Final Project 1

—— KD-Tree

# Abstract

A k-d tree is a binary tree in which every leaf node is a k-dimensional point. In this project, we consider the problems related to two-dimensional KD tree, including establishing KD-tree (insert()), range query(range()), and nearest neighbor query (nearest\_neighbor\_query). In order to explore the time complexity of the range query algorithm, the pruning of the tree involved is explained in detail in this paper. Furthermore, by changing the number of points several times, we compare the time consumed by the range query of KD-tree with that consumed by the naive method. Then this paper visually demonstrates the superiorthe ity of KD-tree in this respect.

**[keywords]: KD-tree, insert/establish, range query, nearest neighbor query, time complexity**

# Question-restatement

A **k-d tree** (short for **k-dimensional tree**) is a space-partitioning data structure for organizing points in a k-dimensional space.

Please complete the code based on kd-tree.py or KDTree.java. In this project, we only consider points in **two dimensions**. Note that you cannot use any third-party library.

Q1. **Explain** the existing code (5 marks).

Q2. Implement and explain **insert() and range()** (English writing style, 22 marks).

Q3. Analyze the time complexity of **range query** (5 marks).

Q4. Visualize the **time performance** between k-d tree method and naive method (5 marks).

Q5. (Bonus) Implement the **nearest neighbor query** (5 marks).

Also note that if you cannot explain your code in Q2, you will risk losing all marks in Q2.

# Question-analysis

## About KD-Tree

A k-d tree is a binary tree in which every leaf node is a **k-dimensional** point. All non-leaf nodes can visually act as a hyperplane to divide the space into two half Spaces. The subtree to the left of the node represents the point to the left of the hyperplane, and the subtree to the right of the node represents the point to the right of the hyperplane. The hyperplane is chosen as follows: **Each node is related to the one dimension in the K-dimension that is perpendicular to the hyperplane.** Therefore, if you choose to divide along the X-axis, all nodes with an x value less than the specified value will appear in the left subtree, and all nodes with an x value greater than the specified value will appear in the right subtree.

## About the Question

The code in this project uses a simple and intuitive example to introduce the KD-tree algorithm in **two-dimension**. The two-dimensional KD-tree is shown below in the figure 1.

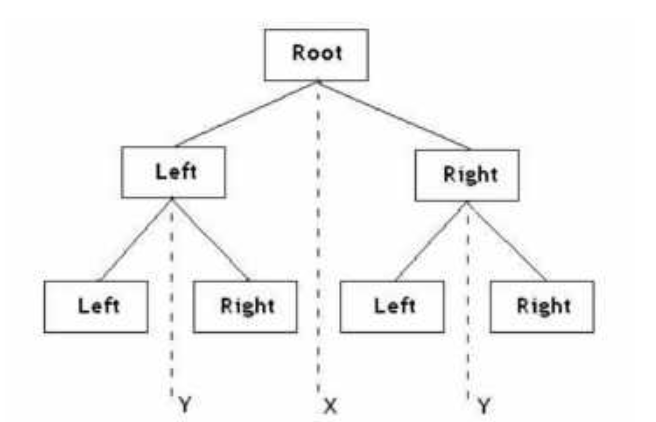


Figure 1: two-dimensional KD-tree

Suppose there are sixtwo-dimensional data points {(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)}, and the data points are located in two-dimensional space (as shown by the black dots in Figure 2).

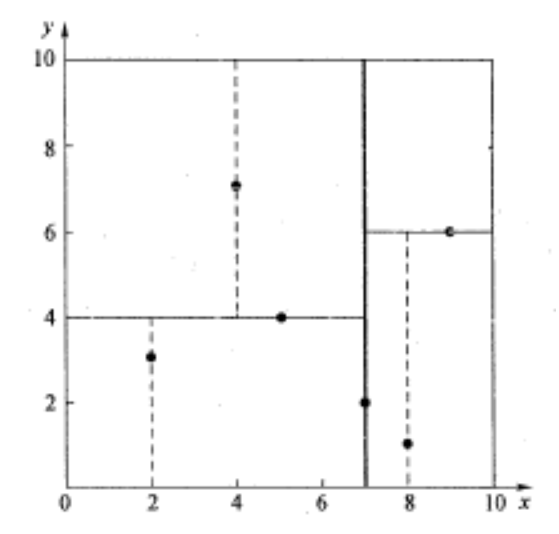


Figure 2: Two-dimensional data points

The KD-tree algorithm is to determine the dividing lines of these segmentation Spaces in Figure 2 (multi-dimensional space is the segmentation plane, usually the hyperplane).

