算法知识点

笔记本： 刷题
创建时间： 2021/1/20 11:26 **更新时间：** 2021/1/21 11:45
作者： 183xosxi216

一. 排序方法

方法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡	O(n^2)	O(n)	O(n^2)	O(1)	In-place	稳定
选择	O(n^2)	O(n^2)	O(n^2)	O(1)	In-place	不稳定
插入	O(n^2)	O(n)	O(n^2)	O(1)	In-place	稳定
希尔	O(nlogn)	O(nlogn)	O(nlogn^2)	O(1)	In-place	不稳定
归并	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	out-place	稳定
快排	O(nlogn)	O(nlogn)	O(n2)	O(n2)	in-place	不稳定
堆排	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	in-place	不 稳定

1. 冒泡排序

（1）算法步骤

> （1）比较相邻的元素。如果第一个比第二个大，就交换他们两个。
> （2）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。这步做完后，最后的元素会是最大的数。
> （3）针对所有的元素重复以上的步骤，除了最后一个。

> （4）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。

（2）算法实现

```
function  bubblesort(s)  {
                        for(var  i = 0; i <= s.length-1; i++)  {
                                for(var  j = 0; j <= s.length-1-i; j++)
                                if(s[j] > s[j+1])  {
                                        var  temp  =  s[j+1];
                                        s[j+1]  =  s[j];
                                        s[j]  =  temp;
                                }
                        }
                return  s;
                }
```

2. 选择排序

（1）算法步骤

> （1）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
> （2）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。
> （3）重复第二步，直到所有元素均排序完毕。

（2）算法实现

```
function  Selectionsort(s)  {
                        var  min_index,  temp;
                        for(var  i=0;i<=s.length-1;i++)  {
                                for(var  j=i;j<=s.length-1;j++){
                                        min_index  =  i;
                                        if(s[min_index]>s[j])  {
                                        min_index  =  j;
                                        }
                                temp  =  s[i];
                                s[i]  =  s[min_index];
```

```
                        s[min_index] = temp;
                        }
                }
        return s;
        }
```

3. 插入排序

（1）算法步骤

> （1）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。
> （2）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。）

（2）算法实现

```
function insertionsort(s) {
        var pre_index, current;
                for(var i=1;i<=s.length-1;i++) {
                        current = s[i];
                        pre_index = i-1;
                        while(pre_index>=0 && s[pre_index]>current) {

                                s[pre_index+1] = s[pre_index]; //
把大于current的元素都往后移
                                pre_index--;
                        }
                        s[pre_index+1] = current; //把current的值
抽出来放到前面使其有序
                }
        return s;
        }
```

4. 希尔排序

（1）算法原理

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

- 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率；
- 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位；

希尔排序的基本思想是：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，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

（2）算法步骤

> （1）选择一个增量序列 t1，t2，……，tk，其中 ti > tj, tk = 1；
> （2）按增量序列个数 k，对序列进行 k 趟排序；
> （3）每趟排序，根据对应的增量 ti，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。

（3）算法实现

```
      function  shellsort(s)  {
                        var  gap  =  1,
                                pre_index,
                                current;
                        while  (gap<s.length/3)  {
                                gap  =  gap*3+1;
                        }
                        for(gap;  gap>0;  gap  =  Math.floor(gap/3))
{      //gap根据划分长度不断调整，直到为1
                                for(var  i=gap;i<=s.length-1;i++)  {
                                        current  =  s[i];
                                        pre_index  =  i-gap;
                                        while(pre_index>=0  &&  s[pre_index]>cur

                                                s[pre_index+gap]  =  s[pre_index
把大于current的元素都往后移
                                                pre_index-=gap;
                                        }
                                s[pre_index+gap]  =  current;  //把current的
值抽出来放到前面使其有序
                                }
                        }
```

```
                return  s;
                }
```

5. 归并排序

（1）算法步骤

> （1）申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列;
> （2）设定两个指针，最初位置分别为两个已经排序序列的起始位置;
> （3）比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置;
> （4）重复步骤 3 直到某一指针达到序列尾;
> （5）将另一序列剩下的所有元素直接复制到合并序列尾。

