前端算法和数据结构

知识点

复杂度

O

空间复杂度，时间复杂度

数组

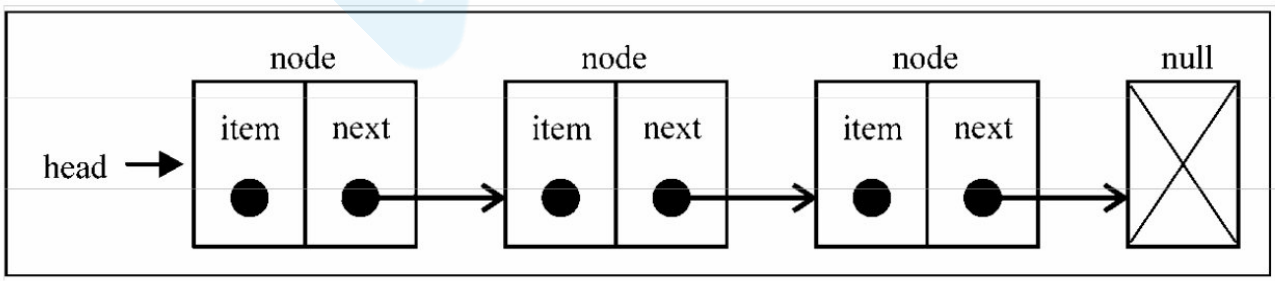
搜索复杂度

删除复杂度

新增复杂度

日常用的最多

链表



搜索复杂度

删除复杂度

新增复杂度

```
// 链表节点
class Node {
    constructor(element) {
```

```
        this.element = element
        this.next = null
    }
}

// 链表
class LinkedList {

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

    // 追加元素
    append(element) {
        const node = new Node(element)
        let current = null
        if (this.head === null) {
            this.head = node
        } else {
            current = this.head
            while(current.next) {
                current = current.next
            }
            current.next = node
        }
        this.length++
    }

    // 任意位置插入元素
    insert(position, element) {
        if (position >= 0 && position <= this.length) {
            const node = new Node(element)
            let current = this.head
            let previous = null
            let index = 0
            if (position === 0) {
                this.head = node
            } else {
                while (index++ < position) {
                    previous = current
                    current = current.next
                }
                node.next = current
                previous.next = node
            }
            this.length++
            return true
        }
        return false
    }

    // 移除指定位置元素
```

```
    removeAt(position) {

        // 检查越界值
        if (position > -1 && position < length) {
            let current = this.head
            let previous = null
            let index = 0
            if (position === 0) {
                this.head = current.next
            } else {
                while (index++ < position) {
                    previous = current
                    current = current.next
                }
                previous.next = current.next
            }
            this.length--
            return current.element
        }
        return null
    }

    // 寻找元素下标
    findIndex(element) {
        let current = this.head
        let index = -1
        while (current) {
            if (element === current.element) {
                return index + 1
            }
            index++
            current = current.next
        }
        return -1
    }

    // 删除指定文档
    remove(element) {
        const index = this.indexOf(element)
        return this.removeAt(index)
    }

    isEmpty() {
        return !this.length
    }

    size() {
        return this.length
    }

    // 转为字符串
    toString() {
        let current = this.head
```

```
        let string = ''
        while (current) {
            string += ` ${current.element}`
            current = current.next
        }
        return string
    }
}
```

