APS Assignment

M.V.

ID: -

Task1.1

Algorithm 1

Let $ListA$ be a list of unique values and $number$ be the input number. Furthermore, let I be an interval with endpoints x_s and x_f, such that $x_s < x_f$.

Algorithm 1

```
1: function AddNewNumbers(ListA, number)
2:     ListA.append(number)
3: end function
```

Explanation

Algorithm 1 takes in two arguments, $ListA$, where we wish to store the new element, and the number $number$, which is the input, then it appends number at the end of $ListA$.

Complexity

The algorithm performs one operation, but it is called for each element of the input. Therefore, the total complexity of building the list, given n elements is $\Theta(n)$. Since we are adding elements to the end of $ListA$, we do not need to change the indices so this takes constant time.

Correctness and Termination

Note that the algorithm always terminates, as it does not have any loops in it and is correct, as it does what it is supposed to.

Algorithm 2

Algorithm 2

```
1: function Query(ListA, StartingPoint, EndPoint)
2:     for element in ListA do
3:         if element ≥ StartingPoint and element ≤ EndPoint then
4:             print element
5:         end if
6:     end for
7: end function
```

Explanation

Algorithm 2 takes in three arguments. A list of numbers $ListA$, and an interval with two points, $StartingPoint$ and $EndPoint$. Then it goes through the elements of $ListA$, checking if each one of them is between the points of the given interval, this takes two operations. If it is, it prints the current element, another operation.

Complexity

Lines 3 and 4 us 2 (+ 1 conditional) operations, which is in the worst case 3. Thus, the complexity of this algorithm is $\Theta(3n) = \Theta(n)$. To prove this for O we just need a $c \geq 3$ and $c \leq 3$ proves Ω.

Correctness and Termination

The algorithm is also guaranteed to terminate, if the $ListA$ has a finite number of elements, as we observe each element only once. It is also correct, as it only returns the values that are between (or equal to) the endpoints.

Task1.2

Algortihm 3

Let $ListA$ be a list of unique values and $number$ be the input number. Furthermore, let I be an interval with endpoints x_s and x_f, such that $x_s < x_f$.

Algorithm 3

```
1: function AddNewNumbers(ListA, number)
2:     ListA.append(number)
3:     Sort ListA
4: end function
```

Explanation

Algorithm 3 takes in the same parameters as Algorithm 1. The new number then is put at the end of $ListA$, then we sort it using a sorting algorithm, since the algorithm will be implemented in python, we can say that this will be Timsort. Note that this algorithm could be improved in a number of ways, such as looking for the correct place of the number using binary search and inserting it there.

Complexity

Given that we want to initialise a list of n values, we call the function n times. We put the new element at the end of $ListA$, $\Theta(1)$ and then we sort it using a sorting algorithm, $\Theta(n \log_2 n)$. So the total complexity is $\Theta(n^2 \log_2 n)$.

Correctness and Termination

Algorithm 3 will always terminate, given that the sorting algorithm terminates. Furthermore, to prove correctness, we can see that for given our $ListA = []$ and $number = input$, we just cons the number to $ListA$, and as $ListA$ has only one element, it is already sorted.

Assuming now that Addnewnumbers works correctly, up to n numbers. That is $|ListA| = n$, and it is sorted. Then for the $(n+1)$-th number, we do as before, putting it at the end of $ListA$, and sorting it again. This will of course result in a sorted list.

Algorithm 4

```
1: function QUERY(ListA, StartingPoint, EndPoint)
2:     i =BinarySearch'(ListA, StartingPoint)
3:     for element in ListA starting from position i do
4:         if element ≤ EndPoint then
5:             print element
6:         else
7:             break
8:         end if
9:     end for
10: end function
```

Algorithm4

Explanation

Algorithm 2 takes in the same three arguments as in Algorithm 2. A list of numbers $ListA$, and an interval with two points, $StartingPoint$ and $EndPoint$. We use binary search to find the $StartingPoint$ in $ListA$ or the number closest to it from above.

Then for each element of $ListA$ starting from the $StartingPoint$, or from the number that is greater than that, we check if the element is less than or equal to the $EndPoint$, if it is, we print the element, if we encounter something that is greater, we break from the loop.

Complexity

As we have seen before BinarySearch' takes $\log_2$n complexity. And then we make k number of comparisons, where k is the number of elements that are in the given interval. This hold only because the list is sorted, and as soon as we encounter an element that is greater than the endpoint, we exit the loop, and so we do not keep on looking. Therefore the total complexity is $\Theta(\log_2 n + k)$, where k is the number of elements in the interval.

