Time Complexity Experimental Evaluation on Range Tree

Xiuge Chen 961392

1. Introduction

Searching is one of the fundamental tasks in computer science, many data structures and algorithms have been developed to support efficient search. For example, a *perfect hash function* outputs a dictionary search in constant time, and a *weighted-balanced binary search tree (WB-BST)* supports range search in $O(log\ n)$ time.

However, it will cost WB-BST $O(n)$ time to search in the multidimensional space. Bentley et al. [1] proposed *Range Tree* to support multidimensional range searches in $O(k + log^{\,d} n)$ time, but it takes more space $O(n\ log^{\,d-1} n)$ to store. Chazelle [2][3] further improved the query time to $O(k + log^{\,d-1} n)$ with space complexity of $O(n\,(\frac{log\ n}{loglog\ n})^{\,d-1})$.

In this experiment, to practically measure the effectiveness of different implementations of *Range Tree*, various *Range Trees* are programmed and tested against various lengths of data and different ranges of search.

2. Preliminaries

2.1 Notations

Let N be the universe size and P be a set of n d-dimensional points drawn from $[1,\ N]^{\,d}$. A range query Q could be represented as $[a_{\,1},\ b_{\,1}] \times ... \times [a_{\,d},\ b_{\,d}]$ with $a_{\,i} \in [1,\ N]^{\,d}$, $b \in [1,\ N]^{\,d}$, and $a_{\,i} \geq b_{\,i}$ for all $i \in d$. We use k to denote the solution size of Q, i.e. $k = |\,P \cap Q\,|$. Without loss of generality, in this experiment, we consider only the two dimensional case, i.e. $d = 2$, let $x_{\,i}$ and $y_{\,i}$ be the first and second dimension of point i. Denote $P(u)$ as the set of elements that are in the subtree rooted at u.

2.2 Problem Definition

Here we assumed that P is static, i.e. it is known and fixed in advance. We defined the range reporting problem and several useful tree terminologies as follows.

d-Dimensional Range Reporting. Given a query range Q, report all the points in $P \cap Q$.

Successor(x): the smallest integer in P that is greater than x.

Predecessor(x): the largest integer in P that is smaller than x.

Lowest Common Ancestor(u, v) or ***LCA(u, v)***: The lowest node w (i.e. with the largest depth) in tree T which is an ancestor of both u and v.

Using binary search and an auxiliary pointer pointing to the parent node, *successor(x)*, *predecessor(x)* and *LCA(u,v)* could be found in $O(log\ n)$ time. Besides, given a range query $Q = [a, b]$, it is equivalent to consider $Q' = [succ(a),\ pred(b)]$.

2.3 Range Tree - Theoretical Analysis

Range Tree supports two operations, *construct(P)* and *query(Q)*, where *construct(P)* constructs a *Range Tree* on P, and *query(Q)* reports the range query result $P \cap Q$. In this experiment, we implemented the following three different types of *Range Tree*.

Naive Original Range Tree: The original version of the *Range Tree* that has *WB-BSTs* in all dimensions, so that it could perform range queries and reduce to search space dimension by dimension. Based on the construction algorithm, we could further categorize it into *Naive Original RT* and *Sorted Original RT*.

construct(P) - Naive Original Range Tree:

(1) Sort P by x and construct a *WB-BST* T_x on P based on x.
(2) For each internal node $u \in T_x$:
    (a) Sort $P(u)$ by y and construct a *WB-BST* $T_{u,y}$ on $P(u)$ based on y.

