AMS 545 Optional Project
KD Tree Range Search

Wenhan Gao

1 Introduction

Kd-trees for points in the plane will be implemented. The input will be a set of n points, $S = \{p_1, p_2, \ldots, p_n\}$, given by mouse clicks. The users also have the option to generate n random points. The program allows one to specify a rectangle, $[x, x'] \times [y, y']$, by mouse input from the user. The output, then, will be a listing of the points that are within the query box.

The following experiment will be performed. For $n = 128$, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, a set S of n random points in the unit square will be generated. The kd-tree for each S will be built. The time it takes to answer a query Q given by a 0.1×0.1 box rectangle: $[0.45, 0.55] \times [0.45, 0.55]$ will be recorded. For each n, the experiment will be repeated for 30 times and the average will be taken. In this report, we compare the speed of the kd-tree method to the "brute force" method of simply testing all n points, one by one, to see which ones are in the query box. Moreover, for the timing experiments, "point search" problem will also be tested. Given a point q in the domain, it is asked to determine if this point is in the set S. This can also be viewed in a sense that the query rectangle is of the form $[x, x] \times [y, y]$; its left bottom corner and right top corner coincide.

Kd-trees will also be used on an input set of axis-aligned rectangles so that those within a query box can be reported/counted. This can be viewed as a range search problem in 4d. Each rectangle with left bottom corner (x_0, y_0) and right top corner (x_1, y_1) can be viewed a point (x_0, y_0, x_1, y_1) in $\mathbb{R}^4$. The query rectangle with left bottom corner (x_{q0}, y_{q0}) and right top corner (x_{q1}, y_{q1}) can be viewed as a query box in $\mathbb{R}^4$: $(x_{q0}, x_{q1}) \times (y_{q0}, y_{q1}) \times (x_{q0}, x_{q1}) \times (y_{q0}, y_{q1})$. A rectangle lies fully in the query rectangle if and only if the corresponding point in $\mathbb{R}^4$ is in the $\mathbb{R}^4$ query box. Therefore, this problem can be mapped to 4d range search problem. In the following content, we will refer to an input rectangle as a point (in 4d) for simplicity, and rectangle refers to the query box.

2 Pseudo-code and Computer Programs

This project is written in Python 3. There are two programs with GUIs; one for taking points as inputs and the other for taking rectangles as inputs. There is also a Jupyter Notebook(Python) file for execution time measurements and plotting. A markdown file user guide is also included.

Here, we present the pseudo-code of the range search KD-tree construction and querying respectively in Algorithm 2.1 and Algorithm 2.2. The construction is done in a recursion; the query is also done in a recursion. We define a class/object called Node(), which has three attributes:

- val: value of the node. A number for non-leaf nodes, and a point for leaf nodes.
- left: a node that is the left child of current node.
- right: a node that is the right child of current node.

Algorithm 2.1: buildKdTree(node, points, depth, k)

```
Result: node
Require: node, points, depth, k (dimension of the an input point)
if len(points) == 0 then
    return;
else if len(points) == 1 then
    node.val := points[0] ;
    return node;
else
    current dimension := depth mod k ;
    split_val := median of the values of points in current dimension;
    node.val = split_val;
    left_list, right_list := split points into two sets based on split_val and current dimension;
    node.left = Node();
    node.right = Node();
    node.left = buildKdTree(node.left, left_list, depth+1, k);
    node.right = buildKdTree(node.right, right_list, depth+1, k);
end
```

Algorithm 2.2: queryRecursion(node, rectangle, output, depth, k)

```
Result: resulting points stored in an array
Require: node, rectangle, output, depth=0, k
if node.val == None then
    return;
if node is a leaf node and the point node.val is in the rectangle then
    append node.val to output array ;
    return;
current dimension = depth mod k;
if the left region of current node intersect the rectangle in the current dimension then
    queryRecursion(node.left, rectangle, output, depth+1, k);
if the right region of current node intersect the rectangle in the current dimension then
    queryRecursion(node.right, rectangle, output, depth+1, k);
```

