前端算法和数据结构

1. 课前准备

2. 课堂主题

复杂度概念

常见算法

常见数据格式

3. 课堂目标

4. 知识点

复杂度数组链表集合hash表栈队列 树图排序冒泡快速排序原地快排序.二分搜索

复杂度

O的概念，来描述算法的复杂度，简而言之，就是算法执行所需要的执行次数，和数据量的关系(时间复杂度)， 占用额外空间和数据量的关系(空间复杂度)

`O(1)`: 常数复杂度 (和数据量无关)

`O(log n)`:对数复杂度 (每次二分)

`O(n)`: 线性时间复杂度 （数组遍历一次）

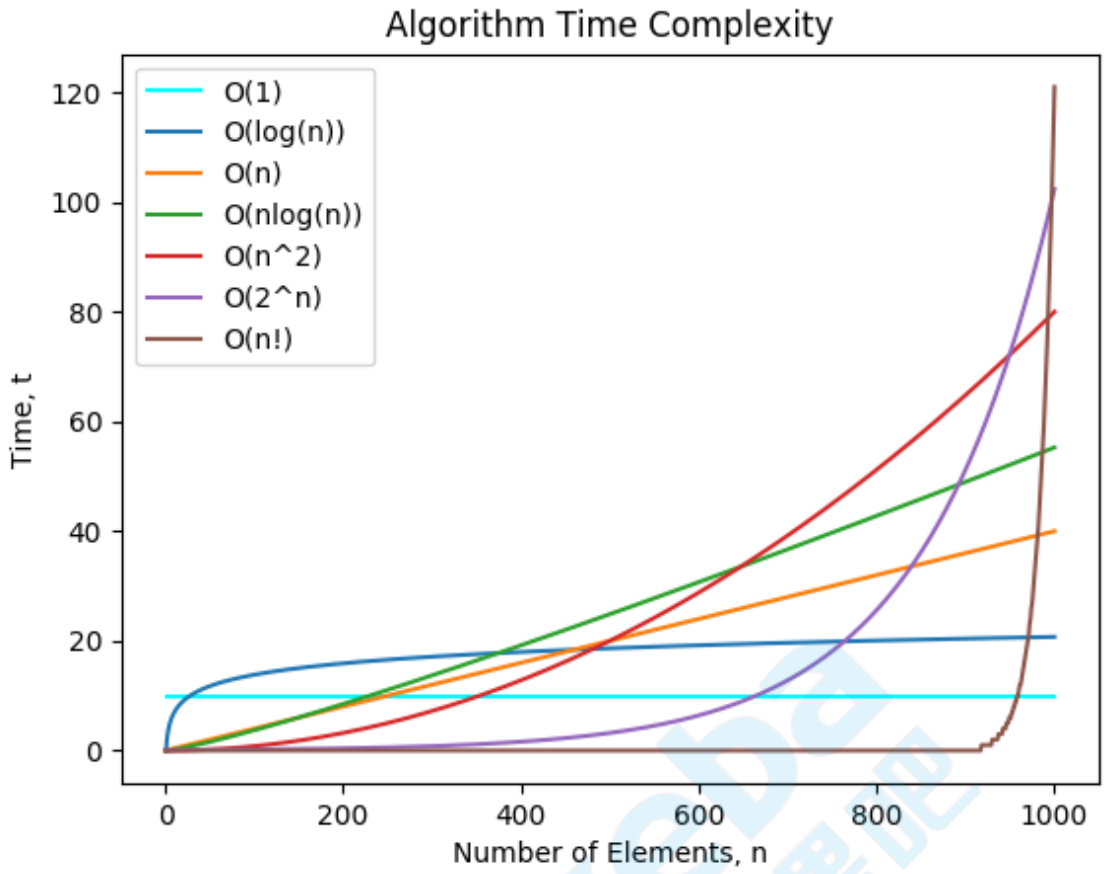
`O(n*log n)`: 线性对数 （遍历+二分）

`O(n^2)`: 平方方 两层遍历

`O(n^3)`: 立立方方

`O(2^n)`: 指数

`O(n!)`: 阶乘



稳定性

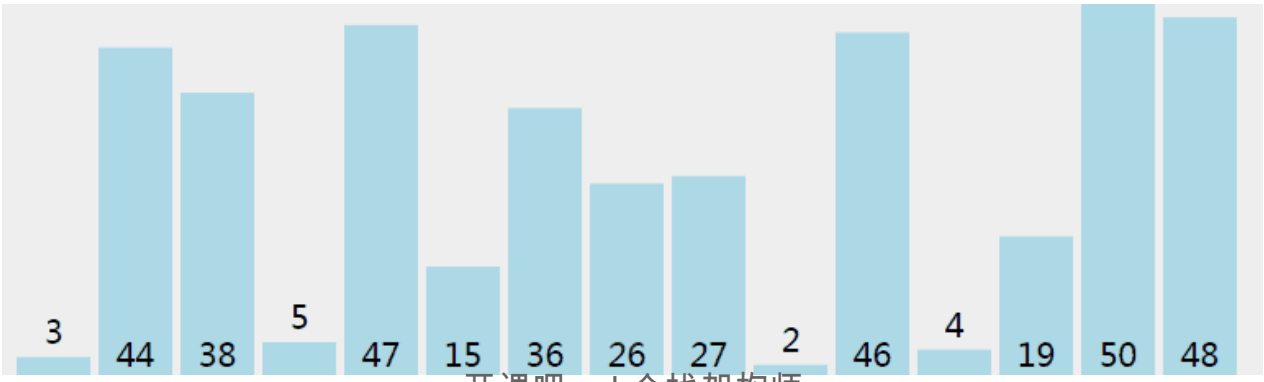
数组中[{name:'xx', age:12}, {name:'kaikeba', age:12}] 如果按照age排序，排序后，xx和kaikeba的相对位置不变，我们成为稳定的算法，否则不稳定