Correctness and Termination

Suppose that BinarySearch' works properly and that $ListA$ is sorted. We prove correctness by induction on k. If $k = 1$, we obtain our starting position, then we check if $k \leq Endpoint$, then return k, if not, terminate. Assume Algorithm 4 works and returns k results. Then for the case of $k+1$, we check if the element is greater than the $EndPoint$, if it is, we exit the loop, if it is not, we return $(k+1)$-th element as well. Hence Algorithm 4 holds for $k+1$ values. This proves correctness.

The algorithm also terminates in every case. The only looping conditions that would keep it running for ever would be an infinite list, where from a point on all elements lie in an interval.

Task 2.1 & Task 2.2

Complexity

The first thing we have to do is to keep the input numbers in a sorted list. For this reason Algorithm 3 is used, which has a complexity of $\Theta(n \log_2 n)$. After we have our list of numbers, we need to make

each number a (leaf) node. That is we have to go iterate through the list of n numbers, and perform some operations.

When a node is created, there are 4 operations taking place. We set the value of the node as given, we set the left and right child to None by default, and finally append it to our list of nodes. Therefore our initial loop takes $\Theta(4n) = \Theta(n)$ time.

After that we call the function on the leaf nodes, which will go through on every second element, call the getrightmost function on it, which on the ith level takes $2i$ operations. Create a new node (4 operations) and connect them correctly (3) and finally store it in an array (1). The function then is called again on the next set of nodes, which if we have n number of nodes then: $|NextLevelNodes| \leq \frac{n}{2}$.

Therefore, in the worst case, we have half of the number of current nodes at each level, so for n number of leaf nodes, we call the function $\log_2 n$ times. And thus:

$$\sum_{i=0}^{\log_2 n} \left(\sum_{j \in \{0,2,4,\ldots\}}^{\frac{n}{2^i}} (8+2i) \right)$$

As we go from the 0-th level up to the $(\log_2 n)$-th level, and on each level we have $\frac{n}{2^i}$ number of nodes, where i is the current level. Since we stop at every second element only, j takes even values. The $(8+2i)$ comes from the operations we perform on the nodes. This all simplifies to:

$$\sum_{i=0}^{\log_2 n} \left(\sum_{j \in \{0,2,4,\ldots\}}^{\frac{n}{2^i}} (8+2i) \right) = \sum_{i=0}^{\log_2 n} \left(\frac{n(8+2i)}{2^{i+1}} \right) =$$

$$= n \sum_{i=0}^{\log_2 n} \left(\frac{8+2i}{2^{i+1}} \right) = n \left(\sum_{i=0}^{\log_2 n} \left(\frac{8}{2^{i+1}} \right) + \sum_{i=0}^{\log_2 n} \left(\frac{2i}{2^{i+1}} \right) \right) \leq$$

$$\leq 4n \left(\sum_{i=0}^{\log_2 n} \left(\frac{1}{2^i} \right) \right) + n \log_2 n \left(\sum_{i=0}^{\log_2 n} \left(\frac{1}{2^i} \right) \right) \leq 8n + 2n \log_2 n$$

Where the 4th inequality uses that $i \leq \log_2 n$ and the last one uses the closed formula for the infinite (convergent) geometric series, which is greater than all of its partial sums. To that we still need to add the complexity of the initialisation, which was $4n + n \log_2 n$. But, we also know that $n \ll n \log_2 n$, so we get that our algorithm is $O(n \log_2 n)$.

Proof by induction on n: For the base case, look at $n = 1$. Then $8n + 2n \log_2 n = 8 \leq 8c$, which holds for any $1 \leq c$. Let us assume now that our claim holds up to n. Then for $n+1$ we have that:

$$8(n+1) + 2(n+1)\log_2(n+1) \leq 8\left(n + \frac{n}{2}\right) + 2(n+n)\log_2(n+n) =$$

$$= 12n + 4n(\log_2 2 + \log_2 n) = 16n + 4n \log_2 n =$$

$$= 2(8n + 2n \log_2 n) \leq c(8n + 2n \log_2 n)$$

for any $1 \leq c$ which proves our assertion.

Task2.3

Complexity

Given that we have build our tree already, a single query consists of FindSplitNode and the OneDRangeQuery algorithms. Firstly, let's look at the FindSplitNode algorithm. We have 3 operations

outside of the loop on line 2,3 and 12, we ignore these for now. We also have a while loop from line 4 to line 11. There are 3 conditions to check at the start of the loop, and 1 more condition and an operation in the body. In the worst case, we do not find a split node, or not until the leaves, and the algorithm goes through the whole height of the tree. Thus, the complexity is:

$$3 + \sum_{i=0}^{\log_2 n} 5 = 5 \log_2 n + 8$$

So our FindSplitNode has complexity $\Theta(\log_2 n)$.

