ReadMe | BIS634 HW5
Olina Zhu | qz258
Note that this readme file contains both analysis, code, and output of the assignment. The code for each exercise is attached below each question's analysis.

Q1

Acknowledgements

- Data was obtained from https://statecancerprofiles.cancer.gov/incidencerates/index.php
- Help generously given by amazing TF Wenxin and Professor McDougal

Data Cleaning

- All descriptors for the data are deleted, including everything except: column headings, state names, and data;
- All data columns that are not state names and age-adjusted incidence rates are deleted as they are irrelevant to the assignment;
- All "(#)" removed from state names;
- The age-adjusted incident rates column was renamed 'Age-Adjusted IR' to allow easier manipulation in code; and
- The state names were all changed to lower case such that, by using name.lower() with all received input names from the website, the input would be able to match to the dataframe's "State" column values despite capitalization of input string.

Validity checks of state name: if the state name entered, despite capitalization, did not match any of the state names within the dataframe, then it is considered invalid. For example, typing errors would lead the user to a failure.html page that says: "You have failed to access state information because the name you entered was invalid!"

Extension

I added a drop-down menu on the index page where users can choose to either 1) enter the state name manually or 2) choose from existing choices. Both would lead to the information being sent to /info and then having the state name displayed. However, I was not able to make this work correctly, as double-curly braces did not function in my code.

*** Please note that prior to changing the input name and df['State'] to all lower case through ".lower()", the index.html page correctly returns the failure.html page when the input name is invalid (spelling error, not capitalized first letter), but after making every state name and inputted "name" into lower case, only the failure page is ever returned. Many other unexpected errors occurred after making this change, which made my output work less than before, yet I was not sure how to resolve them.

Q2:

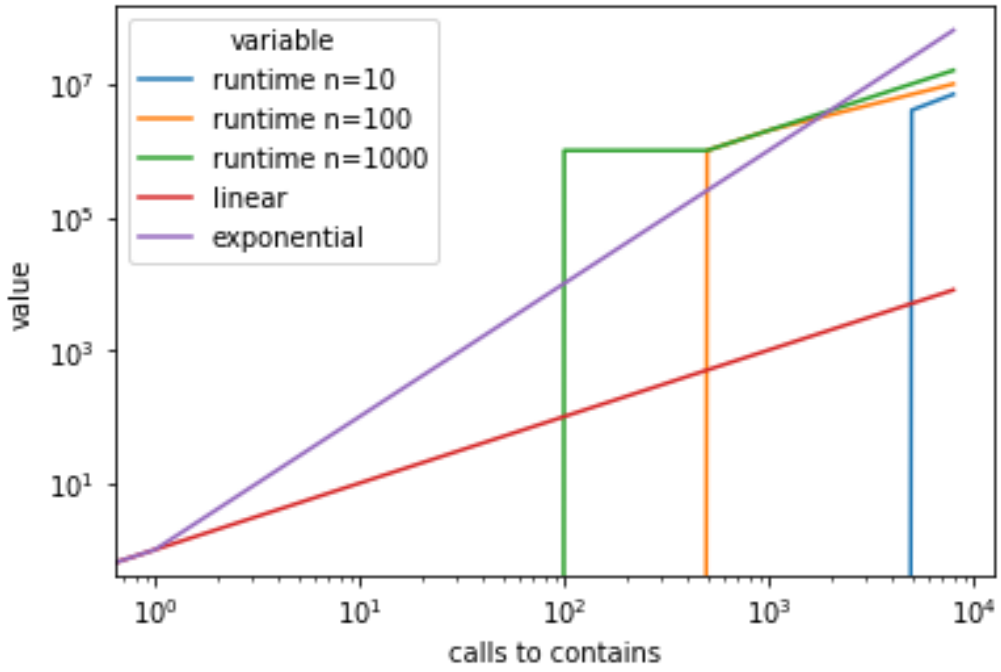
Code, output, and tests are shown in later pages under "**Code and Output on Following Pages**."

Note that t_1, t_2, and t_3 refers to trees with varying sizes of 10, 100, and 1000 respectively. This could be altered as the size of trees with random values generated could be decided by inputting the value in the generate_trees function parameters. This also applies to how many times the range of random numbers to be generated and added to the trees in calc_runtime function.

