# 一、冒泡排序（Bubble Sort）

算法原理


算法稳定性

冒泡排序是一种稳定排序算法。

算法分析

时间复杂度

若文件的初始状态是正序的，一趟扫描即可完成排序。所需的关键字比较次数C和记录移动次数M均达到最小值：

$C_{\min} = n - 1$

$M_{\min} = 0$

所以，冒泡排序最好的时间复杂度为

$O(n)$

若初始文件是反序的，需要进行n-1趟排序。每趟排序要进行n-i次关键字的比较(1≤i≤n-1)，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较和移动次数均达到最大值：

$C_{\max} = \frac{n(n-1)}{2} = O(n^2)$

$M_{\max} = \frac{3n(n-1)}{2} = O(n^2)$

冒泡排序的最坏时间复杂度为

$O(n^2)$

综上，因此冒泡排序总的平均时间复杂度为

$O(n^2)$

算法描述

```
function bubbleSort(arr) {
        var i = arr.length, j;
        var tempExchangVal;
        while (i > 0) {
                for (j = 0; j < i - 1; j++) {
                        tempExchangVal = arr[j];
                        arr[j] = arr[j + 1];
                        arr[j + 1] = tempExchangVal;
                    }
                }
                i--;
    alert(arrSorted);
    console.log(arrSorted);
    var arrSorted = bubbleSort(arr);
    var arr = [3, 2, 4, 9, 1, 5, 7, 6, 8];
    }
    return arr;
}
```

或者

```
function bubbleSort(arr) {
    var len = arr.length;
    for (var i = 0; i < len; i++) {
        for (var 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;
}
```

# 二、选择排序（Selection Sort）

算法原理

现有无序数组[6 2 4 1 5 9]

第一趟找到最小数1,放到最前边(与首位数字交换)
交换后:| 1 | 2 | 4 | 6 | 5 | 9 |

第二趟找到余下数字[2 4 6 5 9]里的最小数2,与当前数组的首位数字进行交换,实际没有交换,本来就在首位
交换后:| 1 | 2 | 4 | 6 | 5 | 9 |

第三趟继续找到剩余[4 6 5 9]数字里的最小数4,实际没有交换,4待首位置无须交换

第四趟从剩余的[6 5 9]里找到最小数5,与首位数字6交换位置
交换后:| 1 | 2 | 4 | 5 | 6 | 9 |

第五趟从剩余的[6 9]里找到最小数6,发现它待在正确的位置,没有交换

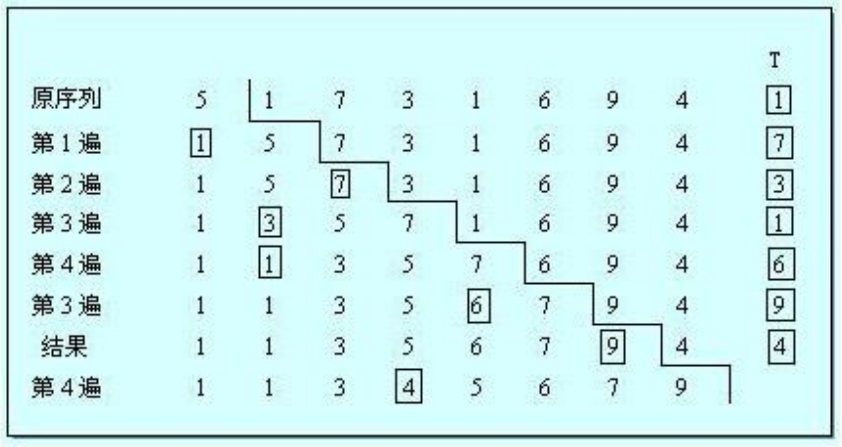
排序完毕输出正确结果[1 2 4 5 6 9]