Proof by induction on n: For our base case, we have $n = 2$. Then $5 \log_2 2 + 8 = 13 \leq clog_2 2$, which holds for $13 \leq c$. Similarly, we have that $5 \log_2 2 + 8 = 13 \geq clog_2 2$, for$13 \geq c \geq 0$. So the base case holds.

Now assume it also holds up to n, then $5log_2(n+1) + 8 \leq 5log_2(n+n) + 8 = 5 \log_2(2n) + 8 = 5 \log_2(n) + 5 \log_2(2) + 8 = 5 \log_2(n) 13 \leq c \log_2(n) + 8$, which holds for $5 \leq c$. Similarly, we could take $13 \geq 1 \geq c \geq 0$, for the other case. This proves that $5 \log_2(n+8) = O(log_2(n))$ and that $5 \log_2(n+8) = \Omega(log_2(n))$, this proves our claim.

Now, let's consider the OneDRangeQuery algorithm. As said before, it firstly finds the split node, since we have proven that this takes $\Theta(log_2 n)$ complexity, we will use this simplification. The algorithm then inspects the left child of the split node, and goes down to the leaves. This takes 4 operations per iteration, and in the worst case, our split node is the root of the whole tree, therefore our summation goes between the same indices. This is however repeated for the right child of the split node, this our final sum is:

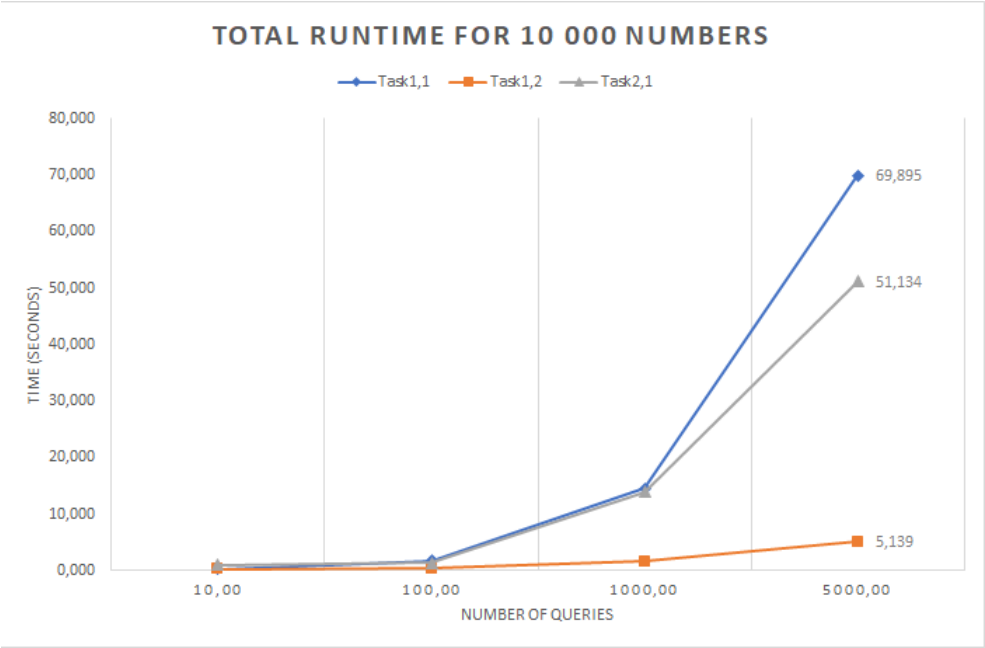
$$4 + 2 \sum_{i=0}^{\log_2 n} 4 = 8 \log_2 n + 8$$

Which looks almost similar to the FindSplitNode complexity, therefore the proof is the same, for O we take $16 \leq c$ and for Ω we again take $1 \geq c \geq 0$ Therefore our overall complexity of a single query is $\Theta(log_2 n)$.