排序

搜索和排序，是计算机的几个基本问题

冒泡

最经典和简单粗暴的排序算法，简而言之，就是挨个对比，如果比右边的数字大，就交换位置 遍历一次，最大的在最右边，重复步骤，完成排序



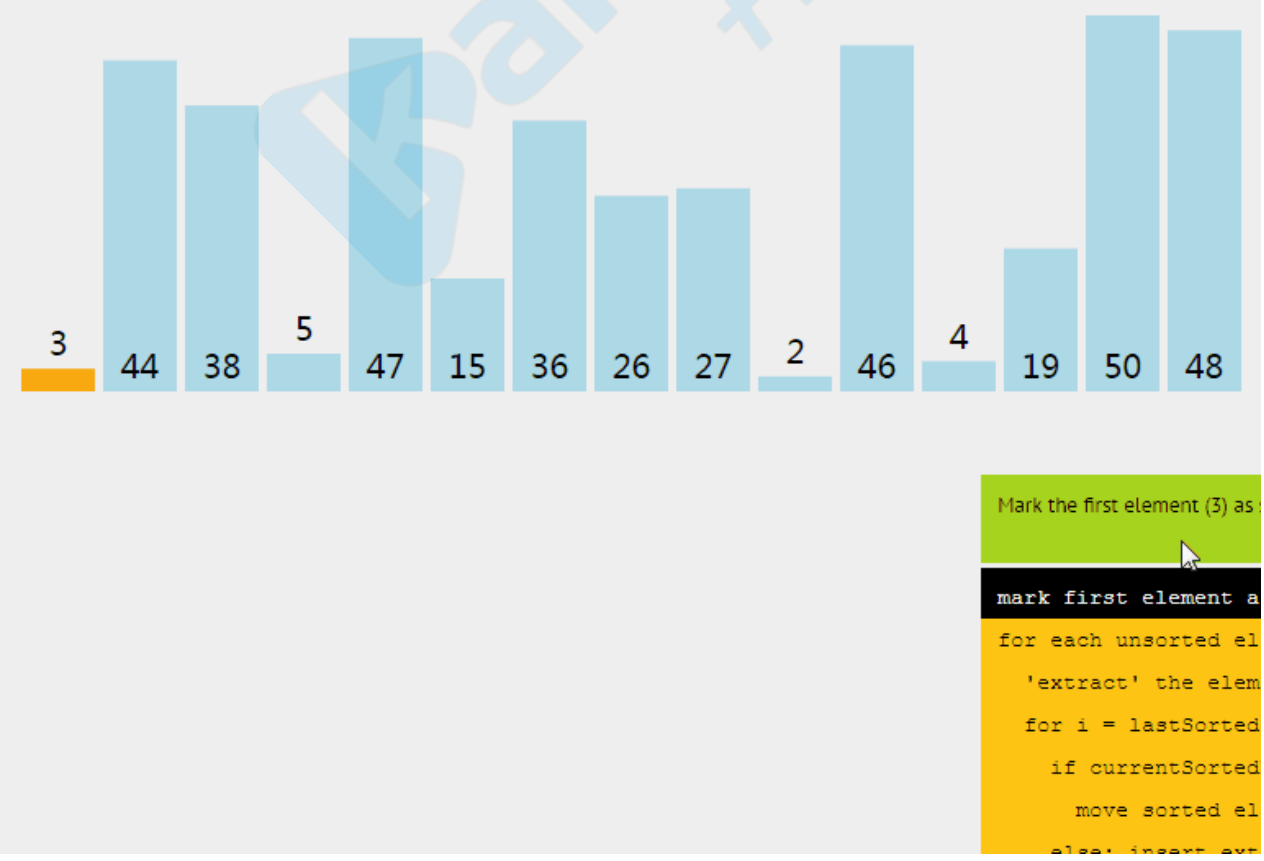
```
function bubleSort(arr) {
    var len = arr.length
    for (let outer = len ; outer >= 2; outer--) {
        for(let inner = 0; inner <=outer - 1; inner++) {
            if(arr[inner] > arr[inner + 1]) {
                [arr[inner],arr[inner+1]] = [arr[inner+1],arr[inner]]
            }
        }
    }
    return arr
}

console.log(bubleSort([4,3,6,1,9,6,2]))
```