**算法描述**

```
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;
}
```

# 三、插入排序（Insertion Sort)

**算法原理**



| | | | | | | | | | T |
|---|---|---|---|---|---|---|---|---|---|
| 原序列 | 5 | 1 | 7 | 3 | 1 | 6 | 9 | 4 | 1 |
| 第1遍 | 1 | 5 | 7 | 3 | 1 | 6 | 9 | 4 | 7 |
| 第2遍 | 1 | 5 | 7 | 3 | 1 | 6 | 9 | 4 | 3 |
| 第3遍 | 1 | 3 | 5 | 7 | 1 | 6 | 9 | 4 | 1 |
| 第4遍 | 1 | 1 | 3 | 5 | 7 | 6 | 9 | 4 | 6 |
| 第3遍 | 1 | 1 | 3 | 5 | 6 | 7 | 9 | 4 | 9 |
| 结果 | 1 | 1 | 3 | 5 | 6 | 7 | 9 | 4 | 4 |
| 第4遍 | 1 | 1 | 3 | 4 | 5 | 6 | 7 | 9 | |

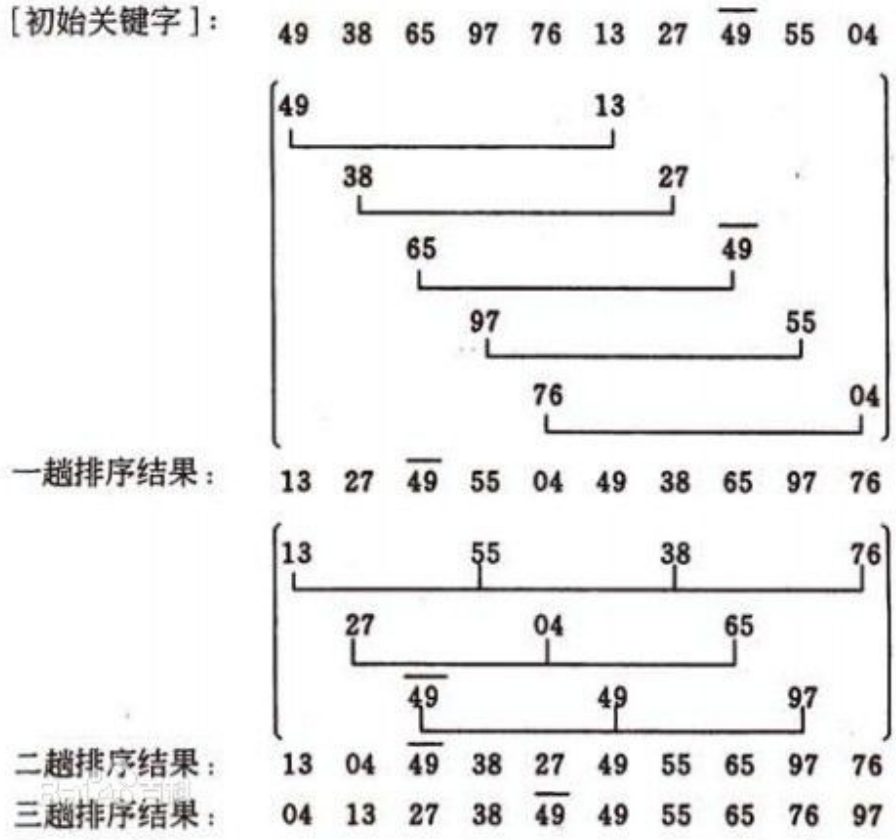


算法描述

```
function insertionSort(arr) {
    var len = arr.length;
    var preIndex, current;
    for (var 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;
}
```

