Exercise 1: Binary Search

Imports

```
import math
import sys
```

Custom Functions

```
def memory_stats(objects):
    """returns sizes of a list of objects thats passed in"""
    return [(sys.getsizeof(o)) for o in objects]
```

List Creation & Declarations

```
# ORDERED List of of values
vals = [i for i in range(1,1001)]

#initial setup
low = 0
mid = 0
hi = len(vals) - 1
```

- binary search uses an ordered list, we will use this to represent the search space for our game

Binary Search Implementation 1

```
def recursive_binary_search(vals, l, h):

    #mid point is recalculated based on reduced bounderies for our list
mid = math.floor((l+h)/2)

    #printing our memory stats
response = input(f"(l={vals[l]}, m={vals[mid]}, h={vals[h]}) | is {vals[mid]} your number
(l=lower, h=higher, y=yes): ").lower()
    print(memory_stats([vals, low, hi]))

    # base case
    if response == "y":
        return "yay!"

    # if users guess is lower, consider everything from low to current guess - 1
    elif response == "l":
        return recursive_binary_search(vals, l, mid-1)
```

```
    # if users guess is higher, consider everything from 1 more than current guess to the high
    elif response == "h":
        return recursive_binary_search(vals, mid+1, h)

    # otherwise input invalid and let user repeat the same entry instruction
else:
        print("INVALID INPUT\n")
        return recursive_binary_search(vals, l, h)
```

Output: Guessing the number 5

```
(l=1, m=500, h=1000) | is 500 your number (l=lower, h=higher, y=yes): l
Memory: vals = 8856 | low = 24 | hi = 28

(l=1, m=250, h=499) | is 250 your number (l=lower, h=higher, y=yes): l
Memory: vals = 8856 | low = 24 | hi = 28

(l=1, m=125, h=249) | is 125 your number (l=lower, h=higher, y=yes): l
Memory: vals = 8856 | low = 24 | hi = 28

(l=1, m=62, h=124) | is 62 your number (l=lower, h=higher, y=yes): l
Memory: vals = 8856 | low = 24 | hi = 28

(l=1, m=31, h=61) | is 31 your number (l=lower, h=higher, y=yes): l
Memory: vals = 8856 | low = 24 | hi = 28

(l=1, m=15, h=30) | is 15 your number (l=lower, h=higher, y=yes): l
Memory: vals = 8856 | low = 24 | hi = 28

(l=1, m=7, h=14) | is 7 your number (l=lower, h=higher, y=yes): l
Memory: vals = 8856 | low = 24 | hi = 28

(l=1, m=3, h=6) | is 3 your number (l=lower, h=higher, y=yes): h
Memory: vals = 8856 | low = 24 | hi = 28

(l=4, m=5, h=6) | is 5 your number (l=lower, h=higher, y=yes): y
Memory: vals = 8856 | low = 24 | hi = 28


Process finished with exit code 0
```

Binary Search Implementation 2

- this binary search reduces the size of the list after each prompt to the user
- memory size of list reduces

```python
def iterative_binary_seach_reducing_list(vals, low, hi):
    while True:
        # starting guess for our range of numbers
        mid = math.floor(len(vals)/2)

        # print memory stats
        stats=memory_stats([vals, low, hi])
        print(f"Memory: vals = {stats[0]} | low = {stats[1]} | hi = {stats[2]}\n")


        response = input(f"is {vals[mid]} your number (l=lower, h=higher, y=yes): ").lower()

        # we guessed it
        if response == "y":
            print("yay!")
            break

        # value is lower (set highest possible guess to values after the current mid point)
        elif response == "l":
            hi = mid - 1
            vals = vals[:mid]

        # value is higher (set lowest possible guess to values before the current midpoint)
        elif response == "h":
            low = mid + 1
            vals = vals[mid + 1:]

        else:  # invalid input...
            print("INVALID INPUT\n")
```

