

前端算法

JavaScript数组去重的12种方法

数组去重，一般都是在面试的时候才会碰到，一般是要求手写数组去重方法的代码。如果是被提问到，数组去重的方法有哪些？你能答出其中的10种，面试官很有可能对你刮目相看。

在真实的项目中碰到的数组去重，一般都是后台去处理，很少让前端处理数组去重。虽然日常项目用到的概率比较低，但还是需要了解一下，以防面试的时候可能被问到。

注：写的匆忙，加上这几天有点忙，还没有非常认真核对过，不过思路是没有问题，可能一些小细节出错而已。

数组去重的方法

一、利用ES6 Set去重（ES6中最常用）

```
1 function unique (arr) {  
2     return Array.from(new Set(arr))  
3 }  
4 var arr = [1,1,'true','true',true,true,15,15,false,false,undefined,undefined,  
5 null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{},{},{}];  
6 console.log(unique(arr))  
6 //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]
```

不考虑兼容性，这种去重的方法代码最少。这种方法还无法去掉“{}”空对象，后面的高阶方法会添加去掉重复“{}”的方法。

二、利用for嵌套for，然后splice去重（ES5中最常用）

```
1 function unique(arr){
2     for(var i=0; i<arr.length; i++){
3         for(var j=i+1; j<arr.length; j++){
4             if(arr[i]==arr[j]){ //第一个等同于第二个，splice方法删除第二个
5                 arr.splice(j,1);
6                 j--;
7             }
8         }
9     }
10    return arr;
11 }
12 var arr = [1,1,'true','true',true,true,15,15,false,false,undefined,undefined,
13            null,null,NaN,NaN,'NaN',0,0,'a','a',{},{},{}];
14 console.log(unique(arr))
15 // [1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {...}, {...}] //NaN和{}没有去重，两个null直接消失了
```

双层循环，外层循环元素，内层循环时比较值。值相同时，则删去这个值。

想快速学习更多常用的ES6语法，可以看我之前的文章《学习ES6笔记——工作中常用到的ES6语法》。

三、利用indexOf去重

```
1 function unique(arr) {
2     if (!Array.isArray(arr)) {
3         console.log('type error!')
4         return
5     }
6     var array = [];
7     for (var i = 0; i < arr.length; i++) {
8         if (array.indexOf(arr[i]) === -1) {
9             array.push(arr[i])
10        }
11    }
```

```

11     }
12     return array;
13 }
14 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
    null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
15 console.log(unique(arr))
16 // [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a", {...},
    {...}] //NaN、{}没有去重

```

新建一个空的结果数组，for 循环原数组，判断结果数组是否存在当前元素，如果有相同的值则跳过，不相同则push进数组。

四、利用sort()

```

1 function unique(arr) {
2     if (!Array.isArray(arr)) {
3         console.log('type error!')
4         return;
5     }
6     arr = arr.sort()
7     var arrry= [arr[0]];
8     for (var i = 1; i < arr.length; i++) {
9         if (arr[i] !== arr[i-1]) {
10             arrry.push(arr[i]);
11         }
12     }
13     return arrry;
14 }
15 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
    null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
16 console.log(unique(arr))
17 // [0, 1, 15, "NaN", NaN, NaN, {...}, {...}, "a", false, null, true, "true",
    undefined] //NaN、{}没有去重

```

利用sort()排序方法，然后根据排序后的结果进行遍历及相邻元素比对。

五、利用对象的属性不能相同的特点进行去重（这种数组去重的方法有问题，不建议用，有待改进）

```
1 function unique(arr) {
2     if (!Array.isArray(arr)) {
3         console.log('type error!')
4         return
5     }
6     var arrry= [];
7     var obj = {};
8     for (var i = 0; i < arr.length; i++) {
9         if (!obj[arr[i]]) {
10             arrry.push(arr[i])
11             obj[arr[i]] = 1
12         } else {
13             obj[arr[i]]++
14         }
15     }
16     return arrry;
17 }
18 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
19           null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
20 console.log(unique(arr))
21 //[1, "true", 15, false, undefined, null, NaN, 0, "a", {...}] //两个true直接去掉了，NaN和{}去重
```

六、利用includes

```
1 function unique(arr) {
2     if (!Array.isArray(arr)) {
3         console.log('type error!')
```

```

4         return
5     }
6     var array = [];
7     for(var i = 0; i < arr.length; i++) {
8         if( !array.includes( arr[i]) ) { //includes 检测数组是否有某个值
9             array.push(arr[i]);
10        }
11    }
12    return array
13 }
14 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
15           null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
15 console.log(unique(arr))
16 //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}, {...}]
   //{}没有去重

```

七、利用hasOwnProperty

```

1 function unique(arr) {
2     var obj = {};
3     return arr.filter(function(item, index, arr){
4         return obj.hasOwnProperty(typeof item + item) ? false : (obj[typeof
5             item + item] = true)
6     })
7 }
8 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
9           null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
10 console.log(unique(arr))
11 //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}] //所有
   的都去重了
12 利用hasOwnProperty 判断是否存在对象属性

```

八、利用filter

```
1 function unique(arr) {
2     return arr.filter(function(item, index, arr) {
3         //当前元素, 在原始数组中的第一个索引==当前索引值, 否则返回当前元素
4         return arr.indexOf(item, 0) === index;
5     });
6 }
7 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
8 null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
9 console.log(unique(arr))
10 // [1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {...}, {...}]
```

九、利用递归去重

```
1 function unique(arr) {
2     var array= arr;
3     var len = array.length;
4     >
5     array.sort(function(a,b){ //排序后更加方便去重
6         return a - b;
7     })
8     >
9     function loop(index){
10         if(index >= 1){
11             if(array[index] === array[index-1]){
12                 array.splice(index,1);
13             }
14             loop(index - 1); //递归loop, 然后数组去重
15         }
16     }
17     loop(len-1);
18     return array;
19 }
```

```

20 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
    null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
21 console.log(unique(arr))
22 //[1, "a", "true", true, 15, false, 1, {...}, null, NaN, NaN, "NaN", 0, "a",
    {...}, undefined]

```

十、利用Map数据结构去重

```

1 function arrayNonRepeatfy(arr) {
2     let map = new Map();
3     let array = new Array(); // 数组用于返回结果
4     for (let i = 0; i < arr.length; i++) {
5         if(map .has(arr[i])) { // 如果有该key值
6             map .set(arr[i], true);
7         } else {
8             map .set(arr[i], false); // 如果没有该key值
9             array .push(arr[i]);
10        }
11    }
12    return array ;
13 }
14 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
    null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
15 console.log(unique(arr))
16 //[1, "a", "true", true, 15, false, 1, {...}, null, NaN, NaN, "NaN", 0, "a",
    {...}, undefined]

```

创建一个空Map数据结构，遍历需要去重的数组，把数组的每一个元素作为key存到Map中。由于Map中不会出现相同的key值，所以最终得到的就是去重后的结果。

十一、利用reduce+includes

```

1 function unique(arr){
2     return arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cur],
3     []);
4 }
5 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
6 null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
7 console.log(unique(arr));
8 // [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}, {...}]

```

十二、[...new Set(arr)]

```

1 [...new Set(arr)]

```

//代码就是这么少----（其实，严格来说并不算是一种，相对于第一种方法来说只是简化了代码）

斐波那契数列

斐波那契数列就是形如0,1,1,2,3,5,8.....的数列。

```

1 let getFibonacci = n =>{

```



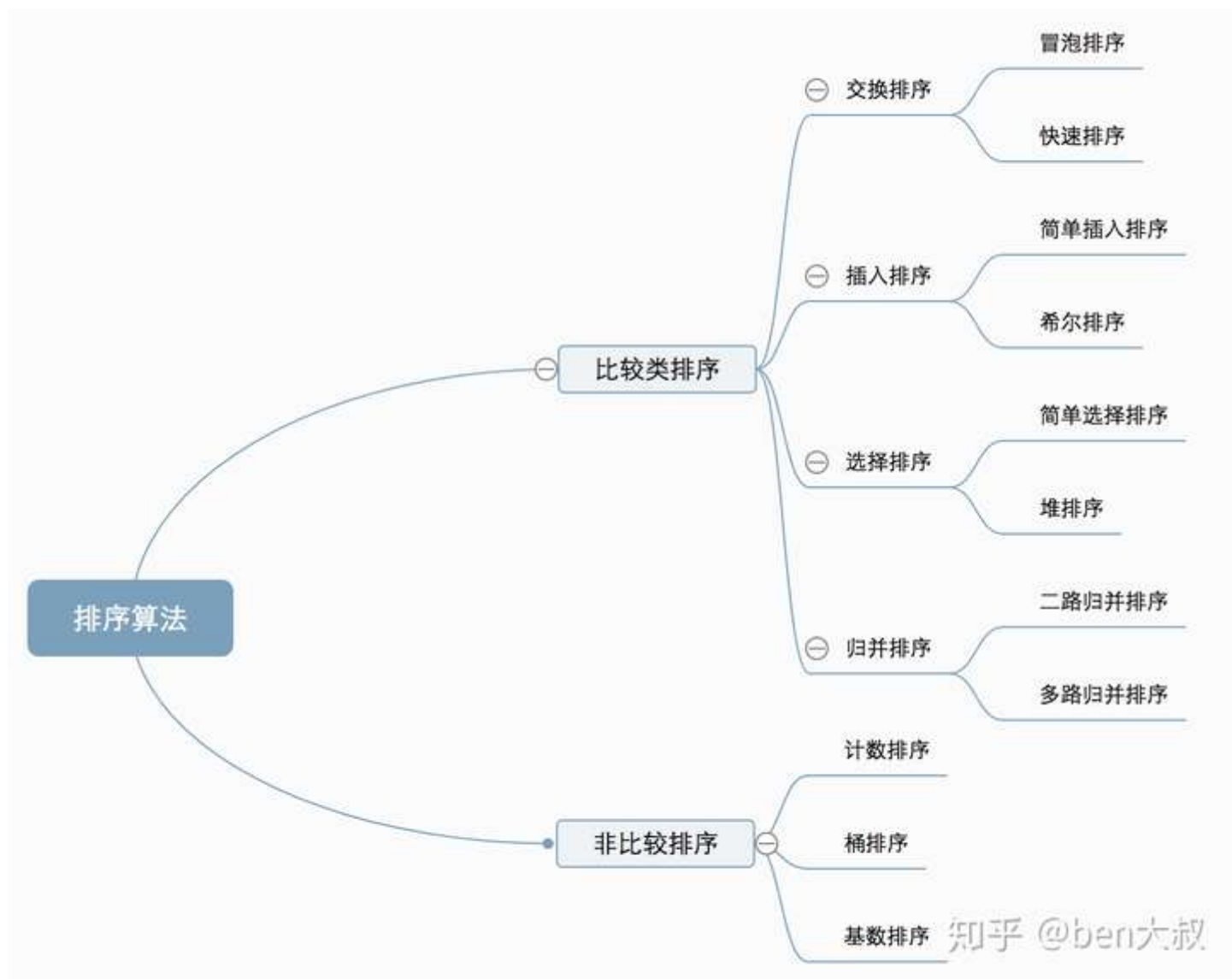
```
2     let arr = [];  
3     for(let i = 0;i<n - 1;i++){  
4         if(i < 2){  
5             arr.push(i);  
6         }else{  
7             arr.push(arr[i-1] + arr[i-2]);  
8         }  
9     }  
10    return arr;  
11 }
```

十大经典排序算法

0.1 算法分类

十种常见排序算法可以分为两大类：

- **比较类排序**：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破 $O(n\log n)$ ，因此也称为非线性时间比较类排序。
- **非比较类排序**：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此也称为线性时间非比较类排序。



0.2 算法复杂度

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n*k)$	稳定

##0.3 相关概念

- **稳定**：如果a原本在b前面，而a=b，排序之后a仍然在b的前面。
- **不稳定**：如果a原本在b的前面，而a=b，排序之后 a 可能会出现在 b 的后面。
- **时间复杂度**：对排序数据的总的操作次数。反映当n变化时，操作次数呈现什么规律。
- **空间复杂度**：是指算法在计算机

内执行时所需存储空间的度量，它也是数据规模n的函数。

1、冒泡排序（Bubble Sort）

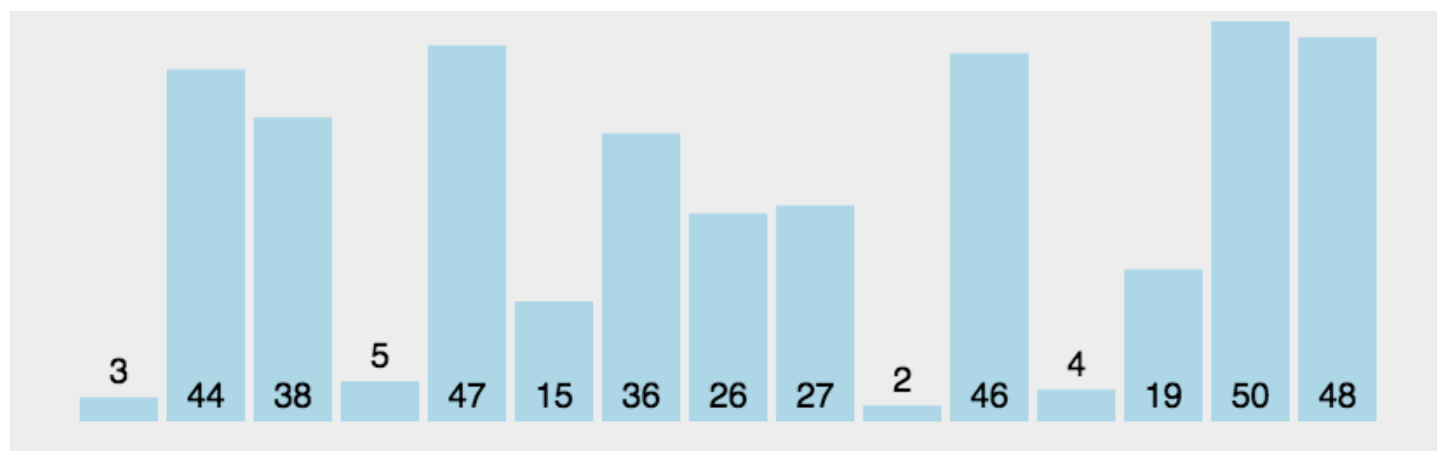
冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经

排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

1.1 算法描述

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
- 针对所有的元素重复以上的步骤，除了最后一个；
- 重复步骤1~3，直到排序完成。

