n-1) + f(n-2)
```

f(6) 计算过程



重复计算太多了

- 次数
  - f(5) 一次
  - f(4) 两次
  - f(3) 三次
  - f(2) 五次
  - f(1) 和 f(0) 共 11 次
  - 左右相加 11 次
  - 总共 33 次操作，拜托，这才是 f(6)
- 很慢
  - 试试 f(40)、f(48) 需要多久算出
  - 千万不要运行 f(100)，没开玩笑
  - 可以通过「记忆化」来优化（空间换时间）

用普通人类思路

秒杀它

斐波那契数列

$$f(n)\begin{cases}0,\ n=0\\1,\ n=1\\f(n-1)+f(n-2),\ n\geqslant 2\end{cases}$$

- 代码

```
f = n => {
  const array = [0,1]
  for(let i=2; i<=n; i++){
    array[i] = array[i-1] + array[i-2]
  }
  return array[n]
}
// f(6) 和 f(48) 都不在话下
```

人类思路总是更好吗

并不是，首先你得定义什么是「好」

对比

- 数学思路更注重形式（结构）
  - ✓ 往往更加优雅、简单
  - ✓ 因此其实更容易优化
  - ✓ 投身于数学，有无限广阔的可能性
- 人类思维更注重过程（命令）
  - ✓ 往往更容易执行、被理解
  - ✓ 对人脑的负担更重
  - ✓ 被人类的经验所局限
- 总结
  - ✓ 复杂度守恒：复杂度不会因为任何原因降低
  - ✓ 你原意把复杂度放在人脑这边，还是机器那边?
  - ✓ 实际上，我们可以结合两种思路，各取所长

递归的缺点及优化

堆栈溢出和重复计算

栈溢出 Stackoverflow

- 场景
  - 一个函数没有结束，却要进入另一个函数
  - 那么在进入之前，就要保存「现场」
  - 另一个函数调用完毕之后，回到「现场」
- 实现
  - 一般使用栈 Stack 数据结构保存现场
  - 现场一般包括 return 到哪、变量、参数等
  - 此栈称为调用栈 Call Stack
  - 存入现场称为压栈，取出现场称为弹栈

举例：阶乘函数

- 代码

```
f = n =>
   n === 1 ? 1 :
             n * f(n-1)