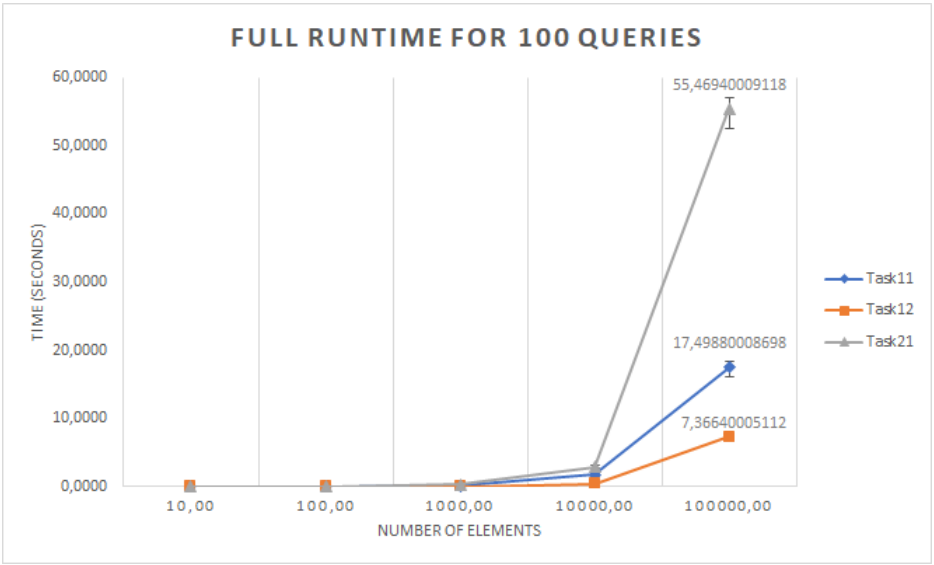
Task2.4

Firstly, note that we can either vary the number of queries, and the number of elements in the list. Note also that lines are drawn only to show the general trend, samples have been taken with the values shown on the x axis. The data points are the average cases out of 5 runs, error bars are also included on Diagram 2, showing the longest and shortest run time, but note that these differences were generally small. For the whole collection of data, see data.xlsx. Let's consider the total run time for 10 000 elements, varying the number of queries.

As we can see on Diagram 1, the naïve approach was bested by both other implementations, Task2.1 however took longer however than Task1.2, despite the results we found from our analysis. This must stem from the implementation of Task2.1.



Now consider the case, when we fix the number of queries to a 100 and vary the number of elements. This is the most unfortunate case for Task2.1, as most of the complexity is coming from building the tree itself, as we can see it is too costly to construct the tree for 100 queries.



In conclusion, our tree-building approach should be avoided, if we only have a small number of queries, but a large number of elements, and as one can see, the best performance was given by Task12 in both cases.

Task 3.1 & Task 3.2

Description and Complexity

Firstly, we have a function to store our numbers, this is just our Algorithm 1 from Task1.1, which has complexity $\Theta(n)$. In this case, the numbers themselves are lists, but the algorithm can handle this case as well.

Then there is a WithinRange function, which takes in 3 points ($PointA, PointB, PointC$), which are lists in our case and a number (i) as parameters. It then checks if the current coordinate of $PointA$ is between the current coordinates of $PointB$ and $PointC$, if it is, it increments i.

It then checks if we have reached the last coordinate, this is our base case, then it returns true. Otherwise, it returns the result of the Boolean expression True and WithinRange(PointA, PointB, PointC, i), which will check if the next coordinates are adequate and etc.

Therefore, we have 6 operations per coordinate per number. Worst case scenario is that the number is within the rectangle, so we have to check every coordinate. This gives a complexity of $O(6m) = O(m)$, where m is the number of dimensions. This function does the heavy lifting for the queries. The query function just calls WithinRange on each of the points and does some string manipulations and returns the result. Therefore, our overall complexity is $O(nm)$, where m is the number of dimensions and n is the number of points.

Correctness and Termination

We prove termination first. When the Query function is called on a set of points, we have 2 loops to consider. One is the for loop, which iterates through the list of points. Assuming that this list of points is finite, this for loop will terminate. The next loop happens via recursion, and it examines the coordinates of the points. In case the current coordinates of the point is not between the endpoints we return false straight away, and if it is, we increment i, which indicates the current coordinate we are looking at. Since we cannot have points with infinite dimensions, this will too terminate at some point. Then finally, we have a for loop to print the results, which is finite as well.

To prove correctness, we must examine both the WithinRange function, which deals with the number of dimensions, and the Query function which deals with the number of points.

First consider the case when the number of dimensions equals 1. Then for each point in our list, the WithinRange function will be called. Then the only coordinate is examined, if it is not between the coordinates of the endpoints, the function returns false, as it should. If it is, i is incremented, then since it equals the number of dimensions, we return true.