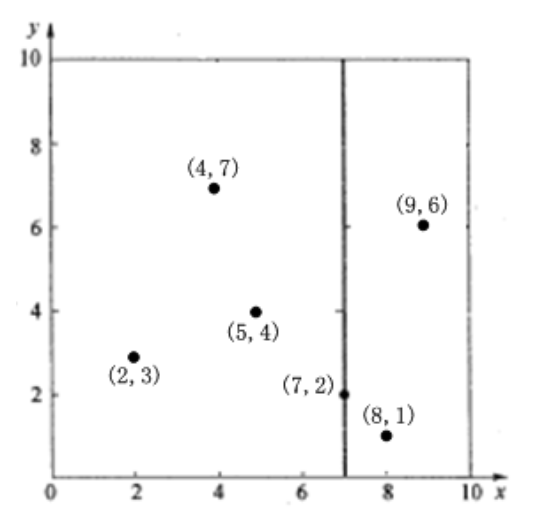


Figure 3: x=7 divides the points

For the first time, the data along the x-axis is segmented. The median value 7 is sorted according to the values 7, 5, 9, 4, 8, 2 along the X-axis, so the root of this KD- tree is (7, 2). Then x=7 divides the points on this hyperplane into two parts. As shown in figure 3, the lower part x<=7 is the left subspace, containing three points {(2,3),(4,7),(5,4)}, and the part x>7 is the right subspace, containing two nodes {(8,1),(9,6)}.

As described in the algorithm, the construction of KD-tree is a recursive process. Then the process of repeating the root node for the data in the left and right subspaces can get the next level of child nodes (5,4) and (9,6) (that is, the 'root' node of the left and right subspaces), and further subdivide the space and data set. This is repeated until the space contains only one data point, as shown in Figure 2. The resulting KD-tree is shown in Figure 4.

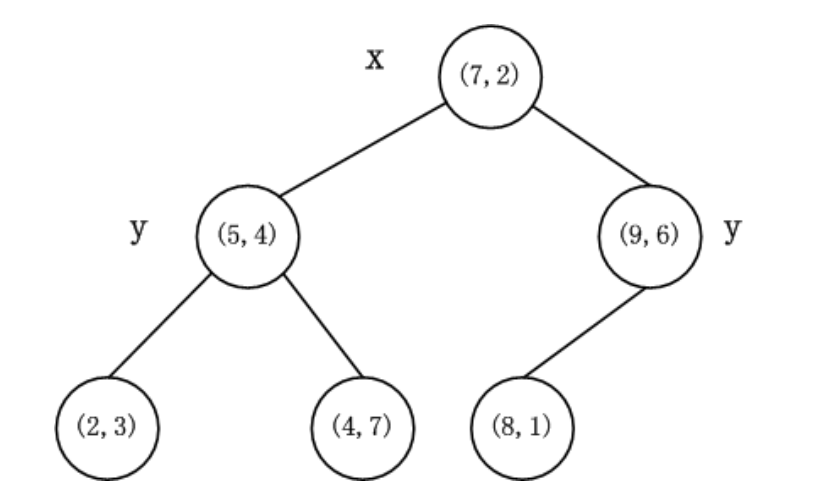


Figure 4: the result KD-tree

In this project, the KD-tree algorithm can be divided into three parts. The first is about the establishment of the data structure of the k-d tree itself **(insert())**. The second is about searching the points in the given range in the established k-d tree **(range())**. The third algorithm is about how to find the nearest neighbor point in the established k-d tree (**nearest\_neighbor\_query()**).

# Question-solving

## Question 1

Explain the existing code.

### class Point

|  |
| --- |
| class Point(namedtuple(**"Point"**, **"x y"**)):  def \_\_repr\_\_(self) -> str:  return **f'Point**{tuple(self)!r}**'** |

To create a Point class with “x” and “y” attributes.

### class Rectangle

|  |
| --- |
| class Rectangle(namedtuple(**"Rectangle"**, **"lower upper"**)):  def \_\_repr\_\_(self) -> str:  return **f'Rectangle**{tuple(self)!r}**'**  def is\_contains(self, p: Point) -> bool:  return self.lower.x <= p.x <= self.upper.x and self.lower.y <= p.y <= self.upper.y |

To create a Rectangle class with “lower” and “upper” attributes.

And “lower” and “upper” stand for the lower left point and the upper right point of the rectangle.