f(4)
= 4 * f(3)
= 4 * (3 * f(2))
= 4 * (3 * (2 * f(1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
24
```

调用栈举例

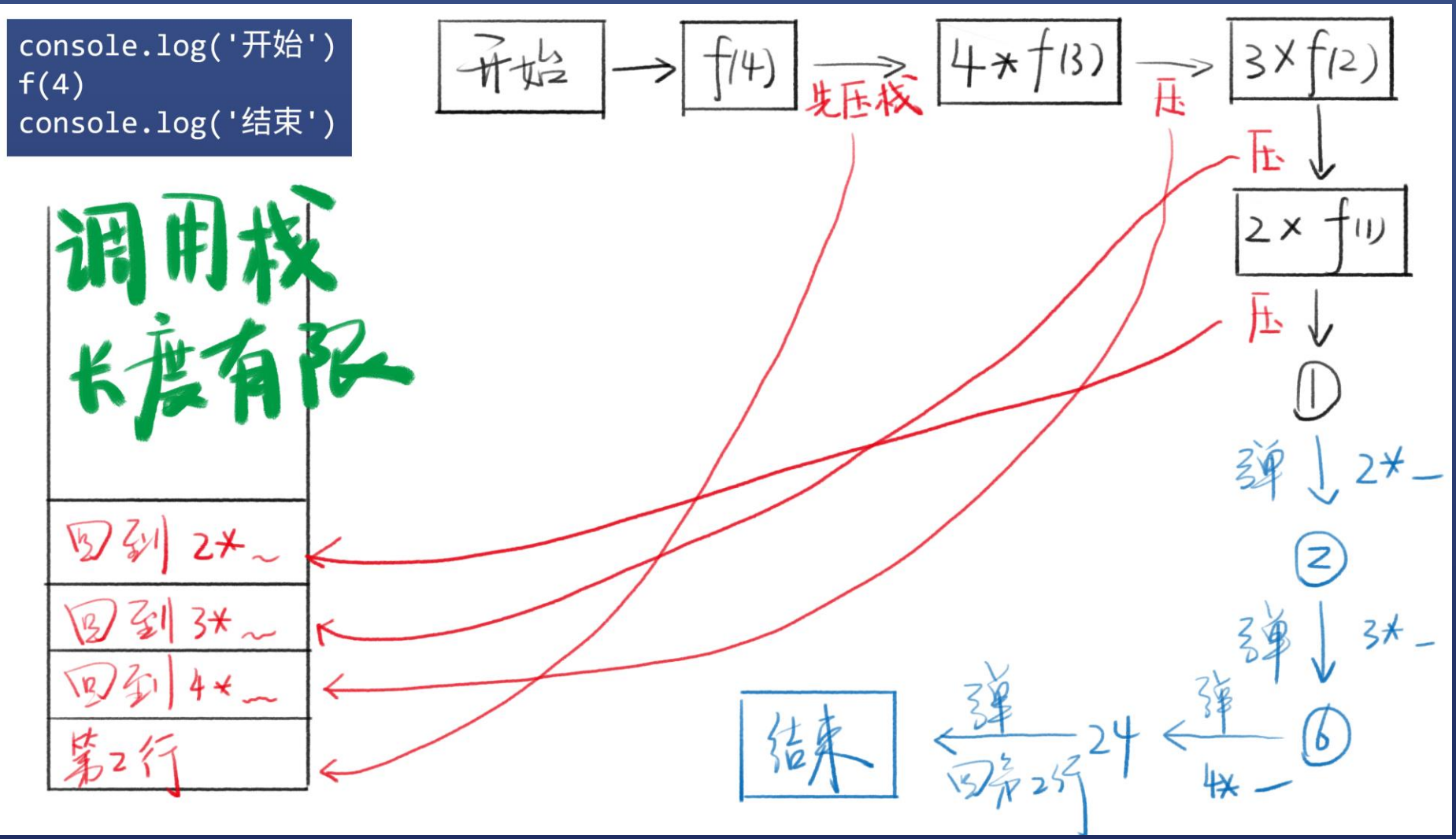
- 代码

```
console.log('开始')
f(4)
console.log('结束')
```

- 分析

调用栈举例

```
console.log('开始')
f(4)
console.log('结束')
```



Call Stack 有多长

每种语言每种实现不一样

以 JS 为例

- 测试代码

```
function computeMaxCallStackSize() {
  try {
    return 1 + computeMaxCallStackSize();
  } catch (e) {
    // 报错说明 stack overflow 了
    return 1;
  }
}
```

- 结果

- ✓ Chrome 12578
- ✓ Firefox 26773
- ✓ Node 12536

大部分情况调用栈够用

一旦不够用，程序就只能中止了

墨菲定律

Anything that can go wrong will go wrong

小概率事件必然会发生

如何减少压栈

- 一、不用递归
  - ✓ 用循环代替递归
- 二、用尾递归+尾递归优化
  - ✓ 用尾递归代替递归
  - ✓ 尾递归就是不用回到现场的递归调用

循环代替递归

- 原代码

```
f = n =>
  n === 1 ? 1 :
            n * f(n-1)
```

- 新代码

```
f = n => {
  let result = 1
  for(let i = 1; i <= n; i++){
    result = result * i
  }
  return result
}
```

所有递归都能写成循环

知乎问答

使用迭代代替递归

- 循环代码

```
f = n => {
  let result = 1
  for(let i = 1; i <= n; i++){
    result = result * i
  }
  return result
}
```

- 分析上面代码

i	1	2	3	4	5
result	1	2	6	24	120

使用迭代代替递归 - 续

i	1	2	3	4	5
result	1	2	6	24	120

- 规律
  - ✓ 只有 i 和 result 在变化，而且是一起变
- 循环迭代代码

```
f = n => {
  let i = 1, result = 1, next_i, next_result
  while(i <= n-1){ // 为什么不是n? 测试出来的
    next_i = i+1
    next_result = next_i * result
    i = next_i
    result = next_result
  }
  return result
}
```

使用迭代代替递归 - 续2

i	1	2	3	4	5
result	1	2	6	24	120

- 规律
  - 只有 i 和 result 在变化，而且是一起变
- 尾递归迭代代码

