排序算法

@authur Bingbing He,
2017/05/05

各种排序算法的比较

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						

以下代码，可以在同级目录中的sortMethods的工程文件夹详细查看，直接import进Eclipse即可，**Main.java**为入口，检测排序算法可行性。

直接插入排序（InsertSort）

```
public static void sort(int[] numbers) {
    int size = numbers.length;
    int tmp = 0;
    for (int i = 0; i < size; i++) {
        tmp = numbers[i];
        int j = i;
        for (; j > 0 && tmp < numbers[j - 1]; j--) {
            numbers[j] = numbers[j - 1];
        }
        numbers[j] = tmp;
    }
}
```

希尔排序（ShellSort）

```
public static void sort(int[] numbers) {
    int size = numbers.length;
    for (int increment = size / 2; increment > 0; increment /= 2) {
        for (int k = 0; k < increment; k++) {
            // 直接插入排序
            for (int i = k; i < size; i += increment) {
                int tmp = numbers[i];
                int j = i;
                while (j - increment >= 0 &&
                    tmp < numbers[j - increment]){
                    numbers[j] = numbers[j - increment];
                    j -= increment;
                }
                numbers[j] = tmp;
            }
        }
    }
}
```

直接选择排序（SelectSort）

```
public static void sort(int[] numbers){
    int size = numbers.length;
    for(int i = 0; i < size - 1  ; i ++){
        int minIndex = i;
        for(int j = i + 1 ; j < size ; j++){
            if( numbers[j] < numbers[minIndex]){
                minIndex = j;
            }
        }
        int tmp = numbers[minIndex];
        numbers[minIndex] = numbers[i];
        numbers[i] = tmp;
    }
}
```

堆排序（HeapSort）

```
public static void sort(int[] numbers) {
    int size = numbers.length;
    for (int i = 0; i < size - 1; i++) {
        int lastIndex = size - 1 - i;
        buildMaxHeap(numbers, lastIndex);
        swap(numbers, lastIndex, 0);
    }
}

public static void buildMaxHeap(int[] numbers, int lastIndex) {
    for (int i = (lastIndex - 1) / 2; i >= 0; i--) {
        int k = i;
        while (k * 2 + 1 <= lastIndex) {
            int biggerIndex = 2 * k + 1;
            if (biggerIndex < lastIndex) {
                if (numbers[biggerIndex] < numbers[biggerIndex + 1]) {
                    biggerIndex++;
                }
            }
            if (numbers[k] < numbers[biggerIndex]) {
                swap(numbers, k, biggerIndex);
                k = biggerIndex;
            } else {
                break;
            }
        }
    }
}

public static void swap(int[] numbers, int i, int j) {
    int tmp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = tmp;
}
```

冒泡排序（BubbleSort）

```
public static void sort(int[] numbers) {
    int size = numbers.length;
    int tmp = 0;
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (numbers[j] > numbers[j + 1]) {
                tmp = numbers[j];
                numbers[j] = numbers[j + 1];
                numbers[j + 1] = tmp;
            }
        }
    }
}
```

快速排序（QuickSort）

```
public static void sort(int[] numbers) {
    if (numbers.length > 0) {
        quickSort(numbers, 0, numbers.length - 1);
    }
}
```

递归

```
public static void quickSort(int[] numbers, int low, int high) {
    if (low < high) {
        int middle = partition(numbers, low, high);
        quickSort(numbers, low, middle - 1);
        quickSort(numbers, middle + 1, high);
    }
}
```

非递归

```
public static void quickSort2(int[] numbers, int low, int high) {
    Stack<Integer> stack = new Stack<>();
    if (low < high) {
        stack.add(low);
        stack.add(high);
        while (!stack.isEmpty()) {
            high = stack.pop();
            low = stack.pop();
            if (low >= high) {
                continue;
            }
            int middle = partition(numbers, low, high);
            stack.add(low);
            stack.add(middle - 1);
            stack.add(middle + 1);
            stack.add(high);
        }
    }
}
```

Partition方法的3种方式

1

```
java
public static int partition(int[] numbers, int low, int high) {
int tmp = numbers[low];
while (low < high) {
while (low < high && numbers[high] >= tmp) {
high--;
}
numbers[low] = numbers[high];
while (low < high && numbers[low] <= tmp) {
low++;
}
numbers[high] = numbers[low];
}
numbers[low] = tmp;
int middle = low;
return middle;
}
```

2

```
java
public static int partition2(int[] numbers, int low, int high) {
```

```
int pivot = numbers[high];
int current = low;
for (int i = low; i <= high; i++) {
if (numbers[i] < pivot) {
swap(numbers, i, current);
current++;
}
}
swap(numbers, high, current);
int middle = current;
return middle;
}
```

3

```
java
public static int partition3(int[] numbers, int low, int high) {
int pivot = numbers[low];
int start = low;
while(low <= high){
while(low<= high && numbers[low] <= pivot){
low++;
}
while(low<=high && numbers[high] >= pivot){
high--;
}
if(low < high){
swap(numbers, low, high);
}
}
swap(numbers, high , start);
int middle = high;
return middle;
}
```

归并排序（MergeSort）

递归

```
public static void sort(int[] numbers) {
    int size = numbers.length;
    mergeSort(numbers, 0, size - 1);
}

public static void mergeSort(int[] numbers, int low, int high) {
    int mid = (low + high) / 2;
    if (low < high) {
        mergeSort(numbers, low, mid);
        mergeSort(numbers, mid + 1, high);
        merge(numbers, low, mid, high);
    }
}

public static void merge(int[] numbers, int low, int mid, int high) {
    int[] tmp = new int[high - low + 1];
    int i = low;
    int j = mid + 1;
    int k = 0;
    while (i <= mid && j <= high) {
        if (numbers[i] < numbers[j]) {
            tmp[k++] = numbers[i++];
        } else {
            tmp[k++] = numbers[j++];
        }
    }
    while (i <= mid) {
        tmp[k++] = numbers[i++];
    }
    while (j <= mid) {
        tmp[k++] = numbers[j++];
    }
    for (int t = 0; t < k; t++) {
        numbers[t + low] = tmp[t];
    }
}
```

非递归

```
public static void merge_sort(int[] numbers) {
    int size = numbers.length;
    int[] tmpArray = new int[size];
    int block = 1;
    while (block < size * 2) {
        for (int start = 0; start < size; start += 2 * block) {
            int low = start;
            int mid = start + block;
            if (mid > size) {
                mid = size;
            }
            int high = start + 2 * block;
            if (high > size) {
                high = size;
            }
            int start1 = low;
            int end1 = mid;
            int start2 = mid;
            int end2 = high;
            int k = low;
            while (start1 < end1 && start2 < end2) {
                if (numbers[start1] < numbers[start2]) {
                    tmpArray[k++] = numbers[start1++];
                } else {
                    tmpArray[k++] = numbers[start2++];
                }
            }
            while (start1 < end1) {
                tmpArray[k++] = numbers[start1++];
            }
            while (start2 < end2) {
                tmpArray[k++] = numbers[start2++];
            }
        }
        for (int t = 0; t < size; t++) {
            numbers[t] = tmpArray[t];
        }
        block *=2;
    }
}
```