六种排序算法的js实现以及性能对比

首先为了对比各个排序算法的性能，这里先写了一个生成大规模数组的方法-generateArray:

exports.generateArray = function(length) {

let arr = Array(length);

for(let i=0; i<length; i++) {

arr[i] = Math.random();

}

return arr;

};

1. 冒泡排序

冒泡排序得名于其排序方式，它遍历整个数组，将数组的每一项与后一项进行对比，如果不符合要求就变换位置，一共遍历n轮，n为数组的长度。n轮之后，数组完全排序。整个过程符合要求的数组项就像气泡从水底冒到水面一样，所以称为冒泡排序。

function bubbleSort(arr) {

//console.time('BubbleSort');

// 获取数组长度，以确定循环次数。

let len = arr.length;

// 遍历数组len次，以确保数组被完全排序。

for(let i=0; i<len; i++) {

// 遍历数组的前len-i项，忽略后面的i项（已排序部分）。

for(let j=0; j<len - 1 - i; j++) {

// 将每一项与后一项进行对比，不符合要求的就换位。

if(arr[j] > arr[j+1]) {

[arr[j+1], arr[j]] = [arr[j], arr[j+1]];

}

}

}

//console.timeEnd('BubbleSort');

return arr;

}

1. 选择排序

选择排序是一种原始比较排序法，大致思路是：找到数组中的最小（大）值，并将其放到第一位，然后找到第二小的值放到第二位，等等，以此类推。

function selectionSort(arr) {

//console.time('SelectionSort');

// 获取数组长度，确保每一项都被排序。

let len = arr.length;

// 遍历数组的每一项。

for(let i=0; i<len; i++) {

// 从数组的当前项开始，因为左边部分的数组项已经被排序。

let min = i;

for(let j=i; j<len; j++) {

if(arr[j]<arr[i]) {

min = j;

}

}

if(min !== i) {

[arr[min], arr[i]] = [arr[i], arr[min]];

}

}

//console.timeEnd('SelectionSort');

return arr;

}

1. 插入排序

插入排序从数组的第二项开始遍历数组的n-1项（n为数组的长度），遍历过程中对于当前项的左边数组项，依次从右到左进行对比，如果左边选项大于（或小于）当前项，则左边选项向右移动，然后继续对比前一项，直到找到不大于（不小于）自身的选项为止，对于所有大于当前项的选项，都在原来位置的基础上向右移动了一项。

// 对于如下数组

var arr = [2,1,3,5,4,3];

// 从第二项（即arr[1]）开始遍历，

// 第一轮：

// a[0] >= 1为true，a[0]右移，

arr = [2,2,3,5,4,3];

// 然后1赋给a[0]，

arr = [1,2,3,5,4,3];

// 然后第二轮：

// a[1] >= 3不成立，该轮遍历结束。

// 第三轮;

// a[2] >= 5不成立，该轮遍历结束。

// 第四轮：

// a[3] >= 4为true，a[3]右移，

arr = [1,2,3,5,5,3];

// a[2] >= 4不成立，将4赋给a[3]，然后结束该轮遍历。

arr = [1,2,3,4,5,3];

// a[4] >= 3成立，a[4]右移一位，

arr = [1,2,3,4,5,5];

// arr[3] >= 3成立，arr[3]右移一位，

arr = [1,2,3,4,4,5];

// arr[2] >= 3成立，arr[2]右移一位，

arr = [1,2,3,3,4,5];

// arr[1] >= 3不成立，将3赋给a[2]，结束该轮。

arr = [1,2,3,3,4,5];

// 遍历完成，排序结束。

Js实现（从小到大排序）

function insertionSort(arr) {

//console.time('InsertionSort');

let len = arr.length;

for(let i=1; i<len; i++) {

let j = i;

let tmp = arr[i];

while(j > 0 && arr[j-1] > tmp) {

arr[j] = arr[j-1];

j--;

}

arr[j] = tmp;

}

//console.timeEnd('InsertionSort');

return arr;

}

1. 归并排序

归并排序的核心思想是分治，分治是通过递归地将问题分解成相同或者类型相关的两个或者多个子问题，直到问题简单到足以解决，然后将子问题的解决方案结合起来，解决原始方案的一种思想。

归并排序通过将复杂的数组分解成足够小的数组（只包含一个元素），然后通过合并两个有序数组（单元素数组可以认为是有序数组）来达到综合子问题解决方案的目的。所以归并排序的核心在于如何整合两个有序数组，拆分数组只是一个辅助过程。

