CSE 2046

ANALYSIS OF ALGORITHMS

HOMEWORK – 1 REPORT

Due Date: 11.05.2022 / 11.59 pm

|  |  |  |  |
| --- | --- | --- | --- |
|  | Student Id | Name Surname | Contribution |
| 1 | 150119033 | Alper Özdemir | Inputs, Report |
| 2 | 150119031 | Eren Başpınar | Code |
| 3 | 150119015 | Emirhan Erdoğan | Code |

PURPOSE and OVERVIEW

Purpose in this project is to analyze different sorting algorithms for finding k’th smallest element in given array. In order to obtain accurate illustration number of trials must be enough and different inputs should be used.

This homework consists of 3 sections: Designing experiment, implementing algorithm, analyzing results. 7 different algorithms will be analyzed during this process and these 3 sections will be applied to all of them:

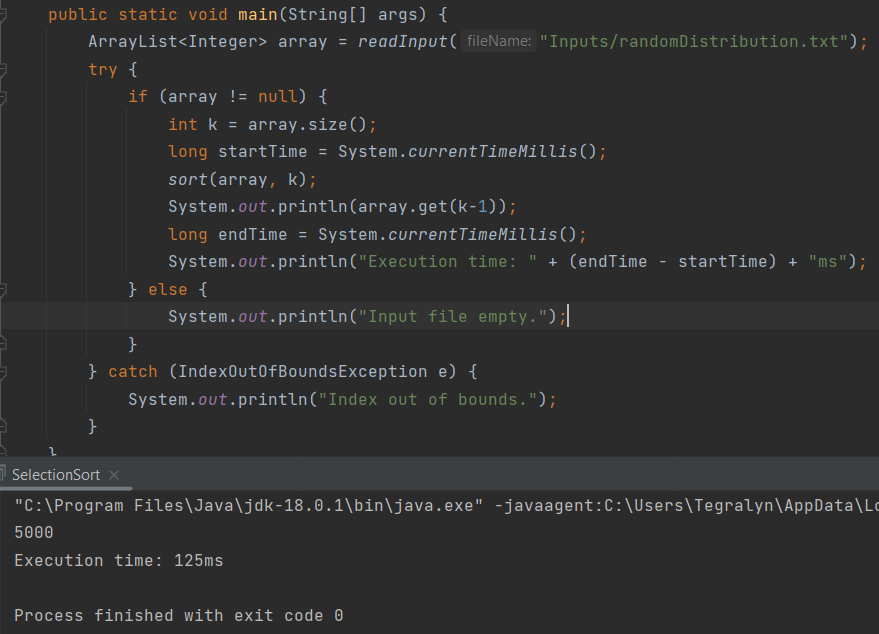
* Insertion Sort
* Merge Sort
* Quick Sort (First element is used as pivot)
* Partial Selection
* Heap Sort
* Quick Select (First element is used as pivot)
* Quick Select (Median of three pivot selection)

INPUTS and METRICS

We chose 13 different inputs for this experiment. Three of them (“ascendingOrder.txt”, “descendingOrder.txt”, “randomDistribution.txt”) should show if distribution of numbers in given array changes algorithm’s efficiency or not. Each of them consists of 5000 numbers so input size does not change. Remaining 10 inputs just vary in terms of input size starting with 1000 to 10000. Also we will make k equal to half of input size. These inputs will be used to obtain input size’s effect on algorithm speed and their quantity is high enough to draw a scatterplot. In the end we will be comparing algorithms’ efficiency by checking distribution’s and size’s effect on runtime speed.

As for the metrics we decided to analyze these algorithms by comparing their runtime speed using *currentTimeMillis()* function in System class (Java). By implementing this function to start and end points of sorting algorithm, results will be obtained in terms of milliseconds but since this function is not completely accurate, we chose to make five trials for each input and take average value as result. Also, this physical timing method may vary due to the CPU so only one computer will be used for all trials.

Example usage of *System.currentTimeMillis()* function:



INSERTION SORT

DATA OBTAINED BY IMPLEMENTATION:

k = N / 2 k = N

* Ascending order: 0ms 0ms
* Descending order: 78.2ms 78.6ms
* Random distribution: 53.2ms 54ms
* N = 1000: 16.8ms 16.8ms
* N = 2000: 28.6ms 28.6ms
* N = 3000: 37.6ms 38ms
* N = 4000: 43.8ms 42ms
* N = 5000: 50ms 49.8ms
* N = 6000: 65.8ms 62.6ms
* N = 7000: 81.6ms 81.6ms
* N = 8000: 108.2ms 108ms
* N = 9000: 131.6ms 134ms
* N = 10000: 151.2ms 150.8ms

COMMENTS:

* Distributions’ effect:

For insertion sort, distribution of numbers effect to run speed is crystal clear. Since the algorithm does not make any change in array for ascending ordered array (for our case it means already sorted) we can take that it is the best case scenario and same logic goes for descending and random ordered arrays, descending is the worst case and random distribution is the average case.