问题：冒泡复杂度和稳定性如何

n^2 空间 1 稳定

插入

插入排序逻辑和冒泡类似，只不过没采用挨个交换的逻辑，而是在一个已经排好序的数组里，插入一个元素，让它依然是有序的




```
function insertSort(arr) {
```

```
    for(let i = 1; i < arr.length; i++) {  //外循环从1开始，默认arr[0]是有序段
        for(let j = i; j > 0; j--) {  //j = i,将arr[j]依次插入有序段中
            if(arr[j] < arr[j-1]) {
                [arr[j],arr[j-1]] = [arr[j-1],arr[j]];
            } else {
                break;
            }
        }
    }
    return arr;
}


console.log(insertSort([11,4,3,6,1,9,7,2,0]))
```

n^2 空间 1 稳定

快速排序

这个逼格略高，使用了二分的思想。可以算最重要的排序算法了

大概就是找一个标志位，先遍历一次，所有个头比他矮的，都站左边，比他个头高的，都站右边，遍历一次，就把数组分成两部分，然后两遍的数组，递归执行相同的逻辑

```
function quickSort(arr) {
    if(arr.length <= 1) {
        return arr;  //递归出口
    }
    var left = [],
        right = [],
        current = arr.splice(0,1); //注意splice后，数组长度少了一个
    for(let i = 0; i < arr.length; i++) {
        if(arr[i] < current) {
            left.push(arr[i])  //放在左边
        } else {
            right.push(arr[i]) //放在右边
        }
    }
    return quickSort(left).concat(current,quickSort(right)); //递归
}
```

上面方便理解，额外占用空间, 原地快拍

```
// 原地版
function quickSort1(arr, low = 0, high = arr.length - 1) {
```

```
    if(low >= high) return
    let left = low
    let right = high
    let temp = arr[left]
    while(left < right) {
        if(left < right && temp <= arr[right]) {
            right --
        }
        arr[left] = arr[right]
        if(left < right && temp >= arr[left]) {
            left ++
        }
        arr[right] = arr[left]
    }
    arr[left] = temp
    quickSort1(arr, low, left - 1)
    quickSort1(arr, left + 1, high)
    return arr
}
console.log(quickSort1([11,4,3,6,1,9,7,2,0]))
```

n*logn 空间 不稳定

其他排序算法还有很多，桶排序，堆排序等，还有一个容易挨揍的排序

```
const list = [11,4,3,6,1,9,7,2,0]
const newList = []
list.forEach(item => {
    setTimeout(function () {
        newList.push(item)
        if(newList.length===list.length){
            console.log(newList)
        }
  }, item * 100)
})
```

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

递归

快排我们了解到，递归就是自己调用自己，形成一个调用栈，逐渐缩小目标，到达截止条件返回执行的逻辑，

talk is cheap，举个小例子

数组打平 （扁平化）

```
Array.prototype.flat = function() {
    var arr = [];
    this.forEach((item,idx) => {
        if(Array.isArray(item)) {
            arr = arr.concat(item.flat()); //递归去处理数组元素
        } else {
            arr.push(item)    //非数组直接push进去
        }
    })
    return arr;    //递归出口
}

arr = [1,2,3,[4,5,[6,7,[8,9]]],[10,11]]
console.log(arr.flat())
```

爬楼

有一楼梯共10级，刚开始时你在第一级，若每次只能跨上一级或二级，要走上第10级，共有多少种走法?

其实就是个斐波那契数列，，只有两种方式 从第9层上一级，或者从第8级上二级， 9和8又各自又两种情况

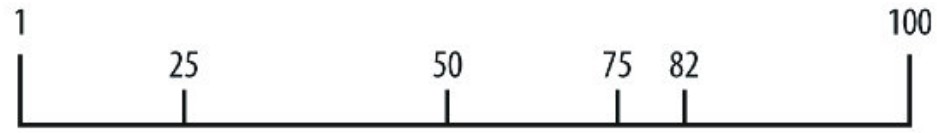
最后推到3级解题，的两种方式1和2 是固定的次数

```
function stairs(n) {
    if(n === 0) {
        return 1;
    } else if (n < 0) {
        return 0
    }
    else {
        return stairs(n-1) + stairs(n-2)
    }
}
console.log(stairs(10))
```

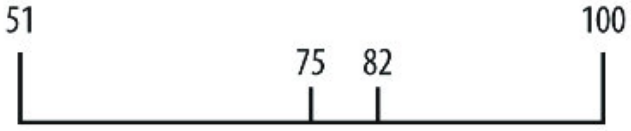
查找

查找比较简单，我们先来看一个经典的二分查找 有点类似幸运52的猜价格，比如让你在1和1000之间猜个数字，挨个猜是很蠢的，要先猜500，如果大了，那就是0~500 ，每次问题减半，很快就能查到

猜数字游戏，目标数字82

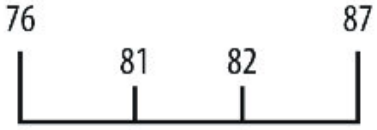


第一次猜测：50，回应：太小

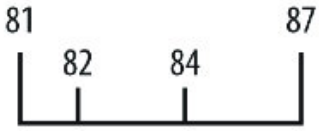


第二次猜测：76，回应：太小

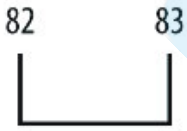
第三次猜测：88，回应：太大



第四次猜测：81，回应：太小



第五次猜测：84，回应：太大



中间值是82.5，记作82

第六次猜测：82，回应：正确

循环版本

```
function binarySearch(arr, target) {
    var low = 0,
        high = arr.length - 1,
        mid;
    while (low <= high) {
        mid = Math.floor((low + high) / 2);
        if (target === arr[mid]) {
            return `找到了${target},在第${mid + 1}个`
        }
        if (target > arr[mid]) {
            low = mid + 1;
        } else if (target < arr[mid]) {
```

```
            high = mid - 1;
        }
    }
    return -1
}

console.log(binarySearch([1,2,3,4,5,7,9,11,14,16,17,22,33,55,65],4))
```