Output: Guess the number 5

```
Memory: vals = 8856 | low = 24 | hi = 28

is 501 your number (l=lower, h=higher, y=yes): l
Memory: vals = 4056 | low = 24 | hi = 28

is 251 your number (l=lower, h=higher, y=yes): l
Memory: vals = 2056 | low = 24 | hi = 28

is 126 your number (l=lower, h=higher, y=yes): l
Memory: vals = 1056 | low = 24 | hi = 28

is 63 your number (l=lower, h=higher, y=yes): l
Memory: vals = 552 | low = 24 | hi = 28

is 32 your number (l=lower, h=higher, y=yes): l
Memory: vals = 304 | low = 24 | hi = 28
```

```
is 16 your number (l=lower, h=higher, y=yes): l
Memory: vals = 176 | low = 24 | hi = 28

is 8 your number (l=lower, h=higher, y=yes): l
Memory: vals = 112 | low = 24 | hi = 28

is 4 your number (l=lower, h=higher, y=yes): h
Memory: vals = 80 | low = 28 | hi = 28

is 6 your number (l=lower, h=higher, y=yes): l
Memory: vals = 64 | low = 28 | hi = 24

is 5 your number (l=lower, h=higher, y=yes): y
yay!

Process finished with exit code 0
```

Conclusion

the binary sort algorithm allows a quick method for quickly finding the element provided that we have a sorted data structure. We can easily reduce the size of the structure by half each iteration to quickly thus reducing the search space.

Exercise 2: Sorting

Imports

```
import json
import math
import time
import random
import timeit
```

Custom Functions

```
def make_stats(function_to_call, num=1):
    """takes in a function and can run it num-times"""
    time = timeit.timeit(function_to_call, number=num)
    return time/num
```

Student Object

```
class Student:
    def __init__(self, student_ID=None, first_name=None, last_name=None, email=None,
major=None):
        self.data = {
            "student_ID": student_ID,
            "first_name": first_name,
            "last_name": last_name,
            "email": email,
            "major": major
        }

        if student_ID is None:
            self.make_random()

    def make_random(self):

        first_names = ["John", "Jane", "Alice", "Bob", "Charlie", "Daisy", "Ella", "Frank",
"Grace", "Henry", "Isabel",
                       "Jack", "Katie", "Leo", "Mia", "Noah", "Olivia", "Paul", "Quinn",
"Riley"]
        last_names = ["Doe", "Smith", "Johnson", "Brown", "White", "Black", "Green", "Gray",
"Adams", "Baker", "Clark",
                      "Davis", "Evans", "Foster", "Garcia", "Harris", "Jackson", "King", "Lee",
"Morris"]
        majors = ["CS", "BIO", "MATH", "PHYS", "HIST", "ENG", "CHEM", "PSYC", "SOC", "PHIL",
"ECO", "MUS",
                         "ANTH", "ME", "EE"]

        #generating the random attributes
        s_id = ''.join(random.choices('0123456789', k=9))
        s_fName = random.choice(first_names)
        s_lName = random.choice(last_names)
        s_major = random.choice(majors)
        s_email = f"{s_fName.lower()[0:2]}{s_lName.lower()[0]}
{''.join(random.choices('0123456789', k=4))}@psu.edu"

        self.__init__(s_id, s_fName, s_lName, s_email, s_major)
    def __repr__(self):
        return self.data
    def __str__(self):
        return str(self.data)
```

Database With Sort Functions

```
class Database:
    def __init__(self, db_name, data=list):
        self.file = self.create_database(db_name, data)
        self.db_name = db_name

        #tracks times of sorting...takes in an entry with key=algorithm name, v=time taken to
sort
```

