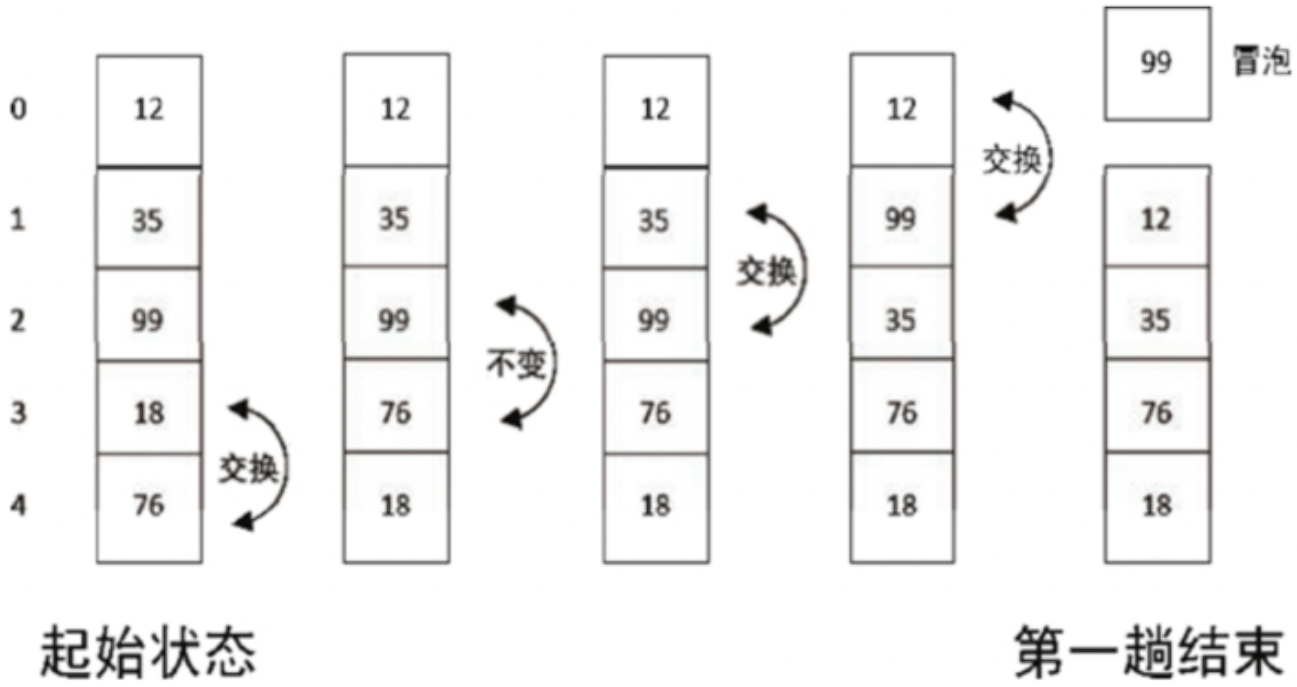
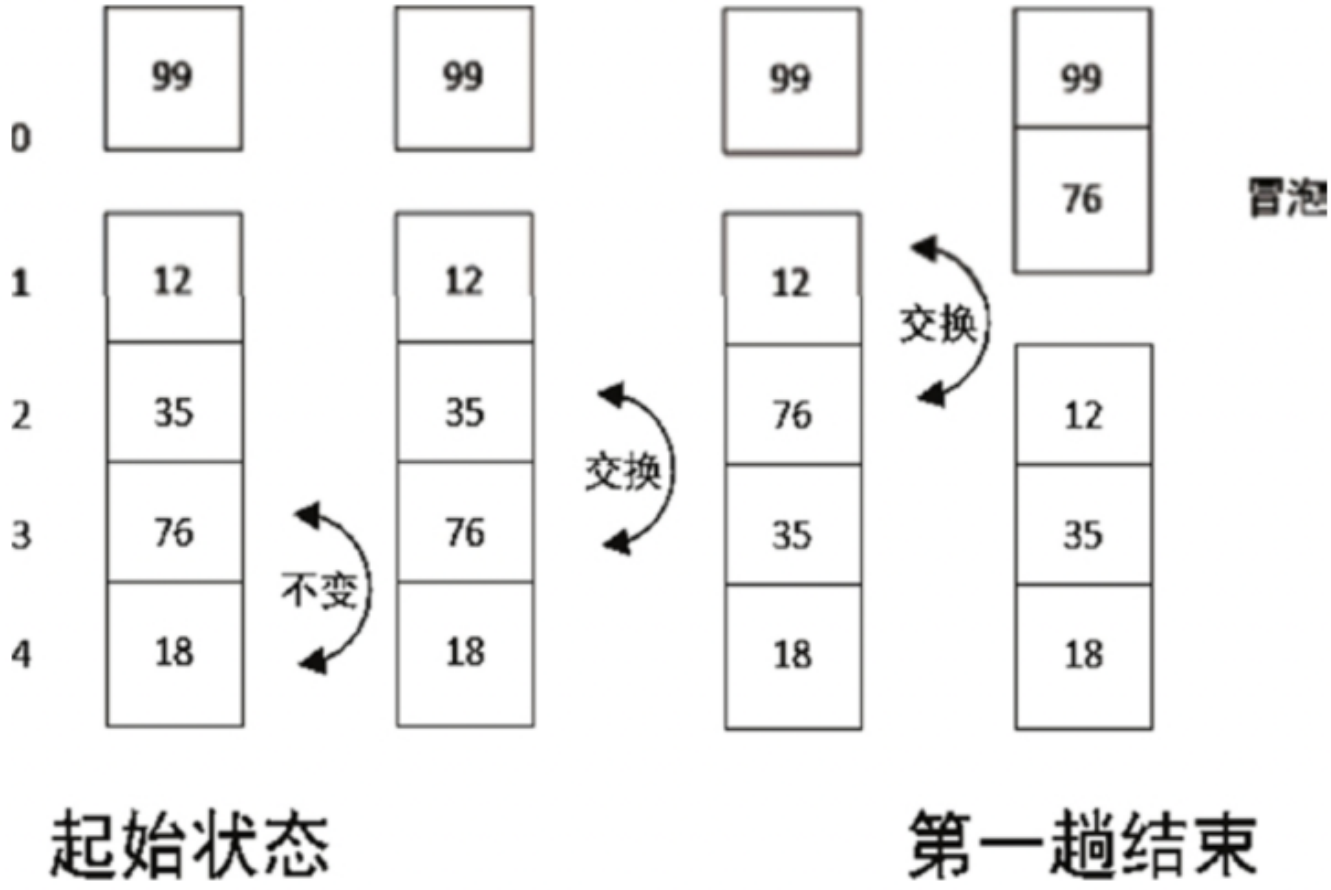


上述可以看到，经过第一趟的排序，可以得到最大的元素，接下来第二趟排序则对剩下的的4个元素进行排序，如下图所示：



经过第 2 趟排序，结果为 99、76、12、35、18

然后开始第3趟的排序，结果为99、76、35、12、18

然后第四趟排序结果为99、76、35、18、12

经过 4 趟排序之后，只剩一个 12 需要排序了，这时已经没有可比较的元素了，这时排序完成

11.2. 如何实现

如果要实现一个从小到大的排序，算法原理如下：

- 首先比较相邻的元素，如果第一个元素比第二个元素大，则交换它们
- 针对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对，这样，最后的元素回事最大的数
- 针对所有的元素重复以上的步骤，除了最后一个
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较