It could be verified that constructing a *WB-BST* on a sorted list $P(u)$ only costs $O(|P(u)|)$ time [4]. Thus, step (1) will take $O(n \log n)$ time if P is not sorted in advance. Similarly, for step (2) we could firstly sum up the cost for all the nodes that are on the same depth, $\sum_{u \in level\ i} |P_u| \log |P_u| = O(n \log n)$. Since we have $O(\log n)$ levels in T_x, the time complexity of *construct(P)* is $O(n \log^2 n)$.

query(Q) - Naive Original Range Tree:

(1) Let $a' \leftarrow successor(a_1)$, $b' \leftarrow predecessor(b_1)$ and $u_{split} \leftarrow LCA(a', b')$.
(2) Report u_{split} if $u_{split} \in Q$.
(3) For each node u on the path from u_{split} to a' (exclude u_{split}):
    (a) Report u if $u \in Q$.
    (b) If $a' < u$, do a range report on the y-coordinate of u's right subtree, $T_{u.right,y}$
(4) For each node u on the path from u_{split} to b' (exclude u_{split}):
    (a) Report u if $u \in Q$.
    (b) If $b' > u$, do range report on the y-coordinate of u's left subtree, $T_{u.left,y}$

Single dimension range reporting will take $O(k + \log n)$ time [4]. Since there are at most $O(\log n)$ nodes on the path from u_{split} to a'(or b'), there are at most $O(\log n)$ y-coordinates range reporting. Summing over all of them, the total time complexity is $O(k + \log^2 n)$.

Sorted Original Range Tree: It has the same underlying data structure as the *Naive Original RT*, except it uses a slightly different construction algorithm to reduce the time cost.

construct(P) - Sorted Original Range Tree:

(1) Sort P by x and construct a *WB-BST* T_x on P based on x.
(2) Sort P on the second dimension, y, denote the sorted list as $L_{y,root}$.
(3) Start from root, for each node $u \in T_x$:
    (a) Use $L_{y,u}$ to construct a *WB-BST* $T_{u,y}$ on $P(u)$ based on y.

(b) Partition $L_{y,u}$ into $L_{y,u.left}$ and $L_{y,u.right}$, where $L_{y,u.left}$ ($L_{y,u.right}$) contains all nodes in the left (right) child subtree of u, sorted by y.

Step (3.a) and (3.b) could be done in $O(|P(u)|)$ time [4]. Since there are $O(\log n)$ levels in the tree and for each level, $\sum_{u \in level\ i} |P(u)| = O(n)$, the time complexity of construct(P) is being reduced to $O(n \log n)$.

Fractional Cascading Range Tree: It imports the *Fractional Cascading* in the last dimension, y. Instead of building a WB-BST, a sorted array arr_u is being created for each internal node u. Each arr_u will contain elements $e \in P(u)$, sorted by y. Besides, each element will have two pointers, e_{left} and e_{right}, pointing to its successor in the $arr_{u.left}$ and $arr_{u.right}$.

construct(P) - Fractional Cascading Range Tree:

(1) Sort P by x and construct a *WB-BST* T_x on P based on x.
(2) Let arr_{root} be the set P sorted by y.
(3) Starting from the root node, for each node $u \in T_x$:
    (a) Create arr_w and arr_v for its two children by partitioning arr_u, points each element in arr_u to its successors in arr_w and arr_v.

Again, partitioning an array a_u into two parts and adding the auxiliary pointers will cost $O(|P(u)|)$ time [4]. Since there are $O(\log n)$ levels in the tree and for each level, $\sum_{u \in level\ i} |P(u)| = O(n)$, the time complexity of construct(P) is $O(n \log n)$.

query(Q) - Fractional Cascading Range Tree:

(1) Let $a' \leftarrow successor(a_1)$, $b' \leftarrow predecessor(b_1)$, and $u_{split} \leftarrow LCA(a', b')$.
(2) Report u_{split} if $u_{split} \in Q$, find the successor of a_2 in arr_{root}, c'.
(3) For each node u on the path from u_{split} to a' (exclude u_{split}):
    (a) Report u if $u \in Q$. Let $c' \leftarrow c'_{right\,(left)}$ if u is the right (left) child of its parent.
    (b) If $a' < u$, report all elements e in $arr_{u.right}$, starting at c'_{right}, until $y_e > b_2$.
(4) For each node u on the path from u_{split} to b' (exclude u_{split}):
    (a) Report u if $u \in Q$. Let $c' \leftarrow c'_{right\,(left)}$ if u is the right (left) child of its parent.
    (b) If $b' > u$, report all elements e in $arr_{u.left}$, starting at c'_{left}, until $y_e > b_2$.