* Input size’s effect:

Obviously increasing the number of inputs makes the algorithm take much longer to sort those inputs and since insertion sort algorithm has O(n2) time complexity we can easily observe the change in time but change in k does not affect the time needed this algorithm to complete since it is not a partial algorithm.

SCATTERPLOT:

MERGE SORT

DATA OBTAINED BY IMPLEMENTATION:

k = N / 2 k = N

* Ascending order: 11.4ms 11.2ms
* Descending order: 10.8ms 11ms
* Random distribution: 12.2ms 10.8ms
* N = 1000: 5.6ms 5.8ms
* N = 2000: 8.4ms 8.8ms
* N = 3000: 9.2ms 8.8ms
* N = 4000: 12.2ms 11.6ms
* N = 5000: 13.8ms 13.2ms
* N = 6000: 16.4ms 16.6ms
* N = 7000: 20.4ms 20ms
* N = 8000: 23.6ms 23.4ms
* N = 9000: 26.4ms 25.4ms
* N = 10000: 28.2ms 28.4ms

COMMENTS:

* Distributions’ effect:

From the data obtained by implementations we can say that distribution of numbers does not effect run speed of merge sort algorithm. k variable does not influence run speed too.

* Input size’s effect:

If we compare run times with previous algorithm (insertion sort) we can see that merge sort is much faster. Also, the increase in input size does not effect run time as much as insertion sort. This is reasonable since merge sort algorithm’s time complexity is O(nlogn). Merge sort is one of the most dependable sorting algorithms if we do not really care about space complexity.

SCATTERPLOT:

QUICKSORT

DATA OBTAINED BY IMPLEMENTATION:

k = N / 2 k = N

* Ascending order: 85ms 89.6ms
* Descending order: 83.2ms 88.8ms
* Random distribution: 10ms 9.2ms
* N = 1000: 4.2ms 4ms
* N = 2000: 5.4ms 5.4ms
* N = 3000: 6ms 5.8ms
* N = 4000: 6.8ms 7.2ms
* N = 5000: 8.4ms 8.2ms
* N = 6000: 9ms 9ms
* N = 7000: 9.2ms 8.8ms
* N = 8000: 11.4ms 10.6ms
* N = 9000: 15.6ms 16ms
* N = 10000: 17.4ms 16.6ms

COMMENTS:

* Distributions’ effect:

This algorithm that we implemented takes the first element of array as pivot point. So, distribution’s affect that we see from data table above makes sense since if the input is already ordered (ascending or descending does not matter) algorithm can not use it’s divide and conquer ability efficiently and run speed becomes even slower than insertion sort.

* Input size’s effect:

Distribution’s affect may vary due to the pivot element selection for this algorithm but if we look at input size’s affect, we can see that it is like merge sort. Also, it is slightly faster but must say it may change due to distribution. Merge sort is more trustworthy algorithm in my opinion.

SCATTERPLOT:

PARTIAL SELECTION SORT

DATA OBTAINED BY IMPLEMENTATION:

k = N / 2 k = N

* Ascending order: 84.2ms 95.8ms
* Descending order: 82.8ms 101ms
* Random distribution: 102.6ms 138.6ms
* N = 1000: 20.8ms 25ms
* N = 2000: 35.6ms 36.6ms
* N = 3000: 42ms 54ms
* N = 4000: 66ms 85.8ms
* N = 5000: 90.8ms 124ms
* N = 6000: 127ms 184ms
* N = 7000: 178ms 234.6ms
* N = 8000: 235ms 282ms
* N = 9000: 276ms 364.2ms
* N = 10000: 364.4ms 404.6ms

COMMENTS:

* Distributions’ effect:

Random distribution does not effect run time speed positively for this algorithm. But for already sorted arrays it does not matter it is ascending or descending. For both of them run time speed is higher. Also k has a huge effect since this is a partial algorithm only sorts the array to the kth number.

* Input size’s effect:

It is slower than merge sort and quick sort for sure. Still it can be faster if k is very small and n is very large. Because with this factors given any partial algorithm will be at advantage since they will not sort whole array. This should be told due to fact that our objective in this report is not to sort the arrays but return the kth smallest numbers in given arrays. Also we can see the increase in time clearly since selection sort has O(n2) time complexity.