用代码表示则如下：

JavaScript | 复制代码

```javascript
function bubbleSort(arr) {
    const len = arr.length;
    for (let i = 0; i < len - 1; i++) {
        for (let j = 0; j < len - 1 - i; j++) {
            if (arr[j] > arr[j+1]) {        // 相邻元素两两对比
                var temp = arr[j+1];        // 元素交换
                arr[j+1] = arr[j];
                arr[j] = temp;
            }
        }
    }
    return arr;
}
```

可以看到：冒泡排序在每一轮排序中都会使一个元素排到一趟， 也就是最终需要 n-1 轮这样的排序

而在每轮排序中都需要对相邻的两个元素进行比较，在最坏的情况下，每次比较之后都需要交换位置，此时时间复杂度为 `O(n^2)`

11.2.1. 优化

对冒泡排序常见的改进方法是加入一标志性变量 `exchange` ，用于标志某一趟排序过程中是否有数据交换

如果进行某一趟排序时并没有进行数据交换，则说明数据已经按要求排列好，可立即结束排序，避免不必要的比较过程

可以设置一标志性变量 `pos` ，用于记录每趟排序中最后一次进行交换的位置，由于 `pos` 位置之后的记录均已交换到位，故在进行下一趟排序时只要扫描到 `pos` 位置即可，如下：

JavaScript | 复制代码

```javascript
function bubbleSort1(arr){
 const i=arr.length-1;//初始时,最后位置保持不变
 while(i>0){
  let pos = 0;//每趟开始时,无记录交换
  for(let j = 0; j < i; j++){
   if(arr[j] > arr[j+1]){
        let tmp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = tmp;
     pos = j;//记录最后交换的位置
   }
  }
  i = pos;//为下一趟排序作准备
 }
 return arr;
}
```

在待排序的数列有序的情况下，只需要一轮排序并且不用交换，此时情况最好，时间复杂度为 `O(n)`

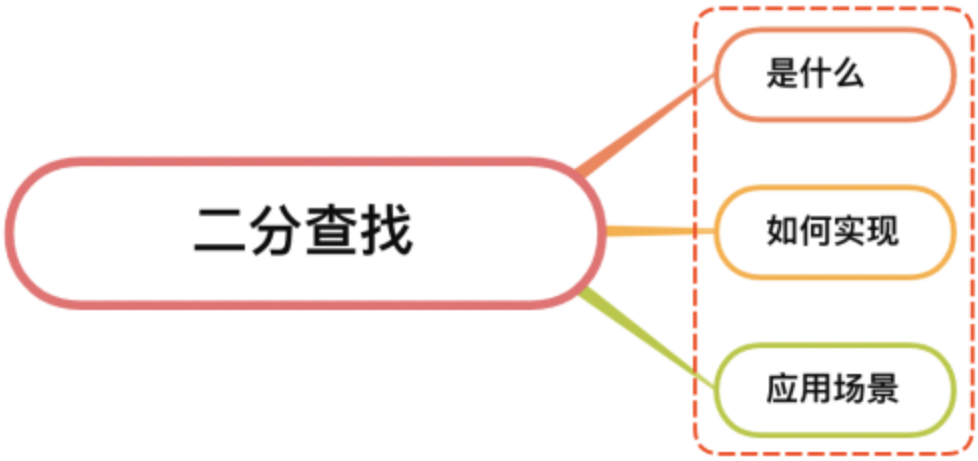
并且从上述比较中看到，只有后一个元素比前面的元素大（小）时才会对它们交换位置并向上冒出，对于同样大小的元素，是不需要交换位置的，所以对于同样大小的元素来说，相对位置是不会改变的，因此， 冒泡排序是稳定的

11.3. 应用场景

冒泡排的核心部分是双重嵌套循环，
时间复杂度是 O(N 2)，相比其它排序算法，这是一个相对较高的时间复杂度，一般情况不推荐使用，由

于冒泡排序的简洁性，通常被用来对于程序设计入门的学生介绍算法的概念

12. 说说你对二分查找的理解？如何实现？应用场景？



12.1. 是什么

在计算机科学中，二分查找算法，也称折半搜索算法，是一种在有序数组中查找某一特定元素的搜索算法

想要应用二分查找法，则这一堆数应有如下特性：

- 存储在数组中
- 有序排序

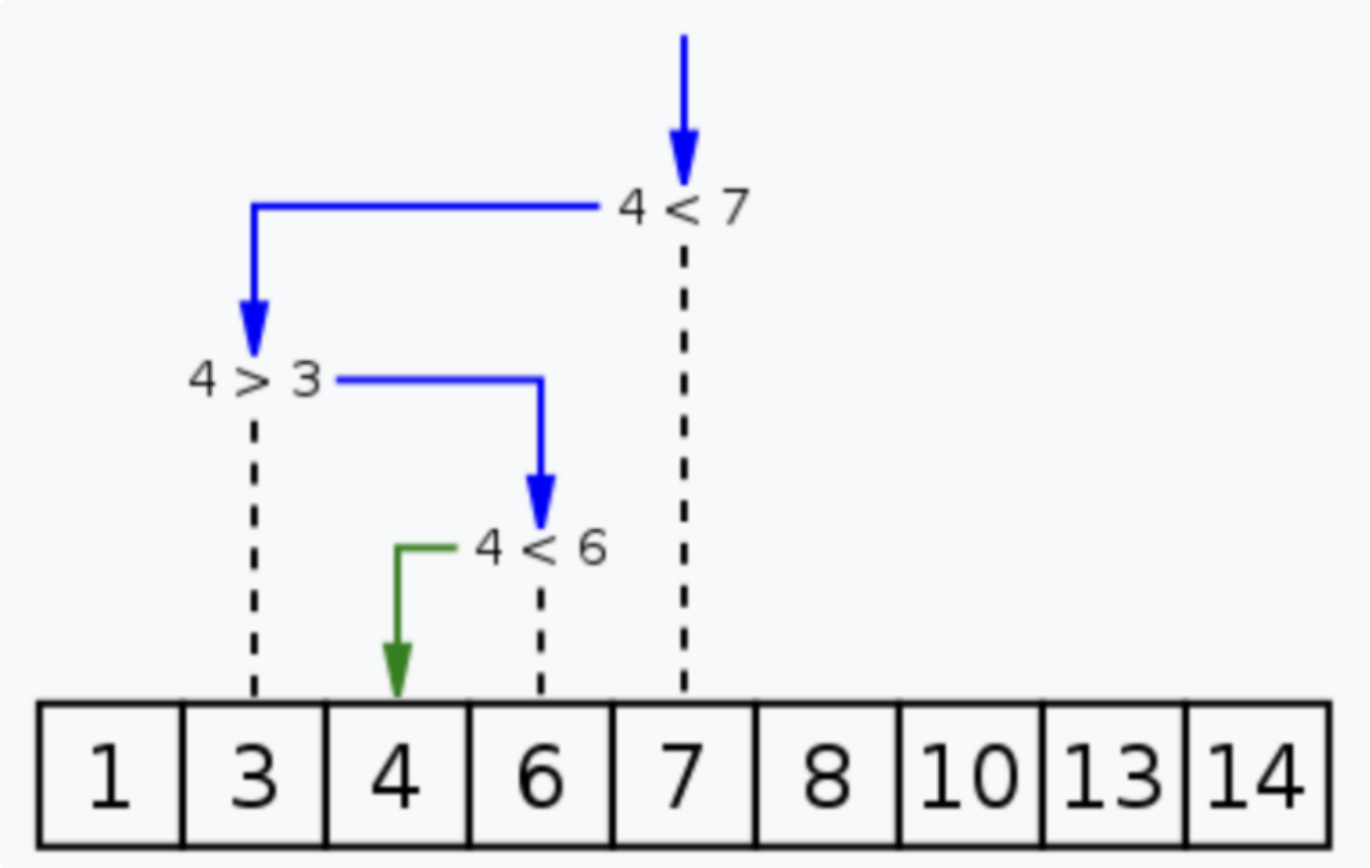
搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束

