I know I have misspelt query throughout my code, however that is inconsequential to the code itself.

1.1.

First we create an empty array of size n. Then within a loop that iterates n times
we go through each of the n values and add them to the array while also doing the comparison xs<value<xf. If that evaluates to true
we add value to a second, dynamic array. Algorithm in the worst case(all values in are also within the range xs..xf)
will have a complexity of 2n, so O(n).

```
array = []      # create an empty array of size n
dynarray = []   # empty dynamic array of size 1 // We assume that the array
doubles itself every time in needs more space
i = 0
loop n times:
    Increment i by 1
    Add ni to array
    Check if ni>=xs and ni<=xf when True:
        add ni to dynarray
    Print dynarray
finish loop
```

Proof:
Base case n = 1, List is [1], xs = 0, xf = 2.
Loop executes once. i = 1. Add n[i], which is n[1] = 1 to the array. Check if 1 is >= 0 and <= 2, evaluates to True. 1 is added to dynarray and array.
Output:
1

Algorithm is correct for the base case.

Case 2:
n = 9, List is [1,5,3,9,6,4,7,2,8] (The numbers from 1 to 9 unsorted). xs = 3, xf = 6.
Loop executes 9 times. For execution number 2,3,5 and 6 the conditional is met and the numbers at the corresponding indices get added to both array and dynarray. All other numbers
are added to array only.
Output:
5 3 6 4

1.2

We start with an initial array and some input values. For the first iteration of the Algorithm
the initial array will be empty. Then all the values get added to the array, worst case time complexity is O(n).
Then we execute mergesort over the array. Worst case time complexity of this is O(nlogn). Overall worst case time complexity is
O(n + logn) = O(nlogn). Whenever we want to add new values to the array we do so and just repeat the merge sort.
For querying - we first find with binary search the smallest number in our already sorted array that falls into the range (xs, xf). Then we go through the following members
of the array until we reach a number that is out of the range. Return the numbers that were in range. Time complexity is - First we do binary search which is in
worst case O(logn) then we go through k+1 numbers where k is the number of output numbers, so O(k+1) = O(k). Overall that is O(logn + k).

```
BuildList(array, n) # default value for array is []
i = 0
Loop n times:
    i = i + 1
    Add ith element of n to array
end Loop
sortedArray = Execute mergesort over array

Querry(sortedArray, xs, xf)
outputArray = []
With binary search find first element of sortedArray that is bigger than xs
Loop until current element is bigger than xf:
    Add current element to outputArray
    Go to next element
end Loop
```

2.2
The input array first needs to be sorted. I've done this through mergesort which has complexity of O(nlogn).
Then the algorithm goes through the list and creates a tree with 2n-1 nodes, i.e. the build algorithm executes 2n-1 times.
Time complexity becomes O(nlogn + 2n-1)= O(nlogn + n) = O(nlogn)

2.3
p - point
xs - start of range
xf - end of range

We first show that any reported p lies between xs and xf and if p is stored where the path to xs or xf ends,
we check if p is inside the range. If it is not on that path, we assume p is reported when ReportSubtree is called.
Let v be the node in ReportSubtree(v.right), such that p is reported in that call.
Assume v lies to the left of vsplit. Then p <= value stored at vsplit. Because the path to xf goes to the right of vsplit,
p < xf. Since p is in the right subtree of v and the path to xs goes to the left of v ,
p > xs. Therefore xs < p < xf. It is proven that p is within the range.
Now to prove that any point in the range will be reported. Let k be the node
in which a point p is stored and v be the closest node upwards that is visited by the query.
If p is to be reported v = k. Assuming v =/= k for a contradiction.
Therefore v cannot be reported by ReportSubtree. That means v is on the search path to xs and/or xf.
I will consider the case that it is on both, but it doesn't make much difference since they cases are similar.
We assume k lies on the left of v. Therefore the search path of xs goes to v, otherwise it wouldn’t be the closest
node to k. However this would imply that xs > p. Similarly if we assume k to be to the right of v, that would imply p > xf. In both cases our (intentionally wrong) assumption is contradicted and p must lie in the range.

Time complexity:
In the worst case scenario all leaves would have to be reported which is O(n). However

the query is output sensitive, i.e. the time heavily depends on the number of points needed to be reported.
If the number of reported points is m, then the total time spent in the ReportSubtree sub-function is O(m). The remaining time is spent traversing the nodes of the tree and since it is
a balanced binary tree that time is O(logn). Total time for a single query is therefore O(m + logn).