Assume now that the WithinRange functions work correctly for points up to dimension n. Then for a given point with $(n + 1)$ coordinates, we have that, in case any of the first n coordinates of the points do not fall between the endpoints of the interval, we return false in that function call, so assume otherwise. Then we have a $(n + 1)$-th function call, the condition is met, and i gets incremented. Since our point has n+1 coordinates, we return true.

The query function calls the WithinRange for each of the points. It does so independently of each point, and then appends the given point to the results, if the WithinRange function returns true on that particular point.

Task 4.1

Algorithm 5

```
1: function MEDIAN(ListA)
2:     Sort ListA
3:     L = length of ListA
4:     if L is odd then
5:         return ListA[L/2]
6:     else
7:         return ListA[L/2 − 1]
8:     end if
9: end function
```

Description

Algorithm 5 is a function called Median, which takes in an unsorted list as input. Since we require the list to be sorted, that is the first thing to do. Again since, the implementation will happen in python, Timsort will be used. Then we check if the length of the list is odd, if it is, we return the middle element. Otherwise, we round down, which in this sense means that we take the smaller out of the two middle elements.

Complexity

Sorting the list, using Timsort takes $\Theta(n \log_2 n)$ time. Acquiring the length of ListA takes $\Theta(1)$, and finally computing the result and checking the if-condition also takes $\Theta(1)$. Therefore our combined complexity is $\Theta(n \log_2 n)$.

Task 4.2

Time complexity

The BuildKDTree algorithm consists of some constant operations, such as checking if P only contains a leaf, and returning it, etc. We omit these, as we already know that the Median algorithm used in BuildKDTree is of complexity $\Theta(n \log_2 n)$ and this dominates constants. Therefore, for each problem the cost is $n \log_2 n$, and then we have a recursive call on Pl and Pr. Since we split on the middle element of the list, (or one of the middle elements) both P_r and P_l have $n/2$ elements, (or $n/2$–1 and $n/2 + 1$ elements). This gives us the recurrence relation: $T(n) = 2T(n/2) + (nlog_2 n)$.

Claim: $T(n) = \Theta(n(\log_2 n)^2)$. We will prove this by induction on n.

For the base case, consider $n = 2$, then we have that $T(2) = 2T(1) + 2\log_2 2 = 4 \leq c(4 \log_2 2)$, so we ta ke4 $\leq c$ for O and $4 \geq c$ for Ω. Assume now that $T(n) = \Theta((nlog_2 n)^2)$, holds up to $(n-1)$. Then for the n-th case, we have that:

$$T(n) = 2T(n/2) + (n \log_2 n) \leq 2(c\frac{n}{2}(\log_2 \left(\frac{n}{2}\right))^2) + (n \log_2 n) =$$
$$= cn(\log_2 n - \log_2 2)^2 + (n \log_2 n) = c(n \log_2 n)^2 - 2c(n \log_2 n) + cn + (n \log_2 n) \leq$$
$$\leq c(n \log_2 n)^2$$

Where the first inequality follows from our induction assumption, as $(n/2) \leq n-1$ for $2 \leq n$. The second equality uses the well-known logarithmic identity. The last inequality uses that because of $4 \leq c$, we have that $c(n \log_2 n) \leq n \log_2 n$, so the difference is negative. The second part of $(-c)2(n \log_2 n$ is used by the fact that $c(n \log_2 n \leq cn$ as $(n \log_2 n) \gg n$, so this difference is also negative. So $T(n) = O((nlog_2 n)^2)$

To show that $T(n) = \Omega((n \log_2 n^2))$ we only need to change direction of the first inequality and the last line. So we have that:

$$T(n) \geq c(n \log_2 n)^2 - 2c(n \log_2 n) + cn + (n \log_2 n) \geq c(n \log_2 n)^2 + cn \geq c(n \log_2 n)^2$$

Where we take a $4 \geq 1 \geq c$, to make the difference $(n \log_2 n) - 2c(n \log_2 n)$ vanish. And thus we proved that $T(n) = \Theta((n \log_2 n)^2))$.

Space complexity

Firstly, we have to analyse Algorithm 5 in terms of space. If we assume the best case scenario, that is our sorting algorithm runs in-place, without claiming any extra bit of memory, then Algorithm 5 takes constant complexity regarding space that is $\Theta(1)$.

For the BuildKDTree algorithm we have to store the following values at each level: current depth d, $axis$, selected $medianPoint$, and finally P_l and P_r. If we are given n numbers, then P_l, P_r and the $medianPoint$ will slice this list into 3 pieces, but this will still require the same amount of space. Since at no point do we store more than necessary the total space complexity of the BuildKDTree algorithm is $\Theta(n+2) = \Theta(n)$.