如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较

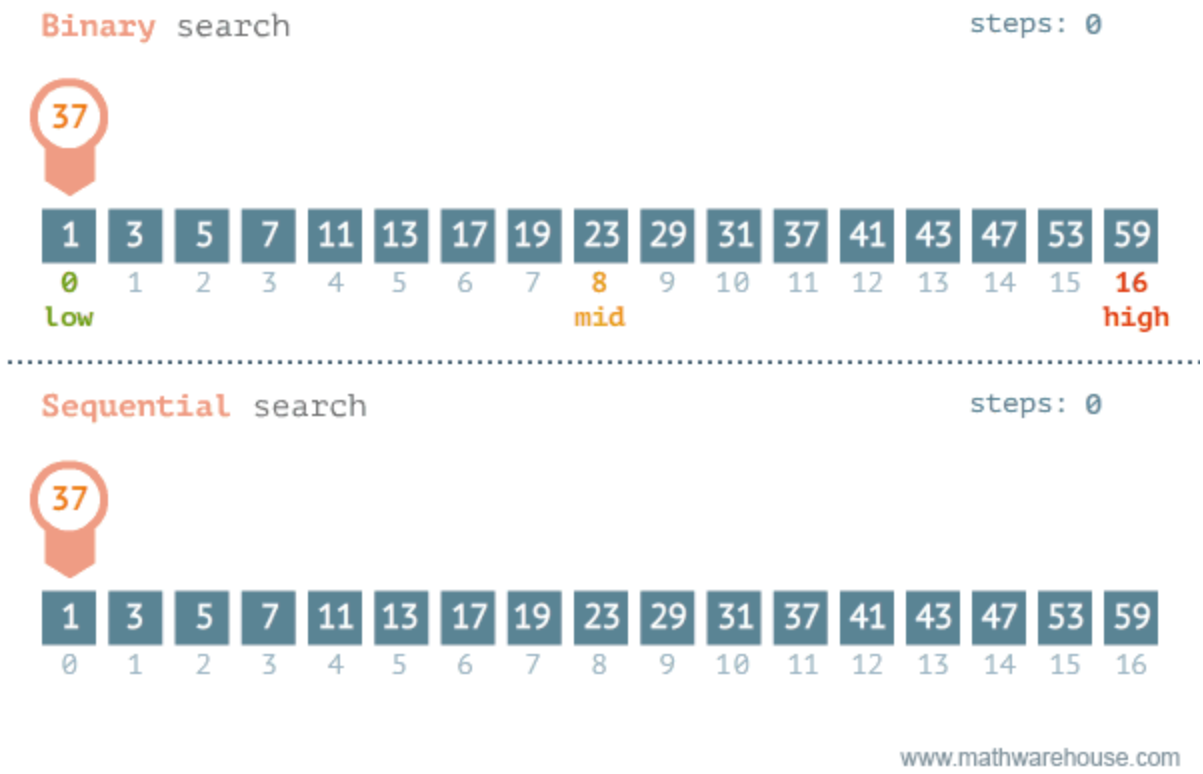
如果在某一步骤数组为空，则代表找不到

这种搜索算法每一次比较都使搜索范围缩小一半

如下图所示：



相比普通的顺序查找，除了数据量很少的情况下，二分查找会比顺序查找更快，区别如下所示：



12.2. 如何实现

基于二分查找的实现，如果数据是有序的，并且不存在重复项，实现代码如下：

JavaScript | 复制代码

```javascript
function BinarySearch(arr, target) {
    if (arr.length <= 1) return -1
    // 低位下标
    let lowIndex = 0
    // 高位下标
    let highIndex = arr.length - 1

    while (lowIndex <= highIndex) {
        // 中间下标
        const midIndex = Math.floor((lowIndex + highIndex) / 2)
        if (target < arr[midIndex]) {
            highIndex = midIndex - 1
        } else if (target > arr[midIndex]) {
            lowIndex = midIndex + 1
        } else {
            // target === arr[midIndex]
            return midIndex
        }
    }
    return -1
}
```

如果数组中存在重复项，而我们需要找出第一个制定的值，实现则如下：

JavaScript | 复制代码

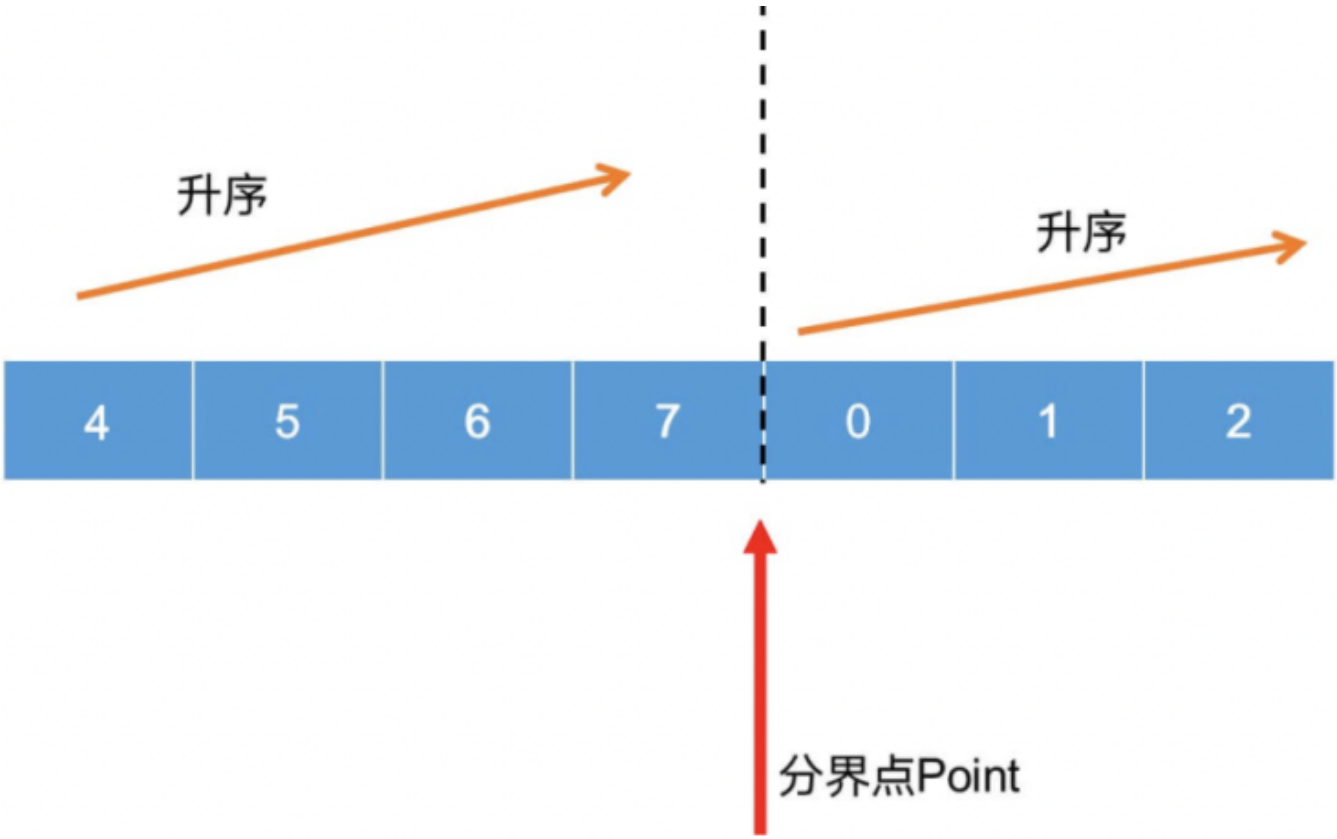
```javascript
function BinarySearchFirst(arr, target) {
    if (arr.length <= 1) return -1
    // 低位下标
    let lowIndex = 0
    // 高位下标
    let highIndex = arr.length - 1

    while (lowIndex <= highIndex) {
        // 中间下标
        const midIndex = Math.floor((lowIndex + highIndex) / 2)
        if (target < arr[midIndex]) {
            highIndex = midIndex - 1
        } else if (target > arr[midIndex]) {
            lowIndex = midIndex + 1
        } else {
            // 当 target 与 arr[midIndex] 相等的时候，如果 midIndex 为0或者前一个数比 target 小那么就找到了第一个等于给定值的元素，直接返回
            if (midIndex === 0 || arr[midIndex - 1] < target) return midIndex
            // 否则高位下标为中间下标减1，继续查找
            highIndex = midIndex - 1
        }
    }
    return -1
}
```

