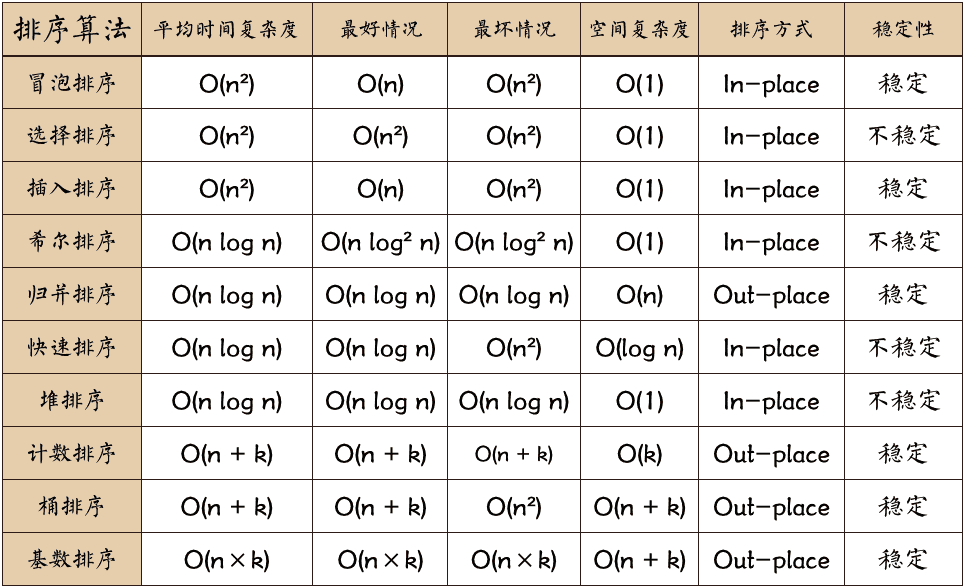
参考文档：<https://www.runoob.com/w3cnote/ten-sorting-algorithm.html>

**各个算法的时间复杂度**



**冒泡排序**

动画演示：<https://www.runoob.com/wp-content/uploads/2019/03/bubbleSort.gif>

冒泡排序由内外2层循环完成，每一次外出循环都会将最大的数据排到后面，如下是数据的变化情况

// 对如下数组进行排序

[4, 2, 5, 3, 1]

// 第1次外层循环

[2, 4, 5, 3, 1]

[2, 4, 5, 3, 1]

[2, 4, 3, 5, 1]

[2, 4, 3, 1, 5]

// 第2次外层循环

[2, 4, 3, 1, 5]

[2, 3, 4, 1, 5]

[2, 3, 1, 4, 5]

// 第3次外层循环

[2, 3, 1, 4, 5]

[2, 1, 3, 4, 5]

// 第4次外出循环

[1, 2, 3, 4, 5]

代码示例

static void BubbleSort(int[] intArray) {

    int temp = 0;

    bool swapped;

    // 外层循环

    for (int i = 0; i < intArray.Length; i++)

    {

        swapped = false;

        // 内侧循环

        for (int j = 0; j < intArray.Length - 1 - i; j++)

        {

            // 如果第 j 位比 j + 1 位大，则调换他们的值

            if (intArray[j] > intArray[j + 1])

            {

                temp = intArray[j];

                intArray[j] = intArray[j + 1];

                intArray[j + 1] = temp;

                if (!swapped)

                    swapped = true;

            }

        }

        if (!swapped)

            return;

    }

}

**选择排序**

动画演示：<https://www.runoob.com/wp-content/uploads/2019/03/selectionSort.gif>

选择排序每次外层循环都会找出一个最小值，将它和前面的值互换

数据变化如下

// 对如下数组进行排序

[4, 2, 5, 3, 1]

// 第 1 次外层循环

// 内层循环在0-4索引中找到数值最小的索引是 4

// 将索引 4 和索引 0 的值进行交换

[1, 2, 5, 3, 4]

// 第2次外层循环

// 内层循环在1-4索引中找到数值最小的索引是 1

// 将索引 1 和索引 1 的值进行交换

[1, 2, 5, 3, 4]