By doing this, for all the internal nodes u, there is no need to perform an extra range query on y-coordinate, therefore the cost becomes just the number of solutions in $P(u)$, $O(k_u)$. Summing over all candidate internal nodes, the total time complexity is $O(k + \log n)$.

Table 2-1 summarize the theoretical upper timer bounds of all the *Range Trees* above:

	Naive Original RT	*Sorted Original RT*	*Fractional Cascading RT*
Construction Time	$O(n \log^2 n)$	$O(n \log n)$	$O(n \log n)$
Query Time	$O(k + \log^2 n)$	$O(k + \log^2 n)$	$O(k + \log n)$

Table 2-1: Theoretical comparison between different *Range Trees* (*RT*).

3. Experiment Preparation

3.1 Implementation

C++ implementation is chosen here for several reasons. Firstly, it is a very powerful middle-level language that could be used to build time-critical programs. Secondly, it is easier to program compared to C as it supports object-oriented programming. Lastly, it provides many useful built-in libraries that could help reduce the workload.

Specifically, besides all the details in section 2, we also include a unique id for each tree node to help break ties between the nodes with the same values.

For better uniform distribution, the random generator of data points is implemented through the *uniform_*_distribution* functions from C++'s built-in random library. And the recording of the overall running time is implemented using C++'s chrono library. Spdlog library [5] is used for fast results logging.

3.2 Experiment Environment

We choose C++ 20 with CMake (version >= 3.15) and Clang compiler (version 11.0.0) to implement our data structures. All experiments are performed at a MacBook Pro, with macOS Catalina (10.15.3) operating system, 2 GHz Dual-Core Intel Core i5 processor, and 8 GB 1867 MHz LPDDR3 memory.

3.3 Data and Query Generation

Both experiment data points and range queries are artificially generated. Each data point contains three integers, a unique identifier id and two values (x, y) indicating the point in two dimensions. To make the id uniquely identify each data point, we also keep track of the next available id, id_{next}, and let the initial id_{next} be 1. Let the size of the universe N be 10^6, different data points and range queries are generated as follows.

1. *generate_a_point(min, max)*: uniformly pick a point $[a_1, b_1] \times [a_2, b_2]$ from the universe $[min, max] \times [min, max]$, assign the point id *i* to be id_{next} and increment id_{next}, return the generated point.
2. *generate_point_set(n)*: call *generate_a_point(*$1, N$*)* n times, return the generated point set.
3. *generate_a_query(s)*: generate a range query with range s. Firstly, invoke *generate_a_point(*$1, N-s$*)* to get a random two-dimensional point q. Then return a range query with range $[q.x, q.x+s] \times [q.y, q.y+s]$. It is assumed that $1 < s < N$.

3.4 Experiment Design and Justifications

To compare the time complexity of different construction methods and different query algorithms, we performed the following three different experiments. All data will be pre-generated and stored in a vector before starting experiments, so that we could accurately

measure the runtime. For each individual experiment, unless otherwise specified, we used the same set of data and queries to test all *Range Trees* to ensure a fair comparison.

1. Construction time v.s. Construction method: Generate 10 different data set with the data length $n_i = 2^i \times 10^3$, for all $i \in [1, ..., 10]$. For each data set, construct the *Naive Original RT* and the *Sorted Original RT* respectively. Repeat this experiment 100 times, 10 new data sets will be generated for each time.
2. Query Time v.s. Query Range: Fixed the data length n to be *1M (M = 10^6)* and vary the range of query s as *1%M, 2%M, 5%M, 10%M and 20%M*. For each range, perform 100 different range-reporting queries on the *Sorted Original RT* and the *Fractional Cascading RT* respectively.
3. Query Time v.s. Data Length: Fixed the query range s to be *5%M (M = 10^6)* and generate 10 different data set with the data length $n_i = 2^i \times 10^3$, for all $i \in [1, ..., 10]$. For each data length, perform 100 different range-reporting queries on the *Naive Original RT* and the *Fractional Cascading RT* respectively.