3 User Guide and Program Demo

3.1 Range Search With 2D Points

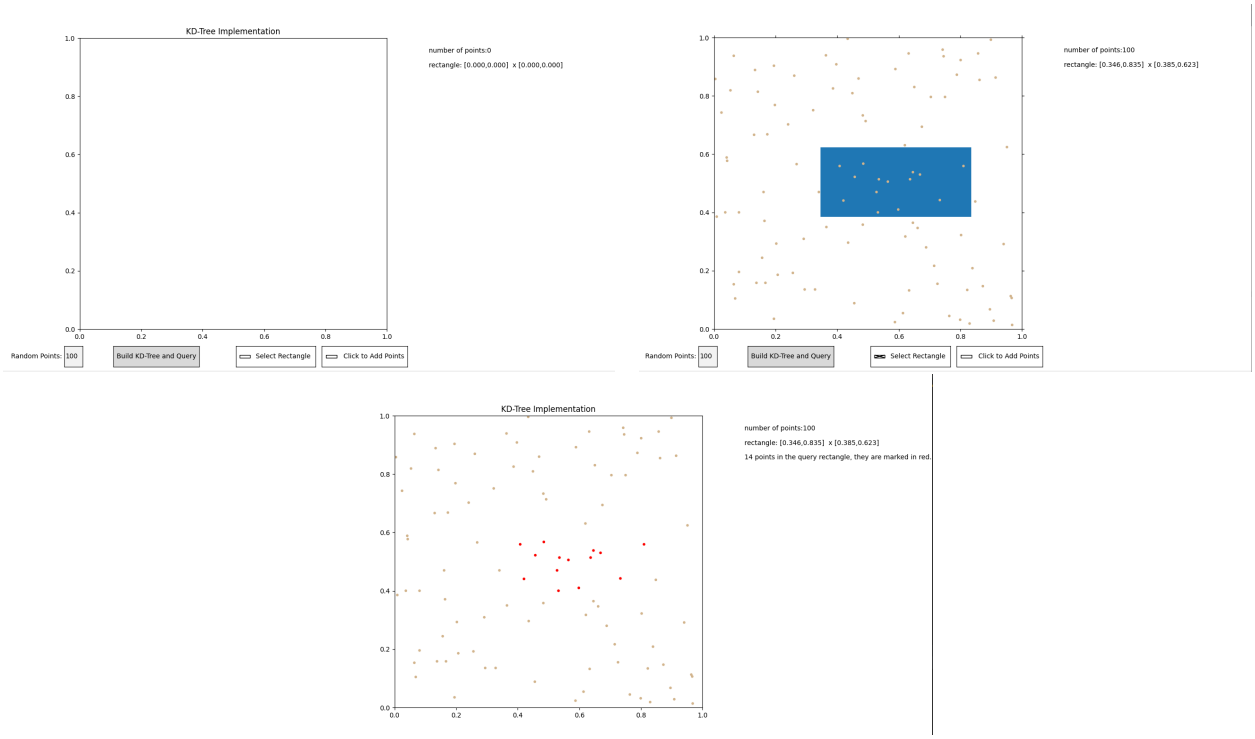
Upon running the program (KD_Tree_GUI.py), a GUI will pop up, and the GUI is quite self-explanatory. The are 4 options in the GUI:

- A check box that allows you to add a point by clicking.

- A check box that allows you to select a rectangle for querying.
- A input box where you can input an integer to add random/uniform points.
- A button to stop adding points or selecting query rectangle and start building the KD Tree and outputting points that is in the query rectangle.

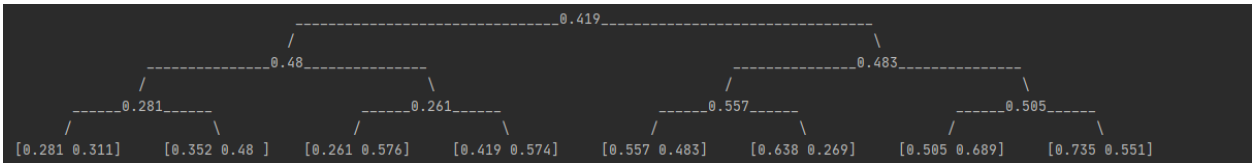
Note that to generate random points, both check boxes have to be unchecked. Also, only one check box can be selected at the same time to avoid conflicts in functionality.

Figure 1: Demo of the GUI (zoom in for a clearer view)



Another function of this program is that, when the input set of points is small (less than or equal to 8), the console will actually print out the KD tree. If the set of points is large, due to display issues, the KD tree will not be displayed. An example is shown below. Note that, in the display, the point values will be truncated to have only 3 floating digits for the purpose of presentation. The original number is in double precision (64 bits) in python.

