算法入门 - 下

接上集

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

复习代码

```
let sort = (numbers) => {
  if(numbers.length > 2){
    let index = minIndex(numbers)
    let min = numbers[index]
    numbers.splice(index, 1)
    return [min].concat(sort(numbers))
  }else{
    return numbers[0]<numbers[1] ? numbers :
           numbers.reverse()
  }
}
```

一句话总结:

每次找到最小的数放前面，然后对后面的数做同样的事情

学算法为什么难

- 说起来容易做起来难
  - ✓ 拿起键盘干！
  - ✓ 运行，发现写错，用 log 调试
  - ✓ 再运行，再写错，再调试
  - ✓ 再运行，再写错，再调试
  - ✓ 运行，没有错！
  - ✓ 继续想更好的写法（循环）
- 动手最重要
  - ✓ 就算你看了正确答案，你自己写也会错
  - ✓ 没有人可以帮你完成这个过程

minIndex

- 你永远都有两种写法
  - 「递归」和「循环」
- 目前的 minIndex

```
let minIndex = (numbers) =>
  numbers.indexOf(min(numbers))
let min = (numbers) => {
  return min(
    [numbers[0], min(numbers.slice(1))]
  )
}
```

- 缺点：看着就繁琐
  - 重写吧

minIndex 重写

```
let minIndex = (numbers) =>
  let index = 0
  for(let i=1; i<numbers.length; i++){
    if(numbers[i] < numbers[index]){
      index = i
    }
  }
  return index
}
```

- 分析
  - ✓ 一目了然，一听就会，一错就错
  - ✓ 写错记得测试几次

所有递归都可以改写成循环

这是真的

那我们把 sort 改写一下

递归变循环

复习代码

```
let sort = (numbers) => {
  if(numbers.length > 2){
    let index = minIndex(numbers)
    let min = numbers[index]
    numbers.splice(index, 1)
    return [min].concat(sort(numbers))
  }else{
    return numbers[0]<numbers[1] ? numbers :
           numbers.reverse()
  }
}
```

思路不变:

每次找到最小的数放前面，~~然后对后面的数做同样的事情~~

然后 i++

循环

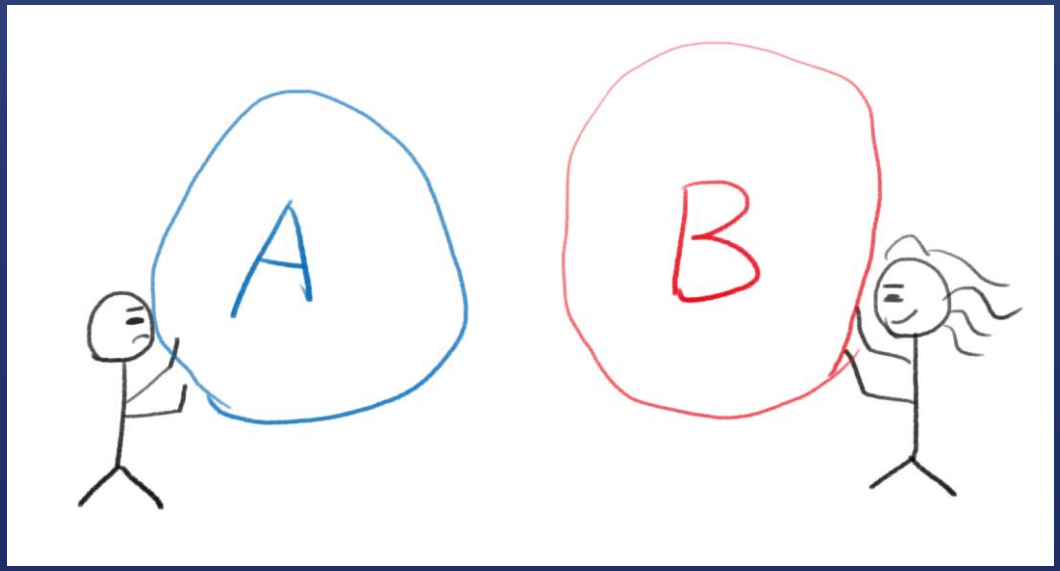
```
let sort = (numbers) => {
  for(let i=0; i<???; i++){
    let index = minIndex(numbers)
    // index 是当前最小数的下标
    // index 对应的数应该放到 i 处
    swap(numbers, index, i) // swap 还没实现
    // index、i 都是 index 的意思，建议 i 改名
  }
}
```