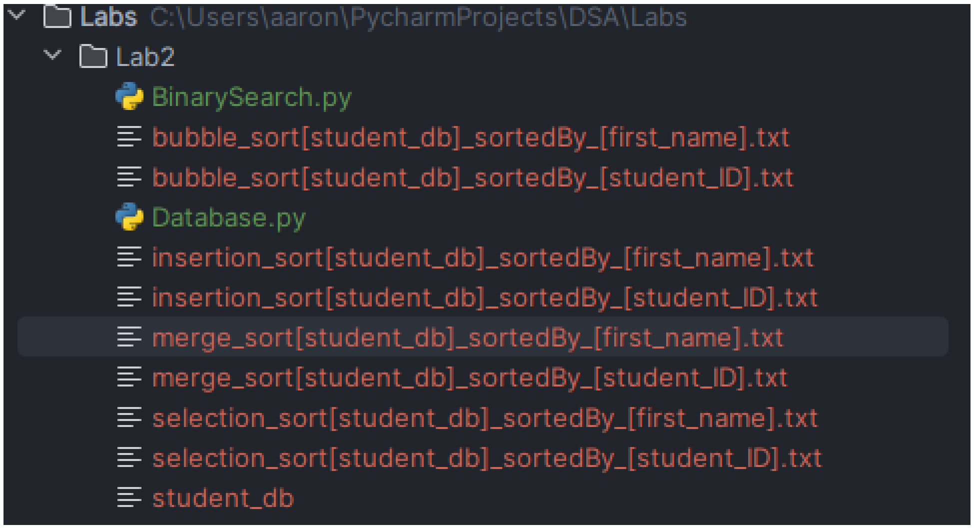
```
        self.sort_times=dict()
    def create_database(self, db_name: str, data):
        with open(db_name, "w") as file:
            for obj in data:
                file.write(json.dumps(obj) + "\n")
        return file

    def write_file(self, name, data):
        """used to write a sorted object to a file"""
        with open(name, "w") as file:
            for obj in data:
                file.write(json.dumps(obj) + "\n")

        return file

    def selection_sort(self, attr):
        """uses the selection sort algorithm to sort the data in the db based on a given
attribute"""
        file_name = f"selection_sort[{self.db_name}]_sortedBy_[{attr}].txt"
        data = self.get_data()


        """Selection sort:
            1. traverse array
            2. find smallest element
            3. swap element with the lowest position
            4. reduce the search size of the array from the left by 1
            5. repeat until search range is 0
        """
        for sorted_line in range((len(data))):
            mindex = sorted_line

            #this traverses the unsorted portion of the list and tries to find the index of the
smallest element
            for i in range(sorted_line, len(data)):
                if data[i][attr] < data[mindex][attr]: #we are comparing the attribute of each
dictionary object
                    mindex = i

            #after the traversal, mindex holds the index of the smallest item
            #so perform a swap
            data[sorted_line], data[mindex] = data[mindex], data[sorted_line]

        self.write_file(file_name, data)
        return data

    def insertion_sort(self, attr):
        """uses the insertion sort algorithm to sort the data in the db based on a given
attribute"""
        file_name = f"insertion_sort[{self.db_name}]_sortedBy_[{attr}].txt"
        data = self.get_data()

        """Insertion sort:
```

```
            1. traverse array
            2. if 2 elements being compared are not in order, swap the smaller element
            with those in the sorted part ot the list until its place is found
            3. expand the sorted line by 1
            4. continue until we have traversed the array
        """

        for sorted_line in range(1, len(data)):
            j = sorted_line
            while j > 0 and data[j - 1][attr] > data[j][attr]:
                data[j - 1], data[j] = data[j], data[j - 1]
                j -= 1

        self.write_file(file_name, data)
        return data

    def bubble_sort(self, attr):
        """uses the bubble sort algorithm to sort the data in the db based on a given
attribute"""
        file_name = f"bubble_sort[{self.db_name}]_sortedBy_[{attr}].txt"
        data = self.get_data()

        """bubble sort:
            1. traverse the list until the len(data) - sorted_line
            2. compare adjacent elements
            3. if n-1 item is greater than n item perform swap
            4. continue until we hit the sorted line
            5. increment sorted line by 1
        """

        #iterate over array...sorted area is formed at the end
        for sorted_line in range(len(data)):
            for i in range(0, len(data)-sorted_line-1):
                if data[i][attr] > data[i+1][attr]:
                    data[i + 1], data[i] = data[i], data[i + 1]

        self.write_file(file_name, data)
        return data

    def merge_sort(self, attr):
        file_name = f"merge_sort[{self.db_name}]_sortedBy_[{attr}].txt"
        data = self.get_data()
        sorted_data = self.merge_sorter(data, attr)

        self.write_file(file_name, sorted_data)
        return sorted_data

    def merge_sorter(self, data, attr):
        """Recursive function to split the lists"""

        """Merge sort:
            1. continously divide the list into 2 equal sublists l,r until each sublist of
length 1
```