递归版本

```
function binarySearch1(arr,target,low = 0,high = arr.length - 1) {
    const n = Math.floor((low+high) /2);
    const cur = arr[n];
    if(cur === target) {
        return `找到了${target},在第${n+1}个`;
    } else if(cur > target) {
        return binarySearch1(arr,target,low, n-1);
    } else if (cur < target) {
        return binarySearch1(arr,target,n+1,high);
    }
    return -1;
}
```

数据结构

队列

这个很好理解 先入先出，有点像排队，通过数组push和shift模拟，通常用作任务管理

栈

先入后出

```
class Stack {
    constructor() {
        this.items = []
    }

    push(item) {
        this.items.push(item)
    }
```

```
    pop() {
        return this.items.pop()
    }

    size() {
        return this.items.length
    }

    clear() {
        this.items = []
    }
}
```

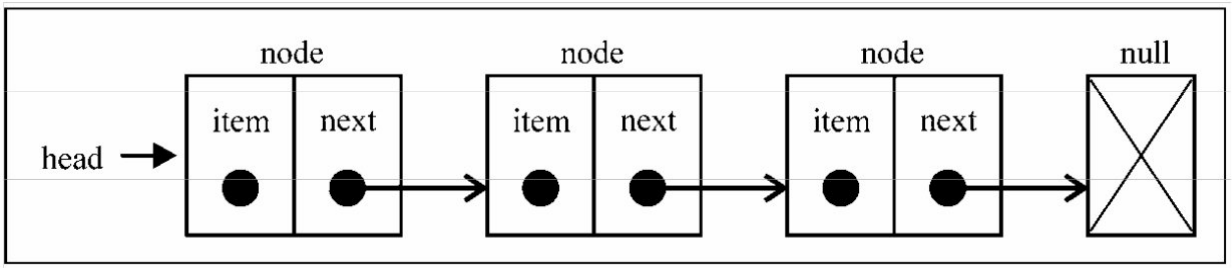
- 索引: `O(n)`
- 搜索: `O(n)`
- 插入: `O(1)`
- 移除: `O(1)`

经典案例： 括号匹配，html标签匹配，表达式计算

```
function isBalance(symbol) {
    const stack = new Stack()
    const left = '{('
    const right = '})'
    let popValue
    let tag = true

    const match = function(popValue, current) {
      if (left.indexOf(popValue) !== right.indexOf(current)) {
        tag = false
      }
    }

    for (let i = 0; i < symbol.length; i++) {
      if (left.includes(symbol[i])) {
        stack.push(symbol[i])
      } else if (right.includes(symbol[i])) {
        popValue = stack.pop()
        match(popValue, symbol[i])
      }
    }
    return tag
  }
  console.log(isBalance('{{(({}))}}'))
  console.log(isBalance('{{(({})}}'))
```

链表

有点像火车，车厢和车厢之间链接，有点是可以随时替换车厢，react最新架构的fiber，就是从树变成了链表，能够让diff任务随时中断




```
class Node{
    constructor(element){
        this.element = element
        this.next = null
    }
}

class LinkedList{
    constructor(){
        this.head = null
        this.current
        this.length = 0
    }

        append(element){
            const node = new Node(element)
            if (this.head === null) {        // 插入第一个链表
              this.head = node
            } else {
              this.current = this.head
              while (this.current.next) {      // 找到最后一个节点
                this.current = this.current.next
              }
              this.current.next = node
            }
            this.length++
    }
        // 移除指定位置元素
        removeAt(position) {
            if (position > -1 && position < this.length) {
              let previous
              let index = 0
              if (position === 0) {          // 如果是第一个链表的话，特殊对待
                this.head = this.head.next
              } else {
```

```
            this.current = this.head
            while (index < position) {  // 循环找到当前要删除元素的位置
              previous = this.current
              this.current = this.current.next
              index++
            }
            previous.next = this.current.next
          }
          this.length--
        }
      }
        // 在指定位置加入元素
insert (position, element) {
    const node = new Node(element)
    let index = 0
    let current, previous
    if (position > -1 && position < this.length + 1) {
      if (position === 0) { // 在链表最前插入元素
        current = this.head
        this.head = node
        this.head.next = current
      } else {
        current = this.head
        while (index < position) { // 同 removeAt 逻辑，找到目标位置
          previous = current
          current = current.next
          index++
        }
        previous.next = node        // 在目标位置插入相应元素
        node.next = current
      }
      this.length++
    }
  }


// 链表中是否含有某个元素，如果有的话返回相应位置，无的话返回 -1
indexOf(element) {
    let index = 0
    this.current = this.head
    while (index < this.length) {
      if (this.current.element === element) {
        return index
      }
      this.current = this.current.next
      index++
    }
    return -1
  }
```

```
    // 移除某元素
    remove(element) {
      const position = this.indexOf(element)
      this.removeAt(position)
    }

    // 获取大小
    size () {
      return this.length
    }

    // 获取最开头的链表
    getHead () {
      return this.head
    }

    // 是否为空
    isEmpty () {
      return this.length === 0
    }

    // 打印链表元素
    log () {
      this.current = this.head
      let str = this.current.element
      while (this.current.next) {
        this.current = this.current.next
        str = str + ' ' + this.current.element
      }
      console.log(str)
      return str
    }
}

  // 测试用例
  var linkedList = new LinkedList()
  linkedList.append(5)
  linkedList.append(10)
  linkedList.append(15)
  linkedList.append(20)
  linkedList.log()          // '5 10 15 20'
  linkedList.removeAt(1)
  linkedList.log()          // '5 15 20'
  linkedList.insert(1, 10)
  linkedList.log()
```