- 分析

✓ 怎么知道 i<??? 处应该写什么
✓ 提前写好 minIndex 能有效简化问题
✓ 用 swap 占位能有效简化问题

实现 swap

```
let swap = (array, i, j) => {
  let temp = array[i]
  array[i] = array[j]
  array[j] = temp
}
swap(numbers, 1, 2)
```



错误地实现 swap

```
let swap = (a, b) => {
  let temp = a
  a = b
  b = temp
}
swap(numbers[1], numbers[2])
```

你会发现，numbers[1] 和 numbers[2] 的值原封不动
因为 a b 是简单类型，传参的时候会复制值
而上一页 PPT 中的 numbers 是对象，传参复制地址
传值 V.S. 传址

循环

```
let sort = (numbers) => {
  for(let i=0; i<???; i++){
    let index = minIndex(numbers)
    // index 是当前最小数的下标
    // index 对应的数应该放到 i 处
    swap(number, index, i) // swap 还没实现
    // index、i 都是 index 的意思，建议 i 改名
  }
}
```

- 分析
  - ✓ 怎么知道 i<??? 处应该写什么

分析代码

```
let sort = (numbers) => {
  for(let i=0; i<???; i++){
    let index = minIndex(numbers)
    swap(number, index, i)
  }
}
```

- 假设 numbers 的长度为 n

n	i	
4	0	swap(numbers, index, 0), index 可能为 0/1/2/3
4	1	swap(numbers, index, 1), index 可能为 0/1/2/3
4	2	swap(numbers, index, 2), index 可能为 0/1/2/3
4	3	swap(numbers, index, 3), index 可能为 0/1/2/3

发现了问题

- minIndex 查找范围有问题
  - ✓ `let index = minIndex(numbers)`
  - ✓ 这句话有问题，如果上次循环已经找到了第一个最小的数字，那么之后找最小数字的时候，就可以忽略第一个
  - ✓ `let index = minIndex(numbers.slice(i)) + i`

i	忽略几个
0	0
1	1
2	2

- 为什么要 +i
  - ✓ 如果不加，那么 index 总是从 0 数起

重新分析代码

```
let sort = (numbers) => {
  for(let i=0; i<???; i++){
    let index = minIndex(numbers.slice(i))+i
    swap(number, index, i)
  }
}
```

- 假设 numbers 的长度 n = 4

n	i	
4	i = 0	swap(numbers, index, 0), index 可能为 1/2/3
4	i = 1	swap(numbers, index, 1), index 可能为 2/3
4	i = 2	swap(numbers, index, 2), index 可能为 3
4	i = 3	number.slice(3) 是 [i]，所以 i = 3 不行，所以 i < n - 1

最终代码

```
let sort = (numbers) => {
  for(let i=0; i< numbers.length -1; i++){
    console.log(`----`) //这个log很精髓
    console.log(`i: ${i}`)
    let index = minIndex(numbers.slice(i))+ i
    console.log(`index: ${index}`)
    console.log(`min: ${numbers[index]}`)
    if(index!==i){
      swap(numbers, index, i)
      console.log(`swap ${index}: ${i}`)
      console.log(numbers)
    }
}
  return numbers
}
```

```
let swap = (array, i, j) => {
  let temp = array[i]
  array[i] = array[j]
  array[j] = temp
}
let minIndex = (numbers) => {
  let index = 0
  for(let i=1; i<numbers.length; i++){
    if(numbers[i] < numbers[index]){
      index = i
    }
  }
  return index
}
```

最终代码删掉log

```
let sort = (numbers) => {
  for(let i=0; i< numbers.length -1; i++){
    let index = minIndex(numbers.slice(i))+ i
    if(index!==i){swap(numbers, index, i)}
  }
  return numbers
}
```

```
let swap = (array, i, j) => {
  let temp = array[i]
  array[i] = array[j]
  array[j] = temp
}
let minIndex = (numbers) => {
  let index = 0
  for(let i=1; i<numbers.length; i++){
    if(numbers[i] < numbers[index]){
      index = i
    }
  }
  return index
}
```

总结

- 所有递归都能改成循环
- 循环的时候有很多细节
  - ✓ 这些细节很难想清楚
  - ✓ 要动手列出表格找规律
  - ✓ 尤其是边界条件很难确定
  - ✓ 我们没有处理长度为 0 和 1 的数组
- 如果 debug
  - ✓ 学会看控制台
  - ✓ 学会打 log
  - ✓ 打 log 的时候注意加标记