实际上，除了有序的数组可以使用，还有一种特殊的数组可以应用，那就是轮转后的有序数组

有序数组即一个有序数字以某一个数为轴，将其之前的所有数都轮转到数组的末尾所得

例如，[4, 5, 6, 7, 0, 1, 2]就是一个轮转后的有序数组

该数组的特性是存在一个分界点用来分界两个有序数组，如下：



分界点有如下特性：

- 分界点元素 >= 第一个元素
- 分界点元素 < 第一个元素

代码实现如下：

JavaScript | 复制代码

```javascript
function search (nums, target) {
  // 如果为空或者是空数组的情况
  if (nums == null || !nums.length) {
    return -1;
  }
  // 搜索区间是前闭后闭
  let begin = 0,
    end = nums.length - 1;
  while (begin <= end) {
    // 下面这样写是考虑大数情况下避免溢出
    let mid = begin + ((end - begin) >> 1);
    if (nums[mid] == target) {
      return mid;
    }
    // 如果左边是有序的
    if (nums[begin] <= nums[mid]) {
      //同时target在[ nums[begin],nums[mid] ]中，那么就在这段有序区间查找
      if (nums[begin] <= target && target <= nums[mid]) {
        end = mid - 1;
      } else {
        //否则去反方向查找
        begin = mid + 1;
      }
      //如果右侧是有序的
    } else {
      //同时target在[ nums[mid],nums[end] ]中，那么就在这段有序区间查找
      if (nums[mid] <= target && target <= nums[end]) {
        begin = mid + 1;
      } else {
        end = mid - 1;
      }
    }
  }
  return -1;
};
```

对比普通的二分查找法，为了确定目标数会落在二分后的哪个部分，我们需要更多的判定条件

12.3. 应用场景

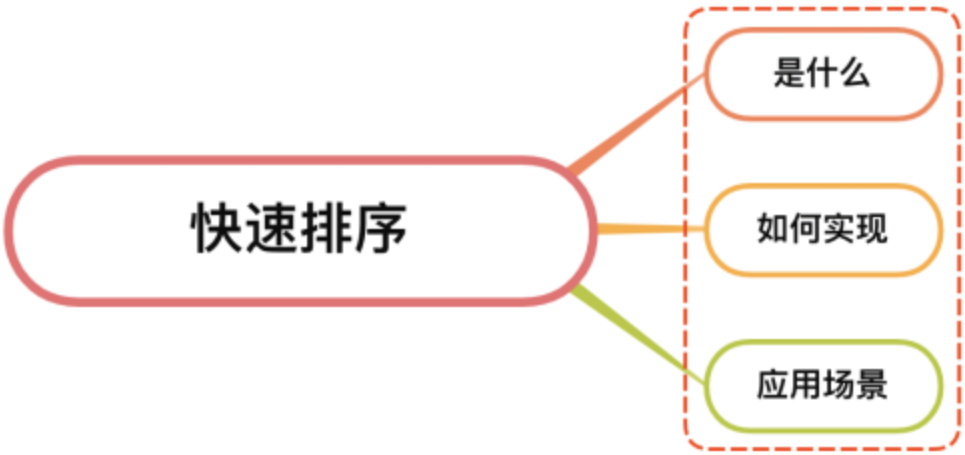
二分查找法的 `O(logn)` 让它成为十分高效的算法。不过它的缺陷却也是比较明显，就在它的限定之上：

- 有序：我们很难保证我们的数组都是有序的
- 数组：数组读取效率是O(1)，可是它的插入和删除某个元素的效率却是O(n)，并且数组的存储是需要连续的内存空间，不适合大数据的情况

关于二分查找的应用场景，主要如下：

- 不适合数据量太小的数列；数列太小，直接顺序遍历说不定更快，也更简单
- 每次元素与元素的比较是比较耗时的，这个比较操作耗时占整个遍历算法时间的大部分，那么使用二分查找就能有效减少元素比较的次数
- 不适合数据量太大的数列，二分查找作用的数据结构是顺序表，也就是数组，数组是需要连续的内存空间的，系统并不一定有这么大的连续内存空间可以使用

13. 说说你对快速排序的理解？如何实现？应用场景？



13.1. 是什么

快速排序（Quick Sort）算法是在冒泡排序的基础上进行改进的一种算法，从名字上看就知道该排序算法的特点是快、效率高，是处理大数据最快的排序算法之一

实现的基本思想是：通过一次排序将整个无序表分成相互独立的两部分，其中一部分中的数据都比另一部分中包含的数据的值小

然后继续沿用此方法分别对两部分进行同样的操作，直到每一个小部分不可再分，所得到的整个序列就变成有序序列

例如，对无序表49，38，65，97，76，13，27，49进行快速排序，大致过程为：

- 首先从表中选取一个记录的关键字作为分割点（称为“枢轴”或者支点，一般选择第一个关键字），例如选取 49

- 将表格中大于 49 个放置于 49 的右侧，小于 49 的放置于 49 的左侧，假设完成后的无序表为：{27，38，13，49，65，97，76，49}
- 以 49 为支点，将整个无序表分割成了两个部分，分别为{27，38，13}和{65，97，76，49}，继续采用此种方法分别对两个子表进行排序
- 前部分子表以 27 为支点，排序后的子表为{13，27，38}，此部分已经有序；后部分子表以 65 为支点，排序后的子表为{49，65，97，76}
- 此时前半部分子表中的数据已完成排序；后部分子表继续以 65 为支点，将其分割为{49}和{97，76}，前者不需排序，后者排序后的结果为{76， 97}
- 通过以上几步的排序，最后由子表{13，27，38}、{49}、{49}、{65}、{76，97}构成有序表：{13，27，38，49，49，65，76，97}