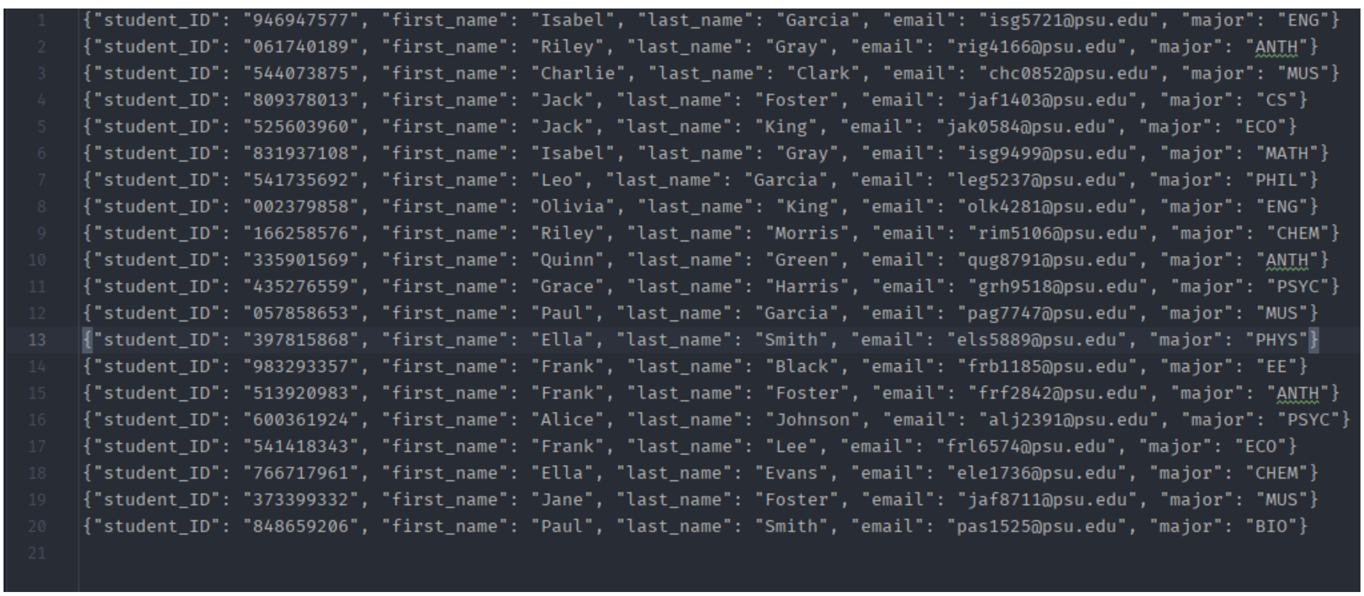
```
            2. sort the sublists and recombine with its partner and call a sort function to
merge those 2 lists back into one
            3. repeat util we have reassembled list back (done automatically through recursion)
        """
        if len(data) <= 1:
            return data

        mid = len(data) // 2
        left = data[:mid]
        right = data[mid:]

        l_sorted = self.merge_sorter(left, attr)
        r_sorted = self.merge_sorter(right, attr)

        return self.make_merge(l_sorted, r_sorted, attr)

    def make_merge(self, left_sublist, right_sublist, attr):
        """helper function...recieves subarrays down from the merging function and sorts
them"""
        merged = []

        #indicie tracking
        l, r = 0, 0

        #first we compare elements from the sublists and select items to be placed into our
fully merged output
        while l < len(left_sublist) and r < len(right_sublist):
            if left_sublist[l][attr] < right_sublist[r][attr]:
                merged.append(left_sublist[l])
                l += 1
            else:
                merged.append(right_sublist[r])
                r += 1

        #if items remain in any of these sublists, add them to the merged list
        while l < len(left_sublist):
            merged.append(left_sublist[l])
            l += 1

        #...emptying right sublist
        while r < len(right_sublist):
            merged.append(right_sublist[r])
            r += 1

        return merged


    def get_data(self):
        if self.file:
            with open(self.file.name, "r") as file:
                return [json.loads(line) for line in file.readlines()]
        else:
            return None
```

```python
    def __str__(self):
        with open(self.db_name) as file:
            students = file.readlines()
            return str(students)
```

Generated Files



Database File



Time Calculation

- I calculated the sorting time before checking the memory consumption of the process
- this allows me to only measure the internals of each sort algorithm without outside variables

Time Calculation Function

```python
def make_stats(function_to_call, num=1):
    """takes in a function and can run it num-times"""
    time = timeit.timeit(function_to_call, number=num)
    return time/num
```

- returns average time for some calls (I use 10 below)

Generating Sorted Files and Getting Times

```python
# generating 20 random student objects (random because params are none)
students = [Student().__repr__() for i in range(20)]


# creating a database and populating
test_db = Database("student_db", students)

sort_attrs = ['student_ID', 'first_name']
results = []

for attr in sort_attrs:
    out = {attr: {
            "insertion": make_stats(str(test_db.insertion_sort(attr)),10),
            "selection": make_stats(str(test_db.selection_sort(attr)), 10),
            "bubble": make_stats(str(test_db.bubble_sort(attr)), 10),
            "merge": make_stats(str(test_db.merge_sort(attr)), 10)
        }}
    results.append(out)

#makes the output more readable by converting results into a string and then concatenating with
\n
print("\n".join(map(str, results)))
```