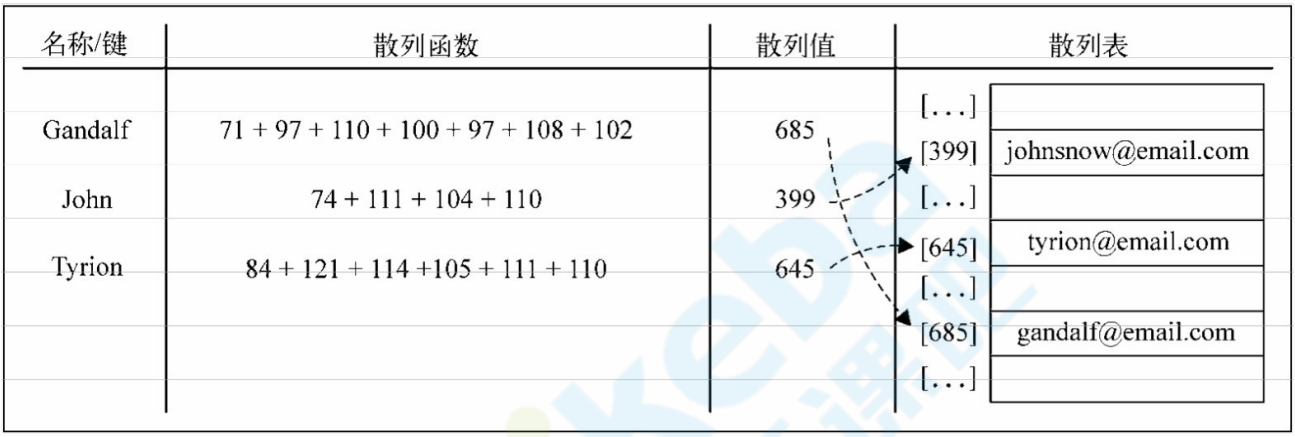
集合

```
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) // []
```

hash表

js的对象，就是hashTable的一种实现




```
class HashTable {

    constructor() {
        this.table = []
    }

    // 散列函数
    static loseloseHashCode(key) {
        let hash = 0
        for (let codePoint of key) {
            hash += codePoint.charCodeAt()
        }
        return hash % 37
    }

    // 修改和增加元素
    put(key, value) {
        const position = HashTable.loseloseHashCode(key)
        console.log(`${position} - ${key}`)
        this.table[position] = value
    }

    get(key) {
        return this.table[HashTable.loseloseHashCode(key)]
    }
```

```
    remove(key) {
        this.table[HashTable.loseloseHashCode(key)] = undefined
    }
}


const hash = new HashTable()
hash.put('Surmon', 'surmon.me@email.com') // 19 - Surmon
hash.put('John', 'johnsnow@email.com') // 29 - John
hash.put('Tyrion', 'tyrion@email.com') // 16 - Tyrion

// 测试get方法
console.log(hash.get('Surmon')) // surmon.me@email.com
console.log(hash.get('Loiane')) // undefined
console.log(hash)
```