Task 5.2

Algorithm 6

```
1: function FullyContained(minMax1,minMax2)
2:     for d in the number of dimensions do
3:         if not(minMax2[d][0] ≤ minMax1[d][0] and minMax2[d][1] ≥ minMax1[d][1]) then
4:             return false
5:         end if
6:         return true
7:     end for
8: end function
```

Description

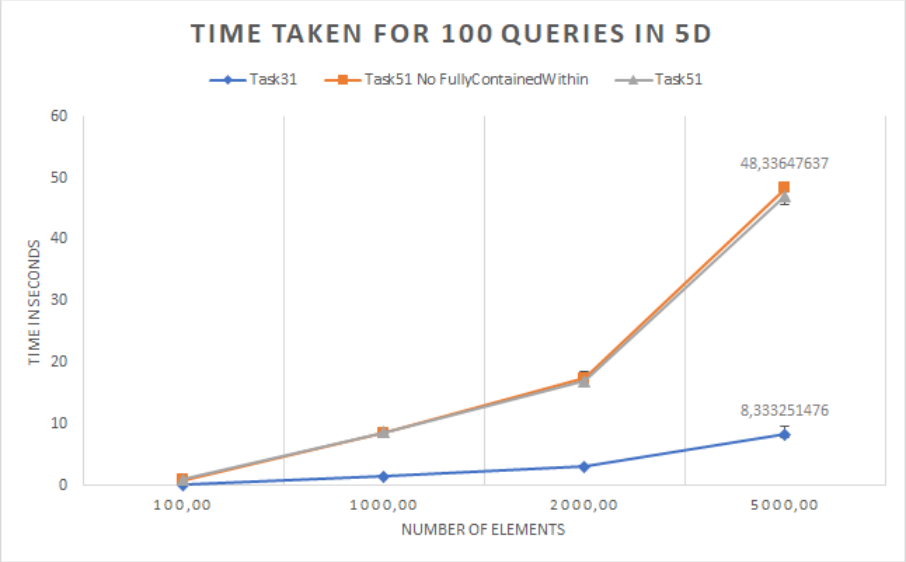
Algorithm 6 uses the same $minMax$ lists as the intersection algorithm from main.pdf. Recall that these are "lists with d pairs (where d is the dimension of the space). Each pair has the minimum and maximum values for that dimension for the region."

These lists will be created to check if the regions intersect anyway, so we do not require to perform any extra computation in this sense. The structure of algorithm 6 is also similar to that of intersection. It checks for every dimension if there exists a point outside the bounds set by the query. That is if there is a maximal element of the region that is greater than the end point of the query, or a minimal element that is less than the starting point of the query. If the algorithm finds such a point on any axis, it returns false. If the for loop successfully terminates, we return true.

Please note that we require $minMax2$ to be the minMax list created from the set of endpoints of the query, otherwise the algorithm does not return a correct answer, i.e. we cannot swap the arguments.

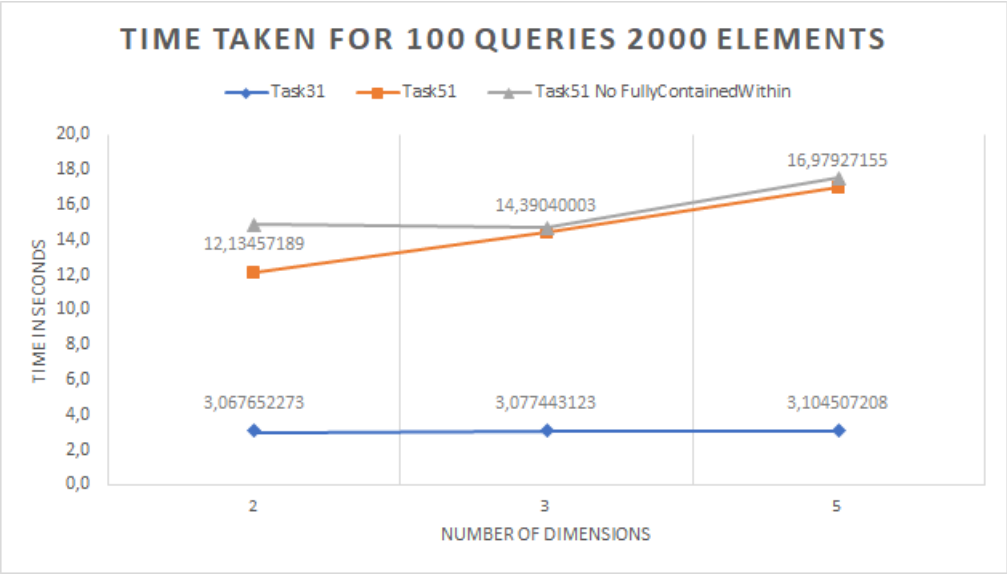
Task5.3

We can now look at the differences in the run time using the different approaches. Each data point is an average of 5 runs on the same data set, which is randomly generated. The maximum and minimum time taken is also saved, but for a clearer picture these are missing from the diagram. First look at the case when the number of dimensions is 3, and we are given 100 number of queries.