Output and Table

Raw

```
{'student_ID': {'insertion': 1.609999999999806e-06, 'selection': 1.5899999999999248e-06,
'bubble': 2.319999999999406e-06, 'merge': 1.5799999999996372e-06}}
{'first_name': {'insertion': 1.659999999999856e-06, 'selection': 1.6799999999997372e-06,
'bubble': 1.6800000000004311e-06, 'merge': 1.6299999999996873e-06}}
```

Table (Student_ID Sort Times)

```
{"student_ID": "004565918", "first_name": "Grace", "last_name": "Baker", "email": "grb7855@psu.edu", "major": "ANTH"}
{"student_ID": "044212550", "first_name": "Quinn", "last_name": "White", "email": "quw7267@psu.edu", "major": "ECO"}
{"student_ID": "046727368", "first_name": "Grace", "last_name": "Green", "email": "grg8294@psu.edu", "major": "PHIL"}
{"student_ID": "053300038", "first_name": "Charlie", "last_name": "White", "email": "chw8992@psu.edu", "major": "MUS"}
{"student_ID": "123696002", "first_name": "Grace", "last_name": "Foster", "email": "grf0488@psu.edu", "major": "PHIL"}
{"student_ID": "164761983", "first_name": "Jane", "last_name": "Davis", "email": "jad4834@psu.edu", "major": "PHYS"}
{"student_ID": "189702528", "first_name": "Quinn", "last_name": "King", "email": "quk9978@psu.edu", "major": "HIST"}
{"student_ID": "281169981", "first_name": "Frank", "last_name": "Baker", "email": "frb8008@psu.edu", "major": "PHIL"}
{"student_ID": "295477397", "first_name": "Frank", "last_name": "Doe", "email": "frd8856@psu.edu", "major": "CS"}
{"student_ID": "341113918", "first_name": "Olivia", "last_name": "Clark", "email": "olc3287@psu.edu", "major": "MATH"}
{"student_ID": "417071002", "first_name": "Isabel", "last_name": "Morris", "email": "ism6326@psu.edu", "major": "ECO"}
{"student_ID": "491875173", "first_name": "Olivia", "last_name": "Doe", "email": "old4673@psu.edu", "major": "SOC"}
{"student_ID": "592430771", "first_name": "Jane", "last_name": "Johnson", "email": "jaj9263@psu.edu", "major": "CHEM"}
{"student_ID": "665315253", "first_name": "Jane", "last_name": "Jackson", "email": "jaj7484@psu.edu", "major": "BIO"}
{"student_ID": "776683649", "first_name": "Riley", "last_name": "Davis", "email": "rid7733@psu.edu", "major": "SOC"}
{"student_ID": "861182912", "first_name": "Alice", "last_name": "Morris", "email": "alm5583@psu.edu", "major": "ENG"}
{"student_ID": "901138636", "first_name": "Paul", "last_name": "Evans", "email": "pae5331@psu.edu", "major": "CHEM"}
{"student_ID": "904777066", "first_name": "Ella", "last_name": "Evans", "email": "ele3031@psu.edu", "major": "ANTH"}
{"student_ID": "974671809", "first_name": "Frank", "last_name": "Gray", "email": "frg8215@psu.edu", "major": "ENG"}
{"student_ID": "999687468", "first_name": "Noah", "last_name": "Harris", "email": "noh5453@psu.edu", "major": "ECO"}
```


```
+-----------------------+-----------------------+-----------------------+-----------------------+
|       insertion       |       selection       |         bubble        |         merge         |
+-----------------------+-----------------------+-----------------------+-----------------------+
| 1.5699999999993497e-06 | 1.6700000000008374e-06 | 1.7199999999994997e-06 | 1.6399999999999748e-06 |
+-----------------------+-----------------------+-----------------------+-----------------------+
```