Figure 2: KD Tree Demo

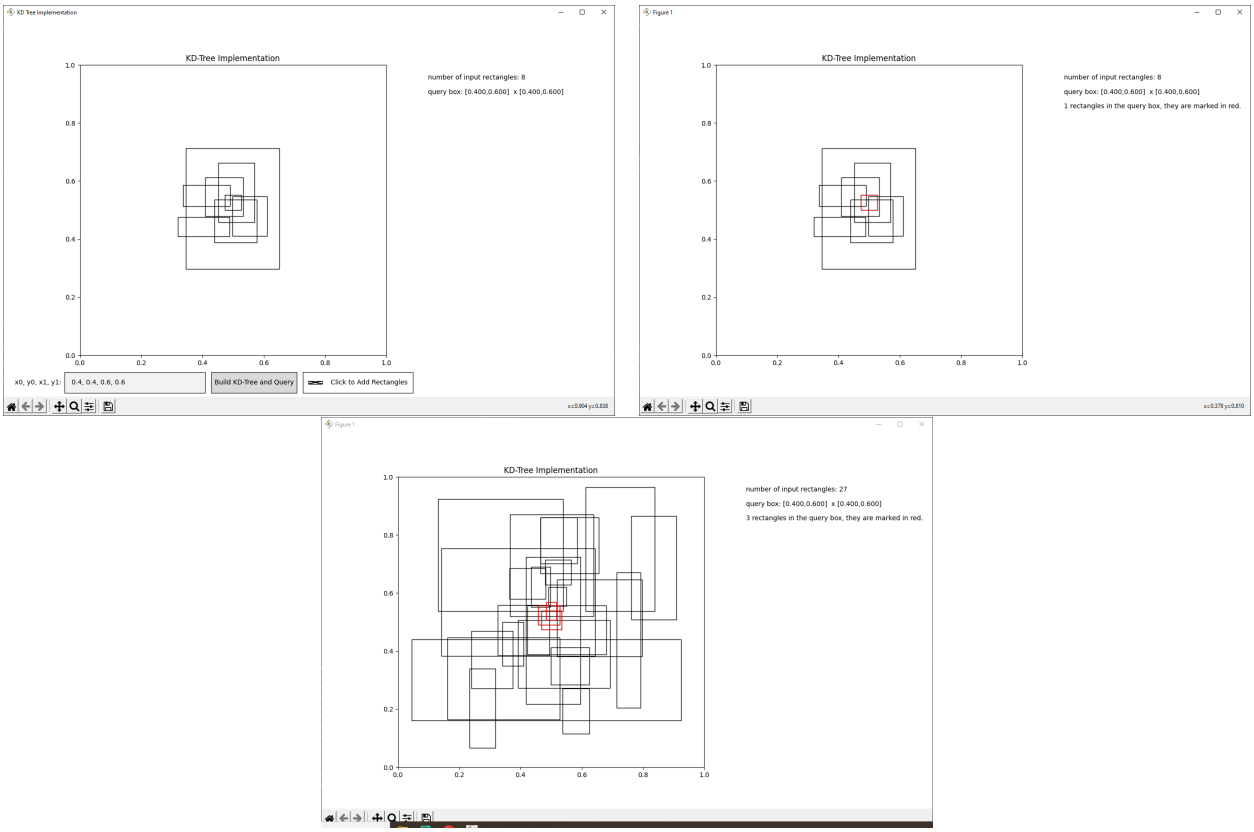


3.2 Range Search With 4D Points (Rectangles as Input "Points")

Upon running the program (KD_Tree_GUI_4D.py), a GUI will pop up, and the GUI is quite self-explanatory. The are 3 options in the GUI:

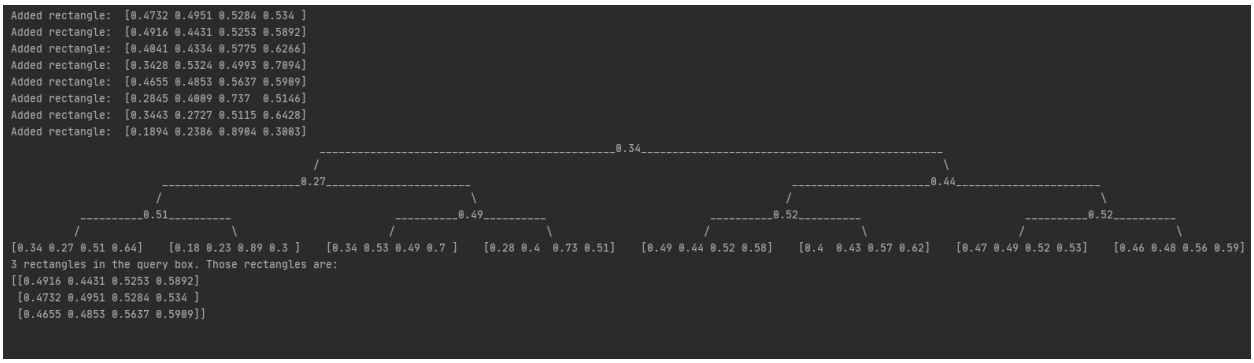
- A check box that allows you to add an input rectangle by clicking.
- A input box that allows you to specify a query box.
- A button to stop adding input rectangles or selecting query box and start building the KD Tree and outputting rectangles that is in the query box.

Figure 3: Demo of the GUI (zoom in for a clearer view)



Another function of this program is that, when the input set of points is small (less than or equal to 8), the console will actually print out the KD tree. If the set of points is large, due to display issues, the KD tree will not be displayed. An example is shown below. All values will be truncated to have only 3 or 2 floating digits for the purpose of presentation. The original number is in double precision (64 bits) in python. It is important to note that when a floating number is truncated, the original double precision number created by clicking will be greater than the truncated value, for example, if 0.6 is displayed, then the original number is greater than 0.6.

Figure 4: KD Tree Demo in 4D; Query Box: $[0.4, 0.6] \times [0.4, 0.6]$

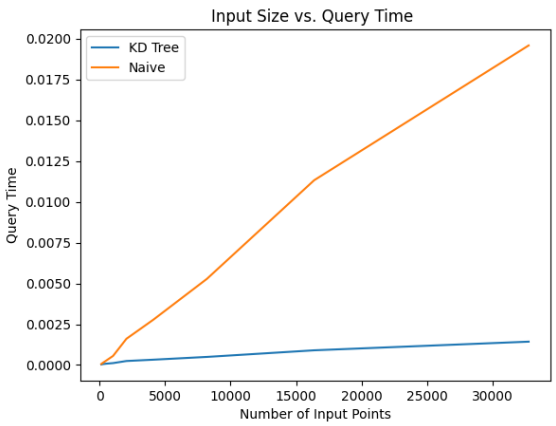


Note: numbers truncated for displaying purposes

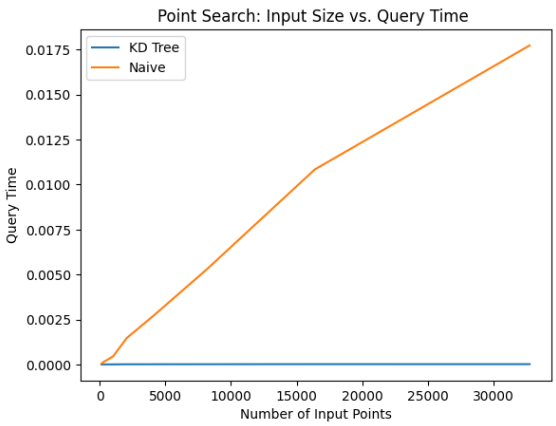
4 Timing Results

For $n = 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768$, a set S of n random points in the unit square will be generated. The kd-tree for each S will be built. The time it takes to answer a query Q given by a 0.1×0.1 box rectangle: $[0.45, 0.55] \times (0.45, 0.55)$ will be recorded. For each n, the experiment will be repeated for 30 times (in each trial, for randomness, the set S will be regenerated) and the average will be taken. KD tree results will be compared with that of the naive method of looping through all input points. In addition, we perform point search (search if a point in the set of points S) to compare KD tree with the naive method. Numerical timing results (in seconds) are provided in Appendix A; here we present the timing plots.

