Yihang Fu
yf329
yihang.fu@yale.edu

Assignment 1
BIS 634 Computational Methods for Informatics

2024/09/28

1 Exercise 1

1.1 1a

We use the package *xml.etree.ElementTree* to deal with the XML file. As indicated by the official document, this package regards the XML file as a hierarchical tree. Since we want to extract the basic information of each patient under *Patients* such as age, name, and gender, we can implement the code like the snippet below.

```
xml_file = '/Users/yihang/Desktop/BIS 634/Week2/hw1-patients.xml'
tree = ET.parse(xml_file)
root = tree.getroot()

## For loop to extract each root and subroot
for patient in root.find('patients').findall('patient'):
    name = patient.attrib.get('name', 'Unknown')
    age = patient.attrib.get('age', 'Unknown')
    gender = patient.attrib.get('gender', 'Unknown')

    print(f"Patient Name: {name}")
    print(f"Age: {age}")
    print(f"Gender: {gender}")
    print()
```

Listing 1: read XML file

To plot the age distribution, we create a blank list and append it by searching all subroots. We use the set function in Python to search for duplicated ages.

```
import matplotlib.pyplot as plt
import pandas as pd
# Extract patient ages
ages = [] #for storing all ages
for patient in root.find('patients').findall('patient'):
    age = float(patient.attrib.get('age', '0'))
    ages.append(age)

plt.figure(figsize=(10,6))
plt.hist(ages, bins=90, color='blue', edgecolor='black')
plt.title("Age Distribution of Patients")
plt.xlabel("Age")
plt.ylabel("Frequency")
plt.grid(True)

# Show the histogram
plt.show()

# Convert the list to a set, which removes duplicates
unique_elements = set(ages)

# Check if the length of the set is different from the length of the original list
if len(unique_elements) != len(ages):
    print("There are duplicates.")
else:
    print("The list does not have duplicates.")
```

Listing 2: plot age distribution and check duplicates

The visualization result is shown in 1, and there are no duplicates after examination.

1.2 Extra Credit Question

If this dataset contains duplicates, it may not influence the performance of the binary search and the sorting algorithm. However, it may require us to continue the exploration after we find one target. Specifically, we need to perform additional searches to the left and right of the found patient to collect all the duplicates (1d, 1e, 1f, 1g, and 1h). If the target age is found, move the search to the right to find the first patient with an age greater than the target age. The detailed implementation is presented below 1.6.

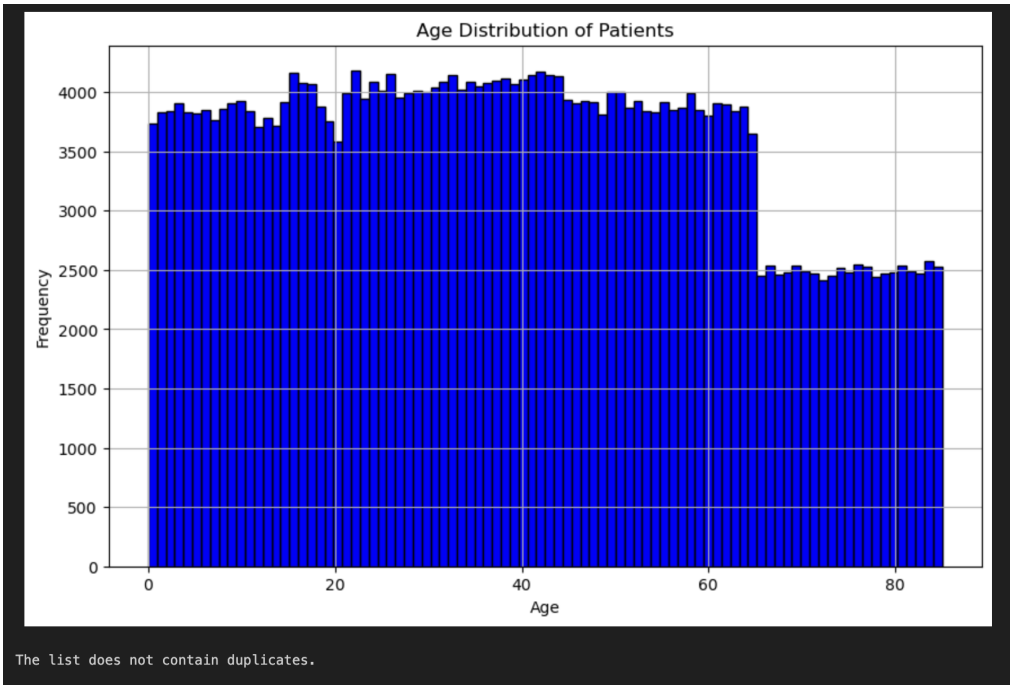


Figure 1: 1a Output

1.3 1b

Using the code snippet below, we can find that there are three keys in *genders* variable which are **Male, Female and Unknown**. If we regard this XML file as a tree structure, the Parents node is called *patientdata*, and its child node is called *patients*. Each child of *patients* is called a *patient* and has several attributes such as age, gender, name, diagnosis, etc. Using the package collections, we can automatically calculate the number of each category and plot the distribution as shown in 2.

```
from collections import Counter

genders = [patient.attrib.get('gender') for patient in root.find('patients').findall('
    patient')]

# Count occurrences
gender_counts = Counter(genders)
bars = plt.bar(gender_counts.keys(), gender_counts.values(), color=['blue', 'green','red'])

# Adding the value labels on top of each bar
#I ask ChatGPT to do this to show value annotation on each bar for better visualization
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, int(yval), va='bottom', ha='center')

plt.title("Gender Distribution")
plt.xlabel("Gender")
plt.ylabel("Count")
plt.show()
```