According to the theoretical bounds in section 2.3, the construction complexity of different *Range Tree* only depends on the length of the data, n. Since the purpose of the experiment is to compare the practical and theoretical performance across various *Range Trees,* it is sufficient to fixe other parameters and vary n. Firstly, in order to observe the asymptotic performance, sufficient large values of n should be chosen, otherwise, the actual time cost might be influenced by the constant factors (or factors that are $o(n \log n)$). Meanwhile, the values of n should be feasible for normal computers, ie. being able to execute within a reasonable amount of time. Therefore, we chose to vary the length as $n_i = 2^i \times 10^3$, for all $i \in [1, ..., 10]$. Lastly, due to the randomness of data generation, each value should be tested for a sufficiently large number of repetitions to avoid outliers, thus we repeat the experiment 100 times, each time we use a fixed random seed to reduce noises.

Similarly, since the theoretical bound of query time relies on both the data length n and the solutions size k, we design our experiments by varying one of them and fixing the other one. Since it is impossible to directly control the solution size when data points and queries are generated randomly, we choose to vary the range s to indirectly change the solution size, it could be verified that for randomly generated data, the expected solution size has a positive linear relationship with the range. The justifications for the choices of values and the number of repetitions are the same as above.

3.5 Evaluation Metrics

To observe the actual running time cost of each operation, it is sufficient to measure the average running time of all repetitions. Therefore, the start and end time (in microseconds) of different operations are recorded.

4. Result and Discussion

All experimental results are shown in the Figure below. Each row represents the results from one experiment, where the left plot indicates the theoretical asymptotic behavior of different data structures/algorithms (values is meaningless), the right plot indicates the actual time cost (in microseconds) of each experiment.