1.2 动图演示



1.3 代码实现

```
1 function bubbleSort(arr) {  
2     var len = arr.length;  
3     for (var i = 0; i < len - 1; i++) {  
4         for (var j = 0; j < len - 1 - i; j++) {  
5             if (arr[j] > arr[j+1]) {           // 相邻元素两两对比  
6                 var temp = arr[j+1];         // 元素交换  
7                 arr[j+1] = arr[j];  
8                 arr[j] = temp;  
9             }  
10        }  
}
```

```
11     }  
12     return arr;  
13 }
```

2、选择排序（Selection Sort）

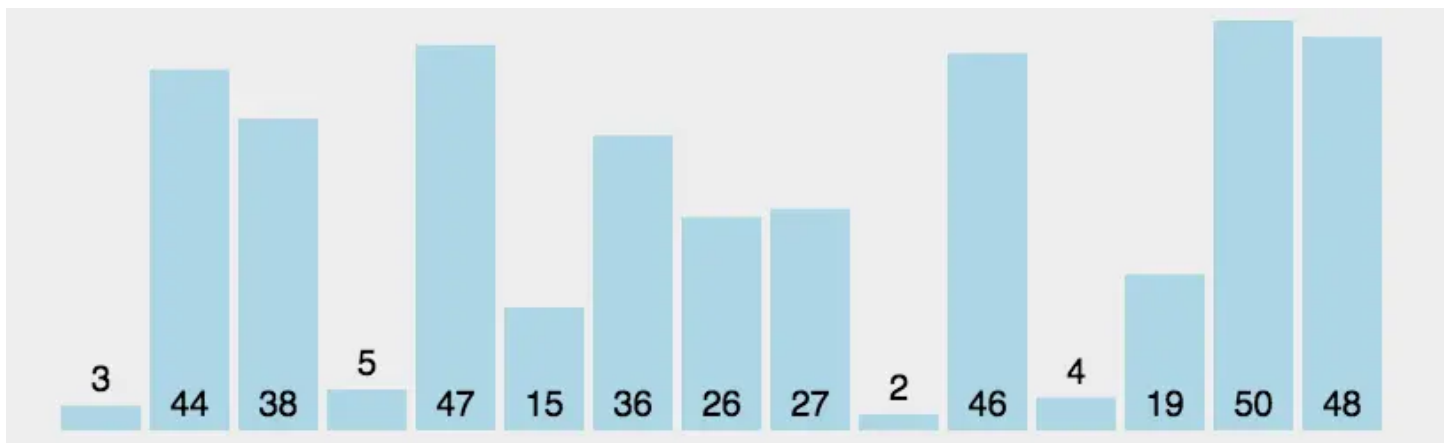
选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

2.1 算法描述

n 个记录的直接选择排序可经过 $n-1$ 趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为 $R[1..n]$ ，有序区为空；
- 第 i 趟排序($i=1,2,3\dots n-1$)开始时，当前有序区和无序区分别为 $R[1..i-1]$ 和 $R(i..n)$ 。该趟排序从当前无序区中-选出关键字最小的记录 $R[k]$ ，将它与无序区的第1个记录 R 交换，使 $R[1..i]$ 和 $R[i+1..n]$ 分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
- $n-1$ 趟结束，数组有序化了。

2.2 动图演示



2.3 代码实现

```
1 function selectionSort(arr) {
2     var len = arr.length;
3     var minIndex, temp;
4     for (var i = 0; i < len - 1; i++) {
5         minIndex = i;
6         for (var j = i + 1; j < len; j++) {
7             if (arr[j] < arr[minIndex]) { // 寻找最小的数
8                 minIndex = j;           // 将最小数的索引保存
9             }
10        }
11        temp = arr[i];
12        arr[i] = arr[minIndex];
13        arr[minIndex] = temp;
14    }
15    return arr;
16 }
```

2.4 算法分析

表现最稳定的排序算法之一，因为无论什么数据进去都是 $O(n^2)$ 的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

3、插入排序（Insertion Sort）

插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

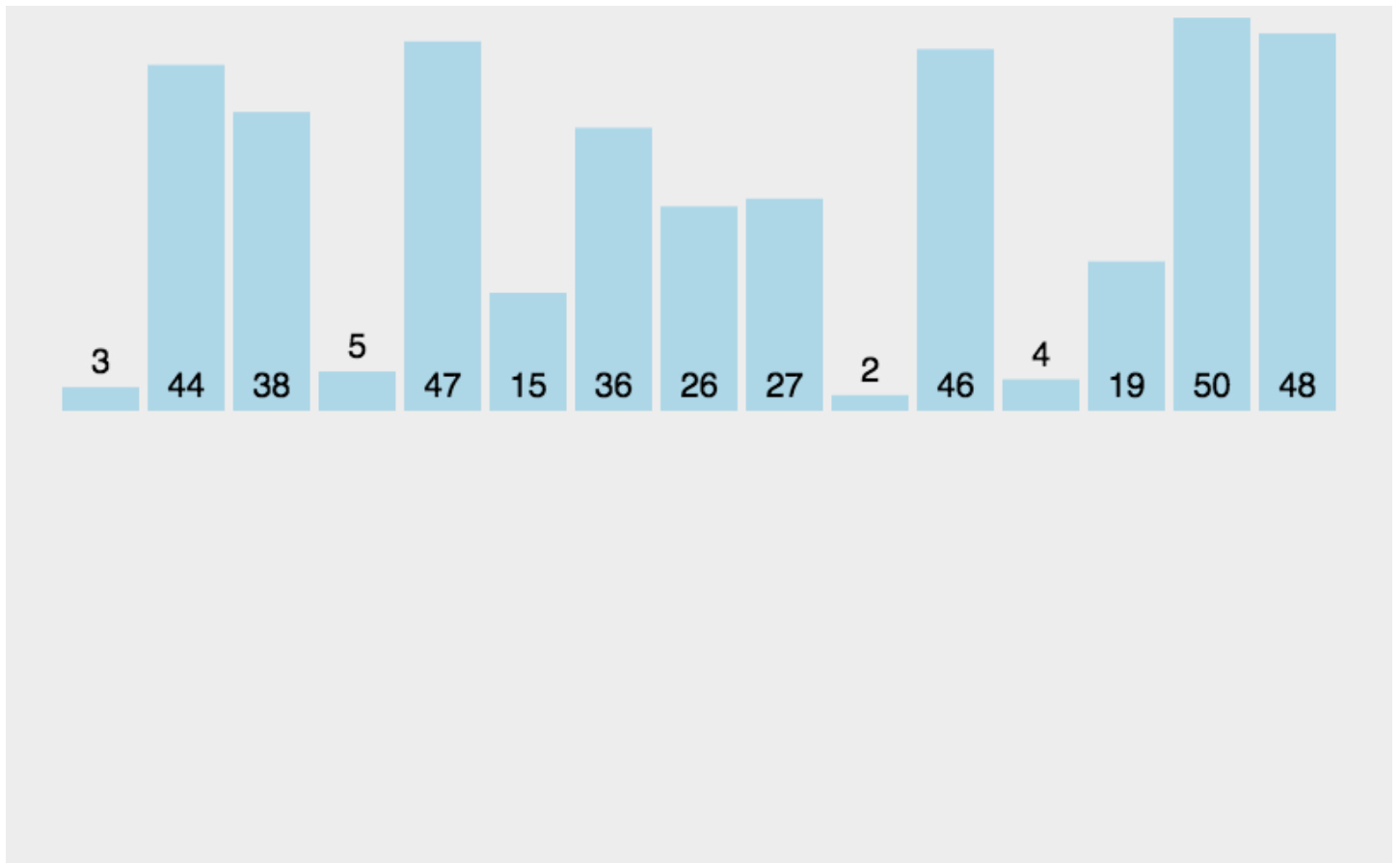
3.1 算法描述

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

- 从第一个元素开始，该元素可以认为已经被排序；

- 取出下一个元素，在已经排序的元素序列中从后向前扫描；
- 如果该元素（已排序）大于新元素，将该元素移到下一位置；
- 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
- 将新元素插入到该位置后；
- 重复步骤2~5。

3.2 动图演示



3.2 代码实现

```
1 function insertionSort(arr) {  
2     var len = arr.length;  
3     var preIndex, current;  
4     for (var i = 1; i < len; i++) {  
5         preIndex = i - 1;  
6         current = arr[i];
```

```
7         while (preIndex >= 0 && arr[preIndex] > current) {
8             arr[preIndex + 1] = arr[preIndex];
9             preIndex--;
10        }
11        arr[preIndex + 1] = current;
12    }
13    return arr;
14 }
```

3.4 算法分析

插入排序在实现上，通常采用in-place排序（即只需用到 $O(1)$ 的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

4、希尔排序（Shell Sort）

1959年Shell发明，第一个突破 $O(n^2)$ 的排序算法，是简单插入排序的改进版。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫**缩小增量排序**。

4.1 算法描述

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

- 选择一个增量序列 t_1, t_2, \dots, t_k ，其中 $t_i > t_j$ ， $t_k = 1$ ；
- 按增量序列个数 k ，对序列进行 k 趟排序；
- 每趟排序，根据对应的增量 t_i ，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1时，整个序列作为一个表来处理，表长度即为整个序列的长度。

4.3 代码实现

```
1 function shellSort(arr) {
2     var len = arr.length;
```



```
3     for (var gap = Math.floor(len / 2); gap > 0; gap = Math.floor(gap / 2)) {
4         // 注意：这里和动图演示的不一样，动图是分组执行，实际操作是多个分组交替执行
5         for (var i = gap; i < len; i++) {
6             var j = i;
7             var current = arr[i];
8             while (j - gap >= 0 && current < arr[j - gap]) {
9                 arr[j] = arr[j - gap];
10                j = j - gap;
11            }
12            arr[j] = current;
13        }
14    }
15    return arr;
16 }
```

4.4 算法分析

希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。动态定义间隔序列的算法是《算法（第4版）》的合著者Robert Sedgewick提出的。

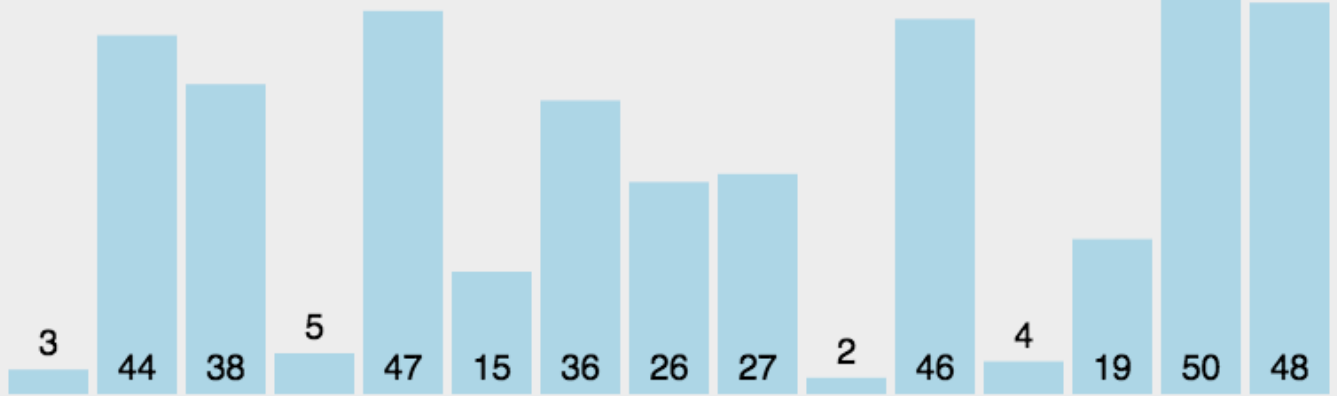
5、归并排序（Merge Sort）

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

5.1 算法描述

- 把长度为 n 的输入序列分成两个长度为 $n/2$ 的子序列；
- 对这两个子序列分别采用归并排序；
- 将两个排序好的子序列合并成一个最终的排序序列。

5.2 动图演示



5.3 代码实现

```
1 function mergeSort(arr) {  
2     var len = arr.length;  
3     if (len < 2) {  
4         return arr;  
5     }  
6     var middle = Math.floor(len / 2),  
7         left = arr.slice(0, middle),  
8         right = arr.slice(middle);  
9     return merge(mergeSort(left), mergeSort(right));  
10 }  
11
```

```
1 function merge(left, right) {  
2     var result = [];  
3  
4     while (left.length > 0 && right.length > 0) {
```

```
5         if (left[0] <= right[0]) {
6             result.push(left.shift());
7         } else {
8             result.push(right.shift());
9         }
10    }
11
12    while (left.length)
13        result.push(left.shift());
14
15    while (right.length)
16        result.push(right.shift());
17
18    return result;
19 }
```

5.4 算法分析

归并排序是一种稳定的排序方法。和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是 $O(n\log n)$ 的时间复杂度。代价是需要额外的内存空间。

6、快速排序（Quick Sort）

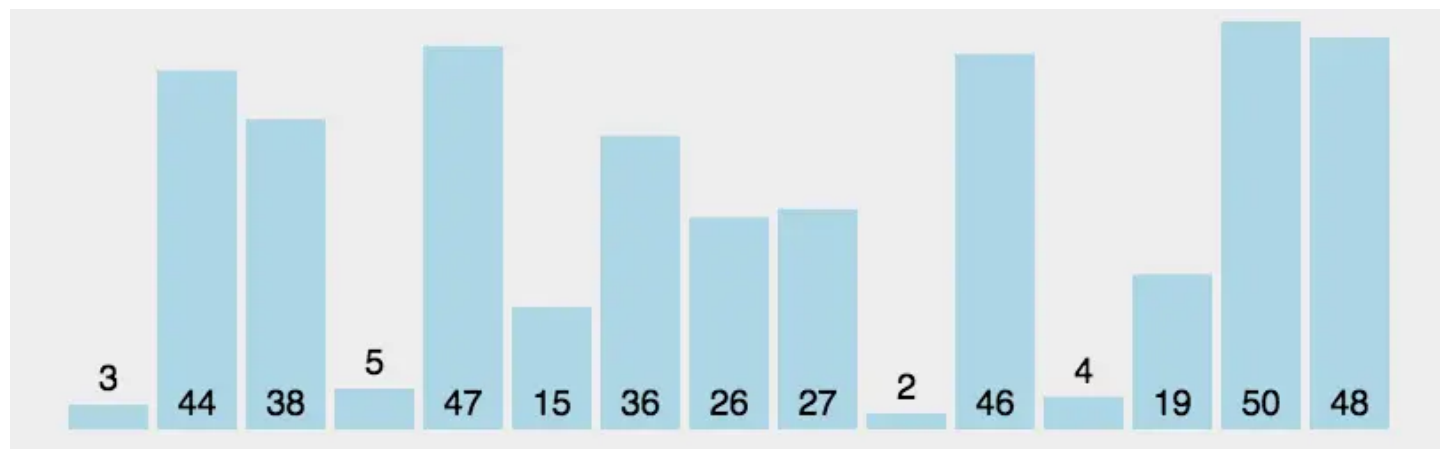
快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

