# ITEC874/COMP733 Report for Assignment 2

*45539669_Francesco Palermo*

## Program execution requirements

The submitted code (*source_code.py*) has been written in Python 3.7.3.
In order to implement an R-tree in Python I downloaded and installed version 1.7.0 of the libspatialindex library from Github (link available at the reference section). Next, the following lines were used in a terminal to install the library.

*$./configure; make; make install*
*$ easy_install Rtree*

## Program documentation

In this section, I am going to describe the main features of my program. First of all, two libraries have been imported:

*import time*
*from rtree import index*

The module "*time*" provides various time-related functions in Python. For the purpose of my project, it has been used to record and display the total running time of answering the 100 queries as well as the average time of each query for both sequential and R-tree methods.

The second module "*index*" has been used to build the R-tree, by inserting records into the index (through the method *insert*), and query the index through the method *intersection.*

The program can be divided into two main parts. At the top, there are four functions, while at the bottom there is the main structure that is responsible to "call" the functions as well as displaying the final results. Each function and its functionality will be briefly described below, however comments inside the function body will help the reader with the general interpretability of the code. For example:

*help(read_file)*

```
Help on function read_file in module __main__:

read_file(filename, val=0)
    This function reads the 2D dataset from a file and return a more manageable
list
```

The four function definitions are:

*def read_file (filename, val=0)*

*def sequential (points, queries, filename):*

*def insert_record (index, points):*

*def number_points (queries, filename):*

***def read_file (filename, val=0)***

Input:

*filename* indicates the name of the file that we want to read.

*val=0* is a keyword parameter, which is a special kind of parameter to which the final user may choose to assign an argument during a function call, or may ignore.
The reason behind this choice is that I wanted to separate the case where we read the dataset points from the case where we want to read the range queries.

The default value 0 refers to reading the two-dimensional data points and therefore eliminate the first row (in fact, the first row only shows the total number of data points). When the user decides to override this default value with an integer different from 0 the goal is reading the range queries.

Description:

This function aims to read data from a file and return a more manageable list to the main program.

Output:

The function returns a list of strings.

***def sequential (points, queries, filename):***

Input:

*points* is a list of strings that denotes the 2d data points. An example of the first value is shown below: ("1","8224", "78217"). The value in position 0 shows the unique identifier, while the other two values show respectively the x- and y-coordinates.

queries is a list of strings that denotes the range query whose rectangle is [x,x'] x [y,y']. The first value is shown below: ("17840", "18840", "13971", "14971").

*filename* indicates the name of the file where the output will be write.

Description:

The function aims to answer the queries by reading the entire dataset sequentially. Two nested loops have been developed; in particular, the first one loops over the 100 queries, while the second loops over the 100000 data points. It is important to note that both lists were converted to integer type in order to perform comparison between values.

Output:

The function does not return anything back to the main program. However, the function generates the output file ("Output_file_Sequential.txt") where it is possible to check the number of points returned by each query.

***def insert_record (index, points):***

Input:

*index* is an instance of the index class that has been created with the default constructor in the main() as follow:

*idx = index.Index()*

*points* is a list of strings that denotes the 2d data points.

Description:

This function inserts data points into the index and it actually builds the R-tree data structure.
The line of code capable of inserting records into the index is shown below:

*idx.insert(0, (left, bottom, right, top))*

In order to insert a single point entry into the index, we have to make sure that left == right && top == bottom. Unfortunately, there is no shortcut to explicitly insert a single point.

Output:

The function does not return anything back to the main program

***def number_points (queries, filename):***

Input:

*queries* is a list of strings that denotes the range queries.

*filename* indicates the name of the file where the output will be write.

Description:

The function aims to answer the queries and to return the number of points retrieved by each range query through an R-tree structure. The method *intersection* has been used to achieve the function's objective. A brief explanation of this method has given below:

Given a range query, the method returns ids that are contained within the range:

*list (idx.intersection((xmin, ymin, xmax, ymax)))*

In order to return the number of points retrieved instead of the details of this points, the method *len* of the class list has been used. In fact, *len* returns the total number of items in a list.

Output:

The function does not return anything back to the main program. However, the function generates the output file ("Output_file_Rtree.txt") where it is possible to check the number of points returned by each query.

# FINAL OUTPUT AND CONCLUSION

The objective of the assignment was to compare the performance of answering queries by using two methods: R-tree and sequential search.

An R-tree approach enables to access data quickly, without searching every row. The R-tree access method allows you to index multidimensional objects. Queries that use an index perform quickly and provide a substantial performance improvement. The R-tree access method speeds access to multidimensional data. It classifies data in a tree-shaped structure, with bounding boxes at the nodes.

An R-tree index usually improves performance because it removes the need to examine objects outside the area of interest, something that sequential search actually does. In fact, without an R-tree index, a query has to assess every object to find those that match the query criteria.

Next, the screenshots of the running results are provided.

```
(base) Tests-Air:Assignment2 Francesco$ python source_code.py
The total time for sequential query is  14.16534 seconds
The average time for sequential query is  0.1416534 seconds
The total time for R-tree query is  0.015285999999999689 seconds
The average time for R-tree query is  0.00015285999999999688 seconds
R-tree is 926.687 times faster than sequential query
```

As expected, the R-tree is a much better method for searching multidimensional data. According to the result, R-tree is 926 times faster than sequential method.

The next and last screenshot will prove that both criteria end up with the same number of points returned by each query. The image below shows that the two output files (only few initial queries are shown) are equal and therefore both criteria produce the same result (even if R-tree has much better performances)

Output_file_Sequential.txt

```
15
13
9
8
8
8
12
10
9
11
11
6
6
10
6
8
10
10
11
7
11
9
4
8
11
12
7
13
9
9
```

Output_file_Rtree.txt

```
15
13
9
8
8
8
12
10
9
11
11
6
6
10
6
8
10
10
11
7
11
9
4
8
11
12
7
13
9
9
```

# References


1) The version 1.7.0 of the libspatialindex library can be found from: http://libspatialindex.github.com
2) https://www.ibm.com/support/knowledgecenter/en/SSGU8G_12.1.0/com.ibm.spatial.doc/ids_spat_071.htm