// 第3次外层循环

// 内层循环在2-4索引中找到数值最小的索引是 3

// 将索引 3 和索引 2 的值进行交换

[1, 2, 3, 5, 4]

// 第4次外出循环

// 内层循环在3-4索引中找到数值最小的索引是 4

// 将索引 4 和索引 3 的值进行交换

[1, 2, 3, 4, 5]

示例代码

function selectionSort(arr) {

    var len = arr.length;

    // minIndex 用于标志最小值所在的索引

    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];

        // 将最小值排到前面

        arr[i] = arr[minIndex];

        arr[minIndex] = temp;

    }

    return arr;

}

**插入排序**

演示动画：<https://www.runoob.com/wp-content/uploads/2019/03/insertionSort.gif>

代码示例

function insertionSort(arr) {

    var len = arr.length;

    // current 保存当前要插入的值

    // preIndex 保存的是要插入位置的前一个位置

    var preIndex, current;

    for (var i = 1; i < len; i++) {

        preIndex = i - 1;

        // 保存当前要插入的值

        current = arr[i];

        // 如果 preIndex 位置的值大于当前值

        while(preIndex >= 0 && arr[preIndex] > current) {

            // 则将 preIndex 位置的值移到后一位

            arr[preIndex+1] = arr[preIndex];

            // preIndex - 1

            preIndex--;

        }

        // 如果 preIndex 位置的值小于当前值，则在 preIndex 插入当前值

        arr[preIndex+1] = current;

    }

    return arr;

}

数据变化

// 对如下数组进行排序

[4, 2, 5, 3, 1]

// 第 1 次外层循环

// 4大于2，将4往后移

// 比较索引小于0，将2插入到索引0

[2, 4, 5, 3, 1]

// 第2次外层循环

// 4小于5，5插入在4 的后面

[2, 4, 5, 3, 1]

// 第3次外层循环

// 5大于3，5往后移

// 4大于3，4往后移

// 2小于3，3插在2的后面

[2, 3, 4, 5, 1]

// 第4次外出循环

// 5大于1，将5往后移

// 4大于1，将4往后移

// 3大于1，将3往后移

// 2大于1，将2往后移

// 比较索引小于0，将1插入到索引0

[1, 2, 3, 4, 5]

**希尔排序**

希尔排序是插入排序的改进版本

我们定义一个数gap，我们把间隔为gap的值分为一组（如对于gap=4的数组[8, 3, 5, 4, 2, 1, 6, 7]，8和2就是一组，3和1就是一种），我们分别对这些组进行插入排序，然后减小gap的值，再进行排入排序，直到gap等于1

数据变化如下

// 如下数组

[8, 3, 5, 4, 2, 1, 6, 7]

// gap = 4，分别对这4个组进行插入排序

[2, 1, 5, 4, 8, 3, 6, 7]

// gap = 2，分别对这2个组进行插入排序

[2, 1, 5, 3, 6, 4, 8, 7]

// gap = 1，分别对这1个组进行插入排序

[1, 2, 3, 4, 5, 6, 7, 8]

示例代码

function shellSort(arr) {

    var len = arr.length,

        temp,

        // 定义 gap

        gap = arr.length / 2;

    // 最外层循环，用于减少 gap 的值

    for (gap; gap > 0; gap = Math.floor(gap / 2)) {

        // 插入排序

        for (var i = gap; i < len; i++) {

            // 将要插入的值保存到临时变量中

            temp = arr[i];

            // 如果 arr[j] 的值大于要插入的值，则将 arr[j] 往后移一位 即移到 arr[j + gap]

            for (var j = i - gap; j >= 0 && arr[j] > temp; j -= gap) {

                arr[j + gap] = arr[j];

            }

            // 否则 要插入的值 插入在 arr[j] 的后面 即 arr[j + gap]

            arr[j + gap] = temp;

        }

    }

    return arr;

}

console.log(shellSort([8, 3, 5, 4, 2, 1, 6, 7]));

**归并排序**

动画演示：<https://www.runoob.com/wp-content/uploads/2019/03/mergeSort.gif>

口头难以说明白，看代码

function mergeSort(arr) {

    var len = arr.length;