6.1 算法描述

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

- 从数列中挑出一个元素，称为“基准”（pivot）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

6.2 动图演示



6.3 代码实现

```
1 function quickSort(arr, left, right) {
2     var len = arr.length,
3         partitionIndex,
4         left = typeof left !== 'number' ? 0 : left,
5         right = typeof right !== 'number' ? len - 1 : right;
6
7     if (left < right) {
8         partitionIndex = partition(arr, left, right);
9         quickSort(arr, left, partitionIndex-1);
10        quickSort(arr, partitionIndex+1, right);
11    }
12    return arr;
13 }
14
15 function partition(arr, left, right) { // 分区操作
16     var pivot = left, // 设定基准值 (pivot)
17         index = pivot + 1;
18     for (var i = index; i <= right; i++) {
19         if (arr[i] < arr[pivot]) {
20             swap(arr, i, index);
21             index++;
22         }
23     }
24     swap(arr, pivot, index - 1);
25     return index-1;
26 }
```

```
26
27 function swap(arr, i, j) {
28     var temp = arr[i];
29     arr[i] = arr[j];
30     arr[j] = temp;
31 }
```

7、堆排序 (Heap Sort)

堆排序 (Heapsort) 是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

7.1 算法描述

- 将初始待排序关键字序列(R_1, R_2, \dots, R_n)构建成大顶堆，此堆为初始的无序区；
- 将堆顶元素 $R[1]$ 与最后一个元素 $R[n]$ 交换，此时得到新的无序区(R_1, R_2, \dots, R_{n-1})和新的有序区(R_n),且满足 $R[1, 2 \dots n-1] \leq R[n]$ ；
- 由于交换后新的堆顶 $R[1]$ 可能违反堆的性质，因此需要对当前无序区(R_1, R_2, \dots, R_{n-1})调整为新堆，然后再次将 $R[1]$ 与无序区最后一个元素交换，得到新的无序区(R_1, R_2, \dots, R_{n-2})和新的有序区(R_{n-1}, R_n)。不断重复此过程直到有序区的元素个数为 $n-1$ ，则整个排序过程完成。

7.3 代码实现

```
1 var len;    // 因为声明的多个函数都需要数据长度，所以把len设置成为全局变量
2
3 function buildMaxHeap(arr) {    // 建立大顶堆
4     len = arr.length;
5     for (var i = Math.floor(len/2); i >= 0; i--) {
6         heapify(arr, i);
7     }
8 }
9
```

```

1 function heapify(arr, i) {      // 堆调整
2     var left = 2 * i + 1,
3         right = 2 * i + 2,
4         largest = i;
5
6     if (left < len && arr[left] > arr[largest]) {
7         largest = left;
8     }
9
10    if (right < len && arr[right] > arr[largest]) {
11        largest = right;
12    }
13
14    if (largest !== i) {
15        swap(arr, i, largest);
16        heapify(arr, largest);
17    }
18 }
19
20 function swap(arr, i, j) {
21     var temp = arr[i];
22     arr[i] = arr[j];
23     arr[j] = temp;
24 }
25
26 function heapSort(arr) {
27     buildMaxHeap(arr);
28
29     for (var i = arr.length - 1; i > 0; i--) {
30         swap(arr, 0, i);
31         len--;
32         heapify(arr, 0);
33     }
34     return arr;
35 }

```

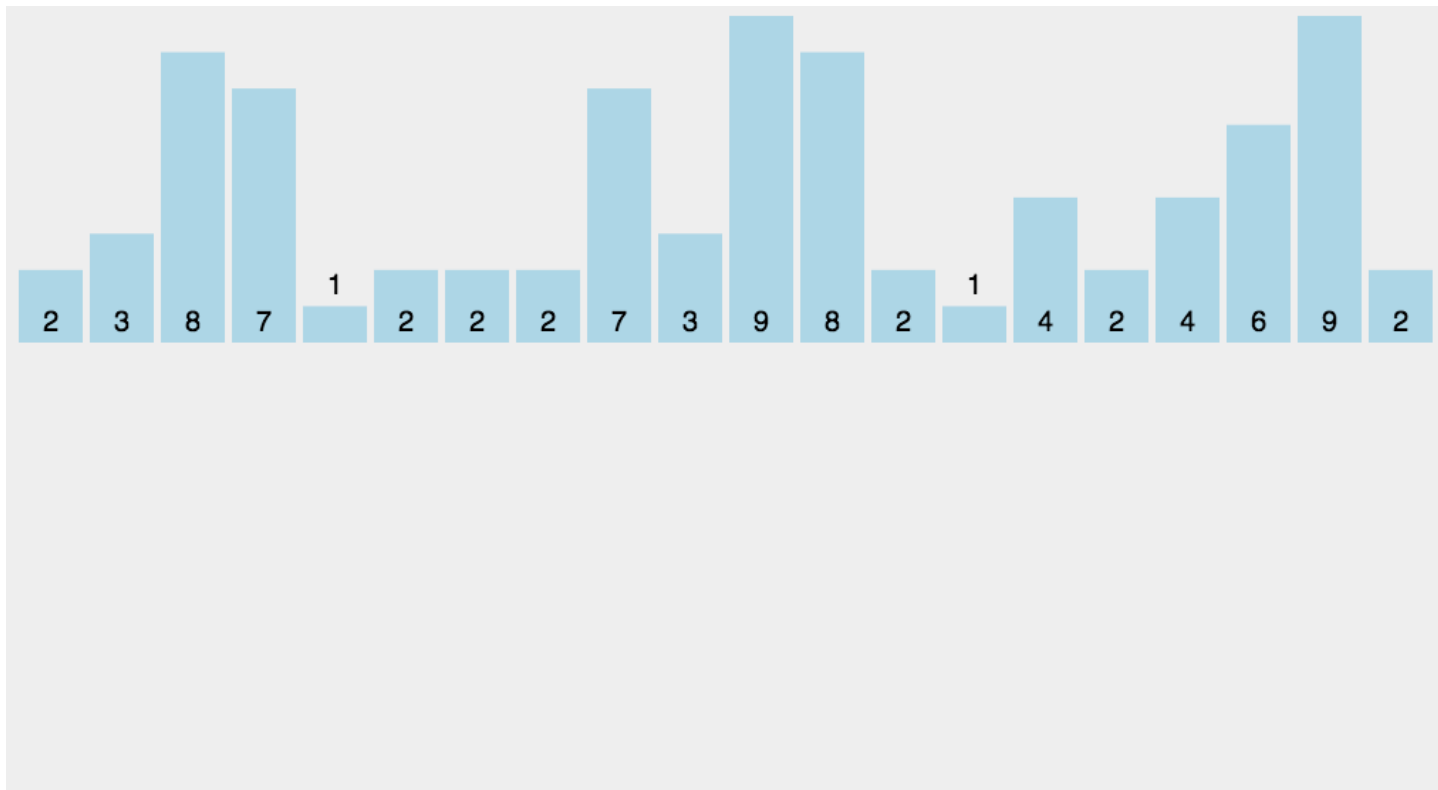
8、计数排序（Counting Sort）

计数排序不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

8.1 算法描述

- 找出待排序的数组中最大和最小的元素；
- 统计数组中每个值为i的元素出现的次数，存入数组C的第i项；
- 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；
- 反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

8.2 动图演示



8.3 代码实现

```
1 function countingSort(arr, maxValue) {  
2     var bucket = new Array(maxValue + 1),  
3     sortedIndex = 0;  
4     arrLen = arr.length,  
5     bucketLen = maxValue + 1;  
6  
7     for (var i = 0; i < arrLen; i++) {  
8         if (!bucket[arr[i]]) {  
9             bucket[arr[i]] = 0;  
10        }  
}
```

```
11         bucket[arr[i]]++;
12     }
13
14     for (var j = 0; j < bucketLen; j++) {
15         while(bucket[j] > 0) {
16             arr[sortedIndex++] = j;
17             bucket[j]--;
18         }
19     }
20
21     return arr;
22 }
```

8.4 算法分析

计数排序是一个稳定的排序算法。当输入的元素是 n 个 0 到 k 之间的整数时，时间复杂度是 $O(n+k)$ ，空间复杂度也是 $O(n+k)$ ，其排序速度快于任何比较排序算法。当 k 不是很大并且序列比较集中时，计数排序是一个很有效的排序算法。

9、桶排序 (Bucket Sort)

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。桶排序 (Bucket sort) 的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排）。

9.1 算法描述

- 设置一个定量的数组当作空桶；
- 遍历输入数据，并且把数据一个一个放到对应的桶里去；
- 对每个不是空的桶进行排序；
- 从不是空的桶里把排好序的数据拼接起来。

9.3 代码实现


```

1 function bucketSort(arr, bucketSize) {
2     if (arr.length === 0) {
3         return arr;
4     }
5
6     var i;
7     var minValue = arr[0];
8     var maxValue = arr[0];
9     for (i = 1; i < arr.length; i++) {
10         if (arr[i] < minValue) {
11             minValue = arr[i];           // 输入数据的最小值
12         } else if (arr[i] > maxValue) {
13             maxValue = arr[i];           // 输入数据的最大值
14         }
15     }
16
17     // 桶的初始化
18     var DEFAULT_BUCKET_SIZE = 5;         // 设置桶的默认数量为5
19     bucketSize = bucketSize || DEFAULT_BUCKET_SIZE;
20     var bucketCount = Math.floor((maxValue - minValue) / bucketSize) + 1;
21     var buckets = new Array(bucketCount);
22     for (i = 0; i < buckets.length; i++) {
23         buckets[i] = [];
24     }
25
26     // 利用映射函数将数据分配到各个桶中
27     for (i = 0; i < arr.length; i++) {
28         buckets[Math.floor((arr[i] - minValue) / bucketSize)].push(arr[i]);
29     }
30
31     arr.length = 0;
32     for (i = 0; i < buckets.length; i++) {
33         insertionSort(buckets[i]);       // 对每个桶进行排序，这里
                                           使用了插入排序
34         for (var j = 0; j < buckets[i].length; j++) {
35             arr.push(buckets[i][j]);
36         }
37     }
38     return arr;
39 }

```

9.4 算法分析

桶排序最好情况下使用线性时间 $O(n)$ ，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为 $O(n)$ 。很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

10、基数排序（Radix Sort）

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。

10.1 算法描述

- 取得数组中的最大数，并取得位数；
- arr为原始数组，从最低位开始取每个位组成radix数组；
- 对radix进行计数排序（利用计数排序适用于小范围数的特点）；

10.2 动图演示



10.3 代码实现

```
1 var counter = [];  
2 function radixSort(arr, maxDigit) {  
3     var mod = 10;  
4     var dev = 1;  
5     for (var i = 0; i < maxDigit; i++, dev *= 10, mod *= 10) {  
6         for(var j = 0; j < arr.length; j++) {  
7             var bucket = parseInt((arr[j] % mod) / dev);  
8             if(counter[bucket]==null) {  
9                 counter[bucket] = [];  
10            }  
11            counter[bucket].push(arr[j]);  
12        }  
13        var pos = 0;  
14        for(var j = 0; j < counter.length; j++) {  
15            var value = null;  
16            if(counter[j]!=null) {  
17                while ((value = counter[j].shift()) != null) {  
18                    arr[pos++] = value;  
19                }  
20            }  
21        }  
22    }  
23    return arr;  
24 }
```

10.4 算法分析

基数排序基于分别排序，分别收集，所以是稳定的。但基数排序的性能比桶排序要略差，每一次关键字的桶分配都需要 $O(n)$ 的时间复杂度，而且分配之后得到新的关键字序列又需要 $O(n)$ 的时间复杂度。假如待排数据可以分为 d 个关键字，则基数排序的时间复杂度将是 $O(d*2n)$ ，当然 d 要远远小于 n ，因此基本上还是线性级别的。

基数排序的空间复杂度为 $O(n+k)$ ，其中 k 为桶的数量。一般来说 $n \gg k$ ，因此额外空间需要大概 n 个左右。

实现把多维数组转为一维数组的几种方式

```
1 /**
2  * 使用转字符串法
3  */
4 let tempArr = [1,[2,3],[4,5,[6,7]]];
5 let result_str1 = tempArr.join(',').split(',');
6 console.log(result_str1); // ["1", "2", "3", "4", "5", "6", "7"]
```

```
1 //使用数组的concat方法，以及apply
2 let tempArr4 = [[1,2], '3', '4', [5,6], [7]];
3 console.log([].concat.apply([], tempArr4)); // [1, 2, "3", "4", 5, 6, 7]
```

算法的时间与空间复杂度（一看就懂）

算法（Algorithm）是指用来操作数据、解决程序问题的一组方法。对于同一个问题，使用不同的算法，也许最终得到的结果是一样的，但在过程中消耗的资源和时间却会有很大的区别。

那么我们应该如何去衡量不同算法之间的优劣呢？

主要还是从算法所占用的「时间」和「空间」两个维度去考量。

- 时间维度：是指执行当前算法所消耗的时间，我们通常用「时间复杂度」来描述。
- 空间维度：是指执行当前算法需要占用多少内存空间，我们通常用「空间复杂度」来描述。

因此，评价一个算法的效率主要是看它的时间复杂度和空间复杂度情况。然而，有的时候时间和空间却又是「鱼和熊掌」，不可兼得的，那么我们就需要从中去取一个平衡点。

下面我来分别介绍一下「时间复杂度」和「空间复杂度」的计算方式。

一、时间复杂度

我们想要知道一个算法的「时间复杂度」，很多人首先想到的的方法就是把这个算法程序运行一遍，那么它所消耗的时间就自然而然知道了。

这种方式可以吗？当然可以，不过它也有很多弊端。

这种方式非常容易受运行环境的影响，在性能高的机器上跑出来的结果与在性能低的机器上跑的结果相差会很大。而且对测试时使用的数据规模也有很大关系。再者，当我们在写算法的时候，还没有办法完整的去运行呢。

因此，另一种更为通用的方法就出来了：「**大O符号表示法**」，即 $T(n) = O(f(n))$

我们先来看个例子：

```
1 for(i=1; i<=n; ++i)
2 {
3     j = i;
4     j++;
5 }
```

通过「大O符号表示法」，这段代码的时间复杂度为： $O(n)$ ，为什么呢？

在大O符号表示法中，时间复杂度的公式是： $T(n) = O(f(n))$ ，其中 $f(n)$ 表示每行代码执行次数之和，而 O 表示正比例关系，这个公式的全称是：**算法的渐进时间复杂度**。