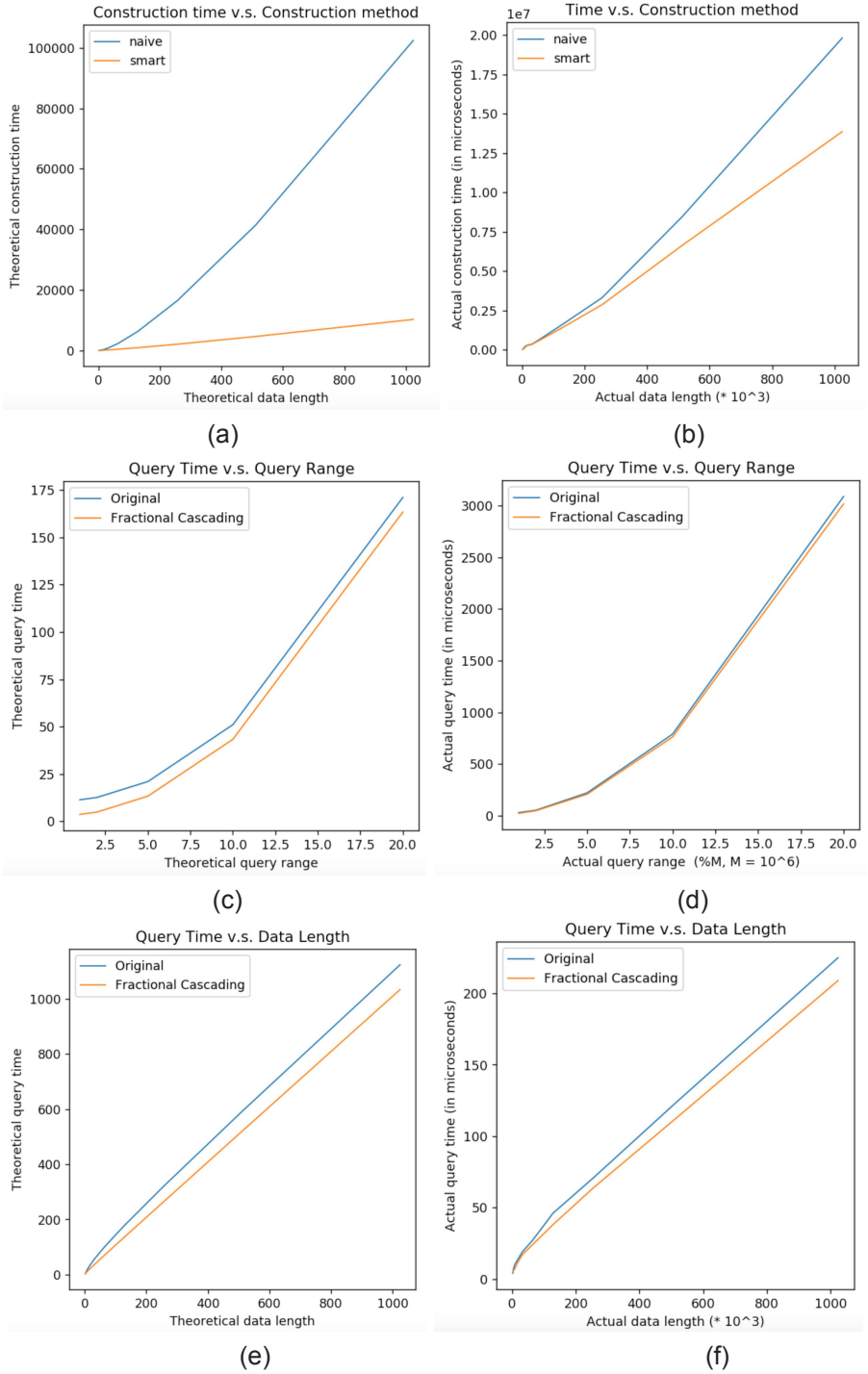

Figure 4: Theoretical (left) and actual (right) time complexity of different data structures / algorithms in each experiment (from top to bottom: experiment 1, 2, 3)

i ($n = 2^i \times 10^3$)	1	2	3	4	5
Naive Org RT	24042 us	47545 us	169501 us	288835 us	354557 us
Sorted Org RT	**20832 us**	**41238 us**	**106932 us**	**272013 us**	**343021 us**

i ($n = 2^i \times 10^3$)	6	7	8	9	10
Naive Org RT	754080 us	1591904 us	3302721 us	8448401 us	19805412 us
Sorted Org RT	**686903 us**	**1391116 us**	**2860969 us**	**6626540 us**	**13857526 us**

Table 4-1: Construction time complexity of different lengths of data points

Search Range s	1% M	2% M	5% M	10% M	20% M
Avg Solution size k	101	399	2500	9997	40019
Sorted Org RT	28 us	51 us	219 us	788 us	3089 us
Fractional Cascading RT	**21 us**	**46 us**	**206 us**	**760 us**	**3018 us**

Table 4-2: Range Reporting time complexity of different sizes of range

i ($n = 2^i \times 10^3$)	1	2	3	4	5	6	7	8	9	10
Avg Solution size k	4	9	19	40	78	158	321	636	1729	2559
Sorted Org RT	**4 us**	7 us	10 us	13 us	19 us	27 us	46 us	71 us	123 us	225 us
Fractional Cascading RT	5 us	**6 us**	**7 us**	**11 us**	**17 us**	**24 us**	**38 us**	**64 us**	**112 us**	**209 us**

Table 4-3: Range Reporting time complexity of different lengths of data points

4.1 Construction time v.s. Construction method

As shown in Figure 4(a), 4(b) and Table 4-1, the results we obtained from experiment 1 align with our theoretical time analysis about the two construction algorithms of the *Original RT*. Theoretically, the naive construction algorithm would cost $O(n \log^2 n)$ time, while the sorted approach would cost only $O(n \log n)$ time. Therefore, as shown in Figure 4(a), when we increase n, we should be able to observe two lines with positive and increasing slopes (due to the $\log n$ and $\log^2 n$ factor), ie. both line should grow faster than the line function $O(n)$. Moreover, because $\log^2 n$ is always greater than $\log n$ for any positive $n > 1$, and the ratio between these two factors ($\frac{\log^2 n}{\log n} = \log n$) is increasing when n becomes larger, the naive construction should grow faster than the one of the sorted algorithm.