hash碰撞

存储复杂度

栈

栈是一种遵从先进后出 (LIFO) 原则的有序集合

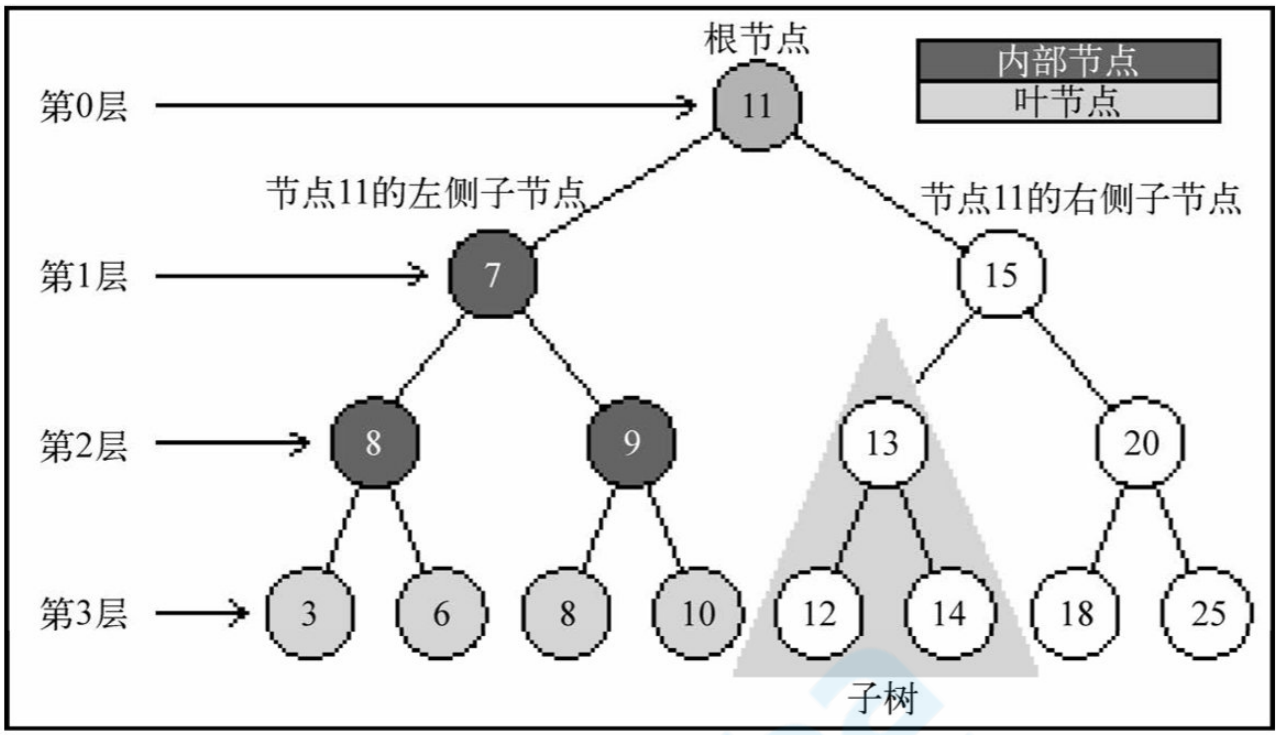
方法调用，作用域

队列

栈是一种遵从先进先出 (LIFO) 原则的有序集合

###

树



树的遍历 比如虚拟dom树

图

闭合的树

排序

冒泡

依次遍历，交换位置

```
function bubble_sort(arr){
    for(let i=0;i<arr.length-1;i++){
      for(let j=0;j<arr.length-i-1;j++){
        if(arr[j]>arr[j+1]){
          let swap=arr[j];
          arr[j]=arr[j+1];
          arr[j+1]=swap;
        }
      }
    }
  }

  let arr=[3,1,5,7,2,4,9,6,10,8];
```

```
bubble_sort(arr);
console.log(arr);
```

快速排序

二分，递归

```
function quick_sort(arr) {
    if (arr.length <= 1) {
        return arr;
    }


    let pivot = arr[0]

    let left = [];
    let right = [];
    for (let i = 1; i < arr.length; i++) {
        if (arr[i] < pivot) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }

    return quick_sort(left).concat([pivot], quick_sort(right));
}

var arr = [5,4,6,7,1,2,8,9,3];
console.log(quick_sort(arr));
```

原地快排序

不占用额外存储空间 原地交换位置

```
function quick_sort1(arr) {
    if (arr.length <= 1) {
        return arr;
    }
    let pivot = arr[0]

    let i = 1
    let j = arr.length-1
    while(i<j){
        let pivot = arr[0]
        while(arr[j]>=pivot && i<j){
            j --
```

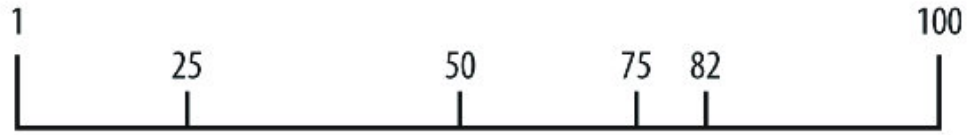
```
        }
        while(arr[i]<=pivot  && i<j){
            i ++
        }
        let temp = arr[i]
        arr[i] = arr[j]
        arr[j] = temp


    }
    let left = arr.slice(1,i+1)
    let right = arr.slice(j+1)
    return [...quick_sort1(left), pivot, ...quick_sort1(right)]

}

console.log(quick_sort1(arr));
```