我们继续看上面的例子，假设每行代码的执行时间都是一样的，我们用1颗粒时间来表示，那么这个例子的第一行耗时是1个颗粒时间，第三行的执行时间是 n 个颗粒时间，第四行的执行时间也是 n 个颗粒时间（第二行和第五行是符号，暂时忽略），那么总时间就是1颗粒时间 + n 颗粒时间 + n 颗粒时间

，即 $(1+2n)$ 个颗粒时间，即： $T(n) = (1+2n) \times \text{颗粒时间}$ ，从这个结果可以看出，这个算法的耗时是随着 n 的变化而变化，因此，我们可以简化的将这个算法的时间复杂度表示为： $T(n) = O(n)$

为什么可以这么去简化呢，因为大O符号表示法并不是用于来真实代表算法的执行时间的，它是用来表示代码执行时间的增长变化趋势的。

所以上面的例子中，如果 n 无限大的时候， $T(n) = \text{time}(1+2n)$ 中的常量1就没有意义了，倍数2也意义不大。因此直接简化为 $T(n) = O(n)$ 就可以了。

常见的时间复杂度量级有：

- 常数阶 $O(1)$
- 对数阶 $O(\log N)$
- 线性阶 $O(n)$
- 线性对数阶 $O(n \log N)$
- 平方阶 $O(n^2)$
- 立方阶 $O(n^3)$
- K次方阶 $O(n^k)$
- 指数阶 (2^n)

上面从上至下依次的时间复杂度越来越大，执行的效率越来越低。

下面选取一些较为常用的来讲解一下（没有严格按照顺序）：

1. 常数阶 $O(1)$

无论代码执行了多少行，只要是没有循环等复杂结构，那这个代码的时间复杂度就都是 $O(1)$ ，如：

```
1 int i = 1;  
2 int j = 2;  
3 ++i;
```

```
4 j++;
5 int m = i + j;
```

上述代码在执行的时候，它消耗的时间并不随着某个变量的增长而增长，那么无论这类代码有多长，即使有几万几十万行，都可以用 $O(1)$ 来表示它的时间复杂度。

1. 线性阶 $O(n)$

这个在最开始的代码示例中就讲解过了，如：

```
1 for(i=1; i<=n; ++i)
2 {
3     j = i;
4     j++;
5 }
```

这段代码，for循环里面的代码会执行 n 遍，因此它消耗的时间是随着 n 的变化而变化的，因此这类代码都可以用 $O(n)$ 来表示它的时间复杂度。

1. 对数阶 $O(\log N)$

还是先来看代码：

```
1 int i = 1;
2 while(i<n)
3 {
4     i = i * 2;
5 }
```

从上面代码可以看到，在while循环里面，每次都将i乘以2，乘完之后，i距离n就越来越近了。我们试着求解一下，假设循环x次之后，i就大于n了，此时这个循环就退出了，也就是说2的x次方等于n，那么 $x = \log_2 n$

也就是说当循环 $\log_2 n$ 次以后，这个代码就结束了。因此这个代码的时间复杂度为： **$O(\log n)$**

1. 线性对数阶 $O(n \log N)$

线性对数阶 $O(n \log N)$ 其实非常容易理解，将时间复杂度为 $O(\log n)$ 的代码循环N遍的话，那么它的时间复杂度就是 $n * O(\log N)$ ，也就是了 $O(n \log N)$ 。

就拿上面的代码加一点修改来举例：

```
1 for(m=1; m<n; m++)
2 {
3     i = 1;
4     while(i<n)
5     {
6         i = i * 2;
7     }
8 }
```

1. 平方阶 $O(n^2)$

平方阶 $O(n^2)$ 就更容易理解了，如果把 $O(n)$ 的代码再嵌套循环一遍，它的时间复杂度就是 $O(n^2)$ 了。

举例：

```
1 for(x=1; i<=n; x++)
2 {
3     for(i=1; i<=n; i++)
4     {
5         j = i;
6         j++;
7     }
```



```
8 }
```

这段代码其实就是嵌套了2层n循环，它的时间复杂度就是 $O(n*n)$ ，即 $O(n^2)$

如果将其中一层循环的n改成m，即：

```
1 for(x=1; i<=m; x++)
2 {
3     for(i=1; i<=n; i++)
4     {
5         j = i;
6         j++;
7     }
8 }
```

那它的时间复杂度就变成了 $O(m*n)$

1. 立方阶 $O(n^3)$ 、K次方阶 $O(n^k)$

参考上面的 $O(n^2)$ 去理解就好了， $O(n^3)$ 相当于三层n循环，其它的类似。

除此之外，其实还有 平均时间复杂度、均摊时间复杂度、最坏时间复杂度、最好时间复杂度 的分析方法，有点复杂，这里就不展开了。

二、空间复杂度

既然时间复杂度不是用来计算程序具体耗时的，那么我也应该明白，空间复杂度也不是用来计算程序实际占用的空间的。

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的一个量度，同样反映的是一个趋势，我们用 $S(n)$ 来定义。

空间复杂度比较常用的有： $O(1)$ 、 $O(n)$ 、 $O(n^2)$ ，我们下面来看看：

1. 空间复杂度 $O(1)$

如果算法执行所需要的临时空间不随着某个变量 n 的大小而变化，即此算法空间复杂度为一个常量，可表示为 $O(1)$

举例：

```
1 int i = 1;
2 int j = 2;
3 ++i;
4 j++;
5 int m = i + j;
```

代码中的 i 、 j 、 m 所分配的空间都不随着处理数据量变化，因此它的空间复杂度 $S(n) = O(1)$

1. 空间复杂度 $O(n)$

我们先看一个代码：

```
1 int[] m = new int[n]
2 for(i=1; i<=n; ++i)
3 {
4     j = i;
5     j++;
6 }
```

这段代码中，第一行new了一个数组出来，这个数据占用大小为 n ，这段代码的2-6行，虽然有循环，但没有再分配新的空间，因此，这段代码的空间复杂度主要看第一行即可，即 $S(n) = O(n)$

算法基础01-数组、链表、跳表

一、数组本质

- 数组的本质是把数据存储在计算机内存管理器开辟的**连续内存地址对应的位置**
 - 所以数组的随机访问时间复杂度为 $O(1)$ ，搜索元素的时间复杂度为 $O(n)$
- 插入删除元素由于平均需要移动半个数组的元素，平均时间复杂度为 $O(n)$

二、链表本质

- 链表的本质是**每个元素靠指针指向其相邻元素**，随机访问需要遍历整个链表
 - 访问和搜索的时间复杂度为 $O(n)$
 - 插入删除元素只需要处理相邻元素的指针指向关系，所以插入和删除的时间复杂度为 $O(1)$

三、跳表本质



- 跳表的本质是在链表的基础上进行**升维**，加入**多级索引**，每级索引不再是跳向相邻元素，而是跳跃 2^k 个元素

- 其底层实现有多种方式，有BST二叉搜索树、AVL平衡二叉树等
- 故其访问和搜索的时间复杂度为 $O(\log n)$
- 插入和删除元素的时间复杂度也是 $O(\log n)$
- redis 的sorted set的底层实现就是跳表[\[1\]](#)

算法基础02-栈、队列、优先队列、双端队列、哈希表、映射、集合

一、栈、队列、双端队列、优先队列本质

- 栈的本质是以后进先出LIFO方式插入删除元素的数据结构
- 队列的本质是以先进先出方式插入删除元素的数据结构
- 双端队列的本质是栈+队列，同时支持先进先出和后进先出
- 优先队列的本质是入队与队列相同，出队按优先级顺序出队，底层实现可能是堆、平衡二叉树等
- 栈、队列、双端队列的插入删除时间复杂度都是 $O(1)$ ，搜索和遍历时间复杂度都是 $O(n)$
- 优先队列的插入时间复杂度是 $O(1)$ ，取出的时间复杂度是 $O(\log n)$

二、哈希表、映射、集合本质

- 哈希表的本质是根据Key直接访问Value的数据结构，通过把Key经过Hash Function映射到表中某个位置，以加快访问速度
- 映射的本质是key-value对，其中key唯一
- 集合的本质是元素唯一
- 哈希表的插入、删除、搜索时间复杂度都是 $O(1)$

三、Java Queue 和 Priority Queue源码分析

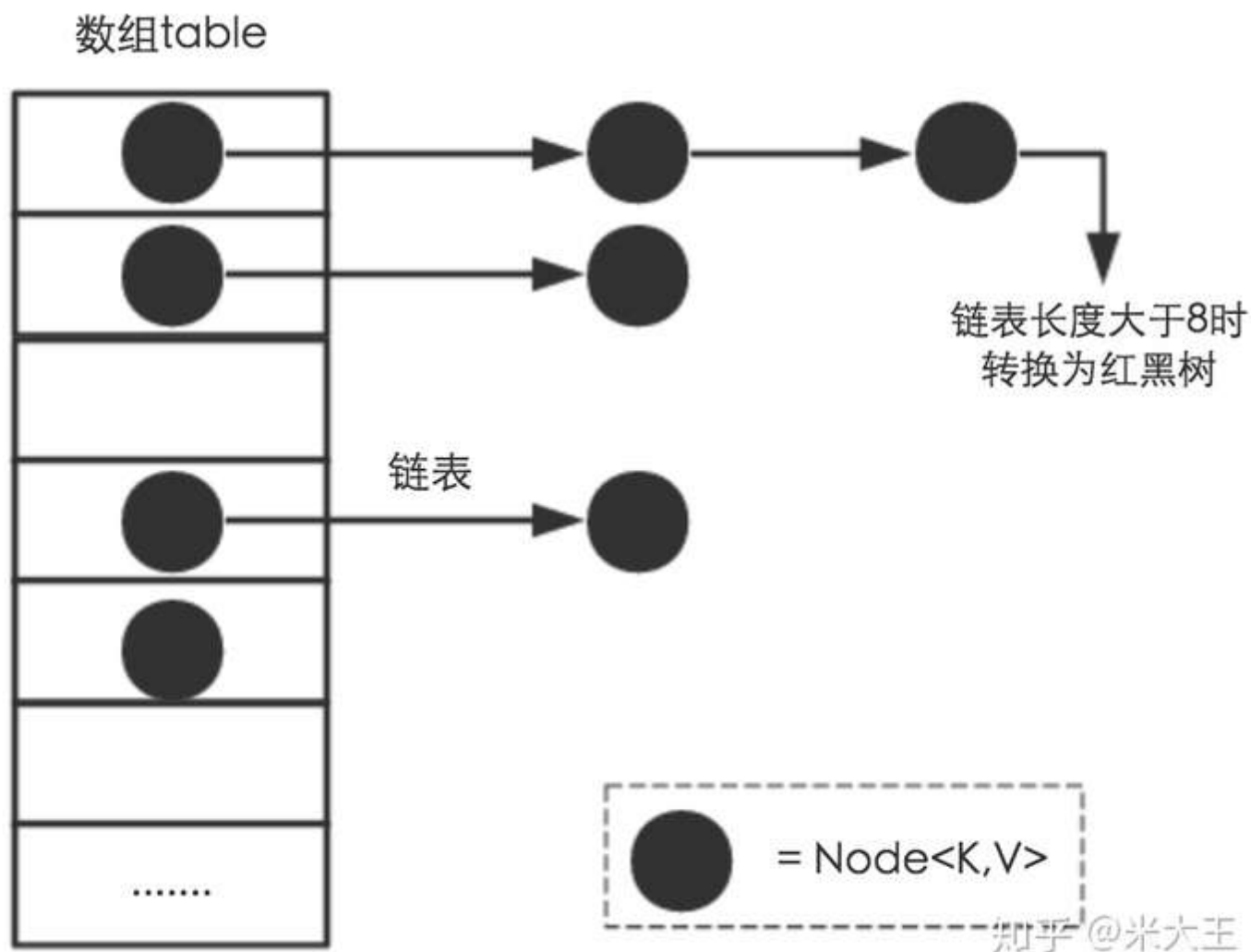
- Priority Queue是通过数组实现一个堆，元素在queue数组中并不是完全有序的，仅堆顶元素最大或最小
- poll方法，实际上是获取堆顶元素，然后调整堆
- 调整堆的方法（大顶堆为例）：

1.判断是否传入comparator，有则按照comparator顺序，否则按照自然顺序排序

2.取节点左右孩子节点的最大值，与父节点交换

四、Java HashMap总结

- 根据键的hashCode值存储数据，访问时间复杂度近似 $O(1)$ ，但遍历顺序不确定
- 非线程安全，任意时刻可以有多个线程同时写HashMap，可能会导致数据混乱，如果需要满足线程安全，可以用 Collections的synchronizedMap方法使HashMap具有线程安全的能力，或者使用ConcurrentHashMap
- 为了解决哈希冲突，Java采用链地址法（另一种方法是开放地址法），底层实现是数组+链表+红黑树的组合



Java 8 HashMap底层实现

- 具体实现过程[1]为：

1. 先调用key的hashCode方法得到hashCode值
2. 再通过Hash算法中的高位运算和取模运算，确定键值对的存储位置
3. 当HashCode值相等时，发生哈希碰撞，此时先判断当前地址下的的链表长度是否大于8，如果大于8就把链表转为红黑树，否则进行链表的插入操作
4. 插入成功后，判断实际存在的键值对数量是否超过最大容量threshold，如果超过就扩容

算法基础03-树、二叉树、递归、分治、回溯、图、堆

一、树的本质

树是一个由根和子树构成的二维数据结构，满足以下几个特点：

- 每个节点都只有有限个子节点或无子节点；
- 没有父节点的节点称为根节点；
- 每一个非根节点有且只有一个父节点；
- 除了根节点外，每个子节点可以分为多个不相交的子树；
- 树里面没有环路(cycle)

!img

由于树的每个节点都具有相同的特点，所以跟树相关的问题几乎都能用**递归**来解决。

二叉树

每个节点最多只有两颗子树的树。

二叉树的遍历

```
1 class Solution {
2     public List<Integer> inorderTraversal(TreeNode root) {
3         List<Integer> res = new ArrayList<>();
4         dfs(res, root);
5         return res;
6     }
7
8     void dfs(List<Integer> res, TreeNode root){
9         if(root == null) return;
10        dfs(res, root.left);
```

```
11         res.add(root.val); // 根据访问根节点的顺序不同，可以分为前序、中序、后序遍历，  
           这里是中序遍历  
12         dfs(res, root.right);  
13     }  
14 }
```