# 四、希尔排序（Shell Sort）

算法原理

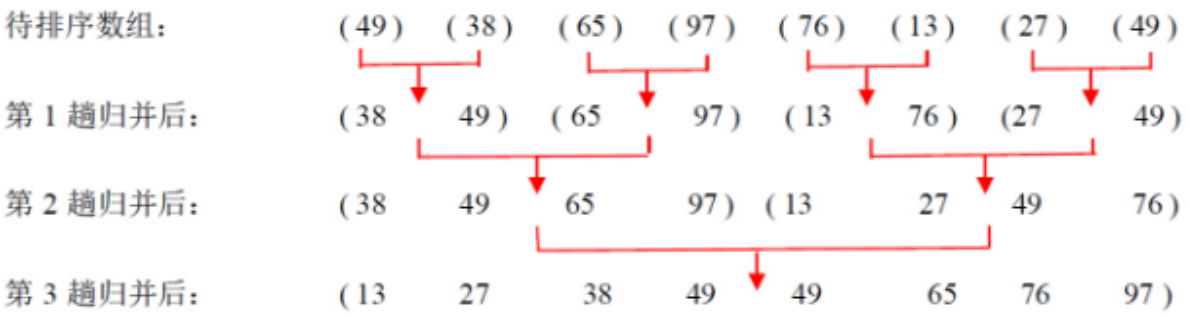



算法描述

```
function shellSort(arr) {
    var len = arr.length,
        temp,
        gap = 1;
    while(gap < len/3) {          //动态定义间隔序列
        gap =gap*3+1;
    }
    for (gap; gap> 0; gap = Math.floor(gap/3)) {
        for (var i = gap; i < len; i++) {
            temp = arr[i];
            for (var j = i-gap; j > 0 && arr[j]> temp; j-=gap) {
                arr[j+gap] = arr[j];
            }
            arr[j+gap] = temp;
        }
    }
    return arr;
}
```