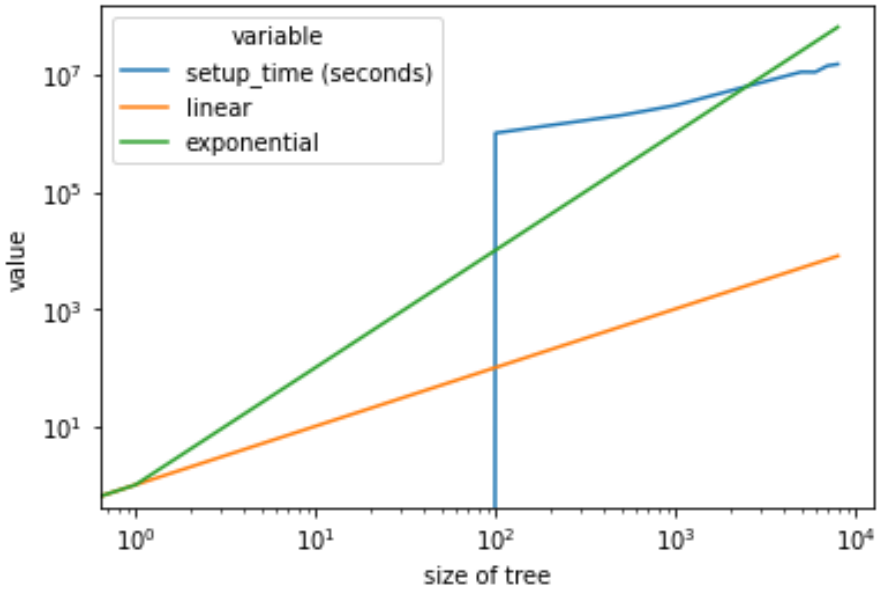
The table and graphs of runtimes are shown below.

	calls to contains	runtime n=10	runtime n=100	runtime n=1000	linear	exponential
0	0	0	0	0	0	0
1	1	0	0	0	1	1
2	10	0	0	0	10	100
3	50	0	0	0	50	2500
4	100	0	0	997900	100	10000
5	500	0	996700	997500	500	250000
6	1000	0	1994600	1994600	1000	1000000
7	5000	4020200	6951100	9975800	5000	25000000
8	6000	4986500	7979000	11968100	6000	36000000
9	7000	5983800	8976800	13958900	7000	49000000
10	8000	6984200	9974100	15957500	8000	64000000

Initially, only the values 0, 1, 10, 100, and 1000 were tested (number of calls to contains (in)), but it seems that most initial runtime values are 0 hence the graph is not very informative, so 0, 1, 10, 50, 100, 500, 1000, 5000, 6000, 7000, and 8000 are the final tested values. Note that with each run of populating trees of different sizes, the runtime is different, shown below is a result specifically picked such that the runtimes become non-zero at different number of calls to contains, such that the graphs do not have two lines overlapping each other and then branching out separately, which could be confusing. Linear and exponential columns were added to the and graphs for comparison with runtime results.



Note that for the log-log graph above , all runtime values for checking “in” (except linear and exponential) eventually fall within the range of linear and exponential, hence runtime is O(logn).



Note that for the log-log graph above, the runtime for setting up trees with 0, 1, 10, 50, 100, 500, 1000, 5000, 6000, 7000, and 8000 randomly generated nodes are shown in blue, compared to linear and exponential lines. The blue line eventually falls within the range of linear and exponential lines.

Q3:

I was not able to produce any deliverable for Q3. The concept of a quad tree is too unfamiliar to me despite discussing in lecture and in the video provided. I could not understand how one should initialize cluster centers and determine which box belongs to which center point. A tree is hierarchical, but KNN neighbors (and their centers) are not, so I do not understand how this could be represented in a tree format. Moreover, the recursion required to add non-center nodes to the tree is also confusing to me. I don't understand the calculation required to determine which node belongs to which subtree, because I do not understand how boxes could be represented by these trees. I tried writing some code, but they are not complete and most likely wrong down to the fundamental logic. If this exercise could be discussed extensively in class, that would be greatly appreciated!

References

National Cancer Institute. (2021, December 3). Incidence Rates Table. State Cancer Profiles. chart. Retrieved December 3, 2021, from https://statecancerprofiles.cancer.gov/incidencerates/index.php?stateFIPS=00&areatype=state&cancer=001&race=00&sex=0&age=001&stage=999&year=0&type=incd&sortVariableName=rate&sortOrder=default&output=0#results.

https://stackoverflow.com/questions/55447599/how-to-send-data-in-flask-to-another-page