    // 如果数组的个数为 1，则数组无法在拆分

    if(len < 2) {

        return arr;

    }

    // 否则将数组拆分为左右两个子数组

    var middle = Math.floor(len / 2),

        left = arr.slice(0, middle),

        right = arr.slice(middle);

    // 对左数组进行归并排序

    var leftarr = mergeSort(left);

    // 对右数组进行归并排序

var rightarr = mergeSort(right);

    // 合并这 2 个数组

    return merge(leftarr, rightarr);

}

function merge(left, right)

{

var result = [];

    while (left.length && right.length) {

        // 哪个数值较小，就将哪个先推入数值中

        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());

    // 返回已合并的数组

    return result;

}

**快速排序**

动画演示：<https://www.runoob.com/wp-content/uploads/2019/03/quickSort.gif>

听到这个名字就知道它的优点就是快

1. 我们选择数组的第一个元素作为参照，将小于该元素的元素放到左边，将大于该元素的元素放到右边
2. 然后我们将左右2边看作2个数组，分别执行步骤1

步骤1的实现

// 如下数组

[4, 3, 5, 8, 2, 1, 6, 7]

// 以 4 作为基准，我们需要一个辅助变量 index

// 索引大于或等于index的，其值大于或等于4

// 所以小于index的，其值小于4

index = 1;

[4, 3, 5, 8, 2, 1, 6, 7]

// 比较索引1，索引1的值3小于4，将索引1的值和index的值交换，这样index所在的位置其值就比4小了，索引index+1

index = 2

[4, 3, 5, 8, 2, 1, 6, 7]

// 比较索引2，索引2的值5大于4，没有改变

index = 2

[4, 3, 5, 8, 2, 1, 6, 7]

// 比较索引3，索引3的值8大于4，没有改变

index = 2

[4, 3, 5, 8, 2, 1, 6, 7]

// 比较索引4，索引4的值2小于4，交换索引4和索引index的值，index++

index = 3

[4, 3, 2, 8, 5, 1, 6, 7]

// 比较索引5，索引5的值1小于4，交换索引5和索引index的值，index++

index = 4

[4, 3, 2, 1, 5, 8, 6, 7]

// 比较所以6和7后，我们的数组变为如下

index = 4

[4, 3, 2, 1, 5, 8, 6, 7]

// 交换我们基准值和index-1的索引的值

[1, 3, 2, 4, 5, 8, 6, 7]

// 如此大于4的在基准值的右边，小于4的在基准值的左边，将左右2边再进行快速排序

快速排序代码实现

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) {

        // 交换数据位置

        partitionIndex = partition(arr, left, right);

        // 对左边进行快速排序

        quickSort(arr, left, partitionIndex-1);

        // 对右边进行快速排序

        quickSort(arr, partitionIndex+1, right);

    }

    return arr;

}

// 交换数据位置

function partition(arr, left ,right) {

    var pivot = left,                       // 设定基准值所在索引

        index = pivot + 1;                  // 分隔数据的index

    for (var i = index; i <= right; i++) {

        // 如果i索引的值小于基准值

        if (arr[i] < arr[pivot]) {

            // 交换i索引和index索引的值

            swap(arr, i, index);

            // index + 1

            index++;

        }

    }

    // 交换基准值和index-1的值

    swap(arr, pivot, index - 1);

    // 返回基准值所在的索引

    return index-1;

}

function swap(arr, i, j) {

    var temp = arr[i];

    arr[i] = arr[j];

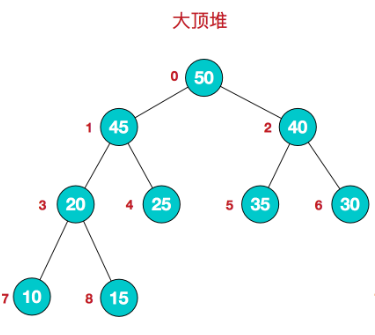
    arr[j] = temp;