# 五、归并排序（Merge Sort）

算法原理

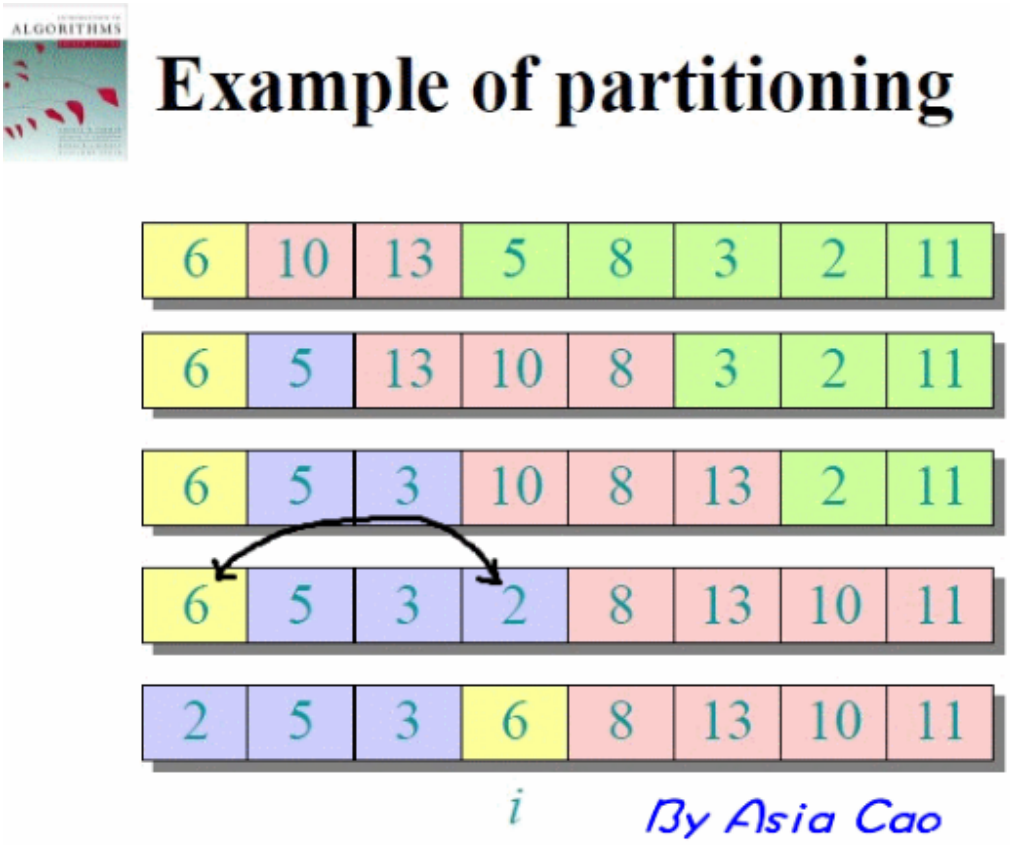



算法描述

```
function mergeSort(arr) {  //采用自上而下的递归方法
    var len = arr.length;
    if(len < 2) {
        return arr;
    }
    var middle = Math.floor(len / 2),
        left = arr.slice(0, middle),
        right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right)
{
    var result = [];

    while (left.length>0 && right.length>0) {
        if (left[0] <= right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }

    while (left.length)
        result.push(left.shift());

    while (right.length)
        result.push(right.shift());

    return result;
}
```

# ? 六、快速排序（Quick Sort）

算法原理




算法描述

```
function quickSort(arr, left, right) {
    var len = arr.length,
        partitionIndex,
        left = typeof left != 'number' ? 0 : left,
        right = typeof right != 'number' ? len - 1 : right;

    if (left < right) {
        partitionIndex = partition(arr, left, right);
        quickSort(arr, left, partitionIndex-1);
        quickSort(arr, partitionIndex+1, right);
    }
    return arr;
}

function partition(arr, left ,right) {      //分区操作
    var pivot = left,                       //设定基准值（pivot）
        index = pivot + 1;
    for (var i = index; i <= right; i++) {
        if (arr[i] < arr[pivot]) {
            swap(arr, i, index);
            index++;
        }
    }
    swap(arr, pivot, index - 1);
    return index-1;
}
```

```
function swap(arr, i, j) {
    var temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

# ? 七、堆排序（Heap Sort)

## 算法原理

堆排序就是把堆顶的最大数取出

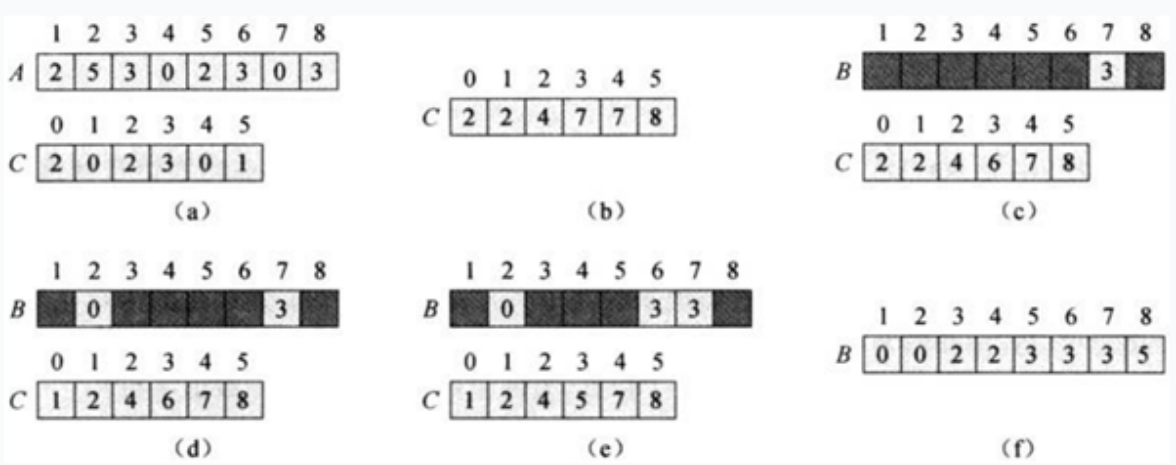
## 算法描述

```
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

function buildMaxHeap(arr) {    //建立大顶堆
    len = arr.length;
    for (var i = Math.floor(len/2); i &gt;= 0; i--) {
        heapify(arr, i);
    }
}

function heapify(arr, i) {      //堆调整
    var left = 2 * i + 1,
        right = 2 * i + 2,
        largest = i;

    if (left < len && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < len && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr, i, largest);
        heapify(arr, largest);
    }
}

function swap(arr, i, j) {
    var temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

function heapSort(arr) {
    buildMaxHeap(arr);

    for (var i = arr.length-1; i > 0; i--) {
        swap(arr, 0, i);
        len--;
```