Code and Output on Following Pages

Q1 code

Please note that IntelliJ stopped working on the device on which I exported code for this problem, so the pdf was generated from code pasted within word document, hence all coloration and most formatting built-into IntelliJ is gone.

```
# import libraries
import jinja2
import pandas as pd
from flask import Flask, render_template, redirect, request, url_for,
jsonify

# import data
# keep only state and CI columns
df = pd.read_csv("incd.csv")
df.drop(df.columns.difference(['State', 'Age-Adjusted IR']), 1,
inplace=True)

# set all state names to lower case
df['State'] = df['State'].str.lower()

# implement a server that provides three routes using flask
app = Flask(__name__)

# the index/homepage written in HTML which prompts the user to enter a
state
# and provides a button, which passes this information to /info
@app.route("/")
def index():
    name = request.form.get("name")
    return render_template("index.html")

# a web page that takes the name of the state as a GET argument and
# (1) if the state name is valid, displays the same information as the
API in an HTML page
# or (2) displays an error if the state name is invalid
# includes a link back to the homepage
@app.route("/info", methods=["GET"])
def info():
    name = request.args.get("name", None)
    name = name.lower()
    if name in df['State']:
        state(name)
        return render_template("info.html", states=list(df['States']))
    else:
        return render_template("failure.html")

# an API that returns JSON-encoded data containing the name of the state
# and the age-adjusted incidence rate (cases per 100k)
# NOTE: in this case, the name of the state is part of the URL itself;
# it is not passed in as a GET or POST argument

@app.route("/state/<string:name>")
def state(name):
    return jsonify(
        state=name,
        age_adjusted_incidence_rate=float(df[df["State"] == name, 'Age-
Adjusted IR'])
    )
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

Q2 Code, Output, and Tests

```
# import libraries
import random
import time as time
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
```

```
# tree class
class Tree:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    # contains method as given in assignment manual
    def __contains__(self, item):
        if self.value == item:
            return True
        elif self.left and item < self.value:
            return item in self.left
        elif self.right and item > self.value:
            return item in self.right
        else:
            return False

    # add method
    # add node into tree in correct space (in-order) if unique
    # if number seen then do nothing
    def add(self, num):
        # if tree is empty
        if not self.value: self.value = num

        # if at leaf node:
        elif self.left == None and self.right == None:
            if self.value > num: self.left = Tree(num)
            elif self.value < num: self.right = Tree(num)


        # not at leaf node but could add
        elif self.value > num and self.left == None: self.left = Tree(num)
        elif self.value < num and self.right == None: self.right = Tree(num)

        # keep recursing until available space
        elif self.value > num: self.left.add(num)
        elif self.value < num: self.right.add(num)

        return self
```

```
# print tree as list in-order
def printTree(tree, list=[]):
    if not tree: return []
    list.append(tree.value)
    if tree.left: printTree(tree.left, list)
    if tree.right: printTree(tree.right, list)
    return list
```

```
print("testing add function...")
print("initializing empty tree...")
test_tree = Tree(None)
```

```
print("Root node value is", test_tree.value)

print("adding 5 to empty tree...")
test_tree.add(5)
print("Output tree is", printTree(test_tree, list=[]))

print("adding 4 to test tree...")
test_tree.add(4)
print("Output tree is", printTree(test_tree, list=[]))

print("adding 10 to test tree...")
test_tree.add(10)
print("Output tree is", printTree(test_tree, list=[]))

print("adding 10 repeatedly to test tree, should not change tree...")
test_tree.add(10)
print("Output tree is", printTree(test_tree, list=[]))

print("Testing contain with existing tree node (5)...")
print(5 in test_tree)

print("Testing contain with non-existent tree node (3)...")
print(3 in test_tree)

print("testing using given code in assignment manual... (Note Tree() is given a parameter of value None b
my_tree = Tree(None)
for item in [55, 62, 37, 49, 71, 14, 17]:
    my_tree.add(item)
print(printTree(my_tree, list=[]))
```

```
testing add function...
initializing empty tree...
Root node value is None
adding 5 to empty tree...
Output tree is [5]
adding 4 to test tree...
Output tree is [5, 4]
adding 10 to test tree...
Output tree is [5, 4, 10]
adding 10 repeatedly to test tree, should not change tree...
Output tree is [5, 4, 10]
Testing contain with existing tree node (5)...
True
Testing contain with non-existent tree node (3)...
False
testing using given code in assignment manual... (Note Tree() is given a parameter of value None because
[55, 37, 14, 17, 49, 62, 71]
```

6
```
# runtime analysis
n_1 = 10
n_2 = 100
n_3 = 1000
t_1 = Tree(None)
t_2 = Tree(None)
t_3 = Tree(None)