The function is\_conains() returns a bool that determines whether the given points are inside the given rectangular region.

### class Node

|  |
| --- |
| class Node(namedtuple(**"Node"**, **"location left right"**)):  *"""  location: Point  left: Node  right: Node  """* def \_\_repr\_\_(self):  return **f'**{tuple(self)!r}**'** |

To create a Node class with “location”, “left”, and “right” attributes.

### class KDTree

|  |
| --- |
| class KDTree:  *"""k-d tree"""* def \_\_init\_\_(self):  self.\_root = None  self.\_n = 0   def insert(self, p: List[Point]):  *"""insert a list of points"""* pass   def range(self, rectangle: Rectangle) -> List[Point]:  *"""range query"""* pass |

To create a KDTree class. The initialized tree has the attribute root to record the root of it and the attribute n to record the number of points in the KD-tree.

The insert function is used to establish a new KD-tree by a list of points.

The range function is used to search the points in the given range in the established tree.

### def range\_test()

|  |
| --- |
| def range\_test():  points = [Point(7, 2), Point(5, 4), Point(9, 6), Point(4, 7), Point(8, 1), Point(2, 3)]  kd = KDTree()  kd.insert(points)  result = kd.range(Rectangle(Point(0, 0), Point(6, 6)))  assert sorted(result) == sorted([Point(2, 3), Point(5, 4)]) |

To test the range() function in the KDTree class.

Use the insert function to establish a new KD-tree with Point(7, 2), Point(5, 4), Point(9, 6), Point(4, 7), Point(8, 1) and Point(2, 3).

Use the range method to sort out the points of the KD-tree in the rectangle whose lower left point is (0, 0) and upper right point is (6, 6)

Finally, assert the result is Point(2, 3), Point(5, 4).

### def performance\_test()

|  |
| --- |
| def performance\_test():  points = [Point(x, y) for x in range(1000) for y in range(1000)]   lower = Point(500, 500)  upper = Point(504, 504)  rectangle = Rectangle(lower, upper)  *# naive method* start = int(round(time.time() \* 1000))  result1 = [p for p in points if rectangle.is\_contains(p)]  end = int(round(time.time() \* 1000))  print(**f'Naive method:** {end - start}**ms'**)   kd = KDTree()  kd.insert(points)  *# k-d tree* start = int(round(time.time() \* 1000))  result2 = kd.range(rectangle)  end = int(round(time.time() \* 1000))  print(**f'K-D tree:** {end - start}**ms'**)   assert sorted(result1) == sorted(result2) |

Store all points of (x, y) which in the x and y range from 0 to 1000 in a list.

Then establish a rectangle with the upper right point (504, 504) and the lower left point (500, 500).

Use result1 to store the points both in the list and the rectangle, and record the consumed time.

Use the points in the list to establish a KD-tree.

Use result2 to store the points both in the KD-tree and the rectangle, and record the consumed time.

Finally, assert result1 equals result2.

### main

|  |
| --- |
| if \_\_name\_\_ == **'\_\_main\_\_'**:  range\_test()  performance\_test() |

To execute range\_test() and performance\_test().

