190004947

Practical 1 - Algorithms

CS3052 - Computational Complexity

Task 1 - First Implementation

My task 1 implementation is written in Java since I have the most experience writing algorithms that map standard input to standard output in Java. The stacscheck tests all passed with my implementation.

The main function uses a buffered reader to take input in which is then parsed into a two-dimensional array (matrix) of points with the columns being x and y values of each point. The ClosestPairs function is then called with this matrix. First, ClosestPairs uses QuickSelect to find the median of the x-coordinates. My QuickSelect function takes advantage of the dynamic storage of ArrayLists to build lists of elements greater than and less than a pivot chosen at random. These ArrayLists are then converted to arrays for the recursive calls. After the median is returned to the ClosestPair algorithm, two arrays of equal size are initialised and filled with points with x-values greater than and less than the median; these arrays represent both sides of the problem. Recursive calls are then made on both sides of the problem until the base case is reached. The base case in my ClosestPairs function is when the input number of points is less than 4, in which case, the BaseCase function is run on those few points. The BaseCase function is a brute force method to calculate the shortest distance between any pair of points. It takes advantage of the distance utility function I wrote which simply returns the distance between two points in a two-dimensional Cartesian space. After the recursive calls in ClosestPairs, we set $d = min(d_l, d_r)$ where d_l is the shortest distance found in the left half of the plane and d_r is the shortest distance found in the right half of the plane. Then two arrays are initialised and populated to store the left and right strips of points within d to the splitting line. The points in these arrays are then sorted by y-coordinate using my sortByY method before the crossStripClosestPair function is called.

My crossStripClosestPair method iterates through both strips taken in as parameters and returns the shortest distance between any two points that are separated by the splitting line (or d if d is shorter). This looks like an $O(n^2)$ operation but a maximum of 6 comparisons are made and so the function is $O(6n)$ which is equivalent to $O(n)$. The output of this method is the solution to the closest pairs problem. This solution is currently of the Double type which needs to be rounded to 9 significant figures. I did not know how to round to a certain number of significant figures using standard Java libraries so I wrote a method called round which takes in two parameters: the number to be rounded and an int representing the desired number of significant figures. After some tweaking and adding special case responses, the function correctly returns a rounded figure. The only issue I found is when running the structured.in test, the shorted distance is returned as 9.14813736e-4 instead of 0.000914813736.

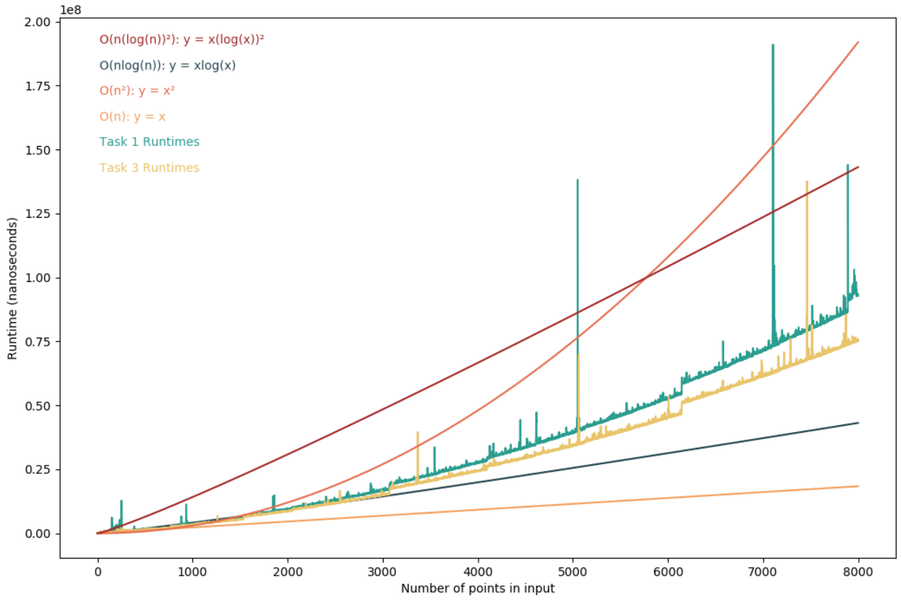
Task 2 - Analysis

My part 1 implementation comprises of four core methods and three utility functions. Starting with the helper methods, my SortByY uses Java's Arrays.sort built in function which is described as a stable, adaptive, iterative mergesort which is known to be $O(nlog(n))$. The distance function is $O(1)$ since it has no dependency on input size. Next, the rounding function works out to be $O(2n') = O(n')$ where n' is the length of the double passed in. This is because I iterate through the digits twice. As we can see, the limiting factor of the utility functions is the $O(nlog(n))$ sort.