SCATTERPLOT:

HEAP SORT

DATA OBTAINED BY IMPLEMENTATION:

k = N / 2 k = N

* Ascending order: 10.2 ms 11.6 ms
* Descending order: 9.6 ms 11.8 ms
* Random distribution: 12 ms 10.4 ms
* N = 1000: 3.8 ms 4 ms
* N = 2000: 5.6 ms 5.4 ms
* N = 3000: 6.8 ms 7.4 ms
* N = 4000: 10.4 ms 9.4 ms
* N = 5000: 12.4 ms 12.2 ms
* N = 6000: 14.2 ms 13.8 ms
* N = 7000: 16 ms 17.4 ms
* N = 8000: 17.2 ms 17.6 ms
* N = 9000: 22 ms 18.8 ms
* N = 10000: 22.6 ms 21.6 ms

COMMENTS:

* Distributions’ effect:

Distribution does not effect run time for this algorithm at all as we see in results above.

* Input size’s effect:

Input size’s affect is not as big as selection an insertion sort as we guessed since heap sort has O(nlogn) time complexity. Also the speed in general is so close to quick sort algorithm which also has O(nlogn) time complexity.

SCATTERPLOT:

QUICKSELECT

DATA OBTAINED BY IMPLEMENTATION:

k = N / 2 k = N

* Ascending order: 142.8 ms 176.4 ms
* Descending order: 111 ms 0.8 ms
* Random distribution: 4 ms 1.2 ms
* N = 1000: 1.6 ms 1.2 ms
* N = 2000: 1.8 ms 1 ms
* N = 3000: 2.4 ms 1 ms
* N = 4000: 2.8 ms 0.8 ms
* N = 5000: 3.8 ms 2.2 ms
* N = 6000: 3.4 ms 3 ms
* N = 7000: 5.6 ms 2.6 ms
* N = 8000: 5.6 ms 4 ms
* N = 9000: 3.2 ms 3.2 ms
* N = 10000: 5.2 ms 5.4 ms

COMMENTS:

* Distributions’ effect:

Random distribution is by far the best case in terms of distribution since pivot selection effects it same as quick sort. k’s affect is reverse compared to the partial algorithms.

* Input size’s effect:

This is one of fastest algorithms as expected. It does not have to sort the array, all it does is return the k’th smallest element. If we do not expect to need sorted version of given input using quick select is a wise option. Also we can see for some inputs that when input size increased run speed gets higher in data table but it is because of the values being so small so that *System.current.millis()* function does not do the best job giving accurate results.

SCATTERPLOT:

QUICKSELECT (Median of Three Pivot Method)

DATA OBTAINED BY IMPLEMENTATION:

k = N / 2 k = N

* Ascending order: 130.6 ms 3 ms
* Descending order: 4.2 ms 2.8 ms
* Random distribution: 6.6 ms 2.8 ms
* N = 1000: 1.6 ms 1.6 ms
* N = 2000: 1.8 ms 1.4 ms
* N = 3000: 3.8 ms 1.8 ms
* N = 4000: 3.6 ms 2.2 ms
* N = 5000: 6.4 ms 2.6 ms
* N = 6000: 4.6 ms 3 ms
* N = 7000: 9 ms 3.2 ms
* N = 8000: 9.8 ms 3.8 ms
* N = 9000: 9.8 ms 4.2 ms
* N = 10000: 4.6 ms 4.2 ms

COMMENTS:

* Distributions’ effect:

In contrary to the quick selects first pivot selection method this median of three methods’ descending order type input also can be classified as the worst case looking at the data table above and yet k’s affect is the same.

* Input size’s effect:

It increases logarithmically unlikely to selection and insertion algorithms. Run speed is as fast as quick sort and merge sort yet the things understood in quick select first pivot selection method goes for also in this method. It can be efficient if we do not expect to use the array again (sorted format) otherwise it is not logical to use this algorithm.

SCATTERPLOT:

CONCLUSION

In the end we saw all given 7 algorithms run time efficiency using *System.current.millis()* function yet we can not say that %100 percent one algorithm is better than the other. Everyone of them has their advantages and disadvantages. As example implementing selection sort is much more easy than implementing quick select but as it comes to run speed quick select is better almost every situation but again if we want to use sorted array again selection might be a better option.

Still if we compare their run speed using our data tables this would be the result; (For every algorithm k = N run time speed is taken)