二叉搜索树（Binary Search Tree）

二叉搜索树，也称二叉搜索树、有序二叉树（Ordered Binary Tree）、排序二叉树（Sorted Binary Tree），是指一棵空树或者具有下列性质的二叉树：

- 左子树上的**所有节点**小于根节点
- 右子树上的**所有节点**大于根节点
- 根节点的左右子树也为二叉搜索树

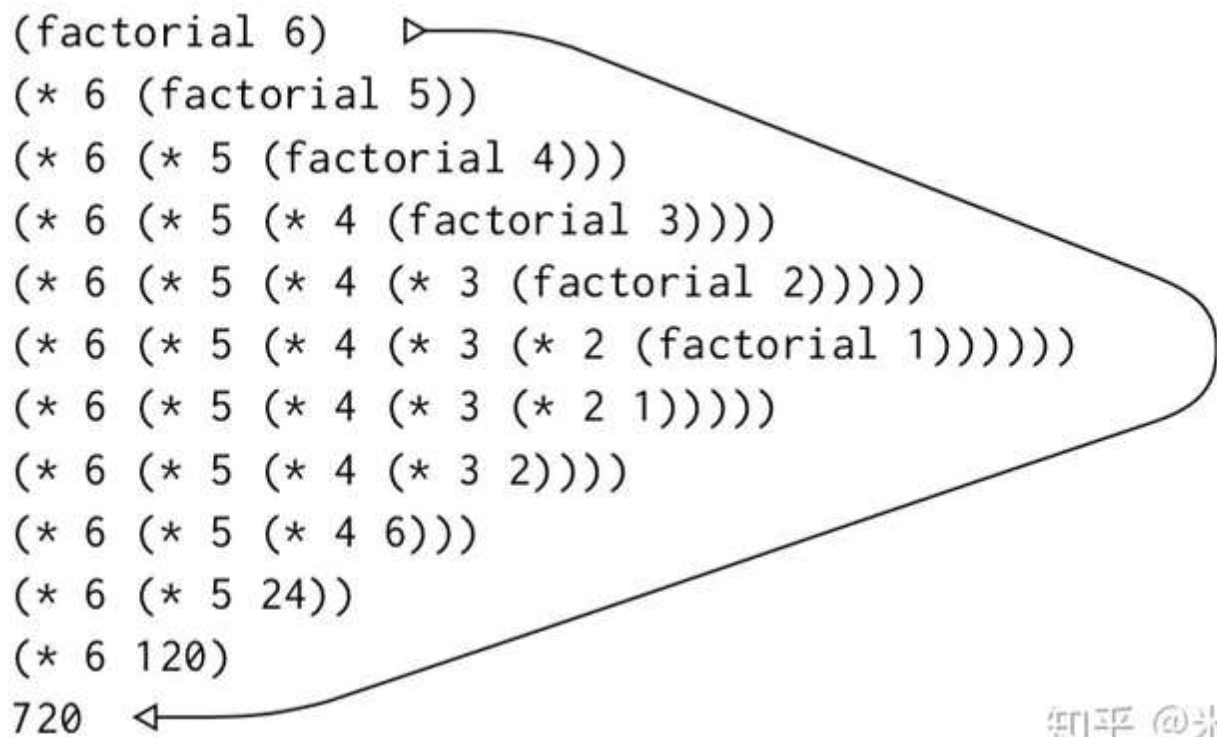
!img

根据这些性质，二叉搜索树最大的特点就是其中序遍历是**升序**的，另外二叉搜索树插入、删除、访问、搜索时间复杂度都是 $O(\log n)$ 。

二、递归

递归，讲起来很简单，先递进再回归。


```
(factorial 6)  ➤
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1))))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720  ←
```



知乎 @米大王

我认为递归是计算机计算效率远超人类的三大原因之一（判断、循环和递归）。因为相比计算机，世代代人类的老祖先们却只能顺序迭代，所以这就导致了递归是非常**反人类天性**的，这也是大部分人觉得递归难的根本原因。

递归模板

这里给出递归的Python代码模板，其他语言类似。

```
1 def recursion(level, param1, param2...):
2     # 1.terminator
3     if level > MAX_LEVEL:
4         process_result
5         return
6     # 2.process
7     process(level, data...)
8     # 3.drill down
9     recursion(level+1, p1, p2...)
10    # 4.restore the current level's states if needed
```

递归的要点

1. 不要一层一层人肉递归，因为人肉递归会很耗脑力，让你非常累（反直觉，但没有办法）
2. 找到最近重复子问题，或者将一个大问题拆成可重复的子问题
3. 数学归纳法思维

分治

分治的思想就是把大问题分成小问题，解决小问题，再把小问题的解答合并得到大问题的解答。

分治的代码模板就是在递归的基础上增加了split the problem和merge subproblems

```
1 def divide_conquer(problem, param1, param2, ...):
2     # recursion terminator
3     if problem is None:
4         print_result
5         return
6     # prepare data
7     data = prepare_data(problem)
8     subproblems = split_problem(problem, data)
9     # conquer subproblems
10    subresult1 = self.divide_conquer(subproblems[0], p1, ...)
11    subresult2 = self.divide_conquer(subproblems[1], p1, ...)
12    subresult3 = self.divide_conquer(subproblems[2], p1, ...)
13    ...
14    # process and generate the final result
15    result = process_result(subresult1, subresult2, subresult3, ...)
16    # revert the current level states
```

回溯

回溯的思想就是在递归解决问题的过程中，遇到不满足条件的情况，返回上一层重新选择路径解决。

回溯的代码模板是在递归的基础上，加入了对选择列表的循环，最重要的是**restore current states**，这一步也是回溯的精髓所在。

```
1 result = []
2 def backtrack(路径, 选择列表):
3     if 满足结束条件:
4         result.add(路径)
5         return
6
7     for 选择 in 选择列表:
8         # 做选择
9         路径.add(选择)
10        (将该选择从选择列表移除)
11        backtrack(路径, 选择列表)
12        # 撤销选择, restore current states
13        路径.remove(选择)
14        (将该选择重新添加到选择列表)
```

三、图的本质

图是由定点和边构成的二维数据结构，用来表示元素之间的关系。从定义的角度来说，树其实是特殊化的图（无向无环），而链表又是特殊化的树（只有单边子树）。图的遍历方法有广度优先搜索(Breadth First, BFS Search)和深度优先搜索(Depth First Search, DFS)。

四、堆的本质

堆是一个顶点元素总是小于其子节点元素的数据结构，所以堆的堆顶元素总是元素列表中最大值或最小值。

二叉堆

二叉堆（英语：binary heap）是一种特殊的堆，二叉堆是完全二叉树或者是近似完全二叉树。二叉堆满足堆特性：父节点的键值总是保持固定的序关系于任何一个子节点的键值，且每个节点的左子树和右子树都是一个二叉堆。因为这些性质，二叉堆的插入、删除、搜索、访问任意元素的操作时间复杂度都是 $O(\log n)$ ，访问最大或最小元素的时间复杂度为 $O(1)$ 。

算法基础04-深度优先搜索、广度优先搜索、二分查找、贪心算法

深度优先搜索DFS、广度优先搜索BFS

比较

- 拿谚语打比方的话，深度优先搜索可以比作**打破砂锅问到底、不撞南墙不回头**；广度优先搜索则对应**广撒网，多敛鱼**
- 两者没有绝对的优劣之分，只是**适用场景不同**
- 当解决方案离树根不远或搜索深度可变时，BFS通常更好，因为只需搜索所有数据中的一部分。另外BFS的一个重要优点是它可以用于找到**无权图**（有权图用Dijkstra算法，贪心思想）中任意两个节点之间的**最短路径**（不能使用DFS）
- 如果树比较宽而且深度有限，DFS可能是更优选项，因为DFS比BSF**更节省空间**，另外由于使用递归，DFS更好写（BFS必须手动维护队列）

时间复杂度

都是 $O(n)$

空间复杂度

都是 $O(n)$

经典题目

1. DFS, 深度优先, 用level记录当前层, 如果当前层记录完毕就return, 结果数组用level索引当前层, 结果数组容量小于等于level就扩容, 再DFS递归到下一层
2. BFS, 广度优先, 用队列先进先出的特性, 先确定当前层的节点数量, 遍历当前层, 把当前层的下一层所有节点入队, 当前层节点出队并放到一个临时数组, 再添加到结果

```
1 // DFS-Java
2 class Solution {
3     public List<List<Integer>> res = new LinkedList<>();
4     public List<List<Integer>> levelOrder(TreeNode root) {
5         dfs(0, root);
6         return res;
7     }
8     private void dfs(int level, TreeNode root){
9         if(root == null) return;
10        if(res.size() <= level){
11            res.add(new LinkedList<>());
12        }
13        res.get(level).add(root.val);
14        dfs(level+1, root.left);
15        dfs(level+1, root.right);
16    }
17 }
```

```
1 // BFS-Java
2 class Solution {
3     public List<List<Integer>> levelOrder(TreeNode root) {
4         List<List<Integer>> res = new LinkedList<>();
5         Queue<TreeNode> queue = new LinkedList<>();
6         if(root == null) return res;
7         queue.add(root);
8         while(!queue.isEmpty()){
9             int level = queue.size();
10            List<Integer> tempList = new LinkedList<>();
11            for(int i = 0; i < level; i++){
12                if(queue.peek().left != null) queue.add(queue.peek().left);
13                if(queue.peek().right != null) queue.add(queue.peek().right);
14                tempList.add(queue.poll().val);
15            }
16            res.add(tempList);
17        }
18        return res;
19    }
20 }
```

```
15         }
16         res.add(tempList);
17     }
18     return res;
19 }
20 }
```

二分查找

本质

二分查找的本质是在一组**单调、有上下界、可索引**的数据中搜索目标值，三大条件缺一不可。

时间复杂度

$O(\log N)$

空间复杂度

$O(1)$

经典题目

2. 1. 二分查找，因为 $y=x^2$ 在 x 正半轴**单调递增**，且存在**上下界1和x**，所以可以使用二分查找逼近

3. 牛顿迭代法，利用切线逼近，公式为 $x = (x + a/x) / 2$

```
1 // 二分查找
2 class Solution {
3     public int mySqrt(int x) {
4         if(x == 0 || x == 1) return x;
5         long left = 1, right = x, mid = 1;
```

```

6         while(left <= right){
7             mid = left + (right - left) / 2;
8             if(mid * mid > x){
9                 right = mid - 1;
10            } else {
11                left = mid + 1;
12            }
13        }
14        return (int) right;
15    }
16 }

```

```

1
2
3
4 ```java
5 // 牛顿迭代法
6 class Solution {
7     public int mySqrt(int x) {
8         long r = x;
9         while (r * r > x) {
10             r = (r + x / r) / 2;
11         }
12         return (int)r;
13     }
14 }

```

贪心算法

本质

贪心的本质是通过每一步的局部最优，期望实现全局最优的一种算法思想。关键在于局部最优是否真的能实现全局最优。如果能实现，那么贪心算法基本上就是问题的最优解。

经典例题

4. 倒序贪心，看最后能不能到达第一个点
5. 正序贪心，如果某个点可以跳到最后，那么它左边的点一定可以

```
1 // 倒序贪心
2 class Solution {
3     public boolean canJump(int[] nums) {
4         if (nums == null) return false;
5         int endReachable = nums.length - 1;
6         for (int i = nums.length - 1; i >= 0; i--){
7             if(nums[i] + i >= endReachable){
8                 endReachable = i;
9             }
10        }
11        return endReachable == 0;
12    }
13 }
```

```
1 // 正序贪心
2 class Solution {
3     public boolean canJump(int[] nums) {
4         int maxPos = 0;
5         for (int i = 0; i < nums.length; i++){
6             if(i > maxLen) return false;
7             maxPos = Math.max(maxPos, nums[i] + i);
8             if(maxPos >= nums.length-1) break;
9         }
10        return true;
11    }
12 }
```

算法基础05-动态规划

定义

让我们看看维基百科中动态规划的定义，截取其中最关键的一部分

*Dynamic programming refers to simplifying a complicated problem by **breaking it down into simpler sub-problems** in a recursive manner.*

*Likewise, in computer science, if a problem can be **solved optimally by breaking it into sub-problems** and then **recursively finding the optimal solutions to the sub-problems**, then it is said to have optimal substructure.*

翻译过来就是：动态规划指的是通过把一个问题**递归拆解**成更加简单的子问题的方式简化一个复杂问题。在计算机科学中，如果一个问题可以通过先拆解成简单子问题，寻递归找到每个子问题的最优解，这样我们就可以认为这个问题存在**最优子结构**。

本质

分治+最优子结构（每步选最优，淘汰次优）

动态规划 “三板斧”

1. 分治，找到最优子结构 `opt[n]=best_of(opt[n-1], opt[n-2], ...)`
2. 状态定义，i 条件时的状态 `f[i]`
3. DP方程，也就是递推公式，例如一维的斐波那契递推公式 `dp[i] = dp[i-1] + dp[i-2]`

二维递推公式例如 `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`，高级的DP公式可能会达到三维甚至三维以上

经典问题

思路：

4. 暴力递归，指数级时间复杂度

5. DP

a. 分治（子问题） $\text{path} = \text{path}(\text{top}) + \text{path}(\text{left})$

b. 状态定义 $f[i, j]$ 表示第*i*行第*j*列的不同路径数

c. DP方程 $\text{dp}[i][j] = \text{dp}[i-1][j] + \text{dp}[i][j-1]$

代码

```
1 class Solution {
2     public int uniquePaths(int m, int n) {
3         int[][] dp = new int[m][n];
4         for(int i = 0; i < n; i++) dp[0][i] = 1;
5         for(int i = 0; i < m; i++) dp[i][0] = 1;
6         for(int i = 1; i < m; ++i){
7             for(int j = 1; j < n; ++j){
8                 dp[i][j] = dp[i-1][j] + dp[i][j-1];
9             }
10        }
11        return dp[m-1][n-1];
12    }
13 }
```

对于我们的DP公式` $\text{dp}[i][j] = \text{dp}[i-1][j] + \text{dp}[i][j-1]$ `，我们还可以继续优化，因为求的是到达终点的不同路径，我们没必要保存到每行每列任意点的不同路径数，其实只需要保存每一行任意点的路径数即可，DP方程可以更新为` $\text{dp}[i] = \text{dp}[i] + \text{dp}[i-1]$ `

```
1 class Solution {
2     public int uniquePaths(int m, int n) {
3         int[] dp = new int[m];
4         dp[0] = 1;
5         for (int i = 0; i < n; ++i) {
6             for (int j = 0; j < m; ++j) {
7                 if (j > 0) dp[j] += dp[j-1];
8             }
9         }
10        return dp[m-1];
11    }
```

