COMPUTATIONAL GEOMETRY
Assignment 4

Aman Choudhary
MTech Coursework, CSA 2020
Sr No: 17920

May 01, 2021

1 Problem 1

- We have the following recurrence relation for the query time of kd-tree,

$$Q(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1, \end{cases}$$

- **Master's theorem** states that if we have a recurrence relation of the form, $T(n) = a\,T(n/b) + \theta(n^k \log^p n)$ where $a \geq 1$, $b > 0$, $k \geq 0$ and p is a real number, such that $a > b^k$, then $T(n) = \theta(n^{\log_b a})$.
- In the above case $a = 2$, $b = 4$, $k = 0$ and $p = 0$. That is $a > b^k$. Hence, the above relation solves to $Q(n) = \theta(n^{\log_4 2}) = \theta(n^{\log_{2^2} 2^1}) = \theta(n^{1/2}) = \theta(\sqrt{n})$. This establishes both the upper i.e. $O(\sqrt{n})$ as well the lower bound, $\Omega(\sqrt{n})$.

2 Problem 2

2.1 (a)

Algorithm 1: BUILD_KD_TREE (P, depth, d)

Input: A set of points P, the current depth, and dimension d.
Output: The root of a kd-tree storing P.

if *P contains only one point* **then**
    return a leaf storing this point
else
    i = (depth % d) + 1 `/* There are 'd' dimensions for every point. We index these dimensions using 1,2,...,d */`
    Find median of P along the i^{th} dimension, let us call it **m**.
    Split P into two subsets using **m**. Let P_1 be the set of points having their i^{th} coordinate less than or equal to that of **m**. Let P_2 be the set of points with their i^{th} coordinate more than that of **m**.
    v_{left} = BUILD_KD_TREE(P_1, depth+1, d)
    v_{right} = BUILD_KD_TREE(P_2, depth+1, d)
    Create a node v storing **m**, make v_{left} the left child of v, and make v_{right} the right child of v.
    return v

Space Complexity:

- We note that each leaf in the kd-tree stores a distinct point of P. Hence, there are n leaves.
- Since the kd-tree is a binary tree, the worst case would occur when the binary tree is full. In that case, the total number of nodes (including internal nodes as well as the leaves) would be $O(n)$.
- Now, each node stores d elements, one for each dimension. However, d is a constant, hence each node takes $O(1)$ space. Therefore, total space complexity $= O(n) * O(1) = O(n)$.

Construction Time Complexity:

- The most expensive step that is performed at every recursive call is finding the median along i^{th} coordinate. Median finding can be done in linear time. Hence, the building time $T(n)$ satisfies the recurrence,

$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ O(n) + 2T(n/2), & \text{if } n > 1, \end{cases}$$

- The above recurrence solves to $O(n \log n)$.

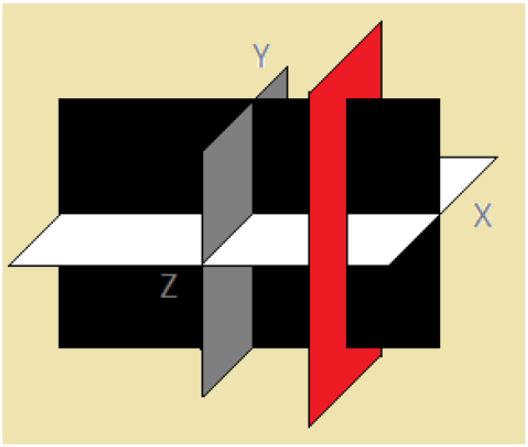
2.2 (b)

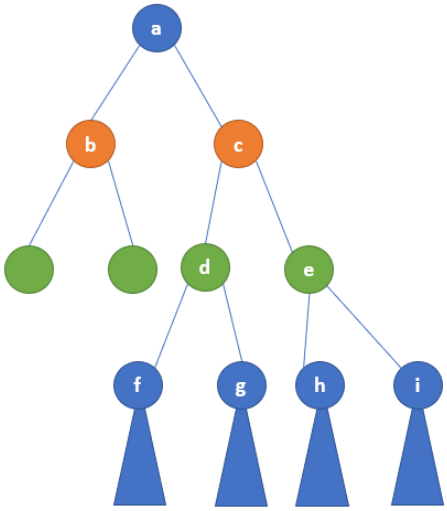
The query algorithm remains unchanged, its same as the 2D case.

Algorithm 2: SEARCH_KD_TREE (v, R)

Input: The root of (a sub-tree of) a kd-tree, and a range R.
Output: All points at leaves below v that lie in the range.

```
1  if v is a leaf then
2      Report the point stored at v if it lies in R.
3  else
4      if region(left_child(v)) is fully contained in R then
5          REPORT_SUBTREE(left_child(v)))
6      else if region(left_child(v)) intersects R then
7          SEARCH_KD_TREE (left_child(v)), R)
8      if region(right_child(v)) is fully contained in R then
9          REPORT_SUBTREE(right_child(v)))
10     else if region(right_child(v)) intersects R then
11         SEARCH_KD_TREE (right_child(v)), R)
```





Analysis:

- Let us consider the case when $d = 3$. We want to calculate how many 3D regions does any arbitrary plane intersect. This analysis follows closely from the 2D case.
- At first, we have the entire 3D region. Let us consider an arbitrary 2D plane (shown in red). We begin with partitioning the space along X-axis. We represent this with node **a** (grey plane).
- The red plane would lie on only one side of the partition (here we arbitrarily choose the right 3D sub-region). We continue to partition the right 3D sub-region along the Y-axis. We represent this with node **c** (white plane).
- The red plane intersects both the 3D sub-regions, which are represented by children of **c**. We continue to partition them along the Z-axis. This is represented by node **d** and **e** (black planes).
- We end up with a total of 4 sub-regions which are intersected by the red plane, represented by the sub-trees rooted at **f**, **g**, **h** and **i**.The resulting problem are smaller versions of the original problem we sought to solve. Note, that we needed to explore 4 nodes, before we reached our subproblems. We can write the recurrence relations for the number of regions as follows,

$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 4 + 4T(n/8), & \text{if } n > 1, \end{cases}$$

- We compare this with case for $d = 2$.