// 假设有以下数组，对其进行归并排序使其按从小到大的顺序排列：

var arr = [8,7,6,5];

// 对其进行分解，得到两个数组：

[8,7]和[6,5]

// 然后继续进行分解，分别再得到两个数组，直到数组只包含一个元素：

[8]、[7]、[6]、[5]

// 开始合并数组，得到以下两个数组：

[7,8]和[5,6]

// 继续合并，得到

[5,6,7,8]

// 排序完成

Js的实现（从小到大排序）

function mergeSort(arr) {

//console.time('MergeSort');

//let count = 0;

console.log(main(arr));

//console.timeEnd('MergeSort');

//return count;

// 主函数。

function main(arr) {

// 记得添加判断，防止无穷递归导致callstack溢出，此外也是将数组进行分解的终止条件。

if(arr.length === 1) return arr;

// 从中间开始分解，并构造左边数组和右边数组。

let mid = Math.floor(arr.length/2);

let left = arr.slice(0, mid);

let right = arr.slice(mid);

// 开始递归调用。

return merge(arguments.callee(left), arguments.callee(right));

}

// 数组的合并函数，left是左边的有序数组，right是右边的有序数组。

function merge(left, right) {

// il是左边数组的一个指针，rl是右边数组的一个指针。

let il = 0,

rl = 0,

result = [];

// 同时遍历左右两个数组，直到有一个指针超出范围。

while(il < left.length && rl < right.length) {

//count++;

// 左边数组的当前项如果小于右边数组的当前项，那么将左边数组的当前项推入result，反之亦然，同时将推入过的指针右移。

if(left[il] < right[rl]) {

result.push(left[il++]);

}

else {

result.push(right[rl++]);

}

}

// 记得要将未读完的数组的多余部分读到result。

return result.concat(left.slice(il)).concat(right.slice(rl));

}

}

归并排序的性能确实达到了应用级别，但是还有一些不足，因为这里的merge函数新建了一个result数组来盛放合并后的数组，导致空间复杂度增加，这里还可以进行优化，使得数组进行原地排序。

1. 快速排序

快速排序的核心思想也是分而治之，递归的将大数组分解为小数组，直到长度为1，不过与归并排序的区别在于其重点在于数组的分解，而归并排序的重点在于数组的合并。

基本步骤如下：

1. 首先，选取数组的中间项作为参考点pivot;
2. 创建左右俩个指针left和right，left指向数组的第一项，right指向最后一项，然后移动左指针，直到其值不小于pivot，然后移动右指针，直到其值不大于pivot。
3. 如果left仍然不大于right，交换左右指针的值（指针不交换），然后左指针右移，右指针左移，继续循环直到left大于right才结束，返回left指针的值。
4. 根据上一轮分解的结果（left的值），切割数组的left和right两个数组，然后分别再分解。
5. 重复以上过程，直到数组长度为1才结束分解。

Js实现（从大到小的排序）

function quickSort(arr) {

let left = 0,

right = arr.length - 1;

//console.time('QuickSort');

main(arr, left, right);

//console.timeEnd('QuickSort');

return arr;

function main(arr, left, right) {

// 递归结束的条件，直到数组只包含一个元素。

if(arr.length === 1) {

// 由于是直接修改arr，所以不用返回值。

return;

}

// 获取left指针，准备下一轮分解。

let index = partition(arr, left, right);

if(left < index - 1) {

// 继续分解左边数组。

main(arr, left, index - 1);

}

if(index < right) {

// 分解右边数组。

main(arr, index, right);

}

}

// 数组分解函数。

function partition(arr, left, right) {

// 选取中间项为参考点。

let pivot = arr[Math.floor((left + right) / 2)];

// 循环直到left > right。

while(left <= right) {

// 持续右移左指针直到其值不小于pivot。

while(arr[left] < pivot) {

left++;

}

// 持续左移右指针直到其值不大于pivot。

while(arr[right] > pivot) {

right--;

}

// 此时左指针的值不小于pivot，右指针的值不大于pivot。

// 如果left仍然不大于right。

if(left <= right) {

// 交换两者的值，使得不大于pivot的值在其左侧，不小于pivot的值在其右侧。

[arr[left], arr[right]] = [arr[right], arr[left]];

// 左指针右移，右指针左移准备开始下一轮，防止arr[left]和arr[right]都等于pivot然后导致死循环。

left++;

right--;

}

}

// 返回左指针作为下一轮分解的依据。

return left;

}

}

性能对比：

各种排序方法在大量的数据下耗时情况如下：