。。。 选择，希尔，归并，堆，桶，基数等等

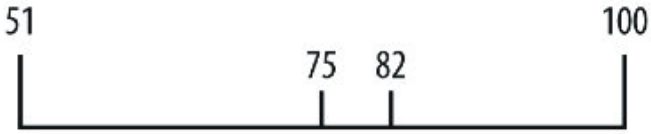
排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	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

.二分搜索

猜数字游戏，目标数字82

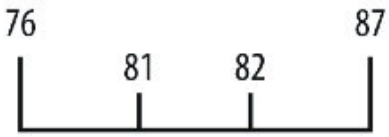


第一次猜测：50，回应：太小

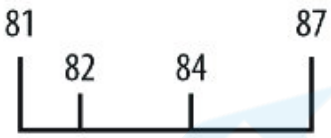


第二次猜测：76，回应：太小

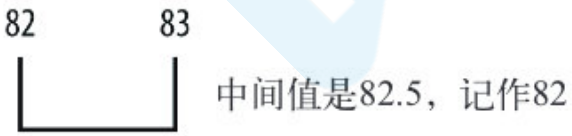
第三次猜测：88，回应：太大



第四次猜测：81，回应：太小



第五次猜测：84，回应：太大



第六次猜测：82，回应：正确

1.将数组的第一个位置设置为下边界（0） 2.将数组最后一个元素所在的位置设置为上边界（数组的长度减1）。 3.若下边界等于或小于上边界，则做如下操作。

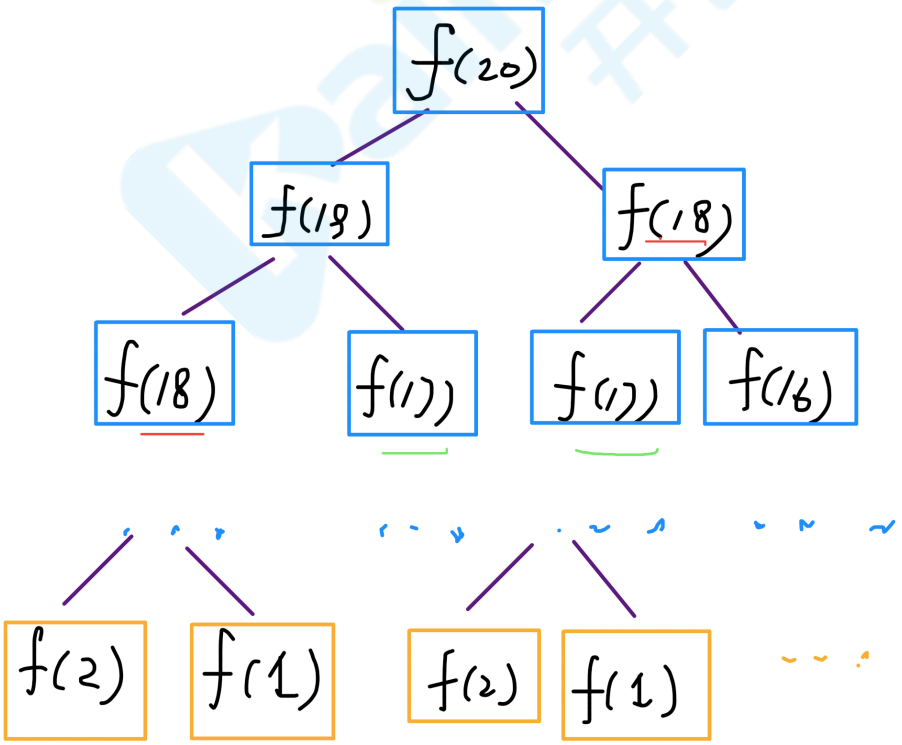
- 将中点设置为（上边界加上下边界）除以2
- 如果中点的元素小于查询的值，则将下边界设置为中点元素所在下标加1
- 如果中点的元素大于查询的值，则将上边界设置为中点元素所在下标减1
- 否则中点元素即为要查找的数据，可以进行返回。

```
function binarySearch(arr, item) {
    count +=1
    let low = 0
    let mid = null
```

```
    let element = null
    let high = arr.length - 1
    while (low <= high){
        mid = Math.floor((low + high) / 2)
        element = arr[mid]
        if (element < item) {
            low = mid + 1
        } else if (element > item) {
            high = mid - 1
        } else {
            return mid
        }
    }
    return -1
}
```