We can observe several things. Firtly note that the naive solution bested the KDtrees for all cases, secondly that there was very little difference between having the FullyContainedWithin part of the algorithm or not. This must be because this algorithm uses the same MinMax lists as the intersection algorithm, which does most of the computations. As noted, the condition itself is similar to the one in the intersection algorithm. Similarly it could be due to a random sample which does not involve cases where full containment could occur. Note that the lines are shown to highlight the overall trend, measurements have only been taken at the points shown on the x-axis.

Let us move on to other number of dimensions. As we can see on the next diagram adding an extra dimensions generates a little more complexity, but each case handles it differently. The naive solution seems to barely notice it, whereas the KDtree approaches seem to increase linearly as the number of dimensions increases. These steps are each small, however noticeable.



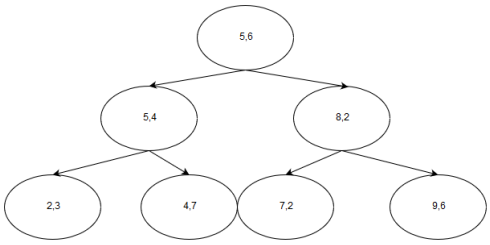
The naive solution seems to barely notice it, whereas the KDtree approaches seem to increase linearly as the number of dimensions increases. These steps are each small, however noticeable.

Task5.4

Our approach handles the case of unique coordinates in each dimension well. There is no need to make such restrictions. For example given the input

```
7 2 1
2 3
4 7
5 4
5 6
7 2
8 2
9 6
[5, 6] [5, 4] [2, 3] [4, 7] [8, 2] [7, 2] [9, 6]
```

our algorithm creates a balanced tree seen on the last line, which in depth first traversal translates to the tree:



Which is just what we wanted.

Task5.5

Description

The sphere version of our algorithm build on top of everything we have so far and it is mostly coded into the previous algorithms. Given the information of our sphere first we find the n dimensional rectangle that circumscribes it, we feed the corners of this rectangle to our previous algorithms. the computation is as follows:

Algorithm 7

```
1: function FINDCORNERS(Radius,Centre)
2:     StartPoint = EndPoint = 0
3:     for each dimension d do
4:         StartingPoint += Centre[d]-Radius
5:         EndPoint += Centre[d]+Radius
6:     end for
7:     return StartingPoint, EndPoint
8: end function
```

Then after we get the results, we filter the points using the equation of an n-dimensional sphere, i.e. we only return the points that satisfy:

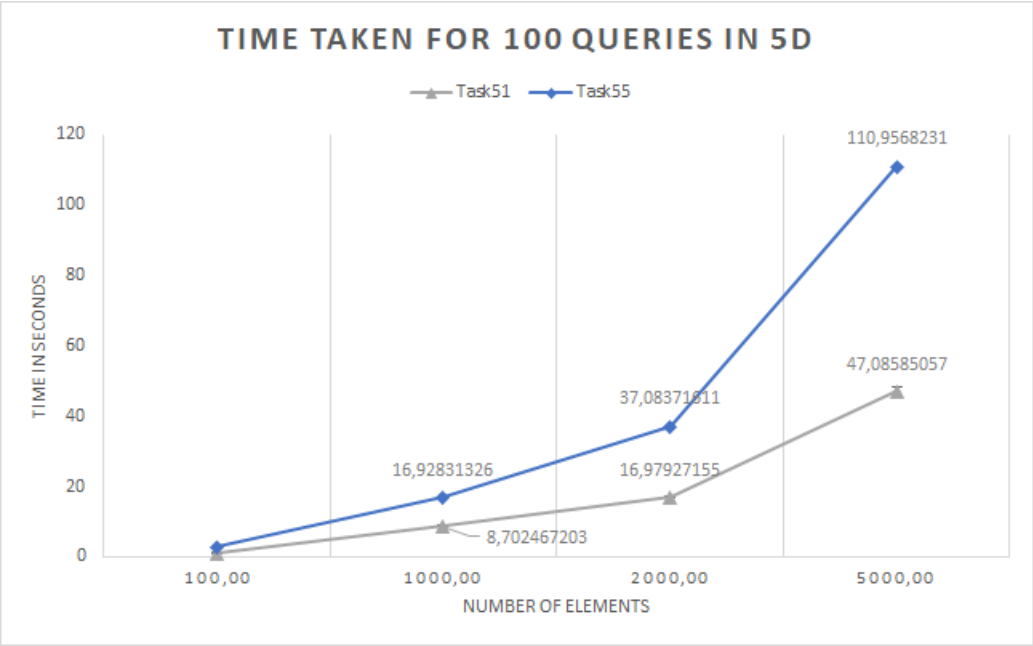
$$(Axis0 - Centre[0])^2 + (Axis1 - Centre[1])^2 + ... + (Axisn - Centre[n])^2 \leq Radius^2$$