```
        heapify(arr, 0);
    }
    return arr;
}
```

# ? 八、计数排序（Counting Sort）

算法原理（）




算法描述

```
function countingSort(arr, maxValue) {
    var bucket = new Array(maxValue+1),
        sortedIndex = 0;
        arrLen = arr.length,
        bucketLen = maxValue + 1;

    for (var i = 0; i < arrLen; i++) {
        if (!bucket[arr[i]]) {
            bucket[arr[i]] = 0;
        }
        bucket[arr[i]]++;
    }

    for (var j = 0; j < bucketLen; j++) {
        while(bucket[j] > 0) {
            arr[sortedIndex++] = j;
            bucket[j]--;
        }
    }

    return arr;
}
```

# 九、桶排序（Bucket Sort）

算法原理

和基数排序（Radix Sort）原理类似

算法描述

```
function bucketSort(arr, bucketSize) {
    if (arr.length === 0) {
```

```
        return arr;
    }

    var i;
    var minValue = arr[0];
    var maxValue = arr[0];
    for (i = 1; i < arr.length; i++) {
      if (arr[i] < minValue) {
          minValue = arr[i];                  //输入数据的最小值
      } else if (arr[i] > maxValue) {
          maxValue = arr[i];                  //输入数据的最大值
      }
    }

    //桶的初始化
    var DEFAULT_BUCKET_SIZE = 5;              //设置桶的默认数量为5
    bucketSize = bucketSize || DEFAULT_BUCKET_SIZE;
    var bucketCount = Math.floor((maxValue - minValue) / bucketSize) + 1;
    var buckets = new Array(bucketCount);
    for (i = 0; i < buckets.length; i++) {
        buckets[i] = [];
    }

    //利用映射函数将数据分配到各个桶中
    for (i = 0; i < arr.length; i++) {
        buckets[Math.floor((arr[i] - minValue) / bucketSize)].push(arr[i]);
    }

    arr.length = 0;
    for (i = 0; i < buckets.length; i++) {
        insertionSort(buckets[i]);                      //对每个桶进行排序，这里使
用了插入排序
        for (var j = 0; j < buckets[i].length; j++) {
            arr.push(buckets[i][j]);
        }
    }

    return arr;
}
```

# 十、基数排序（Radix Sort）

算法原理

待排序数组[62, 14, 59, 88, 16]

分配10个桶, 桶编号为0-9, 以个位数数字为桶编号依次入桶, 变成下边这样

| 0 | 0 | 62 | 0 | 14 | 0 | 16 | 0 | 88 | 59 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

|桶编号

将桶里的数字顺序取出来,

输出结果:[62, 14, 16, 88, 59]

再次入桶,不过这次以十位数的数字为准,进入相应的桶,变成下边这样:
由于前边做了个位数的排序,所以当十位数相等时,个位数字是由小到大的顺序入桶的,就是说,入完桶还是有序

| 0 | 14,16 | 0 | 0 | 0 | 59 | 62 | 0 | 88 | 0 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 桶编号 |

因为没有大过100的数字,没有百位数,所以到这排序完毕,顺序取出即可
最后输出结果:[14,16,59,62,88]

**算法描述**

```
//LSD Radix Sort
var counter = [];
function radixSort(arr, maxDigit) {
    var mod = 10;
    var dev = 1;
    for (var i = 0; i < maxDigit; i++, dev *= 10, mod *= 10) {
        for(var j = 0; j < arr.length; j++) {
            var bucket = parseInt((arr[j] % mod) / dev);
            if(counter[bucket]==null) {
                counter[bucket] = [];
            }
            counter[bucket].push(arr[j]);
        }
        var pos = 0;
        for(var j = 0; j < counter.length; j++) {
            var value = null;
            if(counter[j]!=null) {
                while ((value = counter[j].shift()) != null) {
                      arr[pos++] = value;
                }
          }
        }
    }
    return arr;
}
```