动态规划

暴力递归



这个递归树怎么理解？就是说想要计算原问题 f(20)，我就得先计算出子问题 f(19) 和 f(18)，然后要计算 f(19)，我就要先算出子问题 f(18) 和 f(17)，以此类推。最后遇到 f(1) 或者 f(2) 的时候，结果已知，就能直接返回结果，递归树不再向下生长了。

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，

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

```
➜  dongtai git:(master) X node 01.js
12586269025
fib: 89790.411ms
```

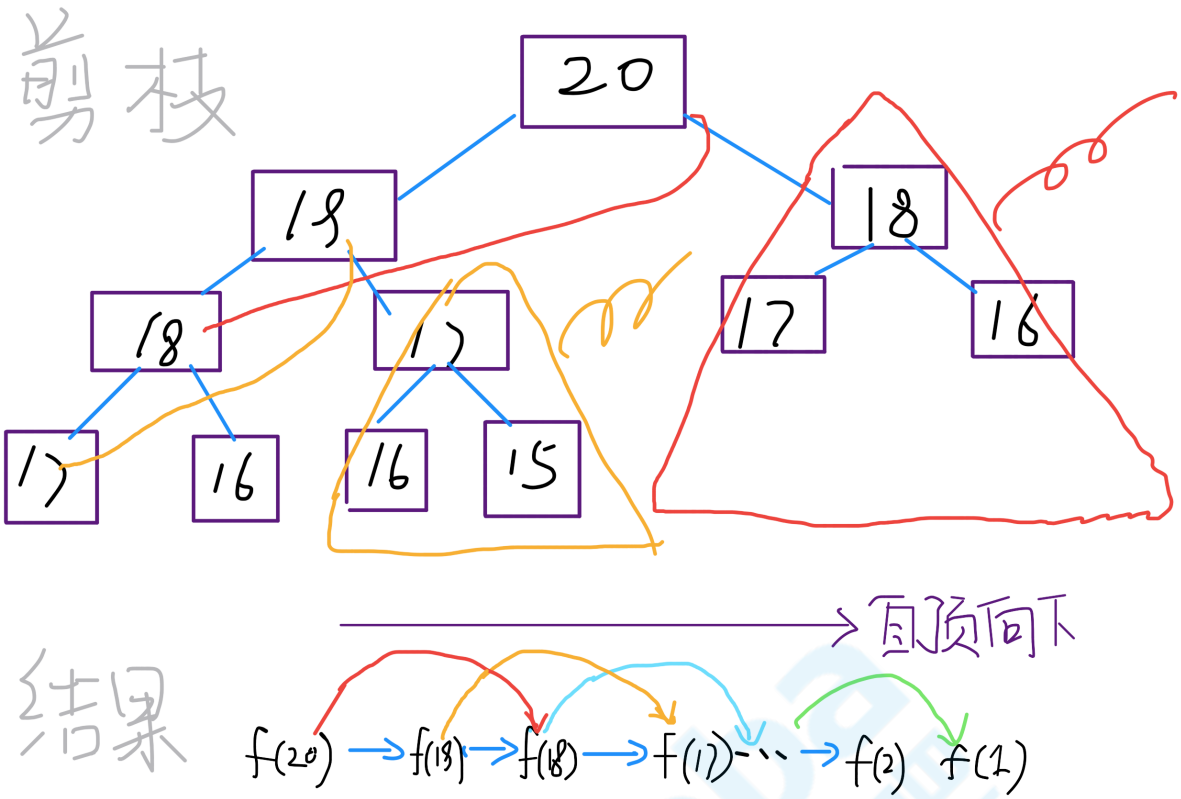
观察递归树，很明显发现了算法低效的原因：存在大量重复计算，比如 f(18) 被计算了两次，而且你可以看到，以 f(18) 为根的这个递归树体量巨大，多算一遍，会耗费巨大的时间。更何况，还不止 f(18) 这一个节点被重复计算，所以这个算法及其低效。