时间复杂度:

- 索引: `O(n)`
- 搜索: `O(n)`
- 插入: `O(1)`
- 移除: `O(1)`

集合

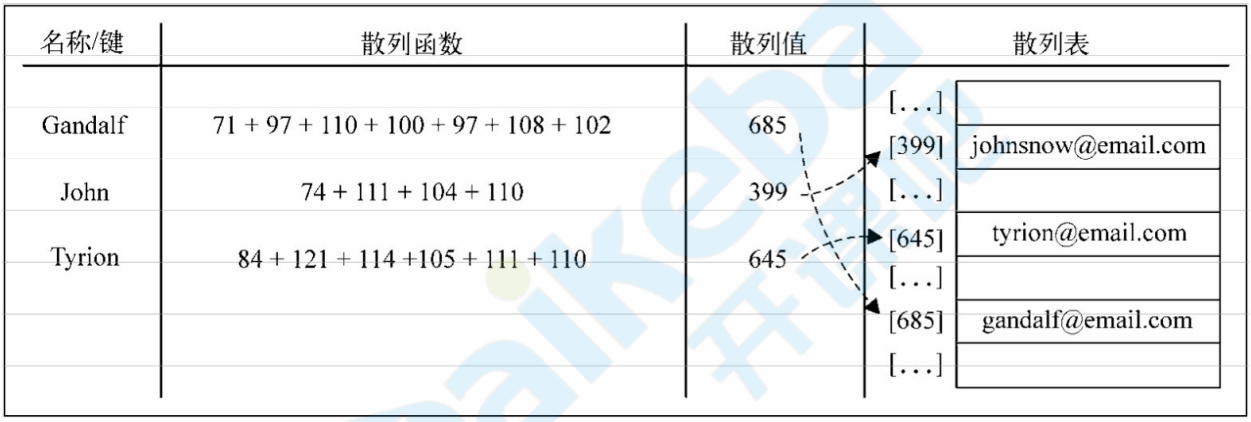
其实就是es6的set，特点就是没有重复数据，也可以用数组模拟

```
class Set {

    constructor() {
        this.items = {}
    }

    has(value) {
        return this.items.hasOwnProperty(value)
    }

    add(value) {
        if (!this.has(value)) {
            this.items[value] = value
            return true
        }
        return false
    }

    remove(value) {
        if (this.has(value)) {
            delete this.items[value]
            return true
        }
        return false
    }

    get size() {
        return Object.keys(this.items).length
    }

    get values() {
        return Object.keys(this.items)
    }
}


const set = new Set()
set.add(1)
```

```
console.log(set.values)  // ["1"]
console.log(set.has(1))  // true
console.log(set.size) // 1
set.add(2)
console.log(set.values)  // ["1", "2"]
console.log(set.has(2))  // true
console.log(set.size) // 2
set.remove(1)
console.log(set.values) // ["2"]
set.remove(2)
console.log(set.values) // []
```

哈希表

哈西其实就是js里的对象，它在实际的键值和存入的哈希值之间存在一层映射。如下例子:

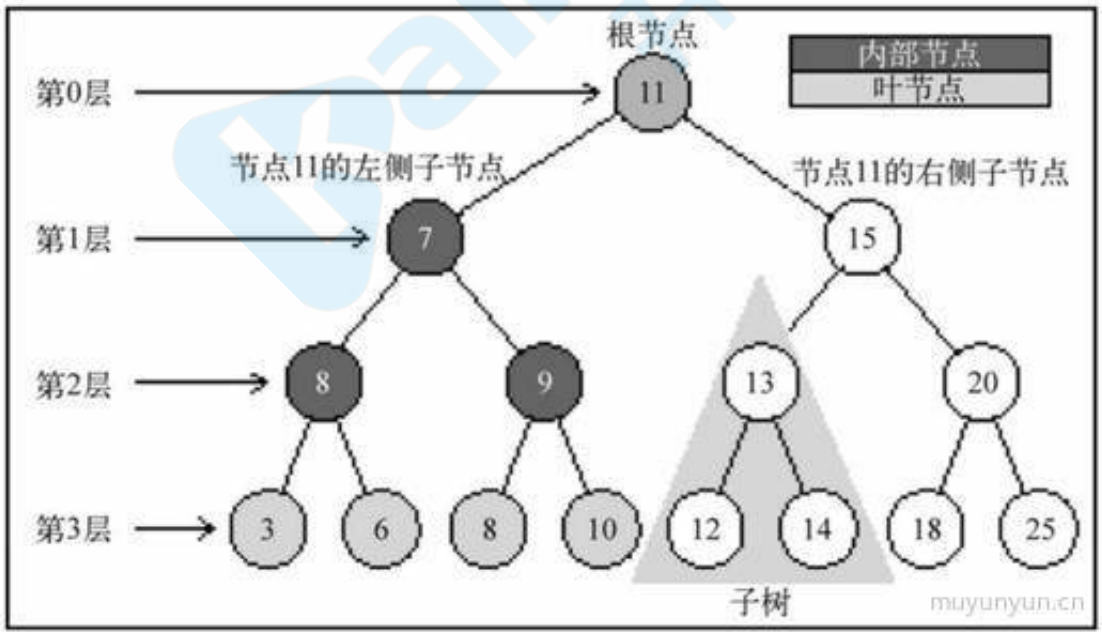



```
class HashTable {
    constructor() {
        this.items = {}

    }
    put(key, value) {
        const hash = this.keyToHash(key)
        this.items[hash] = value
    }

    get(key) {
        return this.items[this.keyToHash(key)]
    }

    remove(key) {
        delete (this.items[this.keyToHash(key)])
    }
    keyToHash(key) {
        let hash = 0
```

```
        for (let i = 0; i < key.length; i++) {
            hash += key.charCodeAt(i)
        }
        hash = hash % 37 // 为了避免 hash 的值过大
        return hash
    }

}

let kkb = new HashTable()
kkb.put('name', 'kaikeba')
kkb.put('age', '6')
kkb.put('best', '大圣老师')

console.log(kkb.get('name') )
console.log(kkb.get('best') )
kkb.remove('name')
console.log(kkb.get('name'))
```