Table (First Name Sort Times)


```
{"student_ID": "861182912", "first_name": "Alice", "last_name": "Morris", "email": "alm5583@psu.edu", "major": "ENG"}
{"student_ID": "053300038", "first_name": "Charlie", "last_name": "White", "email": "chw8992@psu.edu", "major": "MUS"}
{"student_ID": "904777066", "first_name": "Ella", "last_name": "Evans", "email": "ele3031@psu.edu", "major": "ANTH"}
{"student_ID": "281169981", "first_name": "Frank", "last_name": "Baker", "email": "frb8008@psu.edu", "major": "PHIL"}
{"student_ID": "974671809", "first_name": "Frank", "last_name": "Gray", "email": "frg8215@psu.edu", "major": "ENG"}
{"student_ID": "295477397", "first_name": "Frank", "last_name": "Doe", "email": "frd8856@psu.edu", "major": "CS"}
{"student_ID": "046727368", "first_name": "Grace", "last_name": "Green", "email": "grg8294@psu.edu", "major": "PHIL"}
{"student_ID": "123696002", "first_name": "Grace", "last_name": "Foster", "email": "grf0488@psu.edu", "major": "PHIL"}
{"student_ID": "004565918", "first_name": "Grace", "last_name": "Baker", "email": "grb7855@psu.edu", "major": "ANTH"}
{"student_ID": "417071002", "first_name": "Isabel", "last_name": "Morris", "email": "ism6326@psu.edu", "major": "ECO"}
{"student_ID": "665315253", "first_name": "Jane", "last_name": "Jackson", "email": "jaj7484@psu.edu", "major": "BIO"}
{"student_ID": "592430771", "first_name": "Jane", "last_name": "Johnson", "email": "jaj9263@psu.edu", "major": "CHEM"}
{"student_ID": "164761983", "first_name": "Jane", "last_name": "Davis", "email": "jad4834@psu.edu", "major": "PHYS"}
{"student_ID": "999687468", "first_name": "Noah", "last_name": "Harris", "email": "noh5453@psu.edu", "major": "ECO"}
{"student_ID": "341113918", "first_name": "Olivia", "last_name": "Clark", "email": "olc3287@psu.edu", "major": "MATH"}
{"student_ID": "491875173", "first_name": "Olivia", "last_name": "Doe", "email": "old4673@psu.edu", "major": "SOC"}
{"student_ID": "901138636", "first_name": "Paul", "last_name": "Evans", "email": "pae5331@psu.edu", "major": "CHEM"}
{"student_ID": "189702528", "first_name": "Quinn", "last_name": "King", "email": "quk9978@psu.edu", "major": "HIST"}
{"student_ID": "044212550", "first_name": "Quinn", "last_name": "White", "email": "quw7267@psu.edu", "major": "ECO"}
{"student_ID": "776683649", "first_name": "Riley", "last_name": "Davis", "email": "rid7733@psu.edu", "major": "SOC"}
```


```
+-----------------------+-----------------------+-----------------------+-----------------------+
|       insertion       |       selection       |         bubble        |         merge
```

```
|
+-----------------------+-----------------------+-----------------------+------------------------+
| 1.5999999999988247e-06 | 1.5600000000004499e-06 | 1.5600000000004499e-06 | 1.5899999999999248e-06 |
+-----------------------+-----------------------+-----------------------+------------------------+
```

Memory

The Following Algorithms are all of space complexity O(1):

- bubble sort
- insertion sort
- selection sort

That is because we sort in-place and we dont generate any addition lists to consume memory
The only sort here with a higher sort complexity is **Merge Sort**. This sorting algorithm recursively divides the list into n-sub lists.

Memory Usage: Bubble, Insertion, Selection

```
Memory: 248
```

Memory Usage: Merge Sort

- i will list the memory size of each merged sub list here

```
Memory:

Length of   | Size
Sub list    |
------------------------
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
Length = 5  |  104
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
Length = 5  |  104
Length = 10  |  168
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
```

```
Length = 5  |  104
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
Length = 5  |  104
Length = 10  |  168
Length = 20  |  232
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
Length = 5  |  104
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
Length = 5  |  104
Length = 10  |  168
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
Length = 5  |  104
Length = 2  |  72
Length = 2  |  72
Length = 3  |  72
Length = 5  |  104
Length = 10  |  168
Length = 20  |  232
```