这就是动态规划问题的第一个性质：重叠子问题。下面，我们想办法解决这个问题。

带临时存储的递归解法

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。

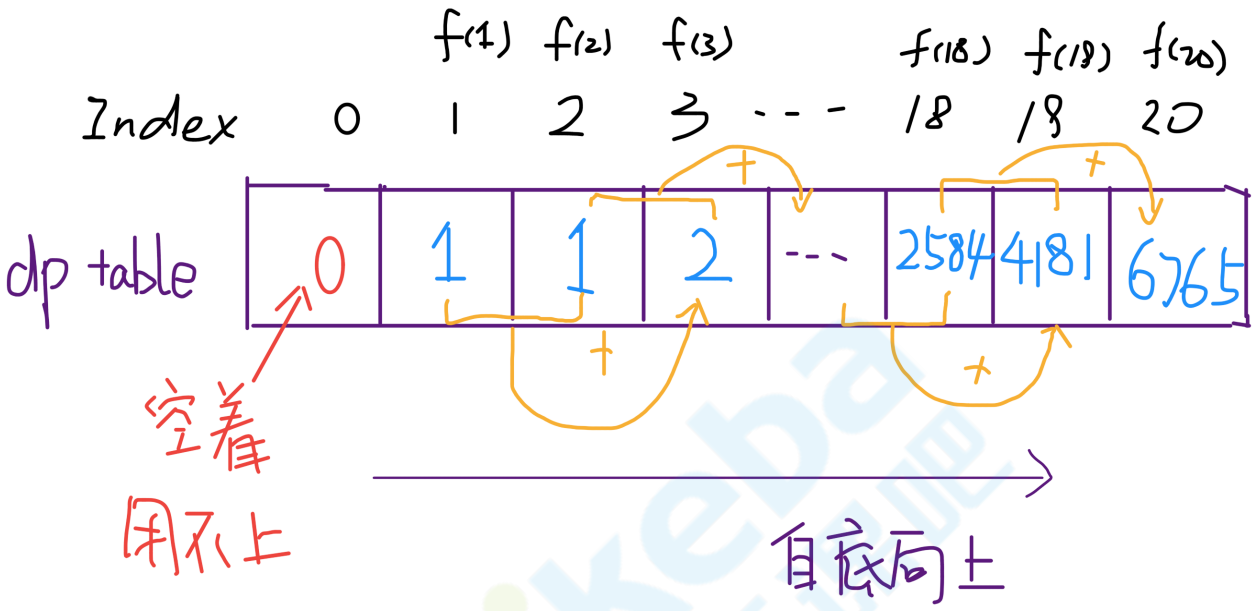
至此，带备忘录的递归解法的效率已经和动态规划一样了。实际上，这种解法和动态规划的思想已经差不多了，只不过这种方法叫做「自顶向下」，动态规划叫做「自底向上」。