13.2. 如何实现

可以分成以下步骤：

- 分区：从数组中选择任意一个基准，所有比基准小的元素放在基准的左边，比基准大的元素放到基准的右边
- 递归：递归地对基准前后的子数组进行分区

用代码表示则如下：

JavaScript | 复制代码

```javascript
function quickSort (arr) {
  const rec = (arr) => {
    if (arr.length <= 1) { return arr; }
    const left = [];
    const right = [];
    const mid = arr[0]; // 基准元素
    for (let i = 1; i < arr.length; i++){
      if (arr[i] < mid) {
        left.push(arr[i]);
      } else {
        right.push(arr[i]);
      }
    }
    return [...rec(left), mid, ...rec(right)]
  }
  return res(arr)
};
```

快速排序是冒泡排序的升级版，最坏情况下每一次基准元素都是数组中最小或者最大的元素，则快速排序就是冒泡排序

这种情况时间复杂度就是冒泡排序的时间复杂度：`T[n] = n * (n-1) = n^2 + n`，也就是`O(n^2)`

最好情况下是`O(nlogn)`，其中递归算法的时间复杂度公式：`T[n] = aT[n/b] + f(n)`，推导如下所示：

$$T(n)=2T(\frac{n}{2})+n$$

$$T(n)=2(2T(\frac{n}{4})+\frac{n}{2})+n=4T(\frac{n}{4})+2n$$

$$T(n)=4(2T(\frac{n}{8})+\frac{n}{4})+2n=8T(\frac{n}{8})+3n$$

……

且 $T(1)=0$

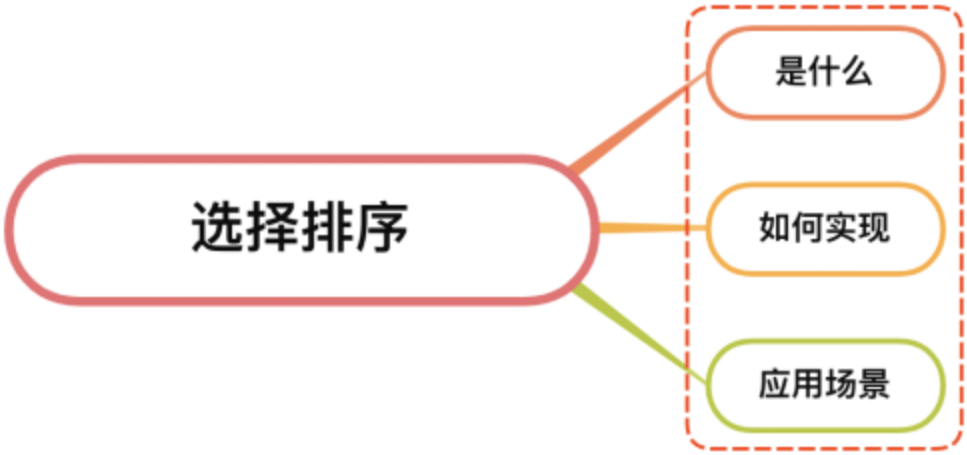
$$T(n)=nT(1)+(\log(n))\times n=O(n\times\log(n))$$

关于上述代码实现的快速排序，可以看到是稳定的

13.3. 应用场景

快速排序时间复杂度为 `O(nlogn)` ，是目前基于比较的内部排序中被认为最好的方法，当数据过大且数据杂乱无章时，则适合采用快速排序

14. 说说你对选择排序的理解？如何实现？应用场景？



14.1. 是什么

选择排序（Selection sort）是一种简单直观的排序算法，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

其基本思想是：首先在未排序的数列中找到最小(or最大)元素，然后将其存放到数列的起始位置

然后再从剩余未排序的元素中继续寻找最小(or最大)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

举个例子，一个数组为 56、12、80、91、29，其排序过程如下：

- 第一次遍历时，从下标为 1 的位置即 56 开始，找出关键字值最小的记录 12，同下标为 0 的关键字 56 交换位置。此时数组为 12、56、80、91、20

12	56	80	91	20

- 第二次遍历时，从下标为 2 的位置即 56 开始，找出最小值 20，同下标为 2 的关键字 56 互换位置，此时数组为12、20、80、91、56

12	20	80	91	56

- 第三次遍历时，从下标为 3 的位置即 80 开始，找出最小值 56，同下标为 3 的关键字 80 互换位置，此时数组为 12、20、56、91、80

12	20	56	91	80

- 第四次遍历时，从下标为 4 的位置即 91 开始，找出最小是 80，同下标为 4 的关键字 91 互换位置，此时排序完成，变成有序数组

12	20	56	91	80

14.2. 如何实现

从上面可以看到，对于具有 `n` 个记录的无序表遍历 `n-1` 次，第 `i` 次从无序表中第 `i` 个记录开始，找出后序关键字中最小的记录，然后放置在第 `i` 的位置上

直至到从第 `n` 个和第 `n-1` 个元素中选出最小的放在第 `n-1` 个位置

如下动画所示：

用代码表示则如下：

```javascript
function selectionSort(arr) {
    var len = arr.length;
    var minIndex, temp;
    for (var i = 0; i < len - 1; i++) {
        minIndex = i;
        for (var j = i + 1; j < len; j++) {
            if (arr[j] < arr[minIndex]) {     // 寻找最小的数
                minIndex = j;                 // 将最小数的索引保存
            }
        }
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
    return arr;
}
```

第一次内循环比较 `N - 1` 次，然后是 `N-2` 次， `N-3` 次，……，最后一次内循环比较1次

共比较的次数是 `(N - 1) + (N - 2) + ... + 1` ，求等差数列和，得 `(N - 1 + 1)* N / 2 = N^2 / 2` ，舍去最高项系数，其时间复杂度为 `O(N^2)`

