插入排序(On2)

```
function insertSort(arr) {
  const n = arr.length;
  for (let i = 1; i < n; i++) {
    let j = i - 1;
    let tmp = arr[j + 1];
    while (j >= 0 && tmp < arr[j]) {
      // 将新加入的元素放入到合适位置，找到第一个小于等于它的元素放它后面
      arr[j + 1] = arr[j--];
    }
    arr[j + 1] = tmp;
  }
  return arr;
}
```

选择排序(On2)

```
function selectSort(arr) {
  const n = arr.length;
  for (let i = 0; i < n; i++) {
    let pos = i;
    for (let j = i; j < n; j++) {
      // 找到最小的元素
      if (arr[pos] > arr[j]) {
        pos = j;
      }
    }
    if (i !== pos) {
      // 交换位置
      swap(arr, i, pos);
    }
  }
  return arr;
}
```

冒泡排序(On2)

```
function propagationSort(arr) {
  const n = arr.length;
  for (let i = 0; i < n; i++) {
    for (let j = 1; j < n - i; j++) {
      if (arr[j] < arr[j - 1]) {
        // 前面的大于后面的交换位置，每次确定一位最大数
        swap(arr, j, j - 1);
      }
    }
  }
  return arr;
}
```

谢尔排序(On1.3)

```
function shellSort(arr) {
  console.log(arr);
  const n = arr.length;
  let gap = Math.floor(n / 2);
  while (gap > 0) {
    for (let i = gap; i < n; i++) {
      let tmp = arr[i];
      let j = i - gap;
      while (j >= 0 && arr[j] > tmp) {
        // 左边的元素大于右边的元素，让右边的元素变为左边的元
素，一直变将更大的元素换到右边
        arr[j + gap] = arr[j];
        j -= gap;
      }
      arr[j + gap] = tmp; // 将小元素放的最后一次变换的位置
    }
    gap = Math.floor(gap / 2);
  }
}
```

二路归并排序(Onlogn)

```
const merge = (arr1, arr2) => {
  let res = [];
  let i = 0,
    j = 0;
  for (; i < arr1.length && j < arr2.length; ) {
    if (arr1[i] > arr2[j]) {
      res.push(arr2[j++]);
    } else {
      res.push(arr1[i++]);
    }
  }
  if (i < arr1.length) {
    res.push(...arr1.slice(i));
  } else {
    res.push(...arr2.slice(j));
  }
  return res;
};

function mergeSort(arr) {
  //采用自上而下的递归方法
  const len = arr.length;
  if (len < 2) {
    return arr;
  }
  // length >> 1 和 Math.floor(len / 2) 等价
  let middle = Math.floor(len / 2),
    left = arr.slice(0, middle),
    right = arr.slice(middle); // 拆分为两个子数组
  return merge(mergeSort(left), mergeSort(right));
}

let mergeSort = (arr, start, end) => {
  // 左闭右闭区间
  // 这个条件是必要的,否则会一直循环,注意返回的是一个数组
  // 左闭右闭区间的终止条件是这样的 [0] 这时候应该返回 [0]
  if (end - start + 1 < 2) return arr.slice(start, end +
1);
  const mid = start + ((end - start) >> 1);
  const res = merge(mergeSort(arr, start, mid),
mergeSort(arr, mid + 1, end));
  return res;
};

mergeSort = (arr, start, end) => {
  // 左闭右开区间
  // 这个条件是必要的,否则会一直循环,注意返回的是一个数组

  // 左闭右开区间的终止条件是这样的 [0, 1) 这时候应该返回 [0]
```

```
  // 左闭右开区间的终止条件是这样的 [0, 1) 这时候应该返回 [0]
  if (end - start < 2) return arr.slice(start, end);
  const mid = start + ((end - start) >> 1);
  const res = merge(mergeSort(arr, start, mid),
mergeSort(arr, mid, end));
  return res;
};
```

快速排序(Onlogn-On2)

```
function quickSort(arr) {
  const n = arr.length;
  const sort = (s, e) => {
    if (s >= e) return;
    let base = arr[s]; // 分界元素
    let i = s,
      j = e;
    while (i < j) {
      while (base <= arr[j] && i < j) j--; // j 是分界元素
右侧开始，第一个小于分界元素的元素下标
      arr[i] = arr[j];
      while (base >= arr[i] && i < j) i++; // i 是分界元素
左侧开始，第一个大于分界元素的元素下标
      arr[j] = arr[i];
    }
    arr[i] = base;
    sort(s, i - 1);
    sort(i + 1, e);
  };
  sort(0, n - 1);
}
```

堆排序(Onlogn)

```
function heapSort(arr) {
  let heapSize = arr.length - 1;
  // 构造大顶堆
  const getMaxHeap = (arr, i, heapSize) => {
    let l = i * 2 + 1; // 它的左儿子
    let r = i * 2 + 2; // 它的右儿子
    let largest = i; // 记录父节点、两个子节点中最大一个
    if (l <= heapSize && arr[l] > arr[largest]) {
      largest = l;
    }
    if (r <= heapSize && arr[r] > arr[largest]) {
      largest = r;
    }
    if (largest !== i) {
      // 将最大的那个换到父节点位置
      [arr[i], arr[largest]] = [arr[largest], arr[i]];
      // 这个主要在下沉后的重新找堆顶元素会用到
      getMaxHeap(arr, largest, heapSize);
    }
  };

  // 初始化大顶堆
  const initMaxHeap = (arr, heapSize) => {
    // 利用了完全二叉树的性质，完全二叉树的节点只能是 2n 个
    // 而每一个`右结点`的父节点的下标值为 n/2-1
    // 将 i 值初始化为第一个末尾节点
    for (let i = (heapSize >> 1) - 1; i >= 0; i--) {
      getMaxHeap(arr, i, heapSize);
    }
  };
  initMaxHeap(arr, heapSize);

  // 执行大顶堆的堆顶值下沉
  while (heapSize >= 0) {
    // 与堆顶元素交换
    // 将堆顶值换到末尾
    [arr[heapSize], arr[0]] = [arr[0], arr[heapSize]];
    heapSize--; // 缩小下一次需要查找的堆范围
    getMaxHeap(arr, 0, heapSize);
  }
}
```

基数排序(Od(r+n)),r 关键字基数,d 长度,n 关键字个数

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

计数排序

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

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

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

  return arr;
}
```

桶排序

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

  let i;
  let minValue = arr[0];
  let 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
  let DEFAULT_BUCKET_SIZE = 5; //设置桶的默认数量为5
  bucketSize = bucketSize || DEFAULT_BUCKET_SIZE;
  let bucketCount = Math.floor((maxValue - minValue) /
bucketSize) + 1;
  let 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++) {
    insertSort(buckets[i]); //对每个桶进行排序，这里使用了插
入排序
    for (let j = 0; j < buckets[i].length; j++) {
      arr.push(buckets[i][j]);
    }
  }

  return arr;
}
```

稳定性

1. 概念

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，ri=rj，且 ri 在 rj 之前，而在排序后的序列中，ri 仍在 rj 之前，则称这种排序算法是稳定的；否则称为不稳定的。

通俗地讲就是能保证排序前 2 个相等的数，其在序列的前后位置顺序，和排序后它们两个的前后位置顺序相同。

2. 比较

插入排序、冒泡排序、二路归并、基数排序、计数排序、桶排序 是稳定排序

选择排序、谢尔排序、堆排序、快速排序 是不稳定排序