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

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]
}
```

动态规划常见问题--背包问题

给定一个固定大小的背包，背包的容量为 capacity，有一组物品，存在对应的价值和重量，要求找出一个最佳的解决方案，使得装入背包的物品总重量不超过背包容量 capacity，而且总价值最大。本题中给出了3个物品，其价值和重量分别是 (3,2),(4,3),(5,4)。括号左边为价值，右边为重量，背包容量 capacity 为5。那么求出其搭配组合，使得背包内总价最大，且最大价值为多少?

Tips:

1. 不一定恰好装满

2. 装满价值不一定最大
3. 每样物品只有一个
4. 最终问题的最终解，由前面几个最优解组合决定

	val	w	j=0	j=1	j=2	j=3	j=4	j=5
i=0	3	2						
i=1	4	3						
i=2	5	4						

> 我们对这个表格做一下说明，左上角 `val` 和 `w` 分别是物品的价值和重量。即上面所描述的3个物品的价值与重量对应关系。

从第三列到最后一列，使用了变量 j，它表示背包总容量，最大值为5，也就是前面问题所说的 capacity 的值。

第二行到最后一行，使用 i 表示，下标从0开始，一共有3个物品，所以 i 的最大值为 2。即我们使用i表示物品，在下面介绍中将 `i=0` 称为 `物品0` ， `i=1` 称为 `物品1` ，以此类推。

除了 j = 0 的情况以外，我们将从左到右，从上到下一步一步去填写这个表格，来找到最大的价值。

表格中未填写的空格，表示背包内物品总价值。我们后面将使用 `T[i][j]` 二维数组来表示它。

容量为0

这个情况很简单，啥都装不进去，填表

	val	w	j=0	j=1	j=2	j=3	j=4	j=5
i=0	3	2	0					
i=1	4	3	0					
i=2	5	4	0					

只能带走一个物品 i=0

> 分析第i行时，它的物品组合仅能是小于等于i的情况。

怎么理解这个原则：比如分析 *i=0* 这一行,那么背包里只能装入 *物品0* ，不能装入其他物品。分析 *i=1* 这一行，物品组合可以是 *物品0* 和 *物品1*

`i=0 j=1` : 背包总容量为1，但是物品0 的重量为 2，无法装下去，所以这一格应该填 *0* 。

`i=0 j=2` : 背包总容量为2，刚好可以装下物品0 ，由于物品0 的价值为3，因此这一格填 **3** 。

`i=0 j=3` : 背包总容量为3，由于根据上面说明的物品组合原则，第0行，仅能放物品0，不需要考虑物品1 和 物品2，所以这一格填 **3** 。

`i=0 j=4` : 同理，填 **3** 。

`i=0 j=5` : 同理，填 **3** 。

	val	w	j=0	j=1	j=2	j=3	j=4	j=5
i=0	3	2	0	0	3	3	3	3
i=1	4	3	0					
i=2	5	4	0					

其实也很好理解，重量一次底层，只能带一件物品的话

i=0 可以装物品0和1

在这一行，可以由物品0 和物品1 进行自由组合，来装入背包。 `i=1 j=1` : 背包总容量为1，但是物品0 的重量为2，物品1重量为3，背包无法装下任何物品，所以填 **0** 。

`i=1 j=2` : 背包总容量为2，只能装下物品0，所以填 **3** 。

`i=1 j=3` : 背包总容量为3，这时候可以装下一个物品1，或者一个物品0，仅仅从人工填表的方式，很容易理解要选择物品1，但是我们该如何以一个确切的逻辑来表达，让计算机明白呢？基于上面说说明的价值和重量在表格中从上到下递增原则，可以确认物品1的价值是大于物品0的，所以默认情况下优先考虑物品1，当选择了物品1之后，把背包剩余的容量和物品1之前的物品重量对比（也就是和物品0的重量对比，如果剩余重量能装下前面的物品，那么就继续装）。所以这里选择物品1，填 **4**

`i=1 j=4` : 选择了物品1之后，物品1 的重量为3，背包容量为4， 减去物品1的重量后， 剩余容量为1，无法装下物品0，所以这里填 **4**

`i=1 j=5` 选择了物品1之后，剩余的容量为2，刚好可以装下物品0，所以一格背包装了物品1，和物品 0，总价值为7，把 **7** 填入表格。

	val	w	j=0	j=1	j=2	j=3	j=4	j=5
i=0	3	2	0	0	3	3	3	3
i=1	4	3	0	0	3	4	4	7
i=2	5	4	0					

i=2

`i=2 j=1` : 填 **0** 。

`i=2 j=2` : 填写这一行时，3种物品都有机会被装入背包。总容量为2时，只能装物品0，所以填 **3** 。

`i=2 j=3` : 物品2的重量为4，大于容量j，所以这里可以参考 T[i-1][j]的值，也就是 `i=1 j=3` 那一格的值，填 **4** 。

`i=2 j=4`：可以装下物品2，价值为5。也可以装下物品1。这一空格需要谨慎一点。我们将使用更严谨的方式来分析。在 `i=1 j=5` 中出现了物品组合一起装入背包的情况，这一空将延续这种分析方式。我们选择了物品2，剩余的容量表达式应为 **j-w[i]** 即 **4 - 4 = 0**，剩余的容量用于上一行的搜索，由于上一行我们是填写完的，所以可以很轻易地得到这个值。表达式可以写成 `val[i] + T[i][j-w[i]]`，可以根据这个表达式得出一个值。但是这并不是最终结果，还需要和上一行同一列数值对比，即 `T[i-1][j]`，对比，取最大值。最后这里填 **5**。

`i=2 j=5`：根据上面计算原理，这里如果选择了物品2，那么最大价值只能5，参照上一行，同一列，价值为7，取最大值。所以放弃物品2，选择将物品0和物品1装入背包，填写7。

	val	w	j=0	j=1	j=2	j=3	j=4	j=5
i=0	3	2	0	0	3	3	3	3
i=1	4	3	0	0	3	4	4	7
i=2	5	4	0	0	3	4	5	7

