Summary Report Lab Assignment 1

Christian Heinrich

General Overview:

I chose to use Java in order to work on this lab, as that is what the majority of my other classes have been using, but at the same time isn’t what I am the most used to using, so I wanted to become more comfortable with the language.

In order to run the program while specifying the size of the data set, pass in the desired data set size through the use of command line arguments, with the command

Javac Quicksort.java or ClosestPoints.java to compile, then

Java Quicksort n or Java ClosestPoints n to run with the desired data size

In order to switch between the randomized set or the poisson set for problem 1, have whichever data set creation is desired uncommented and the other commented inside main.

Problem 1:

I chose to implement the randomized divide and conquer Quicksort algorithm from chapter 13.5 of the book for problem 1, as it seemed to improve the worst case performance of Quicksort in a simple but elegant manner. Essentially the Algorithm seeks to avoid the worst case of Quicksort by forcing the split to be at a relatively median value, rather than a purely random one. The Pseudocode for the algorithm is:

Modified Quicksort(S):

If |S| <= 3

Sort S

Output the sorted list

End If

Else

While no central splitter has been found

Choose a splitter ai ε S uniformly at random

For each element aj of S

Put aj in S- if aj < ai

Put aj in S+ if aj > ai

End For

If |S-| >= |S|/4 and |S+| >= |S|/4 then

Ai is a central splitter

Endif

Endwhile

Recursively call Quicksort(S-) and Quicksort(S+)

Output the sorted set S-, then ai, then the sorted set S+

End if

Although, this pseudo code only works with a set of distinct values, an issue which I ended up running into. If the data set includes multiples of the same value, the unmodified version of this algorithm will actually discard them, as they will be added to neither side of the split. In order to allow for multiples of the same value, which is a potential set of data that might be encountered, I changed the algorithm so that values less than or equal to aj would be added to S-. However, this introduced a new issue, as now there was the possibility of the algorithm reaching an infinite loop, as there would be no possible way to split the list to allow for a central splitter, as all values equal to aj would be added to only one side, so if aj was in a set with four or five of the same value, the program would just hang. In order to solve this, I forced the algorithm to treat aj as always being in the center of a subset of equivalent value, by alternating which side values equivalent to aj would be added to, thus maintaining the intended functionality. I opted to store the split subsets using List<Integer> as the actual size of each data set is unknown, so a data structure with variable size is required in order to properly distribute the values.

Final Data Table Problem 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data Size:** | **1000** | **10K** | **50K** | **100K** | **500K** |
| Average Runtime Random Set | 12.6ms | 62.2ms | 96.3ms | 122.8ms | 426.4ms |
| Average Runtime Poisson Set | 16.2ms | 56.4ms | 99.7ms | 145.2ms | 921.8ms |

My final results for this algorithm showed that Poisson distributed sets were generally slower than a fully random set, and this only increased with the size of the set, leading to the extreme increase in processing time for a 500K size set with Poisson distribution when compared to the fully random set, with nearly double the average time taken. Furthermore, the time taken was far more standardized compared to the fully randomized set, with all of the runs generally staying around the same time taken, while the fully random set had some larger discrepancies between individual runs.

Problem 2:

The algorithm used for problem 2 was:

Closest-Pair(P)

Construct Px and Py

(P0\*, P1\*) = closest-Pair-Rec(Px,Py)

Closest-Pair-Rec(Px, Py)

if P.size <=3

find closest pair by measuring all pairwise distances

endif

construct Qx, Qy,

(Q0\*, Q1\*) = Closest-Pair-Rec(Qx, Qy)

(R0\*, R1\*) = Closest-Pair-Rec(Rx, Ry)

distance delta = min(d(Q0\*, Q1\*), d(R0\*, R1\*))

X\* - maximum x-coordinate of a point in set Q

L = {(x,y) : x = x\*}

S = points in P within distance delta of L

Construct Sy

for each point s E Sy, compute distance from s

to each of next 15 points in Sy

Let s, s' be pair achieving minimum of these distances

if d(s,s') < delta

return (s,s')

else if d(Q0\*, Q1\*) < d(R0\*, R1\*) then

return (Q0\*, Q1\*)

else

return (R0\*, R1\*)

Endif

Which will divide the given set of Points into two subsets, one right and one left of the center, and then find the closest pair on their respective sides via recursion. As such I opted to utilize mostly arrays, as the length of the data structure would always be known. However, I opted to add an additional Pair class in order to save specific pairings of points, and to be able to return multiple values from a single function, as this is the only way to accomplish that. Each Pair object stores the two points which makes up the pair, as well as the method to calculate the distance between the two.

Final Data Table Problem 2:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data Size:** | **1000** | **10K** | **50K** | **100K** | **500K** |
| Average Runtime Poisson Set | 26.6 | 197.7 | 563 | 793.2 | 3010.8 |

The data for problem 2 shows a significant increase in time taken as the data size increased, far more so than for problem 1. Unfortunately I was unable to identify what was causing the large increase in CPU time.