

Report

ADS2 AE1

Umar Farouk Yusuf

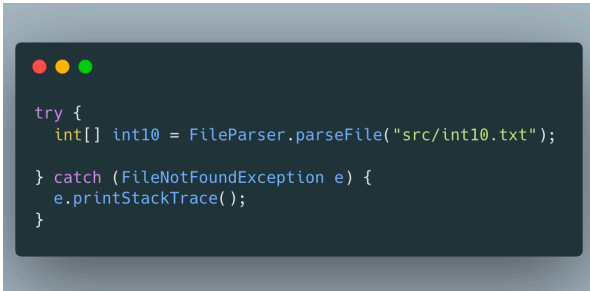
2613065y

9 February 2022

Documentation

There are 4 classes files in the project (FileParser, Sort, Example, and TestSortingAlgorithms), each of which provide different functions and utilities.

FileParser is a class with no constructor. Within it, is a static method with the signature `parseFile(String filePath)` that is used to read data from a file and return it as an `int[]`. This, method throws an error if a file at `filePath` does not exist. It can be used as follows:



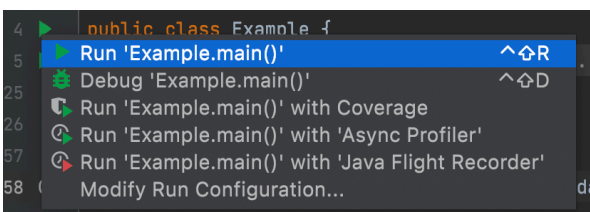
```
try {
    int[] int10 = FileParser.parseFile("src/int10.txt");
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

TestSortingAlgorithm is a class with no constructor. Within it, is a static method with the signature `isSorted(int[] a)` that checks whether input `int[] a` is correctly sorted. It is used as follows:



```
int[] arr = {0, 1, 2, 3, 4, 5}
boolean passed = TestSortingAlgorithms.isSorted(arr); //true
```

The Example class contains an example of the functionality in usage. It is also the file I used to get my readings. It also contains a `testAlgorithms(int[] dataset)` method to test and time the different sorting algorithms with an input `dataset`, and an overload `testAlgorithms(int[][] datasets)` that allows us to test with multiple datasets. To use it just run the main method of the class.



```
4 public class Example {
5     Run 'Example.main()' ^⇧R
25     Debug 'Example.main()' ^⇧D
26     Run 'Example.main()' with Coverage
57     Run 'Example.main()' with 'Async Profiler'
58     Run 'Example.main()' with 'Java Flight Recorder'
    Modify Run Configuration...
```

The Sort class contains the main logic of this exercise with methods for each sorting algorithm as well as a method `generatePathologicalSequence(int size, int max)` that generates a pathological input sequence for the QuickSort algorithm considering a median-of-three partitioning scheme array of integers with no repetition where `size` dictates the length of the output array and `max` dictates the maximum value of any element in the array; `max` must be greater than or equal to `size`. The sorting methods take in an array and sorts it. They have the following signatures:

```
quickSort(int[] arr)
quickSortHybrid(int[] arr, int cutoff)
quickSortMedianOfThree(int[] arr)
quickSortThreeWay(int[] arr)
```

`insertionSort(int[] arr)`

`mergeSort(int[] arr)`

Three Way QuickSort

My Three-Way Quicksort implementation focuses on partitioning the input array into three subarrays that contain all elements less than the pivot, equal to the pivot, and greater than the pivot respectively. To do this it starts with an if statement that ensures that the left index is less than the right index. If that statement does not pass the size of the array is less than 2 so there is nothing to sort and the function returns. This is done to prevent an infinite recursion loop. If the statement passes the check, the algorithm then calls a *threeWayPartition* method.

The partition method is where most of the work is done. It first selects the first element (at index *left*) as its pivot. It focuses on splitting the array into three subarrays, one for all elements less than the pivot, one for all elements that are equal to the pivot, and one for all elements greater than the pivot. It does this by looping through the entire array with an index *mid* that starts at the first index of the array and uses indexes *p1* and *p2* to keep track of where the array is partitioned, where *p1* starts at the beginning of the array, and *right* starts at the end. It then proceeds to increment *mid* and *p1* and swap the elements at *mid* and *p1* when the element at *mid* is less than the *pivot*, increments *mid* without making any swaps when the element at *mid* is equal to the *pivot*, and decrements *p2* and swaps the elements at *mid* and *p2* when the element at *mid* is greater than the *pivot*. After it is done, it returns an `int[]` with two indexes representing where the array is partitioned into the three subarrays.

We can then recursively sort the subarray with elements less than the *pivot* and the subarray with elements greater than our *pivot* using the same method to get our sorted array.

Comparisons and Findings

I will be skipping all datasets with less than 20,000 elements as tests ran in less than 1ms.

	int20k	Int500k	intBig	dutch
QuickSort	1 ms	28 ms	54 ms	148 ms
Hybrid QuickSort (cutOff = 10)	1 ms	26 ms	55 ms	147 ms
Hybrid QuickSort (cutOff = 50)	2 ms	26 ms	50 ms	141 ms
Hybrid QuickSort (cutOff = 5)	2 ms	29 ms	57 ms	140 ms
Median of Three QuickSort	1 ms	28 ms	56 ms	145 ms

	int20k	Int500k	intBig	dutch
3-Way Quicksort	3 ms	31 ms	63 ms	17 ms
InsertionSort	41 ms	13087 ms	53345 ms	12961 ms
MergeSort	2 ms	47 ms	98 ms	36 ms

I found that raising the cutoff value to 50 made the hybrid quick sort faster on average, while dropping it to 5 made it so much slower it actually did worse than the regular quick sort in all the regular datasets. It did however perform better than all but the three-way quicksort in the dutch dataset. It is however worth noting that the three-way quick sort specifically handles datasets with many repeating values like the dutch dataset much better than the rest by design. I also found that the median of three did slightly better than the regular quick sort on average but slightly worse than the hybrid quick sort both of which were within a range of less than 1ms.

The merge sort was slower on average than all the quick sorts in all but dutch where it was only beaten by the three-way quicksort. The insertion sort was decisively slower than the merge sort and all the others in all datasets.

Pathological Sequence Generation

My pathological sequence generation algorithm finds the array's three largest numbers and swaps their positions with that of the elements at the first, middle and last indexes of the array. The median of three partitioning algorithm works with the first, middle and last indexes and so when the median of three partition is executed, the array is divided into two subarrays. One of which contains only two of the biggest elements, and the other with all the rest. This essentially cuts off an entire branch of the recursion tree from an early stage so it takes longer to sort the full array.

To do this, I first went through the array and found the largest element. I swapped this element with the first element in the array. I then went through the array again and found the largest element that was not equal to the first element in the array (remember the first element now holds the largest element in the array) and swapped its position with the middle element of the array. Finally, I found the largest element in the array that was not equal to the first element or the middle element (these now contain the 2 largest elements of the array). I did this instead of finding them all first before swapping because when I tried that sometimes the swaps would mess with the indexes of the other elements and produce an incorrect output.