```
if(j < w[i]){ //容量小于重量 不行
    T[i][j] = T[i-1][j]; //所以值等于上一行，同一列。如果i=0,没有上一行，则T[i][j] 取0
}else{
    T[i][j] = max(val[i] + T[i-1][j-w[i]] , T[i-1][j]);  //参照上面 i=2 j=4 和 i=2 j=5 时的填表分析
}
```

```
function pack(w,val,capacity,n){
    let T = []
    for(let i = 0;i < n;i++){
        T[i] = [];
        for(let j=0;j <= capacity;j++){
            if(j === 0){ //容量为0
                T[i][j] = 0;
                continue;
            }
            if(j < w[i]){ //容量小于物品重量，本行hold不住
                if(i === 0){
                    T[i][j] = 0; // i = 0时，不存在i-1，所以T[i][j]取0

                }else{
                    T[i][j] = T[i-1][j]; //容量小于物品重量，参照上一行
                }
                continue;
            }
            if(i === 0){
                T[i][j] = val[i]; //第0行，不存在 i-1，最多只能放这一行的那一个物品
            }else{
                T[i][j] = Math.max(val[i] + T[i-1][j-w[i]],T[i-1][j]);
            }
        }
```

```
    }
    console.log(123,T)
    findValue(w,val,capacity,n,T);
    return T;
}

//找到需要的物品
function findValue(w,val,capacity,n,T){
    var i = n-1, j = capacity;
    while ( i > 0 && j > 0 ){
        if(T[i][j] != T[i-1][j]){
            console.log('选择物品'+i+',重量: '+ w[i] +',价值: ' + values[i]);
            j = j- w[i];
            i--;
        }else{
            i--;  //如果相等，那么就到 i-1 行
        }
    }
    if(i == 0 ){
        if(T[i][j] != 0){ //那么第一行的物品也可以取
            console.log('选择物品'+i+',重量: '+ w[i] +',价值: ' + values[i]);
        }
    }
}

var values = [3,4,5],
    weights = [2,3,4],
    capacity = 5,
    n = values.length;

console.log(pack(weights,values,capacity,n));
```