Figure 5: Query Time Comparison



(a) Range Search



(b)Point Search

For reference, the query time for KD-tree is $O(\sqrt{n}+k)$, where k is the number of points in the query rectangle. Since the query rectange is a fixed 0.1 by 0.1 box, and the input points are random, the expected size of k is $0.1n$. Therefore, when n is large, k will dominate $\sqrt{n}$ in expectation; KD tree query time is expected $O(\frac{1}{10}n)$ when n is large. The naive method query runs in $O(n)$ time.

However, if we consider the point search problem, the KD-tree querry runs in $O(\sqrt{n})$ time whereas the naive method still runs in $O(n)$ time.

As shown in the plot, the experimental results is consistent with the theoretical complexity analysis. Additional plots are included in Appendix B

5 Conclusion

In this work, we implemented KD Trees in 2D and 4D respectively. The numerical time complexity is consistent with the theoretical time complexity. Overall, the KD Tree data structure is efficient and easy to implement.

Acknowledgement

Some portion of the code is adapted from open source code on StackOverflow under the terms of the Creative Commons license. Specially, the implementation of the median of medians algorithm and the implementation of printing out a balanced binary search tree are adapted. It is also noted in the code comments.

Appendix A Timing Results

Table 1: Range Search: Average Query Time

	KD tree	Naive Method
$N = 2^7$	3.316×10^-5	6.532×10^-5
$N = 2^8$	5.747×10^-5	1.410×10^-4
$N = 2^9$	8.174×10^-5	2.759×10^-4
$N = 2^{10}$	1.165×10^-4	5.560×10^-4
$N = 2^{11}$	2.456×10^-4	1.608×10^-3
$N = 2^{12}$	3.256×10^-4	2.772×10^-3
$N = 2^{13}$	4.959×10^-4	5.277×10^-3
$N = 2^{14}$	9.009×10^-4	1.132×10^-2
$N = 2^{15}$	1.430×10^-3	1.958×10^-2

Time to build the KD Tree is (in ascending order for input sizes from 2^7 to 2^{15}):

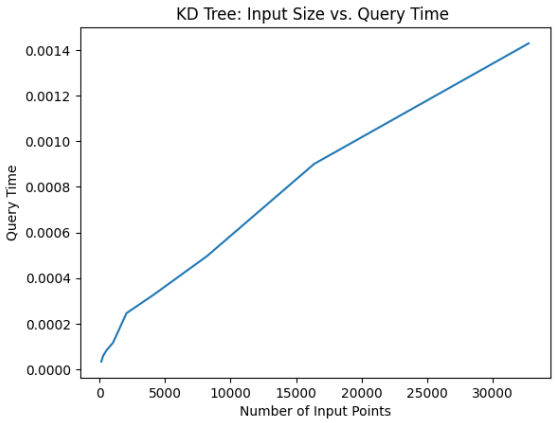
$$\begin{bmatrix}0.002366 & 0.007223 & 0.01139 & 0.02325 & 0.07923 & 0.1508 & 0.3302 & 0.6345 & 1.216\end{bmatrix}$$

Appendix B Additional Timing Plots

Table 2: Point Search: Average Query Time

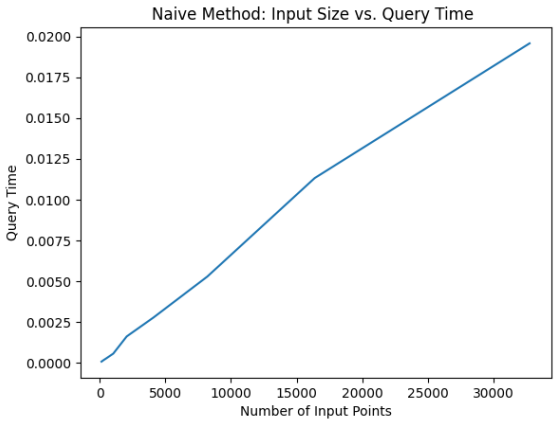
	KD tree	Naive Method
$N = 2^7$	8.66×10^{-6}	5.81×10^{-5}
$N = 2^8$	1.32×10^{-5}	1.34×10^{-4}
$N = 2^9$	1.23×10^{-5}	2.42×10^{-4}
$N = 2^{10}$	1.30×10^{-5}	4.68×10^{-4}
$N = 2^{11}$	1.94×10^{-5}	1.47×10^{-3}
$N = 2^{12}$	2.00×10^{-5}	2.72×10^{-3}
$N = 2^{13}$	2.19×10^{-5}	5.32×10^{-3}
$N = 2^{14}$	2.60×10^{-5}	1.08×10^{-2}
$N = 2^{15}$	2.54×10^{-5}	1.77×10^{-2}

