Project 2

**Introduction**

This project implements quicksort, heapsort, and introsort. It also analyzes their

empirical run times compared to their worst case.

A git repo of the project is at: https://github.com/jmr172/algorithms/tree/master/Project\_2

This project was made with Java 8. It can be run from the command line using

the following from within the src directory:

```

$ javac Project2.java && java Project2 ../input/in.txt ../output/out.txt; cat ../output/out.txt

```

\* in.txt is the input for the closest pairs.

\* input files with 10, 100, 1000, and 10000 data points are provided.

\* out.txt contains the output of the closest pairs calculation if necessary

All my data was generated by the python script I wrote called random\_data\_generator.py.

To run this program, use:

```

python3 random\_data\_generator.py [number\_of\_points]

```

**Intro to Introsort**

Part 1 Pseudocode and run-time analysis:

Quicksort:

// Reference: https://courses.cs.washington.edu/courses/cse373/01sp/Lect18\_2up.pdf

// Java is pass by reference, so all operations are done in place on the original array

Quicksort is a divide-and-conquer approach to sorting an array. The first step is to pick an element in the array called the pivot. In our case, we always choose the last element. Next, we move every element with a value less than the pivot to the left of the pivot and every element with a value greater than the pivot to the right of the pivot. This guarantees that the pivot is in the correct position. Then we call quicksort on the sub-array that is left of the pivot and the sub-array that if right of the pivot. Since this is all done in place, there is no rebuilding step afterwards.

The partitioning step requires iteration through the entire subarray that was passed. This iteration will be for n elements the first time. It will also be for n elements when you sort the left sub-array and the right sub-array. Falling this logic, the performance is dependent on how many recursives call must be made. In the absolutely worst case, quicksort will produce always produce a sub-array that is of size 1 less than was passed to quicksort, resulting in n iterations n times which results in O(n^2). This will happen when the array is sorted in descending order. It will move the pivot to the very front of the sub-array each time, resulting in the maximum amount of recursive calls.

Global int[] input

Quicksort(int x, int y):

// Base condition to escape recursive call

If (x > y):

Return

Else:

int sorted\_index = partition(x, y)

quicksort(x, sorted\_index – 1)

quicksort(sorted\_index + 1, y)

partition(int x, int y):

// choose pivot as last element

Int pivot = input[y]

// keep track of where the pivot will end up

Int index;

For val in sub\_array:

If val < pivot:

// move val to front

Index++;

// move pivot to index

Swap(index, y)

Return (index)

Heapsort:

// Referenced a slide deck from RIT <https://www.cs.rit.edu/~lr/courses/alg/student/1/heapsort.pdf>

Heapsort consists first of building a max heap in O(n) time. Then we remove the root element in O(1) time and recreate the max heap in O( lg n) time. We repeat this process n times. This results in a worst case of O(n) + O(n lg n) -> O(n lg n).

Global int[] input

heapsort():

// build max heap

Heap = Build\_max\_heap(input)

For (int i = input.length; I > 0; i++):

Input[i] = heap.root

Heap = Heapify(heap)

heapify(heap):

if root.left > root.val:

swap(root, root.left)

heapify(heap)

return

if root.right > root.val:

swap(root, root.right)

heapify(heap)

return

Introsort:

// Reference: https://en.wikipedia.org/wiki/Introsort

Introsort combined quicksort, heapsort, and insertion sort. We choose a minimum array length and once the subarrays reach the threshold, insertion sort is used to sort the sub array. We use quicksort until the number of recursive calls have passed another threshold, which we then switch to heapsort. Heapsort is not recursive, so it limits the memory usage of the sorting algorithm.

Quicksort is normally O(n log n) but has a worst case of O(n^2). The worst case is caused by doing n recursive calls and making minimum progress on each. This is addressed by putting an upper limit on the number of recursive calls allowed before switching to heapsort. The worst case for heapsort is O(n log n). Insertion sort’s worst case is O(n^2).

If n is less than the threshold, only insertion sort will be called, resulting in a worst case of O(n^2). That isn’t very practical because that would limit it to very small data sets. When n becomes large, the complexity is dominated by quicksort and heapsort, resulting in O( n log n).