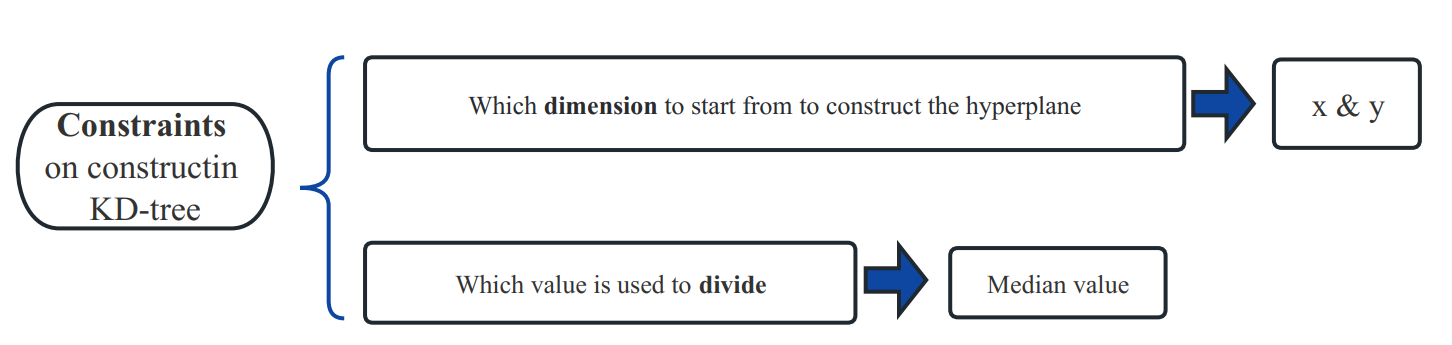
## Question 2

Implement and explain **insert() and range().**

### 1 insert()

#### 1.1 Algorithm description (\_build\_KD\_Tree())

##### 1.1.1 constraints & stipulations



##### 1.1.2 Pseudocode & Explain

In the insert algorithm, I use the **\_bulid\_KD\_Tree** algorithm to establish a KD-tree by a list of points and return the root of the KD-tree. And in the \_bulid\_KD\_Tree algorithm, I use the **get\_left\_position** algorithm to the median value of the points.

###### 1.1.2.1 \_bulid\_KD\_Tree

|  |
| --- |
| **Algorithm: \_build\_KD\_Tree()** |
| **Input:** **point-list**: p = [Point1, Point2, Point3, …, Pointn], **dimension**: axis = depth % 2  **Output:** the root of the KD-Tree |
| 1. **if** p == None then  2. | **return** None  3. **end**  4. axis 🡨 depth % 2  5. p 🡨 p.sort(key = lambda x: x[axis])  6. median 🡨 len(p) >> 1  7. median 🡨 get\_left\_position(p, axis, p[median][axis])  8. left\_child 🡨 \_build\_KD\_Tree(p[:median], depth+1)  9. right\_child 🡨 \_build\_KD\_Tree(p[median+1:], depth+1)  10. n 🡨 n+1  11. **return** Node(location=p[median], left=left\_child, right=right\_child) |

**Explain:**

**Step(1):** Select an axis (x or y) based on depth so that the axis cycles through all valid values.

**Step(2):** Sort the point list by axis and choose the median as the pivot element. And select the median by axis from the point list (using the get\_left\_position algorithm).

**Step(3):** Create the node and construct the left or right subtrees.

**Step(4):** Go to the next layer of the KD-tree, and repeat step(1) ~ step(3).

###### 1.1.2.2 get\_left\_position

|  |
| --- |
| **Algorithm:** get\_left\_position() |
| **Input:** data-set: p = [Point1, Point2, Point3, …, Pointn], axis, target  **Output**: left\_point |
| 1. left 🡨 0  2. right 🡨 len(p) – 1  3. **while** left < right **do**  4. | mid 🡨 left + (right-left)//2  5. | **if** target > p[mid][axis] **then**  6. | | left 🡨 mid+1  7. | **end**  8. | **else** if target < p[mid][axis] **then**  9. | | right 🡨 mid  10. | **end**  11. **end** |

**Explain:**

**Step(1):** The search begins with the middle element of the array. If that element happens to be the target element, the search process ends; otherwise, the next step is performed.

**Step(2):** If the target element is greater than/less than the middle element, look for the half of the array that is greater than/less than the middle element, and repeat step (1).

**Step(3):** If one step array is empty, the target element cannot be found.

#### 1.2 code

|  |
| --- |
| def insert(self, p: List[Point])->Node:  *"""insert a list of points"""* def \_build\_KD\_Tree(p: List[Point], depth: int = 0)->Node:  if not p:  return None axis = depth % 2p.sort(key=lambda x:x[axis])  median = len(p) >> 1 *# 中位数index=list长度除2、向下取整*  median = get\_left\_position(p, axis, p[median][axis])  left\_child = \_build\_KD\_Tree(p[:median], depth + 1)  right\_child = \_build\_KD\_Tree(p[median + 1:], depth + 1)  self.\_n += 1  return Node(  location=p[median],  left=left\_child,  right=right\_child  )   self.\_root = \_build\_KD\_Tree(p) |
| def get\_left\_position(p: List[Point],axis:int=0,target:int=0)->int:  left = 0  right = len(p) - 1;  while(left<right):  mid=left+((right-left)>>1)  if(target>p[mid][axis]):left=mid+1  else:right=mid  return left |