选择排序搞定

每次选择最小/大的，选完就排完

快速排序

quick sort

递归思路

- 以某某为基准
- 想象你是一个体育委员
- 你面对的同学为 [12, 3, 7, 21, 5, 9 4, 6]
- 「以某某为基准，小的去前面，大的去后面」
- 你只需要重复说这句话，就能排序
- 神奇不神奇?
- 用图说明一下

快排源码

- 阮一峰写的版本

```
let quickSort = arr => {
    if (arr.length <= 1) { return arr; }
    let pivotIndex = Math.floor(arr.length / 2);
    let pivot = arr.splice(pivotIndex, 1)[0];
    let left = [];
    let right = [];
    for (let i = 0; i < arr.length; i++){
        if (arr[i] < pivot) { left.push(arr[i])
        } else { right.push(arr[i]) }
    }
    return quickSort(left).concat(
                [pivot], quickSort(right) )
}
```

归并排序

merge sort

递归思路

- 不以某某为基准
  - 想象你是一个体育委员
  - 你面对的同学为 [12, 3, 7, 21, 5, 9 4, 6]
  - 左边一半排好序，右边一半排好序
  - 然后把左右两边合并(merge)起来
  - 神奇不神奇?
  - 用图说明一下

归并排序源码

```
let mergeSort = arr =>{
  let k = arr.length
  if(k===1){return arr}
  let left = arr.slice(0, Math.floor(k/2))
  let right = arr.slice(Math.floor(k/2))
  return merge(mergeSort(left), mergeSort(right))
}
let merge = (a, b) =>{
  if(a.length === 0) return b
  if(b.length === 0) return a
  return a[0] > b[0] ?
     [b[0]].concat(merge(a, b.slice(1))) :
     [a[0]].concat(merge(a.slice(1), b))
}
```

总结

- 目前我们学了三种排序
  - ✓ 选择排序
  - ✓ 快速排序
  - ✓ 归并排序
- 接下来我们学
  - ✓ 计数排序

计数排序

- 思路

- ✓ 用一个哈希表作记录
- ✓ 发现数字 N 就记 N: 1，如果再次发现 N 就加1
- ✓ 最后把哈希表的 key 全部打出来，假设 N: m，那么 N 需要打印 m 次
- ✓ 画图演示

计数排序代码

```
let countSort = arr =>{
  let hashTable = {}, max = 0, result = []
  for(let i=0; i<arr.length; i++){ // 遍历数组
    if(!(arr[i] in hashTable)){
      hashTable[arr[i]] = 1
    }else{
      hashTable[arr[i]] += 1
    }
    if(arr[i] > max) {max = arr[i]}
  }
  for(let j=0; j<=max; j++){ // 遍历哈希表
    if(j in hashTable){
      result.push(j)
    }
  }
  return result
} // 目前代码有 bug
```

计数排序代码2

```
let countSort = arr =>{
  let hashTable = {}, max = 0, result = []
  for(let i=0; i<arr.length; i++){ // 遍历数组
    略...
  }
  for(let j=0; j<=max; j++){ // 遍历哈希表
    if(j in hashTable){
      for(let i = 0; i<hashTable[j]; i++){
        result.push(j)
      }
    }
  }
  return result
}
```

计数排序的特点

- 数据结构不同
  - 使用了额外的 hashTable
  - 只遍历数组一遍（不过还要遍历一次 hashTable）
  - 这叫做「用空间换时间」
- 时间复杂度对比
  - 选择排序 O(n^2)
  - 快速排序 O(n log2 n)
  - 归并排序 O(n log2 n)
  - 计数排序 O(n + max)

算法学习总结

- 心法
  - ✓ 战略上藐视敌人，战术上重视敌人
- 特点
  - ✓ 思路都很简单
  - ✓ 细节都很多
  - ✓ 多画表，多画图，多 log
  - ✓ 如果实在不想陷入 JS 的细节，可以用伪代码

还有哪些排序算法

- 冒泡排序
  - https://visualgo.net/zh/sorting
- 插入排序
  - https://visualgo.net/zh/sorting 点击 INS
- 希尔排序
  - http://sorting.at/ 自己选择 Shell Sort
- 基数排序
  - https://visualgo.net/zh/sorting 点击 RAD

堆排序

下次用一整节课了解这个排序