2.4
Graphs for this task are at the end of the report and the raw data is in the runtimes.xlsx file.

3.1
Time complexity:
There are 3 loops each nested within each other. The outermost loop executes n times, n being the size of the input list.
Middle loop executes d times, d being the dimensionality of the points. And the innermost loop executes d times too.
Final time O(n*d*d)= O(n * d^2)

3.2
input : ["2 2 1", "1 2", "4 4", "[0 0] [2 2]"]
list = [[1,2],[4,4]]
querry = [[0,0],[2,2]]
Assume that both of the points are reported as within the range.
The innermost loop will execute 2*2*2 times = 8. 4 times on each point.
Each coordinate of the point is checked once against the same coordinate on the range.
So for point p, s denoting the first point of the range and f the second, p will be reported if:
xs < xp < xf and ys < yp < yf.
That holds true for [1,2]. However it doesn't for [4,4] which contradicts our assumption.

4.1

```
function Median(list, axis)               // assuming indexing starts from 1
    axis is the axis on which we want the list to be sorted
    if list is not sorted
        Sort the list on the given axis
    end if
    if size of list is even then
        halfLength = size / 2
        median = list[halfLength]
    else if size is odd then
        halfLength = size / 2       // rounded up
        median = list[halfLength]
    end if
end function
```

All the operations in the function except sort take constant time O(1). The time complexity depends entirely on what sorting algorithm we're going to use.
Assuming we use a O(nlogn) sorting algorithm like mergesort, time complexity of Median is also O(nlogn). However if we assume that the list is pre-sorted it would be O(1).

4.2
The most significant operation performed on each call to BuildKDTree is finding the median point. If we assume there are sorted lists for each of the axes, then
finding the median would take linear time O(n). With that in mind the recurrence would like:
T(n)={O(1), if n=1
    {O(n) + 2T([n/2]), if n>1

Which equals O(nlogn) building time. In the end we are left with a binary tree with n nodes, each storing a single point from the list therefore
it uses O(n) space.

5.2
We only need to check if the first and last points in the region are within the query region.

```
function FullyContained(v, R)
    v - the root node of the region
    R - a Range
    firstPoint = first element of the array returned by ReportSubtree(v)
    lastPoint = last element of the array returned by ReportSubtree(v)
    for every dimension d
        if firstPoint[d] and lastPoint[d] are within R then
            return True
        else
            return False
        end if
    end for
end function
```

5.3.
Disclaimer: my implementation was never "naive" so I do not have data reflecting that.
Disclaimer 2: Runtimes here are so incredibly small(printing the output to the screen requires more time than the actual query) with miniscule variance that I decided the data does not require a graph to be presented.
2 dimensions, 3 points avg time: about 2.8e-05 seconds.
2 dimensions, 8 points avg time: about 5e-05 seconds.

5 dimensions, 3 points avg time: about 3.4e-05 seconds
5 dimensions, 9 points avg time: about 6.7e-05 seconds

10 dimensions, 3 points avg time: about 4.2e-05 seconds
10 dimensions, 9 points avg time: about 0.00045 seconds roughly equal to 11e-05 seconds.

The logarithmic ratio at which the times increase is very clear here. Squaring the number of input points roughly doubles the runtime, while
doubling the dimensions gives an increase of about only 25% in runtime.

Now with FullyContained enabled:
2 dimensions, 3 points avg time: about 2.6e-05 seconds.
2 dimensions, 8 points avg time: about 4.1e-05 seconds.

5 dimensions, 3 points avg time: about 3.1e-05 seconds
5 dimensions, 9 points avg time: about 7e-05 seconds

10 dimensions, 3 points avg time: about 4.2e-05 seconds
10 dimensions, 9 points avg time: about 0.00045 seconds roughly equal to 15e-05 seconds.

Enabling the fullyContained portion of the program has very slight performance increase in the average case. However,
if a large portion of the tree satisfies the condition of fullyContained, performance can be greatly increased.

A best case(all points within the range) test on 10 dimensions and 9 points had an average runtime of about 3.2e-05 which is nearly 200% times faster
that without fullyContained.

5.4.