Listing 3: Gender data distribution visualization

1.4 1c

We again extract useful features for the following questions such as name, age, and gender. Figure 3 shows the result of using the built-in function *sorted* to sort the patients. Here, we find out the oldest patient is called Monica Caponera.

1.5 1d

To reach the O(n) time complexity, we traverse the list of patients once. We keep track of the oldest patient and the second oldest patient as going through the list. Then, we compare each patient's age with the current oldest and second oldest and update accordingly.

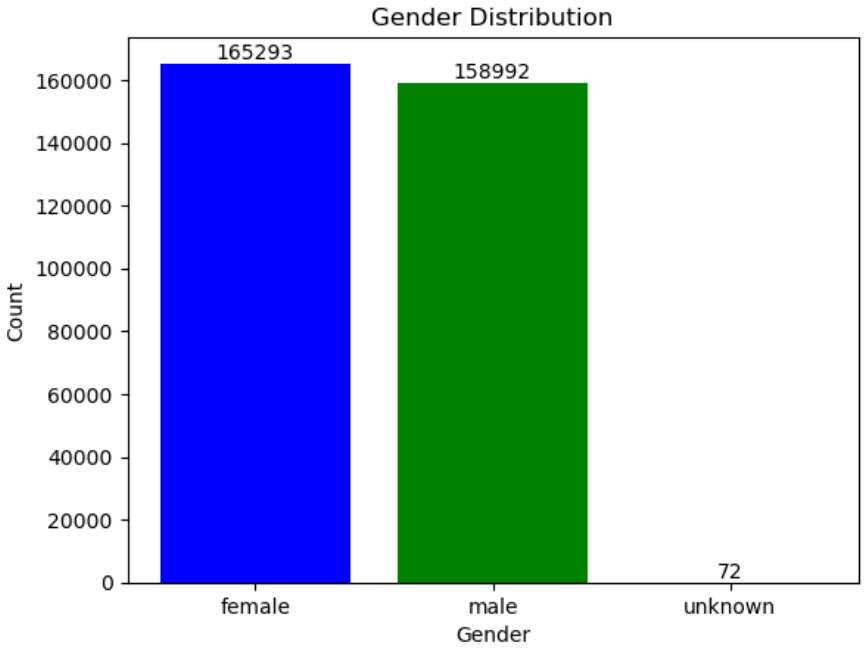


Figure 2: 1b Output

```
# Extract features that used for the following questions
patients = [{'name': patient.attrib['name'], 'age': float(patient.attrib['age']), 'gender': patient.attrib['gender']}
            for patient in root.find('patients').findall('patient')]

# Sort the patients by age
sorted_patients = sorted(patients, key=lambda x: x['age'])

# Identify the oldest patient
oldest_patient = sorted_patients[-1]
print(f"The oldest patient is {oldest_patient['name']}, Age: {oldest_patient['age']}")
```

✓ 0.1s

```
The oldest patient is Monica Caponera, Age: 84.99855742449432
```

Figure 3: 1c Output

The O(n) method is better when only needs to find one target value since it only requires a single pass. However, if multiple values are queried, sorting the whole list will be more efficient.

```
def find_second_oldest(patients):
    max_age = second_max_age = float('-inf')
    oldest_patient = second_oldest_patient = None

    for patient in patients:
        age = patient['age']
        if age > max_age:
            second_max_age = max_age
            second_oldest_patient = oldest_patient
            max_age = age
            oldest_patient = patient
        elif age > second_max_age and age != max_age:
            second_max_age = age
            second_oldest_patient = patient

    return second_oldest_patient

second_oldest = find_second_oldest(patients)
print(f"The second oldest patient is {second_oldest['name']}, Age: {second_oldest['age']}")
```

Listing 4: O(n) for find second oldest patient

1.6 1e

To do a binary search, we first initiate two pointers, left and right. Then we can update the middle index between the left pointer and the right pointer. Then, the logic is continually searching the middle while the

left pointer is always on the left of the right pointer (that means we are still on the right track of the search). At the same time, we update the left or right pointer depending on the comparison between the value of the middle pointer and the targeted value.

```
def binary_search_for_age(sorted_patients, target_age):
    left, right = 0, len(sorted_patients) - 1
    mid = (left+right)//2
    while left <= right:
        if target_age > sorted_patients[mid]['age']:
            left = mid + 1
            mid = (left+right)//2
        elif target_age < sorted_patients[mid]['age']:
             right = mid -1
             mid = (left+right)//2
        else:
             return sorted_patients[mid]



patient_search = binary_search_for_age(sorted_patients, 41.5)
if patient_search:
    print(f"Patient with age 41.5: {patient_search['name']}")
else:
    print("No patient with age 41.5.")
```