### 2 range()

#### 2.1 algorithm description

##### 2.1.1 Pseudocode

|  |
| --- |
| **Algorithm:** \_dfs\_range() |
| **Input:** **rectangle**, the **root** of the KD-Tree  **Output**: result: point list |
| 1. node 🡨 root  2. **if** node = None **then**  3. | **return** None  4. **end**  5. **if** rectangle.is\_contains(node.location) **then**  6. | result.append(node.location)  7. **end**  8. **if** depth % 2 = 0 **then #x层**  9. |  **if** node.location[0] > rectangle.lower[0] **then**  10. | | \_dfs\_range(node.left, depth+1)  11. | **end**  12. | **if** node.location[0] <= rectangle.upper[0] **then**  13. | | \_dfs\_range(node.right, depth+1)  14. | **end**  15. **end**  16. **else if** depth % 2 = 1 **then #y层**  17. | **if** node.location[0] > rectangle.lower[1] **then**  18. | | \_dfs\_range(node.left, depth+1)  19. | **end**  20. | **if** node.location[0] <= rectangle.upper[1] **then**  21. | | \_dfs\_range(node.right, depth+1)  22. | **end**  23. **end**  24. **return** result |

##### 2.1.2 Explain

**Process of the algorithm:**

**Step(1):** If the query range has no intersection with the area of the current node, it pops out directly.

**Step(2):** If the query range includes the region of the current node, jump out directly and append it to the result.

**Step(3):** If the query range intersects with the node but does not contain it completely, the recursive solution will be continued.

Start from the root node of the KD tree and determine whether its location is in the rectangle. If it is, add it to the result; if it isn’t, depth = depth +1, and go to the next layer. And when comes to the last layer, return None, and the recursive process will end.

During the recursive process, the KD tree needs to be **pruned**. For example, at a certain level of the tree by comparing the size of the node x value to divide the left and right subtrees. If the x value of the point is on the left side of the rectangle range, and it is certain that all x values of the points on the left subtree are smaller than that of this point, it is easy to draw the conclusion that all the x values of the points on the left subtree are on the left side of the rectangle range. At this point, in order to improve the performance of range (), the left subtree of this point needs to be pruned.

Hence, when **node.location[axis]<=rectangle.upper[depth%2]**, which means the x value of the point is on the left side of the rectangle range, we need to go to the right subtree to continue the recursive process and pruned the left subtree: **\_dfs\_range(node.right,depth+1)**.

What’s more, we can understand the 2D-tree as using some straight lines (line segments or rays) to divide the whole space into several areas, so as to narrow the search scope and achieve the purpose of pruning.

#### 2.2 code

|  |
| --- |
| **def** range(self, rectangle: Rectangle) -> List[Point]:  *"""range query"""* result = []   **def** \_dfs\_range(node:Node,depth:int=0):  **if** node **is None**:  **return  if** rectangle.is\_contains(node.location):  result.append(node.location)  axis = depth % 2  **if** node.location[axis] > rectangle.lower[axis]:  \_dfs\_range(node.left, depth + 1)  **if** node.location[axis] <= rectangle.upper[axis]:  \_dfs\_range(node.right, depth + 1)  *# if depth%2==0:  # if node.location[0]>rectangle.lower[0]:  # \_dfs\_range(node.left,depth+1)  # if node.location[0]<=rectangle.upper[0]:  # \_dfs\_range(node.right,depth+1)  # else:  # if node.location[0]>rectangle.lower[1]:  # \_dfs\_range(node.left,depth+1)  # if node.location[0]<=rectangle.upper[1]:  # \_dfs\_range(node.right,depth+1)* \_dfs\_range(self.\_root)  **return** result |

## Question 3

Analyze the time complexity of the **range query.**

**🡪analyze the time complexity of rectangle query in 2D-tree.**

The search time depends on **the number of recursive calls** and **the number of sub-regions intersecting with the rectangle at all dimensions (x & y)**.

We can divide the points into 3 parts:

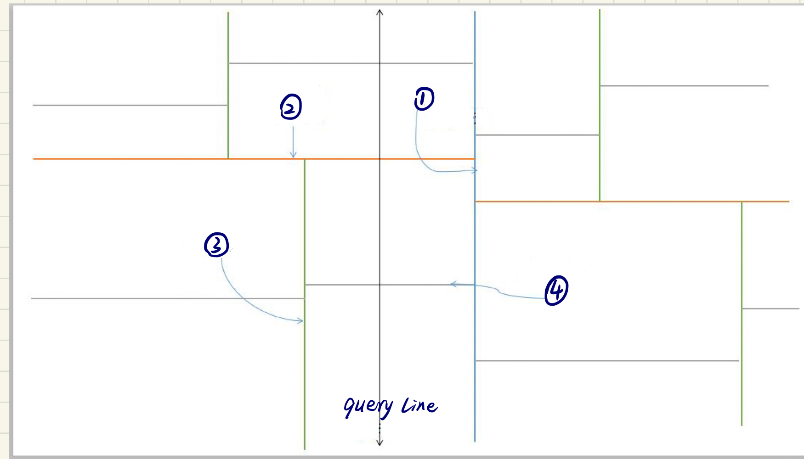
**·** First, the intersection with the rectangle is empty.