$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2 + 2T(n/4), & \text{if } n > 1, \end{cases}$$

- A general recurrence relation can be written in terms of d,

$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2^{d-1} + 2^{d-1}T(n/2^d), & \text{if } n > 1, \end{cases}$$

- **Master's theorem** states that if we have a recurrence relation of the form, $T(n) = a\,T(n/b) + \theta(n^k \log^p n)$ where $a \geq 1$, $b > 0$, $k \geq 0$ and p is a real number, such that $a > b^k$, then $T(n) = \theta(n^{\log_b a})$.
- In the above case $a = 2^{d-1}$, $b = 2^d$, $k = 0$ and $p = 0$. That is $a > b^k$. Hence, the above relation solves to $T(n) = \theta(n^{\log_{2^d} 2^{d-1}}) = \theta(n^{(d-1)/d}) = \theta(n^{1-(1/d)})$.
- After accounting for the time taken to report the points in the output set, the total time complexity for querying is $O(n^{1-(1/d)} + k)$.

2.3 (c)

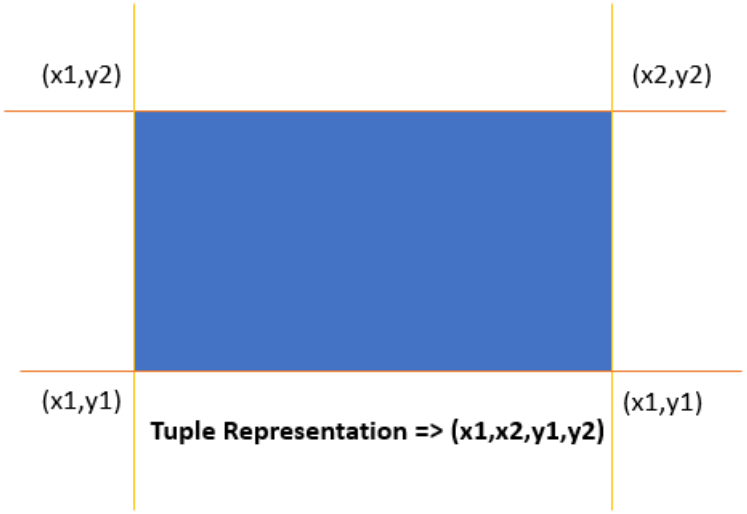
Space Complexity:
As analysed in 2(a), the total number of nodes in the kd-tree would be $O(n)$. Now, each node stores d elements, one for each dimension. Hence each node takes $O(d)$ space. Therefore, total space complexity = $O(n) * O(d) = O(nd)$.

Construction Time Complexity:
As analysed in 2(a), the recurrence relation for constructing the tree is independent of d and the construction time complexity is $O(n \log n)$.

Querying Time Complexity:
As analysed in 2(b), the total number of regions that intersect one boundary of a d-dimensional query range is $O(n^{1-(1/d)})$. A d-dimensional query range has 2*d boundaries, i.e. a 2D rectangle has 4 edges, a 3D space has 6 faces and so on. Therefore, the total number of regions intersected by a d-dimensional query range is $O(dn^{1-(1/d)})$. The overall time complexity for querying the kd-tree is therefore, $O(dn^{1-(1/d)} + k)$.

3 Problem 3

3.1 (a)



- We would use 4-D range trees to solve this problem. Each rectangle is represented using a 4-element tuple of the form $(left, right, top, bottom)$.
- In the first level tree, the points (tuples corresponding to rectangles) are sorted based on the first element of the tuple: *left* (x-coordinate of the left edge). In the second level tree, the points are sorted based on *right* (x-coordinate of the right edge). In the third level tree, the points are sorted based on *bottom* (y-coordinate of the bottom edge). In the last level tree, the points are sorted based on *top* (y-coordinate of the top edge).
- For a query rectangle of the form $[x : x'] \times [y : y']$, we do the following: In the first level we query for all rectangles with $left \geq x$. In the second level we query for all rectangle with $right \leq x'$. In the third level we query for all rectangle with $bottom \geq y$. In the last level we query for all rectangle with $top \leq y'$.
- We know that the space and time complexity for querying a d-dimensional range tree is $O(n \log^{d-1} n)$ and $O(n \log^{d} n + k)$ respectively. Hence, in our case the storage consumed is of the order $O(n \log^{3} n)$, and the query time is $O(n \log^{4} n + k)$.

3.2 (b)

We can solve this problem with just a small amount of extra **pre-processing**. For every polygon in P, we need to find its **smallest enclosing rectangle** and represent the polygon using the 4-tuple $(left, right, top, bottom)$ as described above. To compute the **smallest enclosing rectangle**, we can go over each point in the polygon and compute the extreme x and y coordinates. Since, the rectangle completely covers the polygon entirely, the polygon will also lie in the query rectangle if its enclosing rectangle does. Thus, our current problem reduces to the above problem. Therefore it can also be solved using $O(n \log^{3} n)$ space, while maintaining a query time of $O(n \log^{4} n + k)$.