Figure 6: Plot: Range Search Query Time



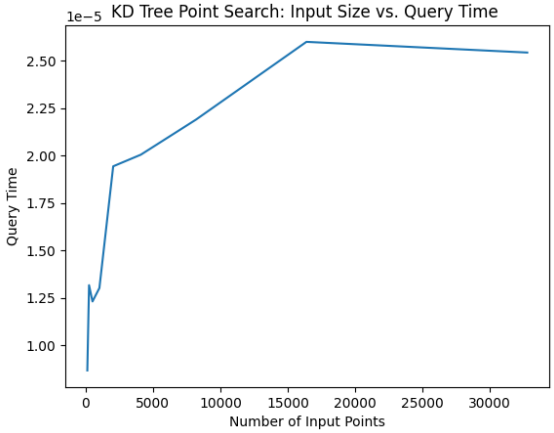
(a) KD Tree; $O(\sqrt{n} + k)$

expected $O(\frac{1}{10}n)$ for large n

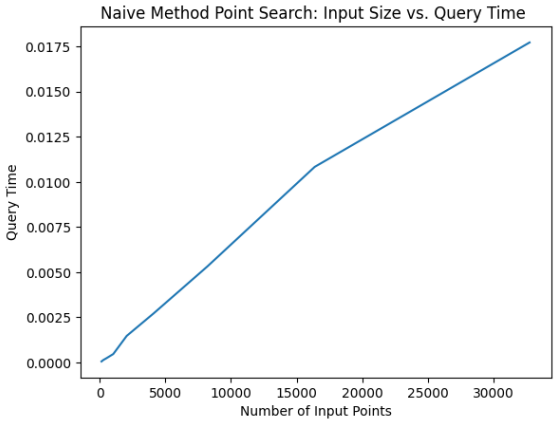


(b)Naive Method; $O(n)$

Figure 7: Plot: Point Search Query Time

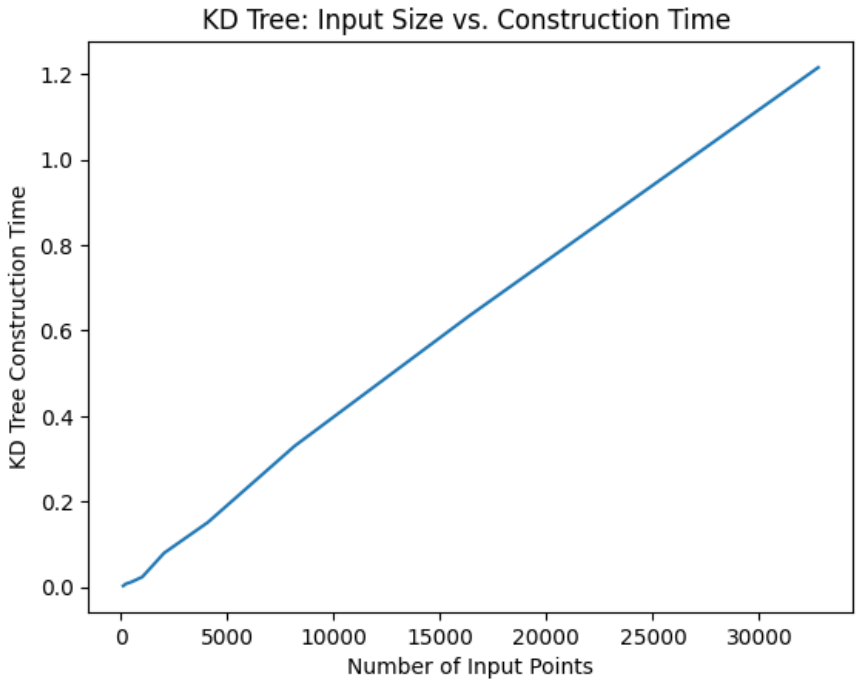


(a) KD Tree; $O(\sqrt{n})$



(b)Naive Method; $O(n)$

Figure 8: KD Tree: Input Size vs. Construction Time



$O(n \log n)$