思路：

1. 暴力
2. DP

		0	1	2	3	4	5	6
	str2 \ str1	"	b	a	b	c	d	e
0	"	0	0	0	0	0	0	0
1	a	0	0	1	1	1	1	1
2	c	0	0	1	1	2	2	2
3	e	0	0	1	1	2	2	3

a. 分治 `LCS[i] = max (LCS(最后一个字母相同), LCS(最后一个字母不相同))`

b. 状态定义 `f[i][j]` 第一个字符串索引 0-i 构成的子串与第二个字符串索引 0-j 子串的最长公共序列

c. DP方程

```
```text
```

```
if text1[-1] == text2[-1]:
```

```
 dp[i][j] = dp[i-1][j-1] + 1
```

```
else:
```

```
 dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

## Java 实现

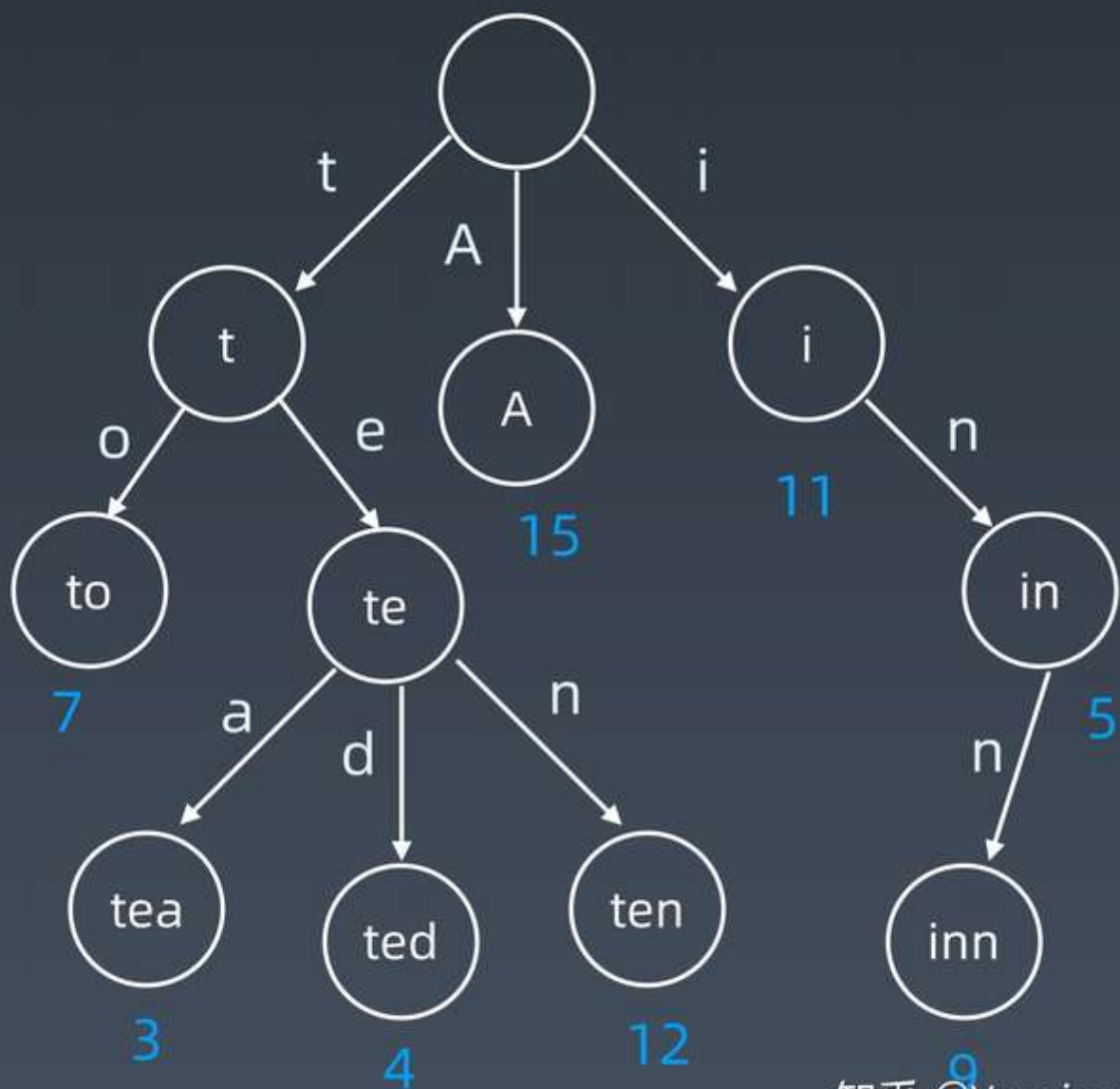
```
1 class Solution {
2 public int longestCommonSubsequence(String text1, String text2) {
3 char[] s1 = text1.toCharArray();
4 char[] s2 = text2.toCharArray();
5 int[][] dp = new int[s1.length+1][s2.length+1];
6 for (int i = 1; i < s1.length+1; ++i) {
7 for (int j = 1; j < s2.length+1; ++j) {
8 if(s1[i-1] == s2[j-1]){
9 dp[i][j] = dp[i-1][j-1] + 1;
10 } else {
11 dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
12 }
13 }
14 }
15 return dp[s1.length][s2.length];
16 }
17 }
```

## 算法基础06 - 字典树、并查集、高级搜索、红黑树、AVL 树

### 字典树（Trie 树）

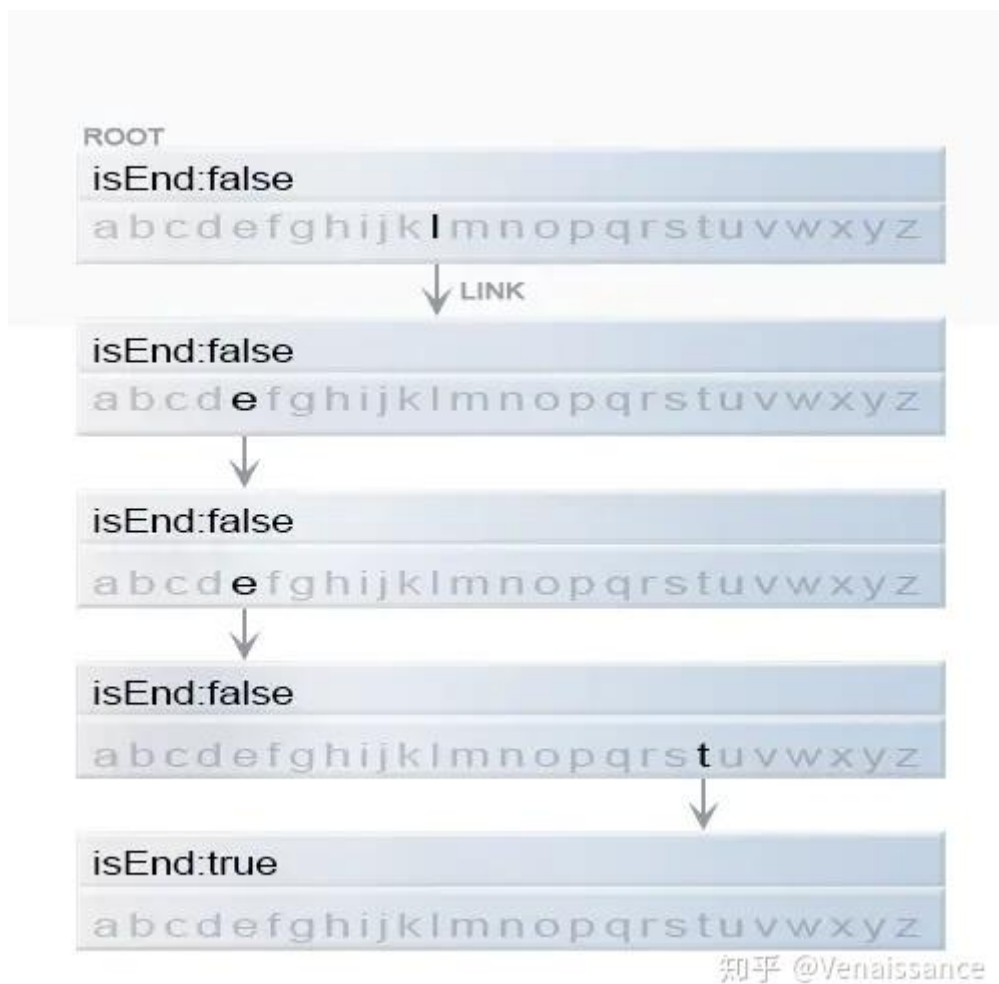
#### 基本结构

字典树是一颗沿着单词中字母排列顺序扩展的多叉树，典型应用是用于统计和排序大量的字符串（但不仅限于字符串），优点是最大限度减少无谓的字符串比较，查询效率比哈希表高。



知乎 @Venaissance

单词 leet 在 Trie 树中的表示



## 核心思想

Trie 树的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

## 主要用途

- 搜索引擎输入栏的自动补全
- 拼写检查
- IP 路由最长前缀匹配
- 打字预测

## 经典问题

思路：

- Trie 树+ DFS + 剪枝，时间复杂度![\[公式\]](#)，其中 L 为单词最大长度，m \* n 为网格单元数，空间复杂度为 O(N)，其中 N 为字典中字母总数

## Python解法

```
1 class Solution:
2 def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
3 # 构建 Trie
4 trie = {}
5 for word in words:
6 node = trie
7 for char in word:
8 node = node.setdefault(char, {})
9 node['#'] = True
10 # DFS
11 def _dfs(i, j, node, pre, visited):
12 if '#' in node:
13 res.add(pre)
14 for (di, dj) in ((-1,0), (1,0), (0, -1), (0, 1)):
15 x, y = i+di, j+dj
16 if 0 <= x < m and 0 <= y < n and board[x][y] in node and (x,y)
not in visited:
17 _dfs(x, y, node[board[x][y]], pre+board[x][y], visited |
{(x,y)})
18 # 主逻辑
19 res, m, n = set(), len(board), len(board[0])
20 for i in range(m):
21 for j in range(n):
22 if board[i][j] in trie:
23 _dfs(i, j, trie[board[i][j]], board[i][j], {(i,j)})
24 return list(res)
```

# 并查集

并查集，英文名 disjoint set，是用于处理不交集的合并以及查询问题的树形结构。

## 基本操作

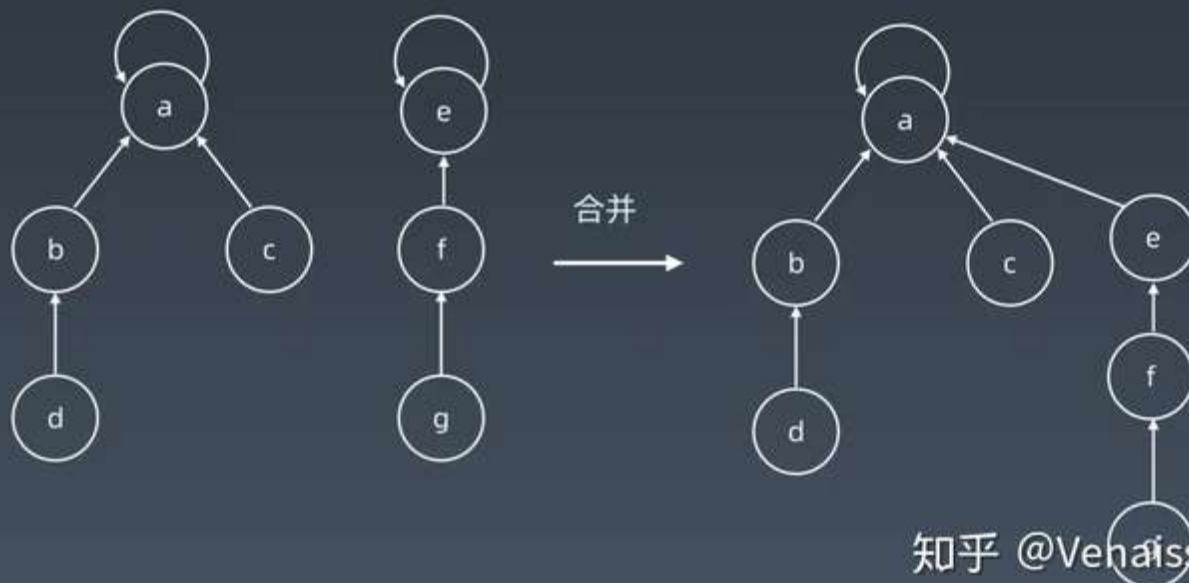
- MakeSet(s)：创建一个新的并查集，其中包含s个单元素集合

## 初始化



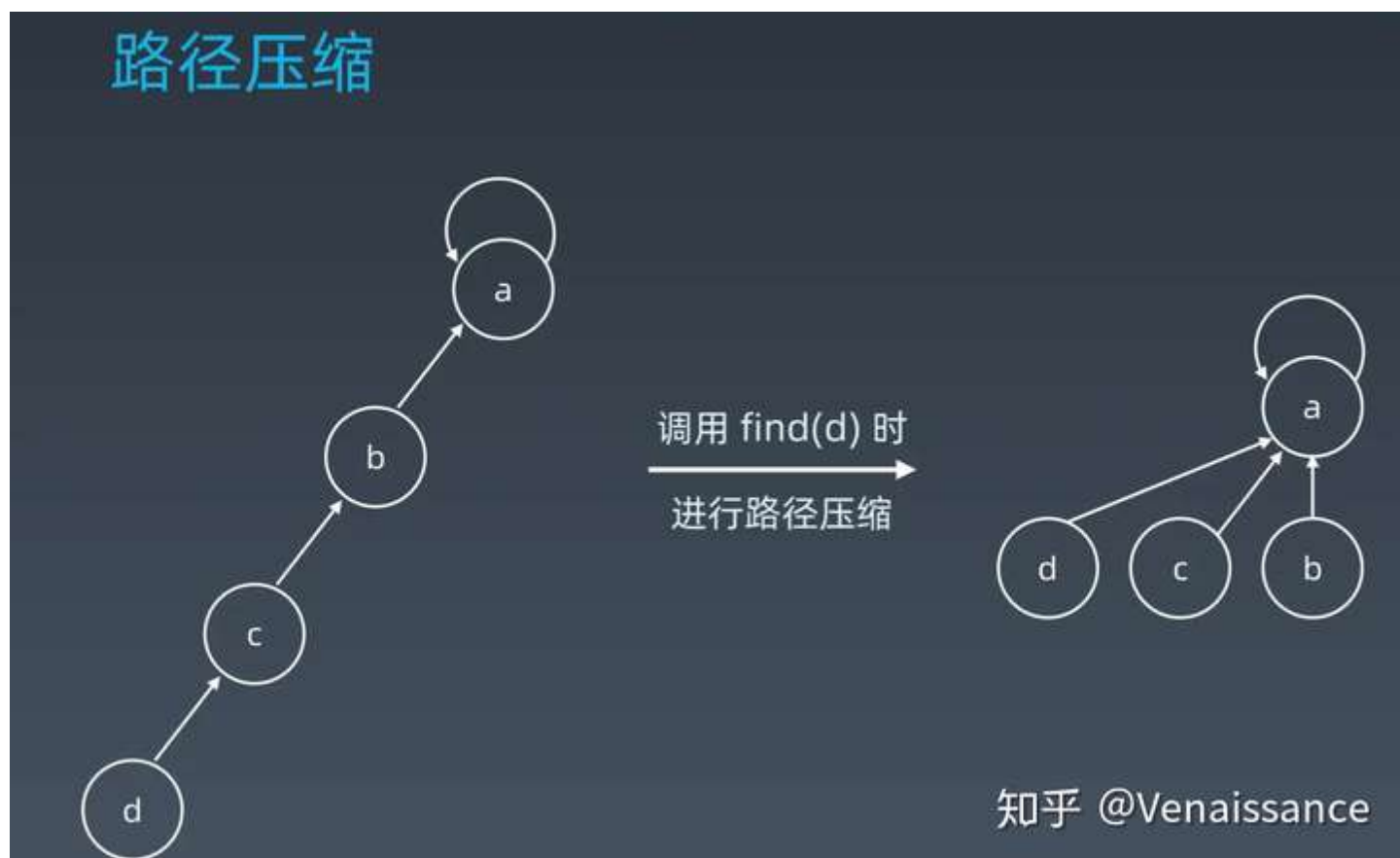
- Union(x, y)：将两个子集合并为一个
- Find(x)：找到元素所在集合的代表，进而确定元素属于哪一个子集