And so our algorithm only checks if this holds for each one of the points, and returns those where it does.

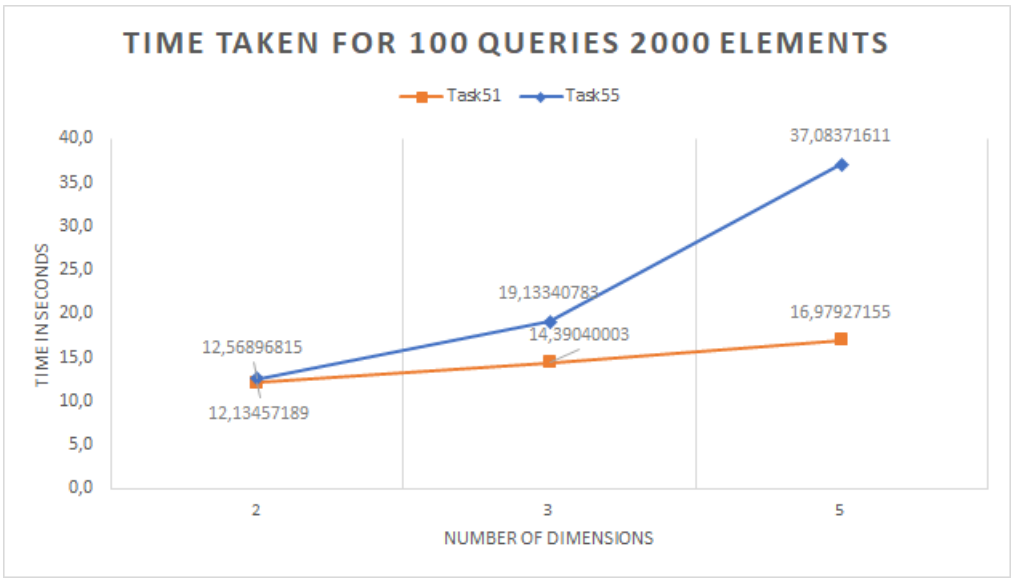
Comparison

As expected the sphere version requires more time, it seems as if the results are approximately doubled in size to the corresponding rectangle data points. However if we look close enough we can see that the difference would grow larger as the number of elements would increase.

Note that the data points are averages out of 5 runs for 100 random queries. Lines are drawn only to show the general trend. The first diagram would also show the maximum and minimum run time at each point, since however these were so close to one another in every case, the differences are negligible and the bars barely visible.



The sphere version of the algorithm is also more sensitive to the change in dimensions and the complexity seems to greatly depend on this as well, a lot more than the other versions.



Task5.6

Description

The approach for Task5.6 would be the same. First, given the points of the polygon as input, we compute the circumscribed rectangle. We use the CreateMinMax algorithm for this as in other cases. Then we perform the search on the rectangle. Then we need to filter our result further and return those points only which are inside the polygon as well.

A way of checking this in 2D could be to find the nearest edge of the polygon for every point inside the rectangle, then given that our point is P and our corner points of the polygon are A and B, we could compute the cross product:

$$\vec{AB} \times \vec{AP} = x_{AB} \cdot y_{AP} - x_{AP} \cdot y_{AB}$$

$$x_{AB} = x_B - x_A,\ x_{AP} = x_P - x_A\ ...$$

and see if it has the same sign as the cross product with the origin. If it does, then P is inside the polygon, otherwise it is not. And then we could generalise this idea to other number of dimensions.

In Task56.py I implemented the first 2 steps of this process, that is the conversion of the polygon to a rectangular region and then performing the search on that. The filtering would still need to be done.