The actual experiment result is shown in Figure 4(b) and Table 4-1, although the discrepancy between two lines is not as big as the one in 4(a), we could still observe expected behavior in both construction algorithms. As shown in Figure 4(b), both algorithm grows faster than linear with increasing slopes, the naive construction method is always slower than the sorted method, and the time differences between these two methods are increasing as n becomes larger. Consequently, the expected time-bound of both construction algorithms on the *Original RT* could be observed in practice.

4.2 Query Time v.s. Query Range

Figure 4(c), 4(d), and Table 5-2 give the results when we fixed the data length n and include different ranges s of queries. In the theoretical analysis, the *Sorted Original RT* could perform a range query $O(k + log^2 n)$ time and the query time complexity of the *Fractional Cascading RT* is $O(k + log\ n)$. Also, since we are picking data points uniformly from the universe, it is not hard to see that the expected number of data points that falls into the specified range should be $E[k] = (\frac{s}{N})^2 n$, where s is the query range and N is the size of the universe. Therefore, since we fixed the length n in this experiment, it is expected to see that for both *Range Trees*, there is a linear relationship between k and the time spent and a quadratic relationship between s and the consumed time. These two lines should be asymptotically the same as the quadratic line. Moreover, the line of *Sorted Original RT* should be slightly above the line of *Fractional Cascading RT* because the extra $O(log\ n)$ factor, the distance between the two lines should stay roughly the same as the n is fixed (ie. the two lines are close to parallel), but this difference might not be obvious because it is dominated by the quadratic term. A visualization of the theoretic behavior of these two data structures is plotted in Figure 4(c).

The actual running time of these two data structures is shown in Figure 4(d) and Table 4-2. Firstly, observe that the time spent for all different lengths of queries is much less than the construction time above, this observation matches with the theoretical analysis that the construction ($O(n\ log\ n)$) is a lot slower than the query ($O(k + log^2 n)$ or $O(k + log\ n)$). Secondly, the query time in both data structures seems to have a quadratic relationship with the range size s and have a linear relationship with the solution size k (this could be verified via Table 4-2). This observation again aligns with our expectations above. Lastly, the distance between two lines does not stay constant as expected, this might be caused by the range size. When the range size is small, all points that are within the specified range might locate on a deep-level small subtree. So even though the theoretical tree search steps (such as find the LCA, trace the path from LCA to successor/predecessor) still will cost $O(log\ n)$, in reality, these operations could be performed in time way smaller than $log\ n$. As the range size increases, the subtree that contains all valid points becomes larger and results in more time in tree searches. Thus this observation does not contradict the theoretical analysis before.

4.3 Query Time v.s. Data Length

As represented by Table 4-3, Figure 4(e), and 4(f), we fixed the range of each query and varied the length of the data. According to the theoretical analysis in section 2.3, the time complexity of range reporting for *Sorted Original RT* is $O(k + log^2 n)$ and for *Fractional Cascading RT* it is $O(k + log\ n)$. Firstly, observe that in this experiment, although we fixed the query range s as 50,000, the value of k is also expected to grow linearly as we increase the n. Since each data point is picked uniformly from the universe, for data length n, the expected number of data

points that fall into the specified range should be $E[k] = (\frac{50000}{N})^2 n$, where N is the size of the universe. Moreover, since we are using the exact same data set and query range, k will be the same for both *Range Trees.* So that these two data structures won't behave the same as the poly-logarithm line $O(log^c n)$ as if k is fixed, instead it will grow asymptotically the same as the linear function $O(n)$. In another word, both lines would have positive slopes, even though their slopes will slightly increase with larger n due to the $log\, n$ and $log^2 n$ factors, it will be dominated by the linear function and negligible. Because the *Sorted Original RT* has an extra $(log\, n)$ factor, its line should be slightly above the line of the *Fractional Cascading RT*, and the difference between those two lines will become larger as n increases, but again this difference might be dominated by the linear function. The theoretical bounds of these two data structures are plotted in Figure 4(e).