When I got to this task I realised that my implementation of the SearchKDTree algorithm handles points which have the same x r y coordinate just fine, without my
conscious intent behind this. Upon analysis I believe this is due to the fact that I use FullyContained(v, R) throughout the program to check if sole points are within R.
Using my implementation of FullyContained(v, R) this way, checks each coordinate on a point twice, once against the minimum and once against the maximum. Which I believe is
the proposed way of overcoming the limitation in section 5.5 of "Computational Geometry: Algorithms and Applications".

5.5. Adapting the algorithm to accommodate circles required only changing the FullyContained(points, R) function and the minMax(points) function.
I have overwritten FullyContained within task55.rb but the minMax changes are within the task51.rb file and are marked with a comment.
The change in FullyContained is only to the conditional:

```
if firstPoint[i] >= r[0][i] && lastPoint[i] <= r[1][i]
    return true
```

```
if coordinate on dimension i from the first point is greater or equal to the
smallest value on the same coordinate of i and coordinate on i from the last
point is smaller
    equal to the biggest point of the same coordinate on i then
    the input(point/s) is within the range
```

I changed it to

```
if firstPoint[i] >= (r[0][i] - r[1][0]) && lastPoint[i] <= (r[0][i] + r[1][0])
    return true
```

Which in pseudocode looks something like:

if coordinate from the first point on dimension i is bigger or equal to the point within the circle with smallest value of the same coordinate on i and
coordinate from last point on dimension i is smaller or equal to the point within the circle with largest value of the same coordinate on i then
    the input(point/s) is within the circle

The change in minMax is to accommodate for the different query format.

```
if points[1] != nil && points[1].size == 1  #adaption for circles/spheres/etc.
        radius = points[1][0]
        for i in 0..d-1
            minMaxArray.push([points[0][i]-radius, points[0][i]+radius])
        end
        return minMaxArray
end
```

```
if there exists a second element of points and its size is 1 then
    radius is equal to that element's value
    for each dimension i
```

```
        add a list  [the value of the center point's coordinate on i minus the
radius, the same coordinate but + radius] to minMaxArray
    end for
    return minMaxArray
end if
```

Which basically makes the 2 points on each axis that are furthest away from the centre the min and max.

Time measurements:
The lists used are the same, the centre of the circle is the first part of the query from the rectangular implementation.
2 dimensions, 3 points avg time: about 2e-05 seconds.
2 dimensions, 8 points avg time: about 3.4e-05 seconds.

5 dimensions, 3 points avg time: about 1.85e-05 seconds
5 dimensions, 9 points avg time: about 5.9e-05 seconds

10 dimensions, 3 points avg time: about 4.8e-05 seconds
10 dimensions, 9 points avg time: output here since to be very sensitive to the radius of the circle:
for very small values(1-10) of the radius average runtime is about 4e-05 seconds,
for a radius of 150, which is exactly the median of possible coordinate values(values range from 0-300) average runtime is about 6.4e-05
and for values of radius so large that all points would be reported runtime is about 2.5e-05 seconds.

Conclusion: generally the algorithm seems to run just a bit faster than the rectangular version, however both versions' Runtimes
are very much dependant on how big a part of the tree will satisfy FullyContained - the bigger it is, the less runtime.

5.6.
Here only changes to the FullyContained(points, R) method were necessary and it is overwritten within task56.rb
I made use of the following expression for checking whether a point(xp, yp) lies to the left of a given edge (x1,y1) - (x2,y2)
D = (x2 - x1) * (yp - y1) - (y2 - y1)*(xp - x1)
For D>0 the point lies to the left and for D=0 the point is on the edge. My implementation of this is done in... reverse, for lack of a better word, and instead points lie on the left when the result is D<0.