哈希的问题也很明显，比如两个数的hash值一样的时候，会发生碰撞，可以用存储链表的方式来解决(重复的值存在链表里) 这些V8帮我们处理的很好了

树



我们浏览器的dom 就是经典的树结构

这幅图中有如下概念:

- 根节点: 一棵树最顶部的节点
- 内部节点: 在它上面还有其它内部节点或者叶节点的节点
- 叶节点: 处于一棵树根部的节点

- 子树: 由树中的内部节点和叶节点组成

  我们其实可以不用模拟，dom操作就是树

dom遍历

```
<body>


    <div id="app">
        <div>123</div>
        <p>2345</p>
        <div class="demo">
            <span>哈喽</span>
        </div>
    </div>
    <script>


function walk(node, func = () => {}) {
  if (node instanceof window.Node) {
    _walk(node, func);
  }
  return node;
}

function _walk(node, func) {
  if (func(node) !== false) {
    node = node.firstChild;
    while (node) {
      _walk(node, func);
      node = node.nextSibling;
    }
  }
}
walk(document.getElementById('app'), node=>{
    console.log(node)
})
</script>
</body>
```

动态规划

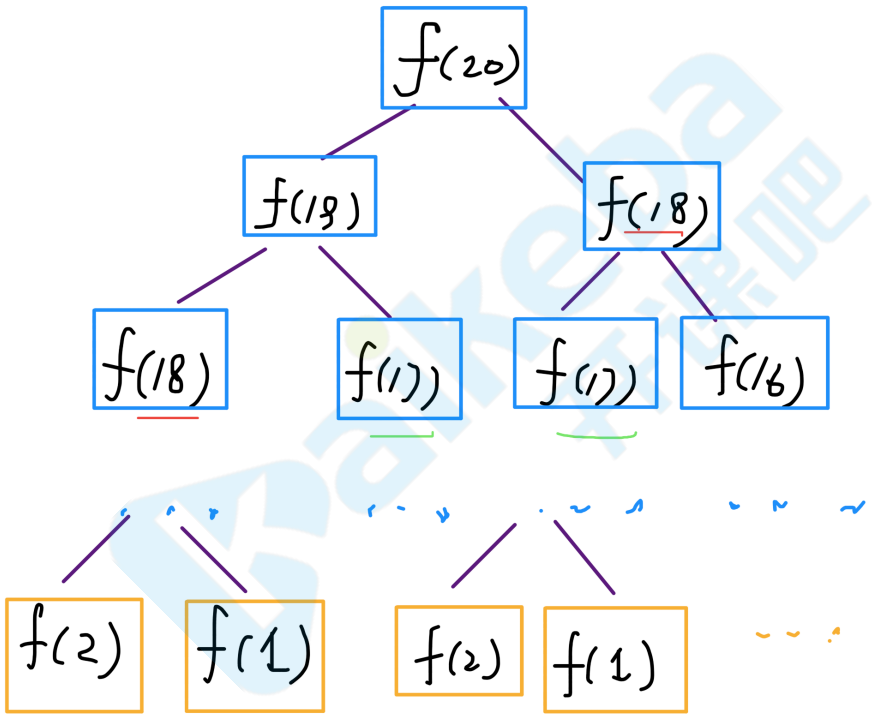
动态规划是一种常见的「算法设计技巧」，并没有什么高深莫测，至于各种高大上的术语，那是吓唬别人用的，只要你亲自体验几把，这些名词的含义其实显而易见，再简单不过了。

至于为什么最终的解法看起来如此精妙，是因为动态规划遵循一套固定的流程：递归的暴力解法 -> 带备忘录的递归解法 -> 非递归的动态规划解法。这个过程是层层递进的解决问题的过程，你如果没有前面的铺垫，直接看最终的非递归动态规划解法，当然会觉得牛逼而不可及了。

举个小栗子，斐波那契数列

暴力递归fib

```
function fib(n){
    if(n==1 || n==2) return 1
    return  fib(n-1) + fib(n-2)
}
```