# generate random numbers to populate each tree:
def generate_trees(tree, lim):
    for i in range(lim):
        num = random.randint(1, 101)
        tree.add(num)

generate_trees(t_1, n_1)
```

```
generate_trees(t_2, n_2)
generate_trees(t_3, n_3)

# storing runtime of random calls of contains to above trees
def calc_runtime(tree, iter, lim):
    start = time.time_ns()
    for i in range(iter):
        num = random.randint(1,lim)
        ret = num in tree
    now = time.time_ns()
    return  now-start

d_1 = {}
d_2 = {}
d_3 = {}
for i in [0, 1, 10, 50, 100, 500, 1000, 5000, 6000, 7000, 8000]:
    d_1[i] = calc_runtime(t_1, i, n_1*2)
    d_2[i] = calc_runtime(t_2, i, n_2*2)
    d_3[i] = calc_runtime(t_3, i, n_3*2)
print(d_1)
print(d_2)
print(d_3)
```

```
{0: 0, 1: 997500, 10: 0, 50: 0, 100: 0, 500: 0, 1000: 1002300, 5000: 3986200, 6000: 5984600, 7000: 598290
{0: 0, 1: 0, 10: 0, 50: 0, 100: 0, 500: 999200, 1000: 997600, 5000: 7948300, 6000: 8977100, 7000: 1096760
{0: 0, 1: 0, 10: 0, 50: 0, 100: 0, 500: 1018400, 1000: 998100, 5000: 4986600, 6000: 5987900, 7000: 700940
```

7

```
# turning time tables into dataframes for later plotting
df_1 = pd.DataFrame.from_dict(d_1, orient="index").reset_index()
df_1.columns=['calls to contains', 'runtime (seconds)']

df_2 = pd.DataFrame.from_dict(d_2, orient="index").reset_index()
df_2.columns=['calls to contains', 'runtime (seconds)']

df_3 = pd.DataFrame.from_dict(d_3, orient="index").reset_index()
df_3.columns=['calls to contains', 'runtime (seconds)']

# merge data frames
run_df = df_1.merge(df_2, how='outer', on='calls to contains')
run_df.columns=['calls to contains', 'runtime n=10', 'runtime n=100']
run_df = run_df.merge(df_3, how='outer', on='calls to contains')
run_df.columns=['calls to contains', 'runtime n=10', 'runtime n=100', 'runtime n=1000']

# add linear and exponential columns
linear = [0, 1, 10, 50, 100, 500, 1000, 5000,6000, 7000, 8000]
exponential = [0, 1, 100, 2500, 10000, 250000, 1000000, 25000000, 6000**2, 7000**2, 8000**2]
run_df['linear'] = linear
run_df['exponential'] = exponential
run_df.head(15)
```

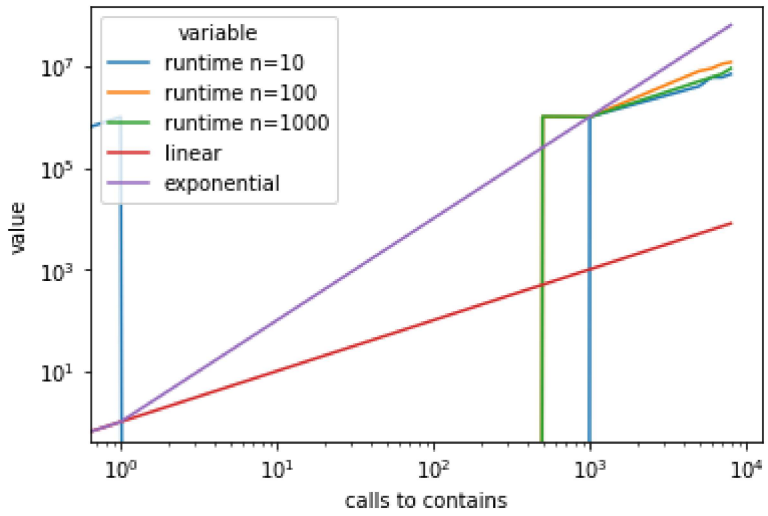
7

	calls to contains	runtime n=10	runtime n=100	runtime n=1000	linear	exponential
0	0	0	0	0	0	0
1	1	997500	0	0	1	1
2	10	0	0	0	10	100
3	50	0	0	0	50	2500
4	100	0	0	0	100	10000
5	500	0	999200	1018400	500	250000

	calls to contains	runtime n=10	runtime n=100	runtime n=1000	linear	exponential
6	1000	1002300	997600	998100	1000	1000000
7	5000	3986200	7948300	4986600	5000	25000000
8	6000	5984600	8977100	5987900	6000	36000000
9	7000	5982900	10967600	7009400	7000	49000000
10	8000	6953100	11967300	9006500	8000	64000000

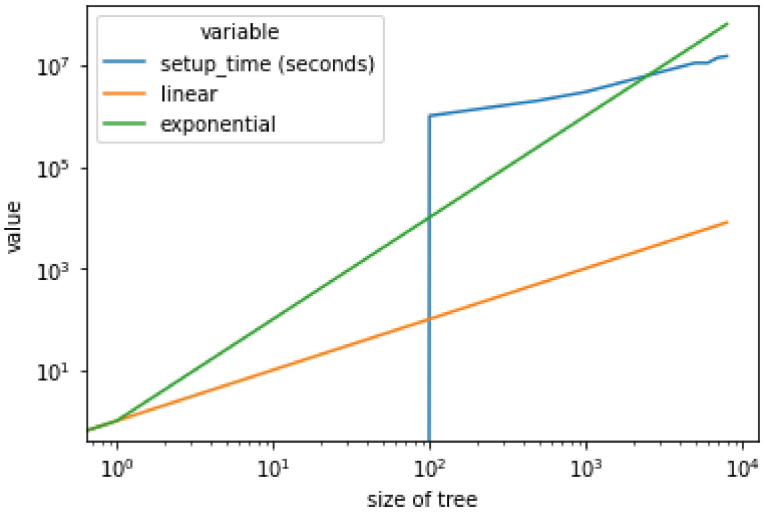
8
```
# plot runtime graph for t_1 with varying number of calls from 1-1000
# plt = sns.lineplot(data = pd.melt(run_df, ['calls to contains']), x = 'calls to contains', y = 'value',
plt = sns.lineplot(x = 'calls to contains', y = 'value', hue='variable', data = pd.melt(run_df, ['calls t
plt.set(xscale="log", yscale="log")
```

8
```
[None, None]
```



15
```
# analyze setup-time of different-sized trees