```
f = n => {
  迭代 = (i,n,result)=>
    i === n ? result
            : 迭代(i+1,n,result*(i+1))
  return 迭代(1, n, 1)
}
```

- 迭代参数 n 可以省略，不过我建议不要省，省了就又变成普通递归了

什么是尾递归

就是递归只出现在 return 语句

且 return 语句里只有递归

代入法理解迭代

- 计算 6 的阶乘

- f(6)
- = 迭代(1, 6, 1)
- = 迭代(2, 6, 2)
- = 迭代(3, 6, 6)
- = 迭代(4, 6, 24)
- = 迭代(5, 6, 120)
- = 迭代(6, 6, 720)
- = 720

还可以这样写

```
f = (n, result) =>
  n === 1 ? result
          : f(n-1, n*result)
f(6, 1)
= f(5, 6)
= f(4, 30)
= f(3, 120)
= f(2, 360)
= f(1, 720)
= 720
```

什么是迭代

注意看「调用的形状」 - 来自SICP

迭，朝代更迭的迭，义为「不停地换」

所以迭代就是「不停地换代」

普通递归 V.S. 迭代

- 代码

```
f(n) = n =>
  n === 1 ? 1 :
            n * f(n-1)

f(4)
= 4 * f(3)
= 4 * (3 * f(2))
= 4 * (3 * (2 * f(1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
24
```

这是递归

- 计算 6 的阶乘
- ✓ f(6)
- ✓ = 迭代(1, 6, 1)
- ✓ = 迭代(2, 6, 2)
- ✓ = 迭代(3, 6, 6)
- ✓ = 迭代(4, 6, 24)
- ✓ = 迭代(5, 6, 120)
- ✓ = 迭代(6, 6, 720)
- ✓ = 720

这是迭代

递归需要归
迭代不需要归

迭代每次都进入一个新的状态，抛弃旧状态

迭代的实现方式

循环或者尾递归

迭代需要压栈吗

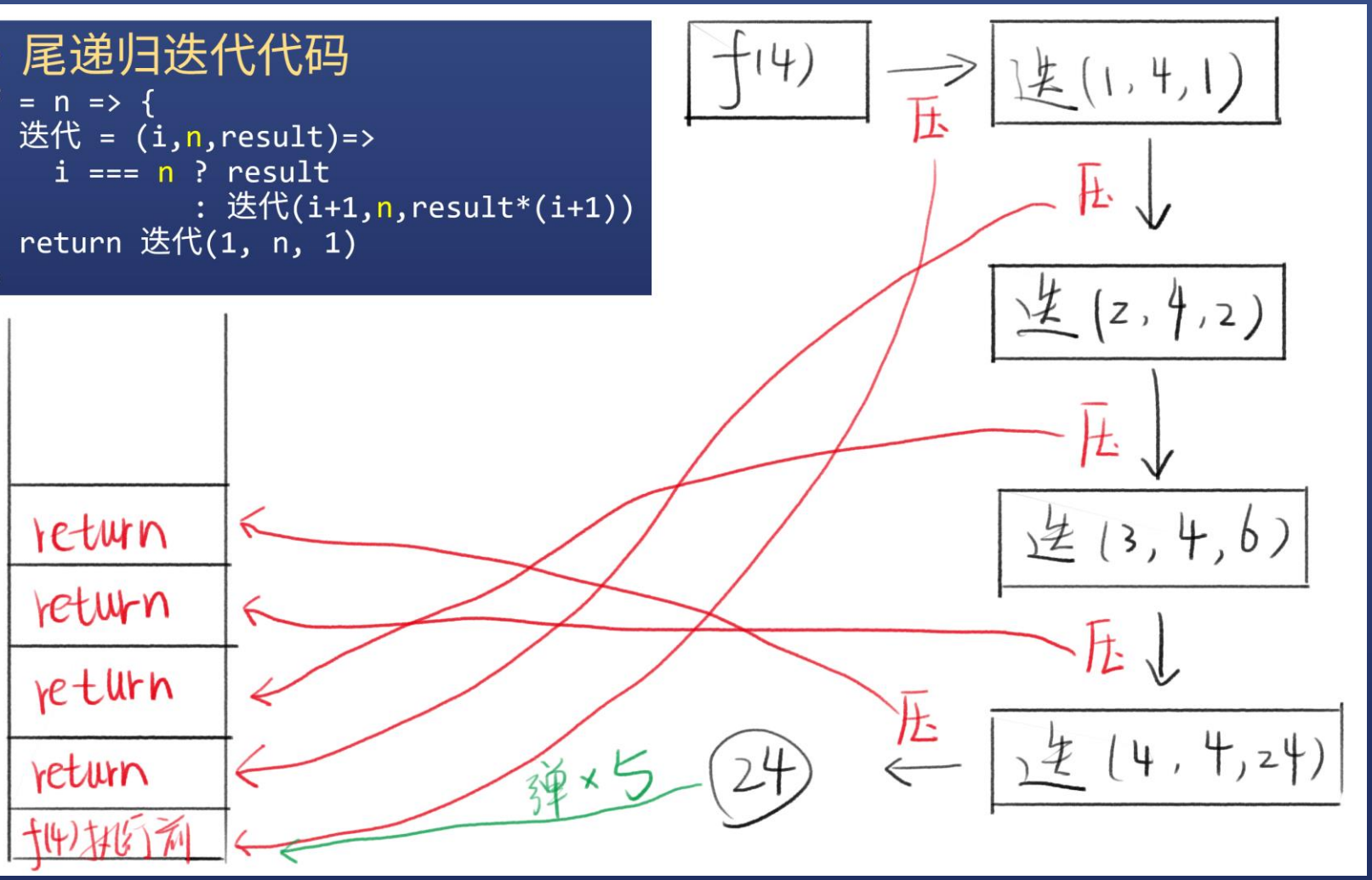
刚刚我们分析了形式

但是实际执行又是另一回事

尾递归迭代的压栈

- 尾递归迭代代码