递归调用很复杂，比如fib(18) 左边和右边就重复计算了

递归算法的时间复杂度怎么计算？子问题个数乘以解决一个子问题需要的时间。

子问题个数，即递归树中节点的总数。显然二叉树节点总数为指数级别，所以子问题个数为 O(2^n)。

解决一个子问题的时间，在本算法中，没有循环，只有 f(n - 1) + f(n - 2) 一个加法操作，时间为 O(1)。

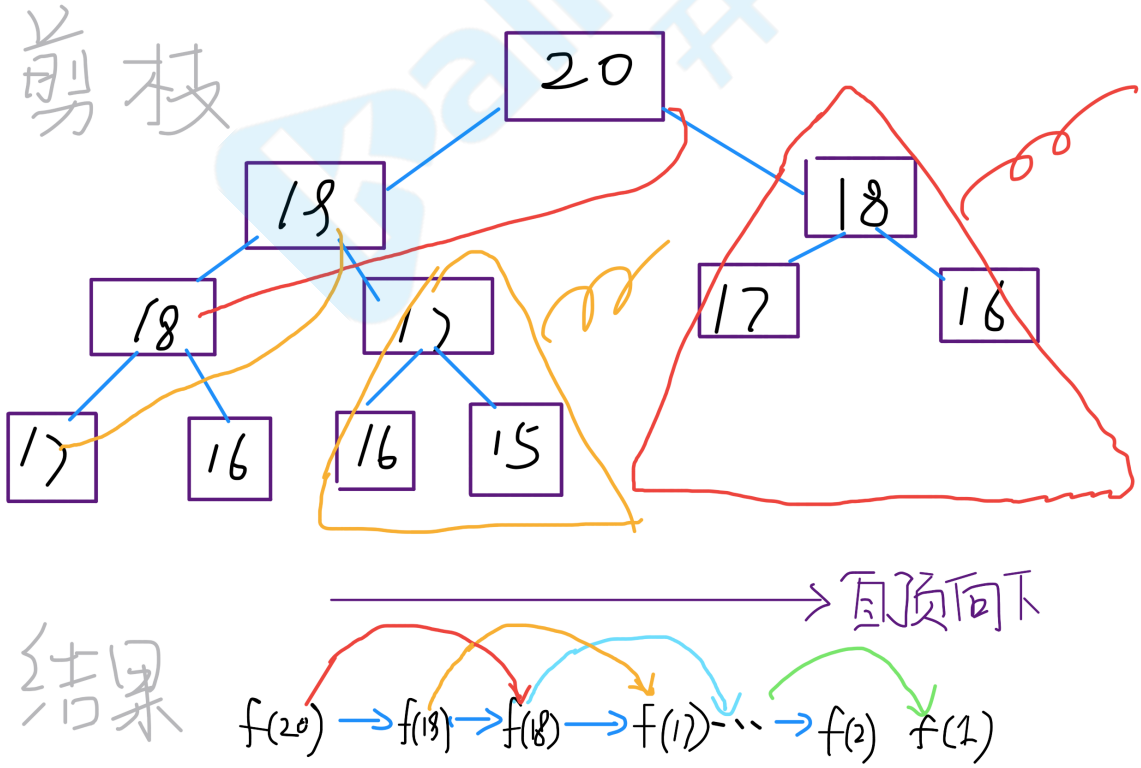
所以，这个算法的时间复杂度为 O(2^n)，指数级别，爆炸。 基本上30，40，

```
console.time('fib')
console.log(fib(40))
console.timeEnd('fib')

➜  newcode git:(master) ✗ node fib1.js
102334155
fib: 680.735ms
```

中间存储fib

明确了问题，其实就已经把问题解决了一半。即然耗时的原因是重复计算，那么我们可以造一个「备忘录」，每次算出某个子问题的答案后别急着返回，先记到「备忘录」里再返回；每次遇到一个子问题先去「备忘录」里查一查，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。

一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。




```
function fib(n){
    let memo = []
    return helper(memo, n)
}
```

```
function helper(memo, n){
    if(n==1 || n==2){
        // 前两个
        return 1
    }
    // 如果有缓存，直接返回
    if (memo[n]) return memo[n];
    // 没缓存
    memo[n] = helper(memo, n - 1) + helper(memo, n - 2)
    return memo[n]
}
```

递归算法的时间复杂度怎么算？子问题个数乘以解决一个子问题需要的时间。

子问题个数，即图中节点的总数，由于本算法不存在冗余计算，子问题就是 f(1), f(2), f(3) ... f(20)，数量和输入规模 n = 20 成正比，所以子问题个数为 O(n)。