## 查询、合并





使用路径压缩可以使未来的查询时间复杂度降为  $O(1)$



## 代码模板

```
1 def init(p):
2 # for i = 0 .. n: p[i] = i;
3 p = [i for i in range(n)]
4
5 def union(self, p, i, j):
6 p1 = self.parent(p, i)
7 p2 = self.parent(p, j)
8 p[p1] = p2
9
10 def parent(self, p, i):
11 root = i
12 while p[root] != root:
13 root = p[root]
```

```
14 while p[i] != i: # 路径压缩
15 x = i; i = p[i]; p[x] = root
16 return root
```

---

## 高级搜索

想知道高级搜索之前，我们先来看看什么是初级搜索

### 初级搜索

1. 暴力搜索
2. 优化方向：剪枝、缓存
3. 搜索方向：DFS、BFS

高级搜索是对初级搜索的进一步改良，比如双向搜索、启发式搜索等。

### 双向搜索

核心思想是开始和结束位置同时开始搜索，如果能在中间相遇，说明可以搜索到，如果无法相遇，说明搜索不到。双端BFS是最典型的双向搜索技巧。

### 双端BFS搜索时间复杂度

$O(b \cdot d)$ ，其中  $b$  为BFS每层的宽度， $d$  为图的深度

普通BFS的时间复杂度为  $O(N^2)$ ，可以通过数学证明双端BFS时间复杂度会比普通BFS快很多。

## 双端BFS代码模板

```
1 public <T> two-endedBFS(begin, end, Container) {
2 if (Container.length == 0 || !Container.contains(end)) return NEGATIVE;
3 Set<T> beginSet = new HashSet<>(), endSet = new HashSet<>();
4 beginSet.add(begin);
5 endSet.add(end);
6
7 Set<T> visited = new HashSet<>();
8 int level = 0;
9 while(!beginSet.isEmpty() && !endSet.isEmpty()) {
10 if (beginSet.size() > endSet.size()){
11 Set<T> set = beginSet;
12 beginSet = endSet;
13 endSet = set;
14 }
15 Set<T> temp = new HashSet<>();
16 for (T item: beginSet){
17 T next = generate_nextLevel_nodes();
18 if (endSet.contains(next)) return POSITIVE;
19 if (Container.contains(next) && !visited.contains(next)){
20 temp.add(next);
21 visited.add(next);
22 }
23 // reverse node states
24 }
25 beginSet = temp;
26 level++;
27 }
28 return NEGATIVE;
29 }
```

## 启发式搜索

启发式搜索，也叫A\*算法，它的本质就是优先搜索。根据问题不同，定义不同的优先级比较器，按照优先级从大到小搜索。由于实现一般比较复杂，这里不作代码方面的展开。

## 红黑树 & AVL树

二叉搜索树的查询时间复杂度只跟树的深度有关，所以为了高效查询，需要保证每个节点子树的深度差不能过大，如此便诞生了平衡二叉树，其中最有名的是红黑树和AVL树。

### AVL树特点

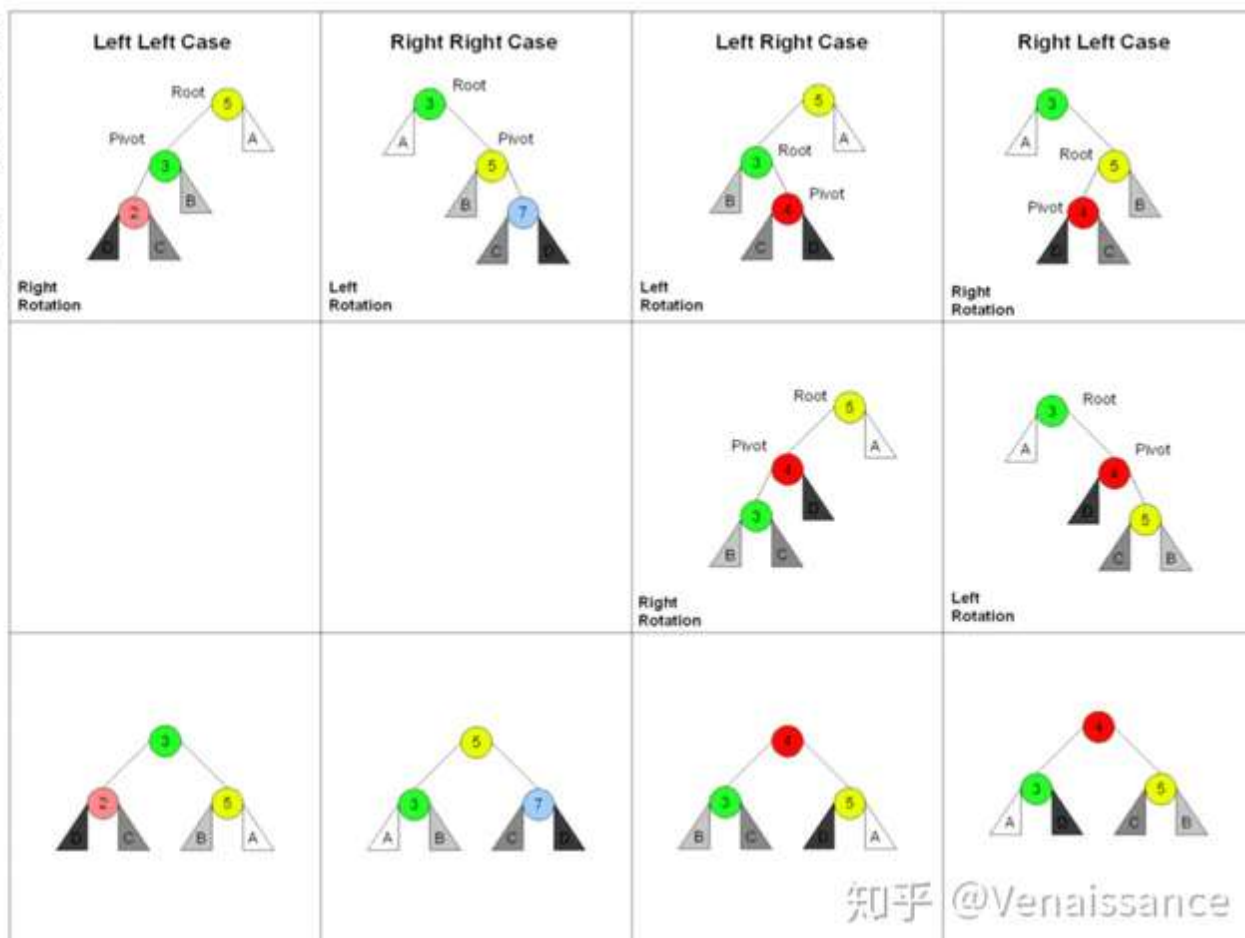
- Balance Factor（平衡因子）：左右子树的高度差，通常限制在  $\{-1, 0, 1\}$  范围内



- 通过旋转操作来进行平衡：左旋、右旋、左右旋、右左旋

There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and Pivot is the child to take the root's place.



## 红黑树特点

红黑树是一种近似平衡的二叉搜索树，它可以确保任何一个节点的左右子树的高度差小于两倍。

- 每个节点要么红，要么黑
- 根节点是黑色
- 每个子叶节点（NIL节点）是黑色的
- 不能有相邻接的两个红色节点
- 从任一节点到其每个子叶的所有路径都包含数量相同的黑色节点

- 最关键的性质是：从根到叶子的**最长可能路径不多于最短的可能路径的两倍长**

## AVL & 红黑树比较

- AVL查询更快，因为更加严格平衡

- 红黑树增加删除更快，因为相比AVL，红黑树需要的旋转操作更少
- AVL要存储平衡因子或子树高度，所以会消耗更多空间，而红黑树每个节点只需要1位信息（红 or 黑）
- 所以读操作非常多写操作不多时，AVL更好；写操作较多或读写参半时红黑树更好；
- 比如主流语言的库函数如map等是用红黑树实现的，而数据库用AVL实现较多

## 算法基础07-位运算、布隆过滤器、LRU Cache

### 位运算

#### 核心要领

想要熟练掌握位运算，记住下面几个公式即可：

- 清除 n 最低位的 1:  $n \& (n - 1)$
- 获取 n 最低位的 1:  $n \& -n$
- 整除 2:  $n \gg 1$
- 判断奇偶:  $n \& 1 == 1 \mid 0$
- $n \& \sim n = 0$

#### 经典例题

1. [N-Queens II](<https://link.zhihu.com/?target=https%3A//leetcode-cn.com/problems/n-queens-ii/>)

思路：

2. 回溯，时间复杂度  $O(n!)$ ，空间复杂度  $O(n)$

2. **位运算**，时间复杂度  $O(n!)$ ，空间复杂度  $O(n)$ ，由于位运算**更接近计算机底层**，所以运行速度会更快，位运算也是N-Queens问题的**终极解决方案**，算法的核心要点如下：

- 0 - 不能放皇后，1 - 能放
- `availPos = (~(cols | pie | na)) & ((1 << n) - 1)` 考虑整行、两条对角线，得到当前能放皇后的位置，用最低的  $n$  位表示
- `p = availPos & -availPos` 取得 `availPos` 最低位的1，用来放皇后
- `availPos &= (availPos - 1)` 清除最低位的 1，表示皇后已放
- `dfs(n, row + 1, cols | p, (pie | p) << 1, (na | p) >> 1)` 对 `cols`, `pie`, `na` 可放皇后的位置做相应的更新，下探下一层 (Drill Down)

## Java 实现

```
1 class Solution {
2 private int count = 0;
3 public int totalNQueens(int n) {
4 dfs(n, 0, 0, 0, 0);
5 return count;
6 }
7
8 private void dfs(int n, int row, int cols, int pie, int na) {
9 if (row == n) {
10 count++;
11 return;
12 }
13 int availPos = (~(cols | pie | na)) & ((1 << n) - 1);
14 while (availPos != 0) {
15 int p = availPos & -availPos;
16 availPos &= (availPos - 1);
17 dfs(n, row + 1, cols | p, (pie | p) << 1, (na | p) >> 1);
18 }
19 }
20 }
```

## Python 实现

```
1 class Solution:
2 def totalNQueens(self, n: int) -> int:
3 self.count = 0
4 self.dfs(n, 0, 0, 0, 0)
5 return self.count
6
7 def dfs(self, n, row, cols, pie, na):
8 if row == n:
9 self.count += 1
10 return
11 availPos = ~(cols | pie | na) & ((1 << n) - 1)
12
13 while availPos:
14 p = availPos & -availPos
15 availPos &= (availPos - 1)
16 self.dfs(n, row + 1, cols | p, (pie | p) << 1, (na | p) >> 1)
```