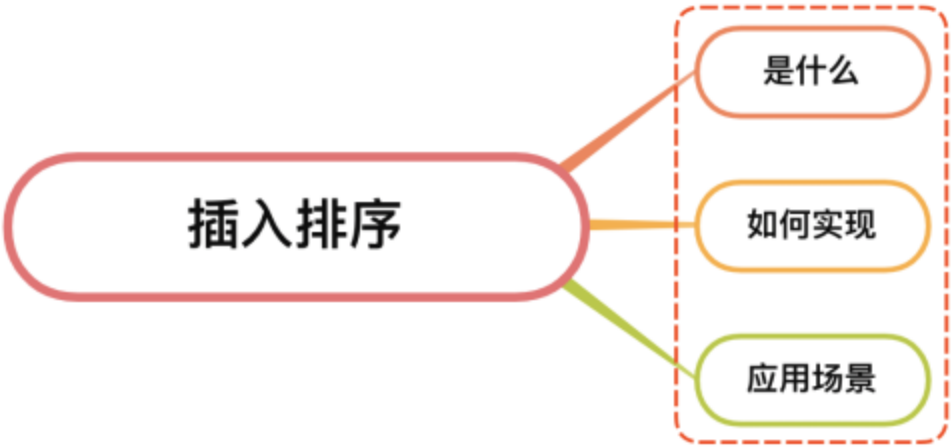
从上述也可以看到，选择排序是一种稳定的排序

14.3. 应用场景

和冒泡排序一致，相比其它排序算法，这也是一个相对较高的时间复杂度，一般情况不推荐使用

但是我们还是要掌握冒泡排序的思想及实现，这对于我们的算法思维是有很大帮助的

15. 说说你对插入排序的理解？如何实现？应用场景？



15.1. 是什么

插入排序（Insertion Sort），一般也被称为直接插入排序。对于少量元素的排序，它是一个有效、简单的算法

其主要的实现思想是将数据按照一定的顺序一个一个的插入到有序的表中，最终得到的序列就是已经排序好的数据

插入排序的工作方式像许多人排序一手扑克牌，开始时，我们的左手为空并且桌子上的牌面向下

然后，我们每次从桌子上拿走一张牌并将它插入左手中正确的位置，该正确位置需要从右到左将它与已在手中的每张牌进行比较

例如一个无序数组 3、1、7、5、2、4、9、6，将其升序的结果则如下：

一开始有序表中无数据，直接插入3

从第二个数开始，插入一个元素1，然后和有序表中记录3比较，1<3，所以插入到记录 3 的左侧

向有序表插入记录 7 时，同有序表中记录 3 进行比较，3<7，所以插入到记录 3 的右侧

向有序表中插入记录 5 时，同有序表中记录 7 进行比较，5<7，同时 5>3，所以插入到 3 和 7 中间

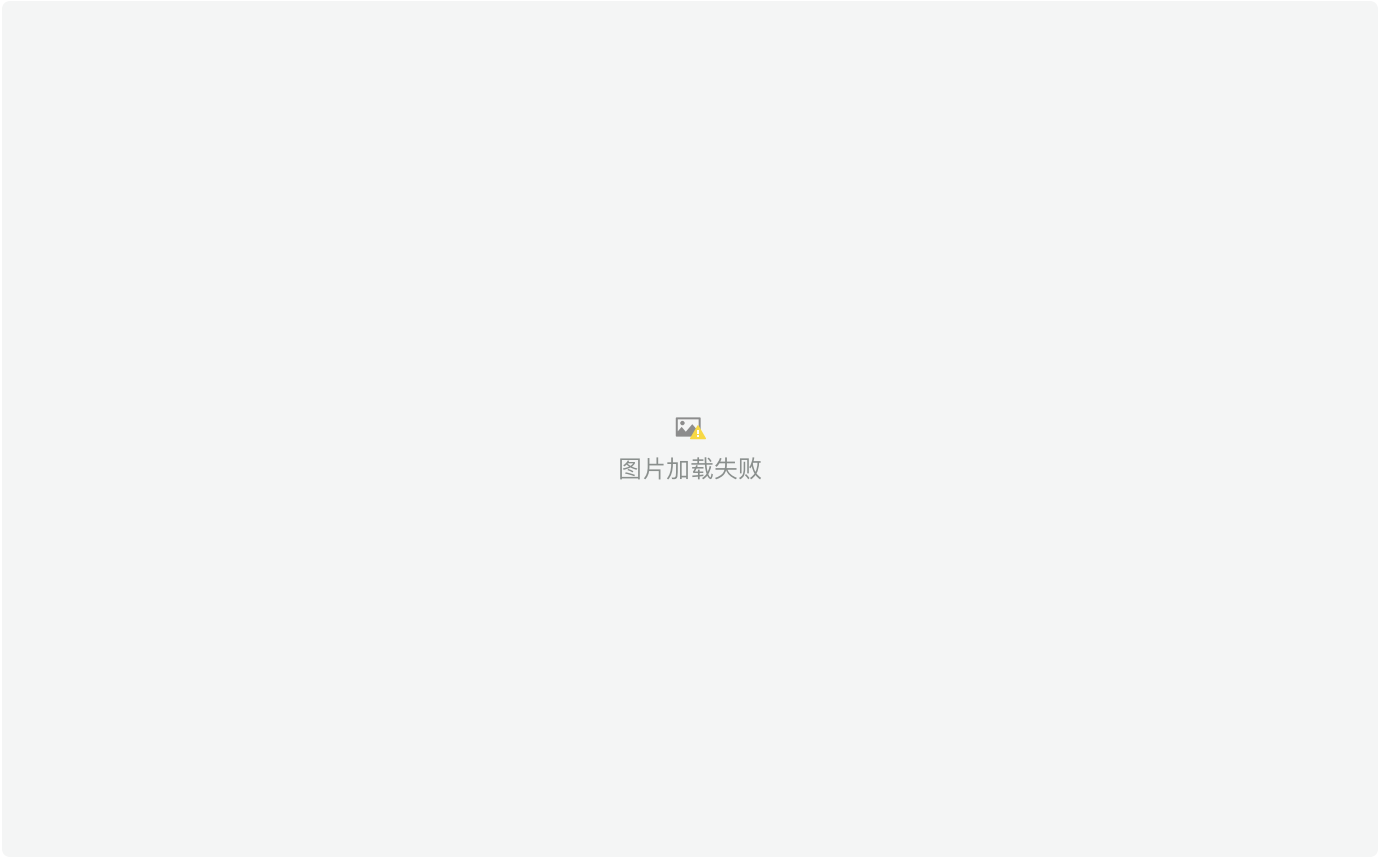
照此规律，依次将无序表中的记录 4，9 和 6插入到有序表中

15.2. 如何实现

将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。

从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置

如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面



用代码表示则如下：

JavaScript | 复制代码

```javascript
function insertionSort(arr) {
    const len = arr.length;
    let preIndex, current;
    for (let i = 1; i < len; i++) {
        preIndex = i - 1;
        current = arr[i];
        while(preIndex >= 0 && arr[preIndex] > current) {
            arr[preIndex+1] = arr[preIndex];
            preIndex--;
        }
        arr[preIndex+1] = current;
    }
    return arr;
}
```

在插入排序中，当待排序数组是有序时，是最优的情况，只需当前数跟前一个数比较一下就可以了，这时一共需要比较 `N- 1` 次，时间复杂度为 `O(n)`

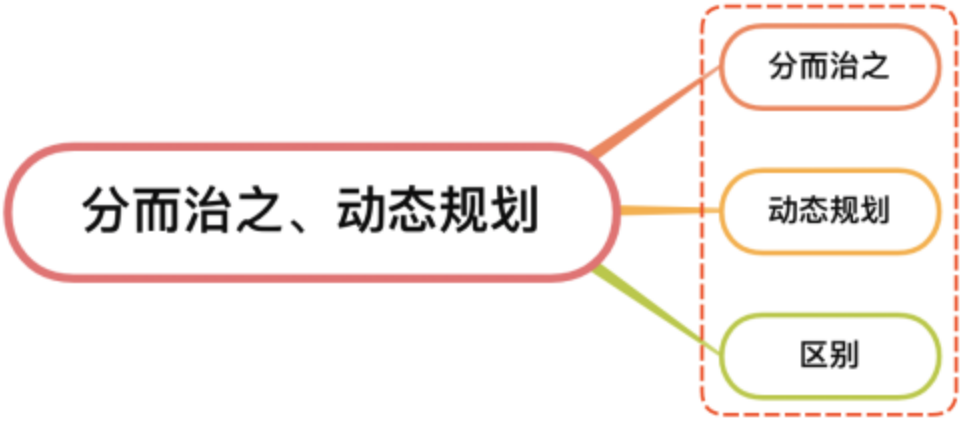
最坏的情况是待排序数组是逆序的，此时需要比较次数最多，总次数记为：1+2+3+…+N-1，所以，插入排序最坏情况下的时间复杂度为 `O(n^2)`