}

**堆排序**

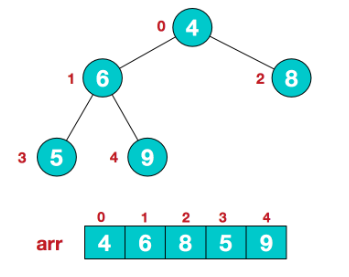
参考：<https://www.cnblogs.com/chengxiao/p/6129630.html>

大顶堆：如下，子节点小于当前节点的二叉树我们称为大顶堆



我们使用大顶堆进行排序，如下示例步骤

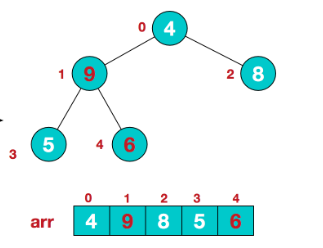
如下数组，我们可以看作一颗树



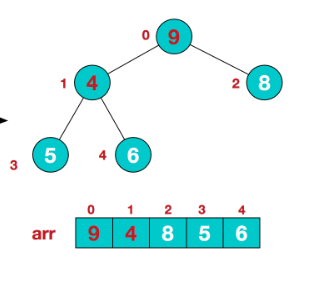
1. 构建大顶堆树

数组元素有5个，5/2等于2，所以我们从第2个元素（索引为1）开始构建堆

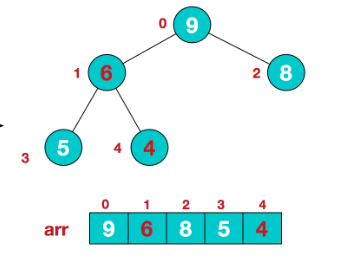
1. 比较节点1的2个子节点，如果子节点比当前节点大，则将最大的子节点和当前节点交换，如下交换节点1和节点4



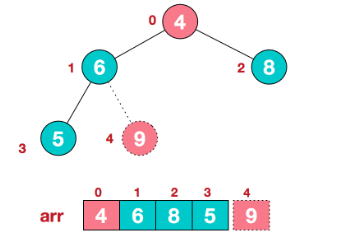
1. 此时以1节点为根的子树满足了大顶堆，接着我们调整1节点的前一个节点，即0节点，调整后如下



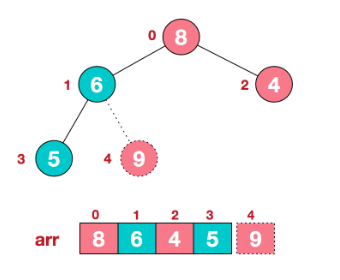
1. 此时以1节点为根的子树不满足大顶堆，所以调整1，调整后如下



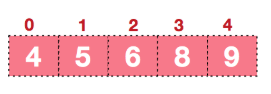
1. 接下来我们应该调整0节点的前一个节点，单0节点没有节点了，所以此时已经构建了个大顶堆
2. 将大顶堆的最后一个元素与第1个元素交换，并将最后一个元素从大顶堆中分离



1. 交换后新的大顶推又不满足大顶堆了，所以我们堆节点0进行调整，的到新的大顶推如下



1. 接着我们重复步骤2，3，将节点0和节点3交换并分离节点3，并重新调整推，如此反复循环，我们可以得到最后的排序结果



**基数排序**

演示动画：<https://www.runoob.com/wp-content/uploads/2019/03/radixSort.gif>

基数排序只能用于数字排序，我们需要准备具有10个队列的数组，然后去数字的个位，将个位与索引对应，然后将数字放入队列中（如数字21，则将数字21放入到队列1中），然后将0到9队列数字依此出列，然后去数字的十位，重复操作

描述很混乱，看演示动画会更好

示例代码

// 队列数组

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++) {

            // 队列的索引

            var bucket = parseInt((arr[j] % mod) / dev);

            // 队列

            if(counter[bucket]==null) {

                counter[bucket] = [];

            }

            // 入队列

            counter[bucket].push(arr[j]);

        }

        var pos = 0;

        for(var j = 0; j < counter.length; j++) {

            var value = null;

            if(counter[j]!=null) {

                // 出队列

                while ((value = counter[j].shift()) != null) {

                      arr[pos++] = value;

                }

          }

        }

    }

    return arr;

}