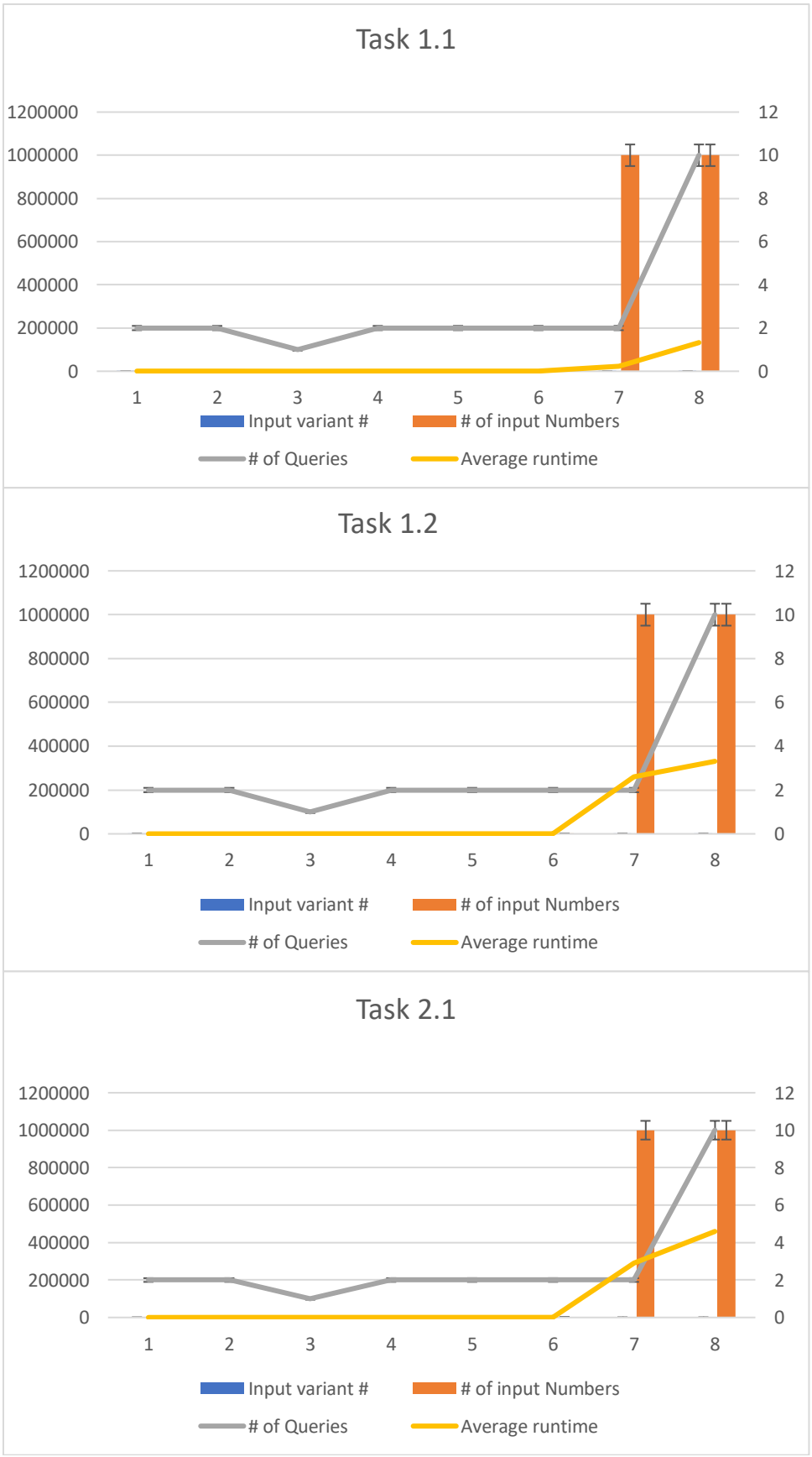
```
def fullyContained(points, r)
    points - list of points to be checked
    r - the polygon's corners
    Let edges be equal to number of corners
    count = 0
    for each point in points do
        for each edge do
            check if current point lies to the left of current edge if it does
                increment count by 1
        end for
    end for
    if count is equal to the number of edges * number of points then
        return true
    else
```

```
        return false
    end if
end function
```

Time evaluations:
Input is the same as the two previous evaluations.
*I believe convex polygons only exist in two dimensional planes, so I will evaluate the algorithm only over two dimensions.

3 points avg time: about 5e-05 seconds.
8 points avg time: about 5e-05 seconds.
19 points avg time: about 8.5e-05 seconds
29 points avg time: about 0.00015 seconds or roughly 15e-05

The running time for the base case is substantially higher than the previous two implementations, which I suspect is because each point is checked 4 times, against the usual 2 times for the previous ones.
However the growth in time in relation to input size still seems to be logarithmic.


Task 1.1
1200000
1000000
800000
600000
400000
200000
0
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
Input variant #
# of input Numbers
# of Queries
Average runtime
Task 1.2
1200000
1000000
800000
600000
400000
200000
0
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
Input variant #
# of input Numbers
# of Queries
Average runtime
Task 2.1
1200000
1000000
800000
600000
400000
200000
0
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
Input variant #
# of input Numbers
# of Queries
Average runtime