```
f = n => {
  迭代 = (i,n,result)=>
    i === n ? result
            : 迭代(i+1,n,result*(i+1))
  return 迭代(1, n, 1)
}
```



可以看到，四次 return 的压栈完全是「多余的」

尾调用优化

尾递归是一种特殊的尾调用

尾调用优化

- 为什么可以优化
  - function a(){ return b() }
  - 因为 b() 算出结果后，不需要任何其他操作（如读变量、计算），直接去 a 该返回的地方
  - 所以为什么不干脆省掉中间的压栈
- 缺点
  - 这样做会在调试代码时，丢失调用历史
  - 目前只有 Safari 实现了 JS 的尾调用优化，V8 没有实现

总结

- 递归需要压栈，而栈的长度有限
- 可以使用循环代替递归
- 可以使用迭代代替普通递归
- 迭代用循环实现，也可以用递归
- 迭代理论上不需要压栈，但实际上有
- 尾调用优化可以消除不必要的压栈
- JS 没有完全普及尾调用优化

递归的缺点及优化

堆栈溢出和重复计算

使用记忆化消除重复计算

Memorize

斐波那契数列

- 记忆化之后的代码

```
f = n =>
  n === 0 ? 0 :
  n === 1 ? 1 :
    mf(n-1) + mf(n-2)

memorize = fn => {
  cache = {}
  return (first, ...args) => {
    if(!(first in cache)){
      cache[first] = fn(first, ...args)
    }
    return cache[first]
  }
}

mf = memorize(f)
```

斐波那契数列

- 记忆化之后的代码简写

```
memorize = fn => {
  cache = {}
  return (first, ...args) => {
    if(!(first in cache)){
      cache[first] = fn(first, ...args)
    }
    return cache[first]
  }
}

f = memorize(
  n =>
    n === 0 ? 0 :
    n === 1 ? 1 :
      f(n-1) + f(n-2)
)
```

现在试试 f(47)

是不是快了许多，因为减少了大批重复计算

总结

- 算法的思路
  - 人类思路：根据人类活动类比出思路
  - 数学思路：根据数学知识得到思路
- 递归
  - 先递进，再归纳
  - stack overflow
  - 尾调用优化
  - 循环很有用
  - 记忆化很有用
  - 迭代要理解

再见

下节课开始排序算法