通过上面了解，可以看到插入排序是一种稳定的排序方式

15.3. 应用场景

插入排序时间复杂度是 O(n2)，适用于数据量不大，算法稳定性要求高，且数据局部或整体有序的数列排序

16. 说说你对分而治之、动态规划的理解？区别？



16.1. 分而治之

分而治之是算法设计中的一种方法，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并

关于分而治之的实现，都会经历三个步骤：

- 分解：将原问题分解为若干个规模较小，相对独立，与原问题形式相同的子问题
- 解决：若子问题规模较小且易于解决时，则直接解。否则，递归地解决各子问题
- 合并：将各子问题的解合并为原问题的解

实际上，关于分而治之的思想，我们在前面已经使用，例如归并排序的实现，同样经历了实现分而治之的三个步骤：

- 分解：把数组从中间一分为二
- 解决：递归地对两个子数组进行归并排序
- 合并：将两个字数组合并称有序数组

同样关于快速排序的实现，亦如此：

- 分：选基准，按基准把数组分成两个字数组
- 解：递归地对两个字数组进行快速排序
- 合：对两个字数组进行合并

同样二分搜索也能使用分而治之的思想去实现，代码如下：

JavaScript | 复制代码

```
function binarySearch(arr,l,r,target){
    if(l> r){
        return -1;
    }
    let mid = l + Math.floor((r-l)/2)
    if(arr[mid] === target){
        return mid;
    }else if(arr[mid] < target ){
        return binarySearch(arr,mid + 1,r,target)
    }else{
        return binarySearch(arr,l,mid - 1,target)
    }
}
```

16.2. 动态规划

动态规划，同样是算法设计中的一种方法，是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法

常常适用于有重叠子问题和最优子结构性质的问题

简单来说，动态规划其实就是，给定一个问题，我们把它拆成一个个子问题，直到子问题可以直接解决

然后呢，把子问题答案保存起来，以减少重复计算。再根据子问题答案反推，得出原问题解的一种方法。

一般这些子问题很相似，可以通过函数关系式递推出来，例如斐波那契数列，我们可以得到公式：当 n 大于 2的时候，F(n) = F(n−1) + F(n−2) ，

f(10)= f(9)+f(8),f(9) = f(8) + f(7)...是重叠子问题，当n = 1、2的时候，对应的值为2，这时候就通过可以使用一个数组记录每一步计算的结果，以此类推，减少不必要的重复计算

16.2.1. 适用场景

如果一个问题，可以把所有可能的答案穷举出来，并且穷举出来后，发现存在重叠子问题，就可以考虑使用动态规划

比如一些求最值的场景，如最长递增子序列、最小编辑距离、背包问题、凑零钱问题等等，都是动态规划的经典应用场景

关于动态规划题目解决的步骤，一般如下：

- 描述最优解的结构
- 递归定义最优解的值
- 按自底向上的方式计算最优解的值
- 由计算出的结果构造一个最优解

16.3. 区别

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解

与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往**不是互相独立**的，而分而治之的子问题是相互独立的

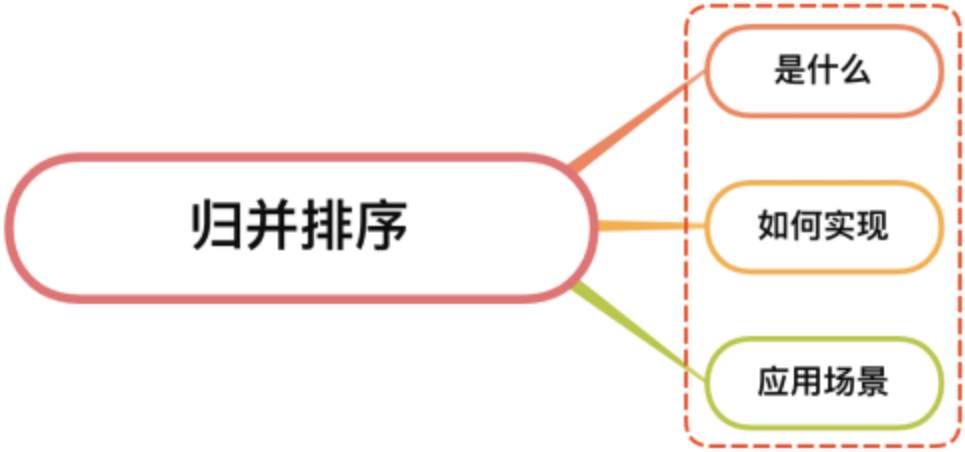
若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次

如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间

综上，可得：

- 动态规划：有最优子结构和重叠子问题
- 分而治之：各子问题独立

17. 说说你对归并排序的理解？如何实现？应用场景？



17.1. 是什么

归并排序（Merge Sort）是建立归并操作上的一种有效，稳定的排序算法，该算法是采用分治法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