## 布隆过滤器（Bloom Filter）

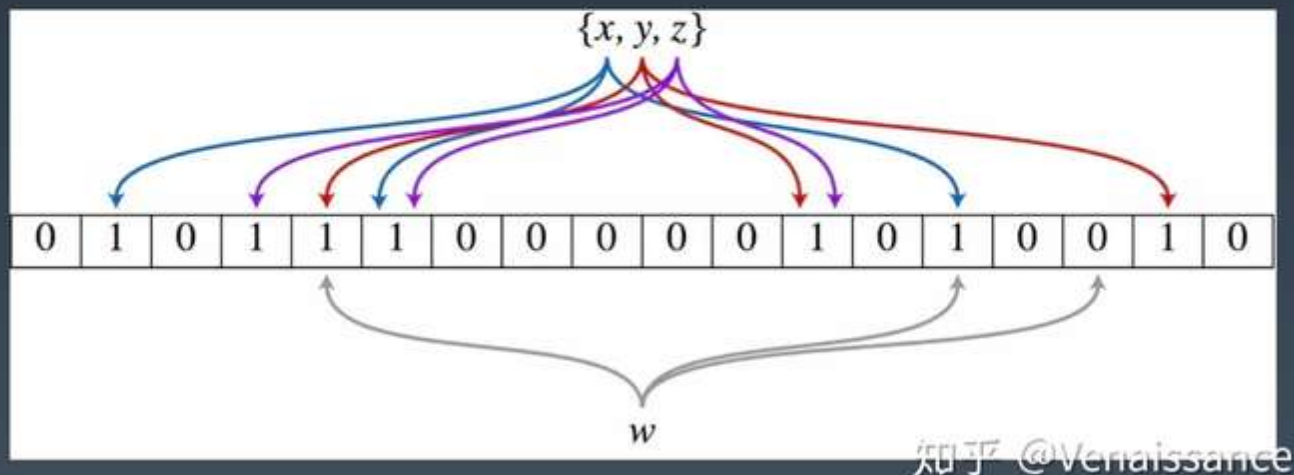
### 本质

布隆过滤器由一个很长的二进制向量和一系列随机映射函数组成。用于检索一个元素是否在一个集合中。

- 优点是空间效率和查询时间远超一般算法
- 缺点是有一定的错误识别率和删除困难



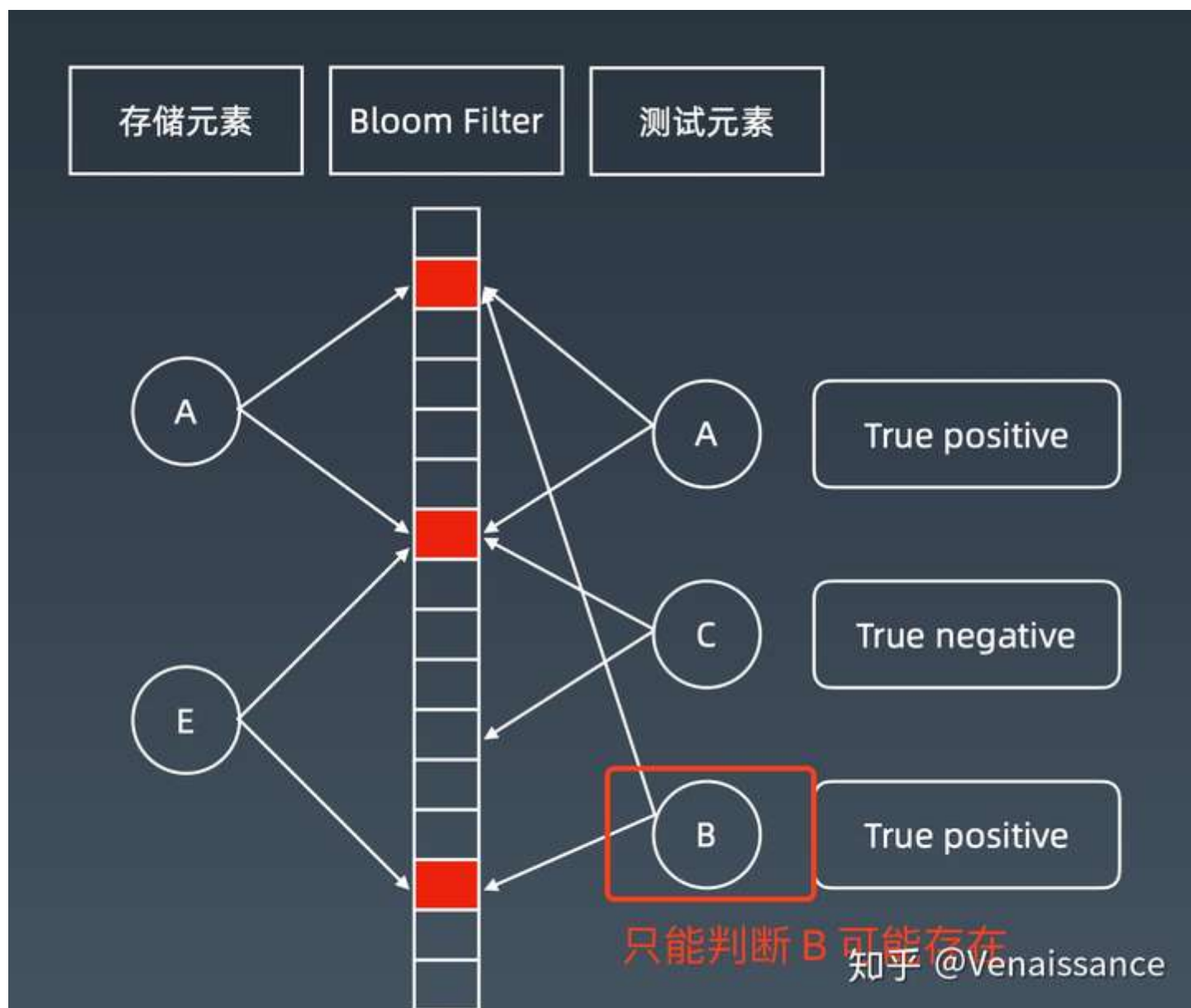
# 布隆过滤器示意图



## 要点

布隆过滤器是一个快速判断元素是否存在集合的算法，特点是：

3. 不需要像哈希表一样存额外的信息
2. 只能判断**肯定不存在**或**可能存在**
3. 适合用作**高速缓存**，如判断为可能存在，再到数据库中查询
4. 每个元素的存在用几个**二进制位置 1** 来表示
4. 多用于大型分布式系统如比特币网络、Redis缓存、垃圾邮件过滤器、评论过滤器等

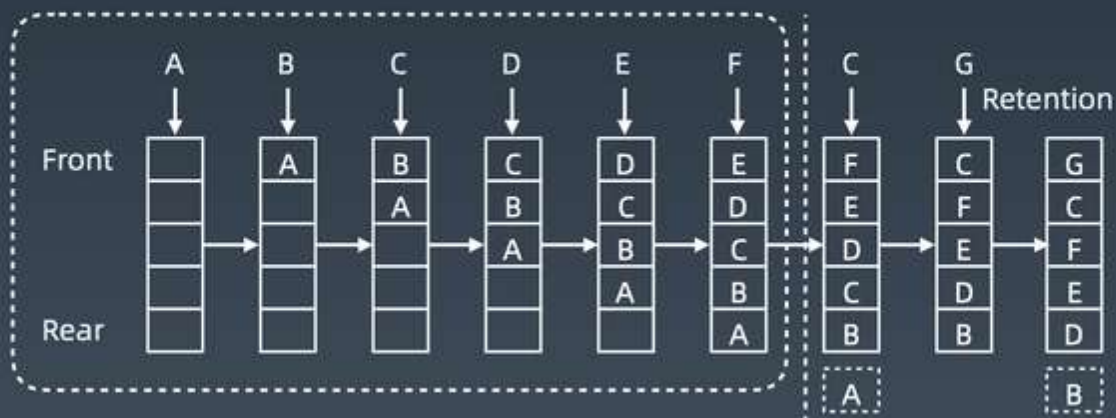


## LRU Cache

最近最少使用缓存替换策略，是一种 [缓存替换策略](#)，其他缓存策略还有 FIFO、LFU、RR 等等。

LRU: Latest Recently Used 最近最少使用。

# LRU cache 工作示例



被移除 知乎 @Verdissance

## 硬核实现 LRU Cache

接下来让我们自己造轮子，硬核实现一个 LRU Cache，它应该支持以下两个操作：

- 获取数据 `get(key)` - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。
- 写入数据 `put(key, value)` - 如果密钥已经存在，则变更其数据值；如果密钥不存在，则插入该组「密钥/数据值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

还有两个需求：

- 增加删除数据的时间复杂度为  $O(1)$
- 随机访问数据的时间复杂度为  $O(1)$

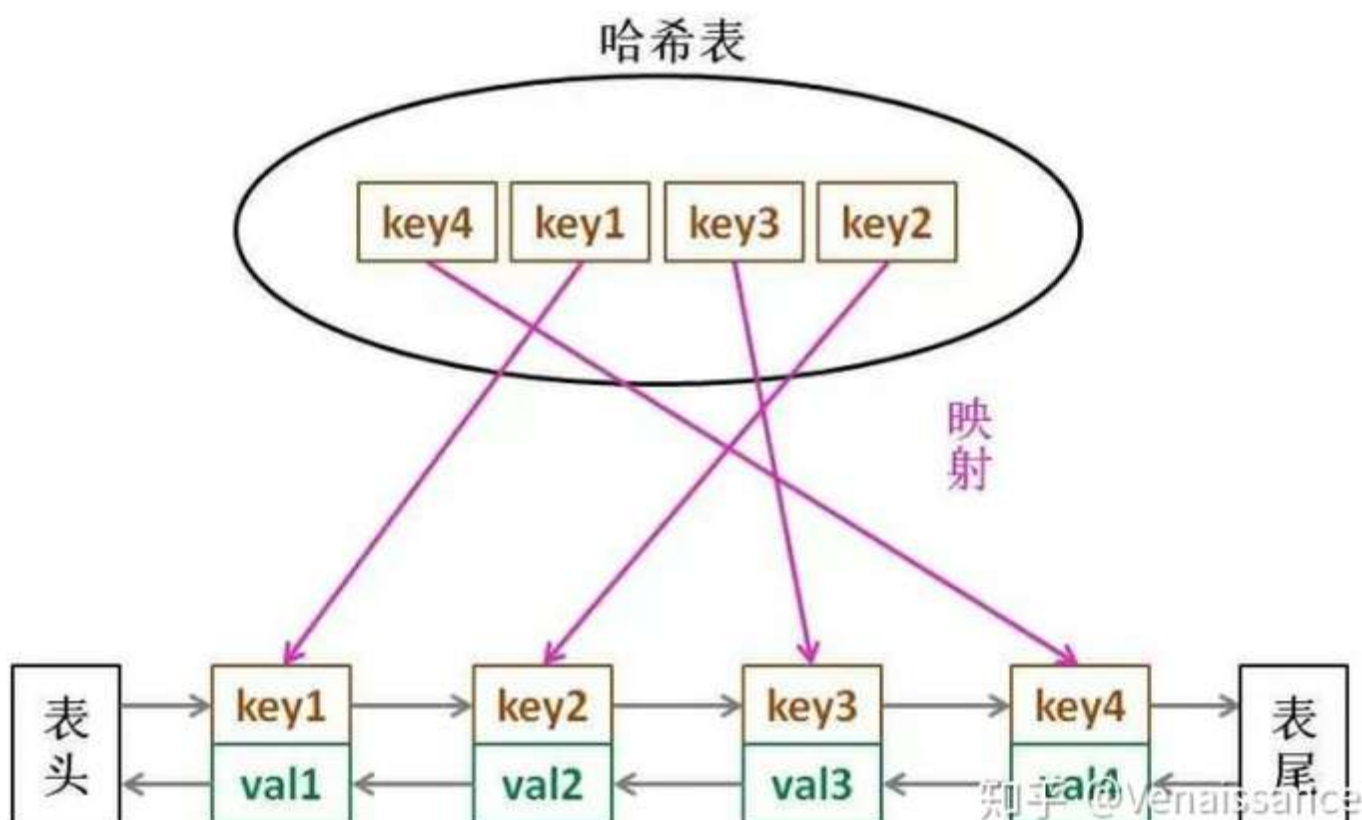
增删  $O(1)$ ，我们第一个想到要用 LinkedList，访问  $O(1)$ ，我们第一个想到要用 HashTable，那么怎么把两者结合起来呢？让我们从思考这两个问题开始：

5. 链表有特殊要求吗？单链表还是双向链表？

6. 链表中的节点存什么，只存 value 够吗？

先回答第一个问题，因为我们需要删除数据，链表删除节点需要找到当前节点的**前驱**，单链表想要找到节点前驱需要从头开始遍历，存在  $O(n)$  的时间复杂度；而双向链表本身就保存了节点的前驱，删除时间复杂度就是  $O(1)$ ，所以我们使用**双向链表**。

接下来回答第二个问题，由于缓存空间有限，当缓存存满时，根据 LRU 策略，我们需要移除缓存里保存最久的未被访问节点，比如上图中的 A 和 B，与此同时，我们还需要移除 HashMap 中的该节点（已不在缓存中，下次无法查到），如果节点只存 value，那么 HashMap 就无法移除该节点（哈希表删除节点需要 key，`map.remove(key)`），所以链表节点需要**同时存 key 和 value**，就像下图这样。



解答了这两个问题，我们就可以开始动手了，先实现一个同时保存 key 和 value 的双向链表。

```
1 class Node {
2 public int key, val;
3 public Node next, prev;
4 public Node(int k, int v) {
5 this.key = k;
6 this.val = v;
7 }
8 }
9
10 // 双向链表
11 class DoubleLinkedList {
12 private Node head, tail; // 头尾虚节点
13 private int size; // 链表元素数
14
15 public DoubleLinkedList() {
16 head = new Node(0, 0);
17 tail = new Node(0, 0);
18 head.next = tail;
19 tail.prev = head;
20 size = 0;
21 }
22
23 // 在链表头部添加节点
24 public void addFirst(Node node) {
25 node.next = head.next;
26 node.prev = head;
27 head.next.prev = node;
28 head.next = node;
29 size++;
30 }
31
32 // 删除链表中的节点（该节点一定存在）
33 public void remove(Node node) {
34 node.prev.next = node.next;
35 node.next.prev = node.prev;
36 size--;
37 }
38
39 // 删除链表中最后一个节点，并返回该节点
40 public Node removeLast() {
41 if (tail.prev == head)
42 return null;
43 Node last = tail.prev;
44 remove(last);
45 return last;
46 }
47 }
```

```
45 // 返回链表长度
46 public int size() {
47 return size;
48 }
49 }
```

```
1
2 接下来就是正式实现 LRU Cache 了，核心逻辑写在注释上了。
3
4 ```java
5 class LRUCache {
6 private HashMap<Integer, Node> map;
7 private DoubleLinkedList cache;
8 private int cap; // 最大容量
9 public LRUCache(int capacity) {
10 this.cap = capacity;
11 map = new HashMap<>();
12 cache = new DoubleLinkedList();
13 }
14
15 // 访问节点
16 public int get(int key) {
17 if (!map.containsKey(key)) {
18 return -1;
19 }
20 int val = map.get(key).val;
21 // 使用put方法把最近访问的节点提前
22 put(key, val);
23 return val;
24 }
25
26 public void put(int key, int value) {
27 // 生成新的节点
28 Node node = new Node(key, value);
29 if (map.containsKey(key)) {
30 // 删除旧的节点，新的插到头部
31 cache.remove(map.get(key));
32 cache.addFirst(node);
33 // 更新 map 中对应的数据
34 map.put(key, node);
35 } else {
36 // 如果缓存满了
37 if (cap == cache.size()) {
```

```
38 // 删除链表最后一个数据
39 Node last = cache.removeLast();
40 // 哈希表也要删
41 map.remove(last.key);
42 }
43 // 新节点添加到头部
44 cache.addFirst(node);
45 map.put(key, node);
46 }
47 }
48 }
```

## 算法基础08-排序算法

说到排序算法，它可能是最接近程序员日常工作的算法，像 Java、Python 里 `sort()` API 的实现，都离不开快速排序。非常经典的逆序对问题的最佳解法也需要用到归并排序，所以排序算法很大程度上能体现一个程序员的基础扎实程度。这里对排序算法做一个全面的归纳总结。

### 复杂度总览

除了最下面三种特殊排序算法，目前所有排序算法都无法突破  $O(n \log n)$  的时间复杂度下限。根据时间复杂度是  $O(n^2)$  还是  $O(n \log n)$ ，我们把排序算法分为初级排序算法和高级排序算法。

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

知乎 @Vernaisance