def calc_setup_runtime(size):
    start = time.time_ns()
    tree = Tree(0)
    generate_trees(tree, size)
    now = time.time_ns()
    return  now-start

d_setup = {}
for i in [0, 1, 10, 50, 100, 500, 1000, 5000,6000, 7000, 8000]:
    d_setup[i] = calc_setup_runtime(i)

df_setup = pd.DataFrame.from_dict(d_setup, orient='index').reset_index()
df_setup['linear'] = linear
df_setup['exponential'] = exponential
df_setup.columns = ['size of tree', 'setup_time (seconds)', 'linear', 'exponential']
df_setup.head(15)

# plot
plt = sns.lineplot(x = 'size of tree', y = 'value', hue='variable', data = pd.melt(df_setup, ['size of tr
plt.set(xscale="log", yscale="log")
```

```
15 [None, None]
```


variable
setup_time (seconds)
linear
exponential
10^7
10^5
10^3
10^1
value
10^0
10^1
10^2
10^3
10^4
size of tree

Q3 Code, no output were generated because code is incomplete

```
1  # import libraries
   import pandas as pd
   import seaborn as sns
   from matplotlib import pyplot as plt
   import random
```

```
6  # modifying code from Q2 to build quad-tree class
   class QTree:
       def __init__(self, value):
           self.value = value
           self.children = []
           self.xlo = None
           self.xhi = None
           self.ylo = None
           self.yhi = None
           self.type = None
```

```
7  # implementing 2-dimensional k-nearest neighbors using quad-tree
   def quad_tree(tree, data, val, k):
       # initialize n clsuter centers
       n = random.randint(0,max(2,len(data)//10))
       centers = random.choices(data, n)
       for i in range(len(centers)):
           # attaching new node to existing non-empty tree via comparison of coordinates
           if tree:
               tree.children.append(centers[i])
               if centers[i][0] <= tree.value[0]:
                   if centers[i][1] <= tree.value[1]: tree = tree.xlo
                   else: tree = tree.xhi
               else:
                   if centers[i][1] <= tree.value[1]: tree = tree.ylo
                   else: tree = tree.yhi
           # filling in information in new node
           # I got completely lost by this part
           tree.xlo = QTree(0)
           tree.xhi = QTree(0)
           tree.ylo = QTree(0)
           tree.yhi = QTree(0)
           tree.value = centers[i]
           tree.type = centers[i][2]
       # go into recursive helper function for adding nodes
       output_tree = recurse(tree, data[:n], k)
       return output_tree

   # recursive helper function
   def recurse(tree, data, k):
       if not tree: return tree
       # recursively add children nodes here but I do not know how to do it
       return tree
```