// Pseudocode references quicksort and heapsort functions from previous pseudocode

Global int[] input

introsort(int x, int y, recursive\_count):

// build max heap

If ( y – x < insertion\_limit) :

Insertion\_sort(x, y)

elif (recursive\_count == 0):

heapsort()

else:

int sorted\_index = partition(x, y)

introsort(x, sorted\_index – 1, recursive\_count - 1)

introsort(sorted\_index + 1, y, recursive\_count - 1)

Runtime analysis:

Below is a table of the number of comparisons on n sized data sets sorted in descending vs random order:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Random order | | | | | |
| Input Size | 10 | 100 | 500 | 1000 | 5000 | 10000 |
| Quicksort | 19 | 645 | 5102 | 10963 | 68702 | 164619 |
| Heapsort | 33 | 629 | 4266 | 9579 | 59599 | 129123 |
| Introsort | 18 | 668 | 5505 | 11626 | 72224 | 172393 |
| n log n | 33 | 664 | 4483 | 9966 | 61439 | 132877 |
| n^2 | 100 | 10000 | 250000 | 1000000 | 25000000 | 1E+08 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Descending order | | | | | |
| Input Size | 10 | 100 | 500 | 1000 | 5000 | 10000 |
| Quicksort | 45 | 4950 | 124750 | 499500 | 12497500 | 49995000 |
| Heapsort | 26 | 566 | 3926 | 8816 | 55936 | 121696 |
| Introsort | 45 | 1617 | 11698 | 26575 | 175584 | 381275 |
| n log n | 33 | 664 | 4483 | 9966 | 61439 | 132877 |
| n^2 | 100 | 10000 | 250000 | 1000000 | 25000000 | 1E+08 |

All three sorting algorithms run in O(n log n) on random data, but this clearly falls apart when the data is sorted in descending order. This is the worst case for quicksort, and it always takes 0.5\*n^2 comparisons. Introsort also takes a small performance hit on descending order data due to utilizing quicksort at first, but it avoids losing the asymptotic relationship to O(n log n) even in this case. This point is illustrated in the following graphs:

From the graph, it is clear that on random data, they all asymptotically track n log n. However, for descending data both linearly and logarithmically:

Quicksort falls away from the others and begins asymptotically tracking n^2. The graph illustrates that introsort does use more comparisons than heapsort in this case, but it does still track n log n.

**Median-of-Three Partitioning**

Part 1 Pseudocode for median-of-three in the partition function

Median-of-three is the strategy of choosing a pivot that is the median out of the first value in the array, the last value in the array, and the middle value in the array. This would replace the int pivot = input[y] line of the partition function.

partition(int x, int y):

// choose pivot as median of three

Int pivot = median(input[x], input[y], input[y-x])

swap(y, median\_val)

// keep track of where the pivot will end up

Int index;

For val in sub\_array:

If val < pivot:

// move val to front

Index++;

// move pivot to index

Swap(index, y)

Return (index)

Median-of-three partitioning adds a constant time operation to every partition function call, but prevents the worst case scenario of always choosing the largest element. Since the worst case is elinimated, the runtime of median-of-three is O(n log n).

Runtime analysis:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Random order | | | | | |
| Input Size | 10 | 100 | 500 | 1000 | 5000 | 10000 |
| Quicksort | 25 | 784 | 5661 | 12056 | 77795 | 175617 |
| Introsort | 18 | 662 | 5378 | 11516 | 73926 | 168211 |
| n log n | 33 | 664 | 4483 | 9966 | 61439 | 132877 |
| n^2 | 100 | 10000 | 250000 | 1000000 | 25000000 | 1E+08 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Descending order | | | | | |
| Input Size | 10 | 100 | 500 | 1000 | 5000 | 10000 |
| Quicksort | 43 | 1571 | 19503 | 66259 | 1306385 | 5040571 |
| Introsort | 45 | 1325 | 10431 | 24079 | 163755 | 359214 |
| n log n | 33 | 664 | 4483 | 9966 | 61439 | 132877 |
| n^2 | 100 | 10000 | 250000 | 1000000 | 25000000 | 1E+08 |

Quicksort and introsort took a small performance loss in the average case as shown by the random order tests, but both improved when ran on lists sorted in descending order. Specifically, quicksort was taking 0.5\*n^2 comparisons and it gained a factor of 10 increase in performance. In practice, it is still quite a bit above n log n as illustrated in the following graphs.

In conclusion, quicksort outperforms introsort when operating on random data, but introsort dominates quicksort when operating on descending order data with or without the median-of-three implementation.