Figure 4(f) and Table 4-3 demonstrate the actual running results of this experiment. Similar to the previous experiment, the results of this experiment also help verify that the query time should be a lot faster than the construction time. Not surprisingly, the results we obtained is actually very similar to the theoretical analysis above, for both data structures, the query time grows linearly as we increase n, the query in the *Sorted Original RT* is slightly slower than the query in the *Fractional Cascading RT*, and the difference of them will slightly increase when n becomes larger. Therefore, combining the result from section 4.2, for both *Sorted Original RT* and *Fractional Cascading RT,* the time bounds of range reporting could be achieved in practice.

5. Conclusion

In conclusion, for all three variants of the *Range Tree,* the theoretical expected construction and range reporting time bounds could be successfully achieved in practice.

On the one hand, we compared two construction methods of the *Original Range Tree*, both constructs a *WB-BST* tree T_x on x dimension and a *WB-BST* tree on y dimension for each node of T_x. The only difference is the naive approach takes $O(n\, log^2 n)$ time, while the sorted method takes $O(n\, log\, n)$ time by sorting all data points by y first. When we vary the data length n, the actual time cost of these two construction methods aligns with their theoretical analysis, both of them grow faster than the linear function $O(n)$ but slower than the quadratic function $O(n^2)$. The naive approach is always slower than the sorted approach and the difference between them is growing as we increase n. Thus the sorted approach would be preferable in any length.

On the other hand, we experiment about the range reporting time complexity of the *Sorted Original Range Tree* and the *Fractional Cascading Range Tree*. The *Sorted Original Range Tree* requires $O(k + log^2 n)$ time to report all points within a certain range, while the *Fractional Cascading Range Tree* improves it to $O(k + log\, n)$ by incorporating the idea of *Fractional Cascading*. Since both complexities involve two parameters, k, and n, we performed two experiments with either one of the parameters being fixed.

When n is fixed, we vary the query range s to indirectly vary the solution size k, the experimental results again match with the expectation. Since we are generating data points and

queries uniformly, the expected size of the solution should be $E[k] = (\frac{s}{N})^2 n$. This is verified by the experiment outcomes, the time spent for both data structure has a quadratic relationship with s and a linear relationship with k. The *Sorted Original Range Tree* is expected to be slightly slower than the *Fractional Cascading Range Tree* by a constant factor, but we observed that the difference between them is slowly increasing, this is because the tree search operations are cheaper (only in practice, not asymptotically) to perform when the range of query is small.

Also, if we fixed the query range s, the actual time costs of both data structures support the theoretical analysis. First, notice that even though we fixed s, k is still linearly related to n, so the time spent by both data structures are linearly correlated to n. Also because of the extra $(log\ n)$ factor, the *Sorted Original Range Tree* is slower than the *Fractional Cascading Range Tree* all the time, and the difference between them is slowly increasing. Consequently, in practice, for all valid values of k and n, the *Fractional Cascading Range Tree* would outperform the *Sorted Original Range Tree*.

To sum up, all three types of *Range Tree* could support fast range reporting with prefixed data set and relatively slow construction time, so the *Range Tree* should be used only when the data set is known and fixed in advance (ie. static, not change over time) and there are a lot of range reporting queries. Moreover, considering that the theoretical construction time of the *Fractional Cascading Range Tree* is also $O(n\ log\ n)$, it would be the better choice for any values of k and n.

6. References

[1] Bentley, Jon Louis, and James B. Saxe. "Decomposable searching problems I: Static-to-dynamic transformation." *J. algorithms* 1.4 (1980): 301-358.

[2] Chazelle, Bernard. "Lower bounds for orthogonal range searching: I. the reporting case." *Journal of the ACM (JACM)*37.2 (1990): 200-212.

[3] Chazelle, Bernard. "Lower bounds for orthogonal range searching: part II. The arithmetic model." *Journal of the ACM (JACM)* 37.3 (1990): 439-463.

[4] Junhao, G. and Wirth, T. (n.d.). *Week 7 Slides, COMP90077*. [online] Melbourne: University of Melbourne. Available at: https://canvas.lms.unimelb.edu.au/courses/80169/files/3313991?module_item_id= 1896991.

[5] Melman Gabi, spdlog, (2020), GitHub repository, https://github.com/gabime/spdlog.