Conclusion and Issues

I had learned a lot implementing these algorithms during this lab. I made the unfortunate mistake of overcomplicating the work by creating classes and objects. This was made duly worse by the fact that the time my algorithms took didnt line up with what the time complexity postulates. below i have included an implemenation of the sort algortihms in my database class but instead set to work with integers.

```python
import random
import timeit


def selection_sort(data):
    """uses the selection sort algorithm to sort the data in the db based on a given attribute"""


    """Selection sort:
        1. traverse array        2. find smallest element        3. swap element with the lowest position        4. reduce the search size of the array from the left by 1        5. repeat until search range is 0
```

```
    """    for sorted_line in range((len(data))):
        mindex = sorted_line

        # this traverses the unsorted portion of the list and tries to find the index of the
smallest element
        for i in range(sorted_line, len(data)):
            if data[i] < data[mindex]:  # we are comparing the attribute of each dictionary
object
                mindex = i

        # after the traversal, mindex holds the index of the smallest item
        # so perform a swap        data[sorted_line], data[mindex] = data[mindex],
data[sorted_line]


    return data


def insertion_sort( data):
    """uses the insertion sort algorithm to sort the data in the db based on a given
attribute"""



    """Insertion sort:
        1. traverse array
        2. if 2 elements being compared are not in order, swap the smaller element        with
those in the sorted part ot the list until its place is found        3. expand the sorted line
by 1        4. continue until we have traversed the array    """
    for sorted_line in range(1, len(data)):
        j = sorted_line
        while j > 0 and data[j - 1] > data[j]:
            data[j - 1], data[j] = data[j], data[j - 1]
            j -= 1


    return data

def bubble_sort(data):
    """uses the bubble sort algorithm to sort the data in the db based on a given attribute"""



    """bubble sort:
        1. traverse the list until the len(data) - sorted_line        2. compare adjacent
elements
        3. if n-1 item is greater than n item perform swap        4. continue until we hit the
sorted line        5. increment sorted line by 1    """

    # iterate over array...sorted area is formed at the end
    for sorted_line in range(len(data)):
        for i in range(0, len(data) - sorted_line - 1):
            if data[i] > data[i + 1]:
                data[i + 1], data[i] = data[i], data[i + 1]
```

```
    return data

def merge_sort(data):

    sorted_data = merge_sorter(data)

    return sorted_data


def merge_sorter(data):
    """Recursive function to split the lists"""

    """Merge sort:
        1. continously divide the list into 2 equal sublists l,r until each sublist of length 1
2. sort the sublists and recombine with its partner and call a sort function to merge those 2
lists back into one        3. repeat util we have reassembled list back (done automatically
through recursion)    """    if len(data) <= 1:
        return data

    mid = len(data) // 2
    left = data[:mid]
    right = data[mid:]

    l_sorted = merge_sorter(left)
    r_sorted = merge_sorter(right)

    return make_merge(l_sorted, r_sorted)


def make_merge(left_sublist, right_sublist):
    """helper function...recieves subarrays down from the merging function and sorts them"""
    merged = []

    # indicie tracking
    l, r = 0, 0

    # first we compare elements from the sublists and select items to be placed into our fully
merged output
    while l < len(left_sublist) and r < len(right_sublist):
        if left_sublist[l] < right_sublist[r]:
            merged.append(left_sublist[l])
            l += 1
        else:
            merged.append(right_sublist[r])
            r += 1

    # if items remain in any of these sublists, add them to the merged list
    while l < len(left_sublist):
        merged.append(left_sublist[l])
        l += 1

    # ...emptying right sublist
    while r < len(right_sublist):
```

```
        merged.append(right_sublist[r])
        r += 1

    return merged

def make_stats(function_to_call, *args, num=1):
    """takes in a function and can run it num-times"""
    time = timeit.timeit(lambda: function_to_call(*args), number=num)
    return str(time / num)

fixed = [random.randint(0, 10000) for i in range(2000)]

test_data = fixed.copy()
print("Merge:"+make_stats(merge_sorter, test_data))
test_data = fixed.copy()
print("Buble:"+make_stats(bubble_sort, test_data))
test_data = fixed.copy()
print("Insertion: "+make_stats(insertion_sort, test_data))
test_data = fixed.copy()
print("Selection: "+make_stats(selection_sort, test_data))
```

Output

```
Merge:0.0035518000000000001
Buble:0.16691229999999999
Insertion: 0.13455069999999997
Selection: 0.06331689999999995
```

From this output, its clear that MergeSort outperforms all the other algorithms. This is due to it having to do at most O(nlogn) splitting operations. Selection sort also performed reasonably well. This is likely due to the fact that we didnt have to swap items as much as the other sorting algorithms.