解决一个子问题的时间，同上，没有什么循环，时间为 O(1)。

所以，本算法的时间复杂度是 O(n)。比起暴力算法，是降维打击。

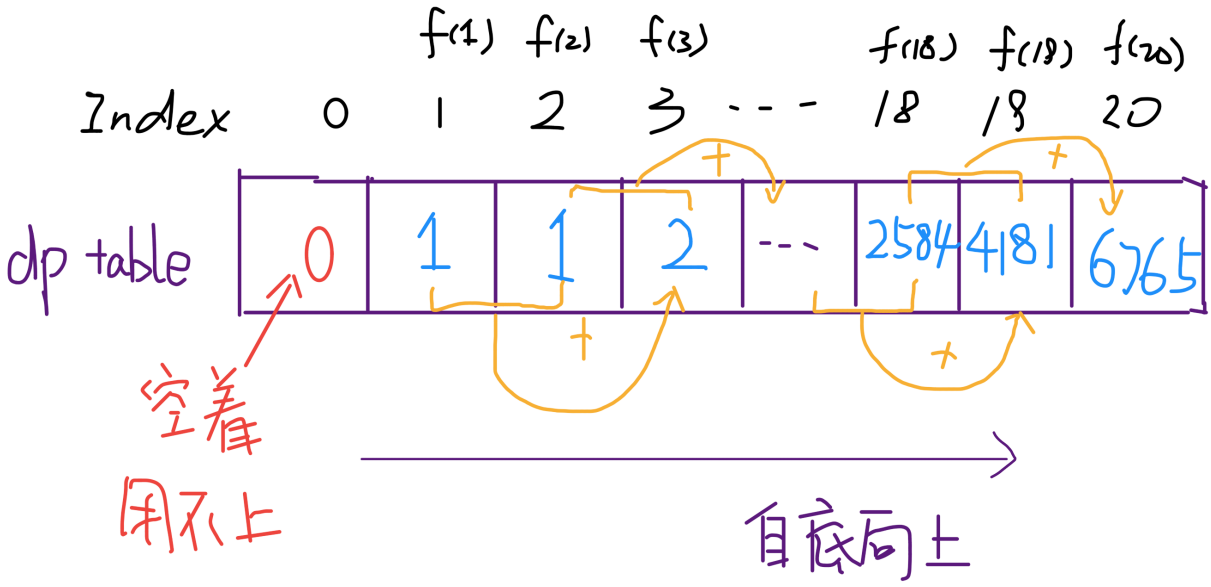
至此，带备忘录的递归解法的效率已经和动态规划一样了。实际上，这种解法和动态规划的思想已经差不多了，只不过这种方法叫做「自顶向下」，动态规划叫做「自底向上」。

啥叫「自顶向下」？注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说 f(20)，向下逐渐分解规模，直到 f(1) 和 f(2) 触底，然后逐层返回答案，这就叫「自顶向下」。

啥叫「自底向上」？反过来，我们直接从最底下，最简单，问题规模最小的 f(1) 和 f(2) 开始往上推，直到推到我们想要的答案 f(20)，这就是动态规划的思路，这也是为什么动态规划一般都脱离了递归，而是由循环迭代完成计算。

动态规划fib

我们可以把这个「备忘录」独立出来成为一张表，就叫做 DP table 吧，在这张表上完成「自底向上」的推算岂不美哉！




```
// 斐波那契
function fib(n){
    let dp = []
    dp[1] = dp[2] = 1
    for (let i = 3; i <=n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n]
}
```

动态规划找零

再举个找零的小栗子，:假如有 1, 5, 10, 20 ,50,100的人民币

```
4
[1, 1, 1, 1]   // 需 4 个 1

5
[5]            // 需 1 个 5

36
[20, 10, 5, 1] // 需 20、10、5、1各一个
```

```
// 找零
```

```
class Change{
    constructor(changeType){
        this.changeType = changeType
        this.cache = {}
    }
    makeChange (amount) {
        let min = []
        if (!amount) {
          return []
        }
        if (this.cache[amount]) {    // 读缓存
          return this.cache[amount]
        }

        for (let i = 0; i < this.changeType.length; i++) {
          const leftAmount = amount - this.changeType[i]
          let newMin
          if (leftAmount >= 0) {
            newMin = this.makeChange(leftAmount) // 这一句是动态规划的提现
          }
          if (leftAmount >= 0
            && (newMin.length < min.length - 1 || !min.length)) { // 如果存在更
小的找零硬币数, 则执行后面语句
            min = [this.changeType[i]].concat(newMin)
          }
        }
        return this.cache[amount] = min
      }
}


const change = new Change([1, 5, 10, 20,50,100])

console.log(change.makeChange(2))
console.log(change.makeChange(5))
console.log(change.makeChange(13))
console.log(change.makeChange(35))
console.log(change.makeChange(135))
```

贪心算法

贪心算法是一种求近似解的思想。当能满足大部分最优解时就认为符合逻辑要求。

还用找零 这个案例为例, 考虑使用贪心算法解题: 比如当找零数为 36 时, 从硬币数的最大值 20 开始填充, 填充不下后再用 10 来填充, 以此类推, 找到最优解。

```
场景：假如有 1, 5, 10, 20,50,100 的人民币

36              // 找零数
[20, 10, 5, 1] // 需 20、10、5、1
```

```
// 贪心
class Change {
    constructor(changeType){
        this.changeType = changeType.sort((r1, r2) => r2 - r1)

    }
    makeChange(amount) {
        const arr = []
        for (let i = 0; i < this.changeType.length; i++) {
          while (amount - this.changeType[i] >= 0) {
            arr.push(this.changeType[i])
            amount = amount - this.changeType[i]
          }
        }
        return arr
      }
}

const change = new Change([1, 5, 10, 20,50,100])

console.log(change.makeChange(36))
console.log(change.makeChange(136))

console.log('-'.repeat(100))
const change1 = new Change([1, 3, 4])

console.log(change1.makeChange(6)) // 其实33最好
```

贪心算法相对简单，就是先怼最大的，大部分情况都OK，但是有些情况不是最优解，所以人不要太贪心哦

```
console.log('-'.repeat(100))
const change1 = new Change([1, 3, 4])

console.log(change1.makeChange(6)) // 其实33最好
```

前端的数据结构

virtual-dom

fiber

hooks

推荐书目

啊哈算法

https://book.douban.com/subject/26979890/

强推算法第四版

https://book.douban.com/subject/10432347/

5.扩展

6总结

7作业

8 预告