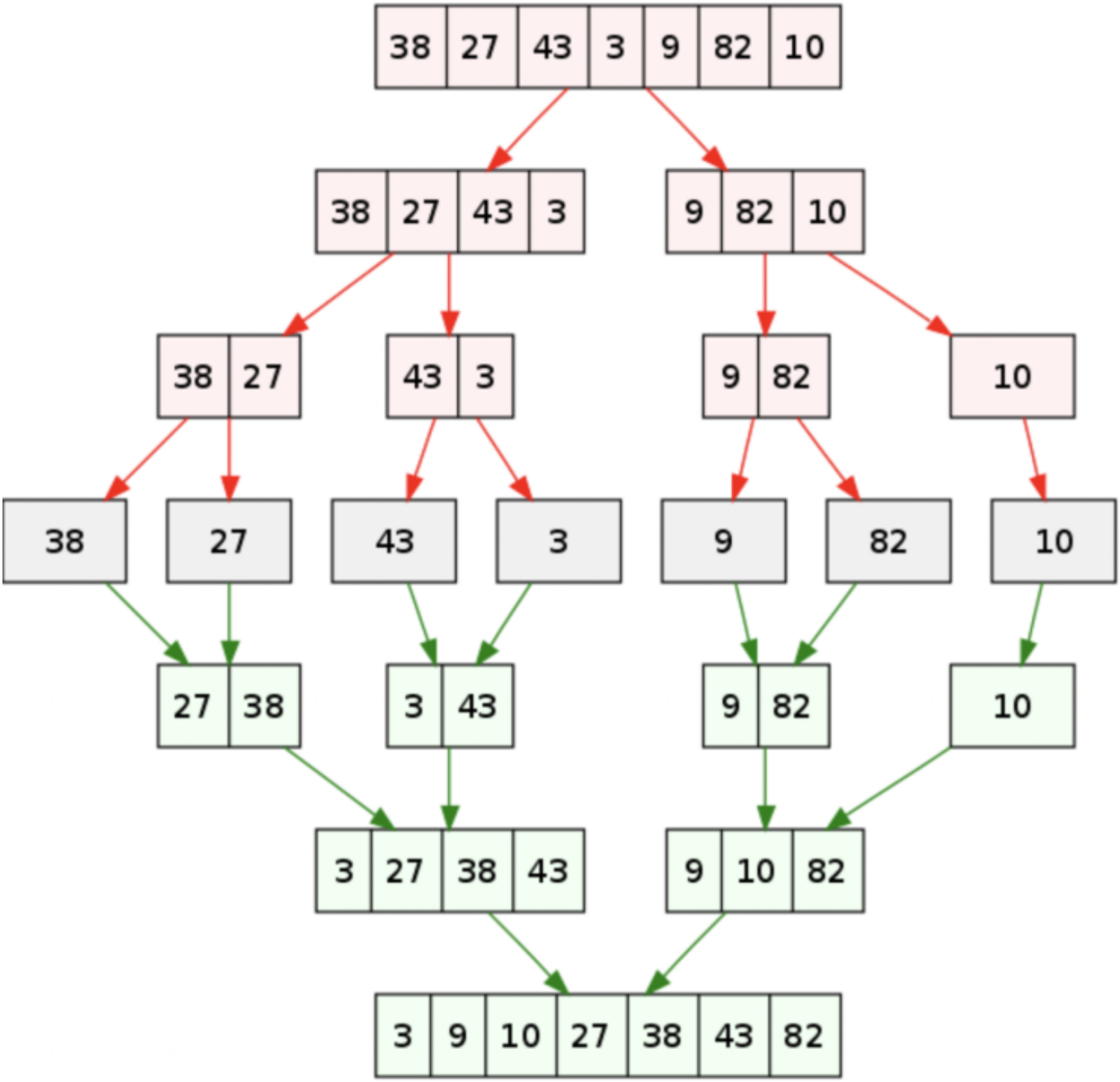
例如对于含有 `n` 个记录的无序表，首先默认表中每个记录各为一个有序表（只不过表的长度都为 1）

然后进行两两合并，使 `n` 个有序表变为 `n/2` 个长度为 2 或者 1 的有序表（例如 4 个小有序表合并为 2 个大的有序表）

通过不断地进行两两合并，直到得到一个长度为 `n` 的有序表为止

例如对无序表{49，38，65，97，76，13，27}进行归并排序分成了分、合两部分：

如下图所示：



归并合过程中，每次得到的新的子表本身有序，所以最终得到有序表

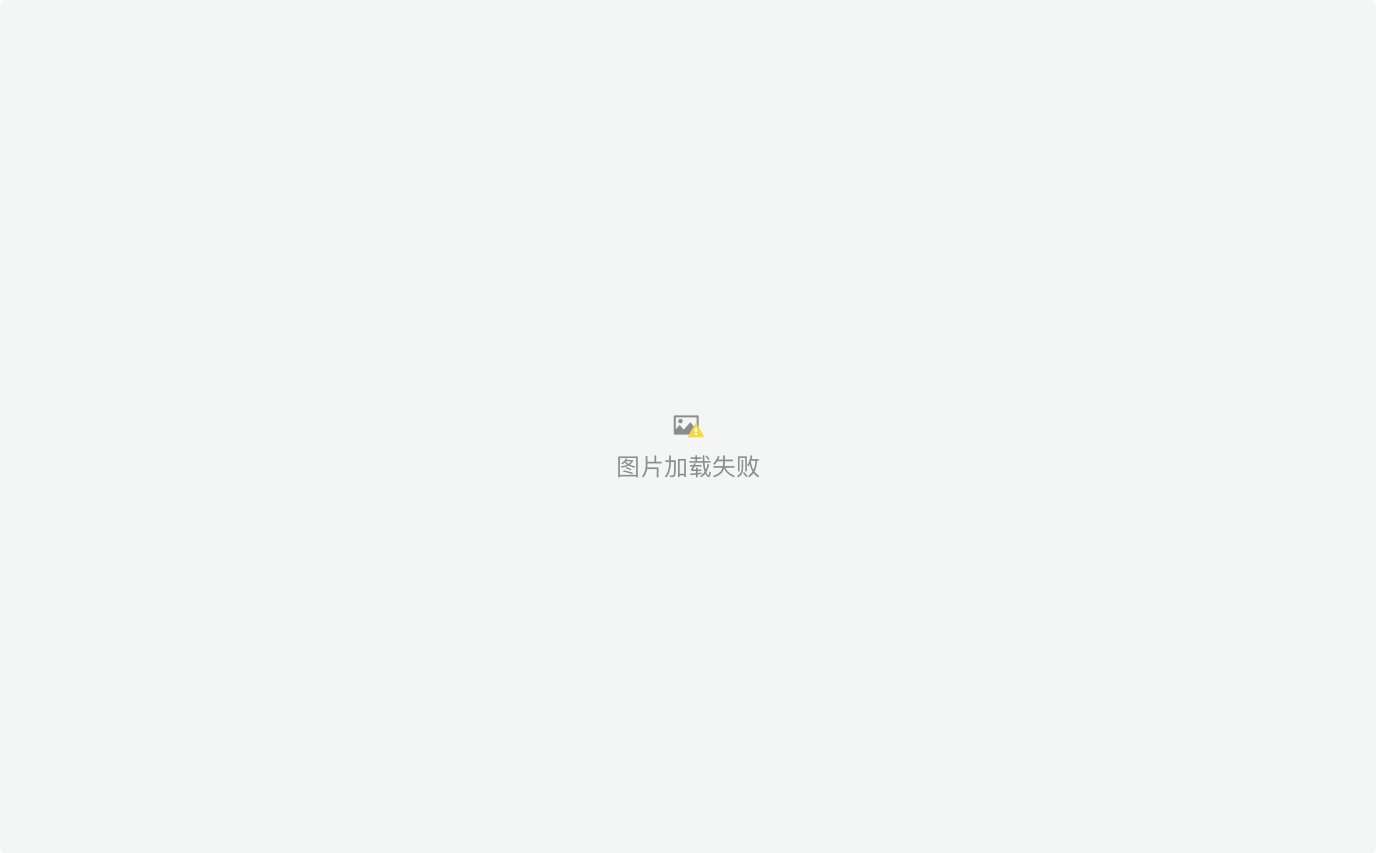
上述分成两部分，则称为二路归并，如果分成三个部分则称为三路归并，以此类推

17.2. 如何实现

关于归并排序的算法思路如下：

- 分：把数组分成两半，再递归对子数组进行分操作，直至到一个个单独数字
- 合：把两个数合成有序数组，再对有序数组进行合并操作，直到全部子数组合成一个完整的数组
  - 合并操作可以新建一个数组，用于存放排序后的数组

- 比较两个有序数组的头部，较小者出队并且推入到上述新建的数组中
- 如果两个数组还有值，则重复上述第二步
- 如果只有一个数组有值，则将该数组的值出队并推入到上述新建的数组中



用代码表示则如下图所示：