（2）算法实现

```
    function  mergeSort(s)  {
                        var  len  =  s.length;
                        if(len  <=  1)  {
                                return  s;
                        }
                        var  m  =  Math.floor(len/2),
                        left  =  s.slice(0,m),
                        right  =  s.slice(m);
                        return  merge(mergeSort(left),  mergeSort(right));
                }
                function  merge(l,  r)  {
                        var  result  =  []; //首先创建一个空数组，大小为M+N
                        while(l.length  &&  r.length)  {
                                if(l[0]<=r[0])  {
                                        result.push(l.shift());
                                }  else{
                                        result.push(r.shift());
                                }          //比较l和r中第一个值，将较小的值存入到result中
                        }
```

```
                while(l.length)  {
                        result.push(l.shift());
                }
                while(r.length)  {
                        result.push(r.shift());
                }
                return  result;
        }
```

6. 快速排序

（1）算法思想

> （1）从数列中挑出一个元素，称为 “基准”（pivot）；
> （2）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；
> （3）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；

（2）算法实现

```
function  quickSort(nums,  left,  right)  {
        if  (left  >=  right)  return;
        let  pivotIndex  =  partition(nums,  left,  right)
        quickSort(nums,  left,  pivotIndex  -  1)
        quickSort(nums,  pivotIndex  +  1,  right)
        return  nums;
}
function  partition  (nums,  left,  right)  {
        let  pivot  =  right;
        let  leftIndex  =  left;
        for  (let  i  =  left;  i  <  right;  i++)  {
                if  (nums[i]  <  nums[pivot])  {
                        [nums[leftIndex],  nums[i]]  =  [nums[i],  nums[leftInde

                        leftIndex++;
                }
```

```
        }
        [nums[leftIndex], nums[pivot]] = [nums[pivot], nums[leftIndex]];
        return leftIndex;
}
```

7. 堆排序

（1）实现思路

> （1）将初始二叉树转化为大顶堆（heapify）（实质是从第一个非叶子结点开始，从下至上，从右至左，对每一个非叶子结点做shiftDown操作），此时根结点为最大值，将其与最后一个结点交换。
> （2）除开最后一个结点，将其余节点组成的新堆转化为大顶堆（实质上是对根节点做shiftDown操作），此时根结点为次最大值，将其与最后一个结点交换。
> （3）重复步骤2，直到堆中元素个数为1（或其对应数组的长度为1），排序完成。

（2）算法实现

```
//定义一个交换函数，用于后面的两次交换
                function swap(A, i, j) {
                        let temp = A[i];
                        A[i] = A[j];
                        A[j] = temp;
                }
                //堆的建立过程：传入参数为A，当前根节点序号i，长度
                function shiftDown(A, i, len) {
                        let temp = A[i]; //将根节点暂存
                        for( let j =2*i+1; j< len; j = j*2+1 ) { //j
是i的子节点，比较大小
                                if (j+1<len && A[j]<A[j+1]) {      //先
找到i的子节点中的较大值和i比较
                                        j++;
                                }
                                if (A[j]>temp) { //如果根节点的值小于子节
点，就交换，并且将子节点的序号交给根节点
                                        swap(A, i, j);
                                        i = j;
```

```
                        } else {
                                break;
                        }
                }
        }
        //堆排序过程，将最大的根节点依次移到末尾
        function heapSort(A) {
                //初始化一个非叶子节点
                for(let i = Math.floor(A.length/2)-1; i>=0; i--) {
                        shiftDown(A, i, A.length); //将所有节点调整，使其变为大根堆
                }
                //交换过程：将最大的根节点移到末尾，注意当前移动后要重新调整剩余元素使其变为大根堆
                for(let i = Math.floor(A.length)-1; i>0; i--) {
                        swap(A, 0, i); //i是最后一个元素，i和0交换
                        shiftDown(A, 0, i); // 从根节点开始调整，并且最后一个结点已经为当
                                            // 前最大值，不需要再参与比较，所以第三个参数
                                            // 为 i，即比较到最后一个结点前一个即可
                }
                return A;
        }
```