Now for the core methods, BaseCase looks like $O(n^2)$ because of the nested for loop, however, only 3 or less values are ever passed in at once and BaseCase can be considered constant or $O(1)$. QuickSelect has an average time complexity of $O(nlog(n))$ due to the input size in the recursive calls being approximately half the size of the parent call. However, in the worst case we can hit $O(n^2)$ when the input size of successive recursive calls only decreases by one each time. My implementation chooses a random pivot in each call which makes the probability of hitting the worst-case extremely low and unrealistic for large n. Choosing a random pivot yields almost certain linear time[1]. I investigated how implementing PivotSelect for QuickSelect would affect runtime by avoiding the worst case but the theoretical improvement (produces the recurrence equation $QS(n) \leq QS(2n/3) + O(n)$ which falls into case 3 of the master theorem and so $QS(n) \in O(n)$) didn't translate well to practical results and so I left the function in the submission but didn't end up using it for pivot selection. The crossStripClosestPair method proves to be $O(6n) = O(n)$ as described in the previous section.

Now for the ClosestPair function, we have multiple single-layer loops through the input so it is at least $O(n)$. There are also two calls to sortByY which we know to be $O(nlog(n))$. Using this as the limiting factor, we consider the recursive calls on inputs of $n/2$ in the recurrence equation $C(n) = 2C(n/2) + O(nlog(n))$ where $C(n)$ is the time to find the closest pair among n points. The $O(nlog(n))$ factor includes the time to find the median, split the points, sort by y-values, and do the final check. Here, we are in case 2 of the master theorem with $k = 2$ and so $C(n) = \Theta(n(log(n))^2)$ which is clearly $o(n^2)$.

Once my implementation was complete, I wrote a function to generate random inputs of increasing size and used python to plot these values on a graph. Anomalies plagued my results and so I re-wrote my program to take the average runtime of five runs for each input size from 1 to 8000. This was not a quick computation and so I automated the data collection process and let it run for an hour. These experimental results can be seen in the plot below.



The plot includes the average runtimes for both task 1 and task 3 which we will come back to. The other lines plotted are $y = x(log(x))^2$, $y = xlog(x)$, $y = x^2$, and $y = x$. These lines clearly show that the runtime trend increases with n at a faster rate than both $nlog(n)$ and n itself (complexity is $\omega(n)$). Additionally, the plot obviously depicts n^2 growing at a much faster rate than the runtimes (complexity is $o(n^2)$).

There were still anomalies in my results despite my efforts to combat this however an interesting result can be seen where the number of points is 5000. Both implementations struggled with this input size much more than both smaller and larger inputs. After searching for a culprit, I am not sure why this happened.

Task 3 - Further Implementations

In task 3, I took the call to QuickSelect out for finding the median and replaced it with an $O(1)$ call to the central element in the list of points sorted by x. I also upgraded the algorithm by avoiding sorting by y-value in each recursive call. I achieved this by giving ClosestPairs two parameters: a list of the points ordered by x-value and the same list of points but sorted by y-value. This ordering is maintained throughout the recursion. To do this sorting I updated by sortByY method to be a sortByColumn method which can sort by either x or y depending on the parameters passed in. Since the sorting was the limiting factor, we should see a clear complexity improvement over the first implementation. This can be seen in the plot above; task 3 runtimes are very close to the task 1 runtimes for small n. However, after approximately 3000 points in the input, the algorithm upgrades start to show and the task 1 runtimes line pulls away from the task 3 runtimes line. If I had access to more computing power I would have collected experimental results from a larger range of inputs to more clearly show the difference for large n.

References

[1] https://en.wikipedia.org/wiki/Quickselect#Variants