**·** Second, all are contained in the rectangle.

**·** Third, intersections with the rectangle but not all included in it.

In the query process, the algorithm will not continue the recursive subtree when it encounters the first and second classes of points. Therefore, the time complexity of the algorithm is related to the number of the third class of points

Assuming there is a 2-dimensional plane with n points, and we need to query the points left to the query line. And the blue line, orange line, green line, and gray line stand for the first, second, third, and fourth time of pruning. As shown in the following figure.



An example 2D-tree

Firstly, the first time of pruning is effective, the right half will be cut off, so the number of knots down will not double.

Then, the second time of pruning is ineffective, and the number of knots down is doubled.

The third time is valid, and the fourth time is invalid...

This way, only odd-numbered layers will be pruned effectively, not even-numbered layers.

Therfore, a 2D-tree with  points and  layer will have  times of double.

🡪

And since the 2D-tree is balanced, so 

🡪 

**Hence, the time complexity of the range query is **

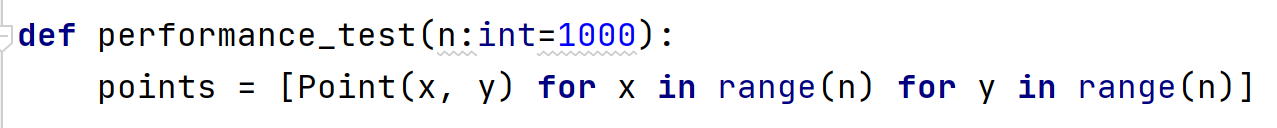
## Question 4

Visualize the **time performance** between KD-tree method and naive method.

### solution

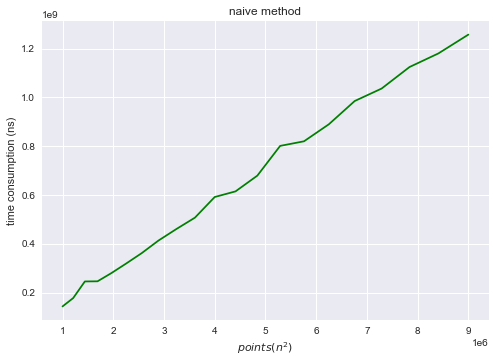
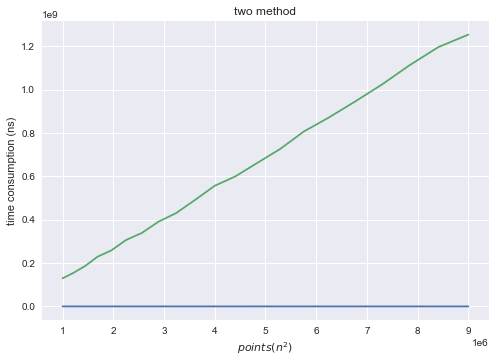
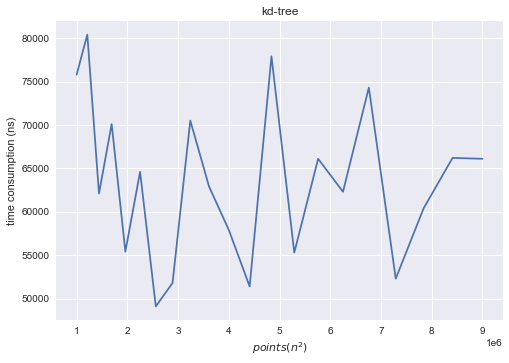
In order to test and compare the time performance of the two methods, I have fine-tuned performance\_test —— change the number of points in the list, then use them to build 2D trees to test the time consumption of naive method and KD-tree searches.

( = the number of points)



Assuming **n increasing linearly**, and  (, ).

The change of time consumed by the two methods as n changes is shown in the figure below. ( Time is measured in nanoseconds)

Hence, as shown in the figures above, it is easy to draw the conclusion that **KD-tree greatly improves the time performance of range query compared with the naive method.** Moreover, in a certain range, the change of the n value has no obvious relationship with the change of KD-tree range query time.

### Code

|  |
| --- |
|  |
|  |

## Question 5

Implement the **nearest neighbor query.**

### code

|  |
| --- |
|  |