Listing 5: Binary Search for Specific Age

As the extra credit question mentioned, if there exists duplicates in the list, while the binary search still works, we should double-check the left or the right of the searched target to see if there are duplicates.

```
    i = found - 1
    while i >= 0 and sorted_patients[i]['age'] == target_age:
        result.append(sorted_patients[i])
        i -= 1


    i = found + 1
    while i < len(sorted_patients) and sorted_patients[i]['age'] == target_age:
        result.append(sorted_patients[i])
        i += 1
```

Listing 6: Additional code used for checking duplicates in binary search

We find that the patient with age = 41.5 is John Braswell.

1.7 1f

Since the final middle pointer is the targeted value we are looking for, a simple subtraction will calculate the number of patients above a certain age.

```
def above_certain_age(sorted_patients, target_age):
    left, right = 0, len(sorted_patients) - 1
    mid = (left+right)//2
    while left <= right:
        if target_age > sorted_patients[mid]['age']:
            left = mid + 1
            mid = (left+right)//2
        elif target_age < sorted_patients[mid]['age']:
             right = mid -1
             mid = (left+right)//2
        else:
             return len(sorted_patients) - mid -1
    ## If no match here
    return len(sorted_patients) - left
```

Listing 7: Count Patients Above a Certain Age

Also, we consider the case when there are duplicates. The only thing we do is to find the leftmost of the duplicates. Both two functions returned the same answer, Number of patients above 41.5: 150470.

```
def above_certain_age(sorted_patients, target_age):
    left, right = 0, len(sorted_patients) - 1

    # Perform binary search to find the leftmost occurrence of target_age
    while left <= right:
        mid = (left + right) // 2
        if sorted_patients[mid]['age'] <= target_age:  # Check if we need to go right for
    further check
```

```
            left = mid + 1
        else:
            right = mid - 1

    # After the loop, 'left' will point to the first patient whose age is strictly greater
    than target_age
    return len(sorted_patients) - left
```

Listing 8: Count Patients Above a Certain Age when duplicates

1.8 1g

Similarly, to find the number of patients who are at least *low age* years old but strictly less than *high age* years old, we use binary search separately to find the lowest and the highest value. Then we do subtraction.

```
def age_range(sorted_patients, low_age, high_age):
    def above_equal_certain_age(sorted_patients, target_age):
        left, right = 0, len(sorted_patients) - 1
        mid = (left+right)//2
        while left <= right:
            if target_age > sorted_patients[mid]['age']:
                left = mid + 1
                mid = (left+right)//2
            elif target_age < sorted_patients[mid]['age']:
                right = mid -1
                mid = (left+right)//2
            else:
                return len(sorted_patients) - mid
    ## If no match here
        return len(sorted_patients) - left
    return above_equal_certain_age(sorted_patients, low_age) - above_equal_certain_age(
    sorted_patients, high_age)
```

Listing 9: Function for Age Range Query

Based on the question description, we are looking for the range [minage, maxage). Therefore, to do the thorough testing, we use this function to recalculate the number of patients in the database. Figure 4 shows the testing result.

```
min_age = sorted_patients[0]['age']
max_age = sorted_patients[-1]['age']

# 1. Test for total population in the range [min_age, max_age)
total_pop1 = age_range(sorted_patients, min_age, max_age)
print(f"Total population in the range [min_age, max_age): {total_pop1}")

# 2. Test for total population in the range [min_age, max_age + 1)
total_pop2 = age_range(sorted_patients, min_age, max_age + 1)
print(f"Total population in the range [min_age, max_age + 1): {total_pop2}")

# 3. Check total population matches the length of sorted_patients
total_pop_check = len(sorted_patients)
print(f"Total population of the dataset: {total_pop_check}")

# 4. Double-Check if total_pop1 + patients with max_age = total_pop2
patients_with_max_age = len([p for p in sorted_patients if p['age'] == max_age])
print(f"Patients with max_age: {patients_with_max_age}")
print(f"Total population should match: {total_pop1 + patients_with_max_age == total_pop2}")

# 5. Additional tests for edge cases
no_patients_in_range = age_range(sorted_patients, max_age + 1, max_age + 2)  # Should be
    zero
print(f"Total population in an empty range: {no_patients_in_range}")
```

Listing 10: Generality testing

```
Total population in the range [min_age, max_age): 324356
Total population in the range [min_age, max_age + 1): 324357
Total population of the dataset: 324357
Patients with max_age: 1
Total population should match: True
Total population in an empty range: 0
```

Figure 4: 1g Output

1.9 1h

In order to query gender data as well, the basic idea is to search for *results['gender'] == 'male'*.

```
def age_and_gender_range(sorted_patients, low_age, high_age):
    def above_equal_certain_age(sorted_patients, target_age):
        left, right = 0, len(sorted_patients) - 1
        while left <= right:
            mid = (left + right) // 2
            if target_age > sorted_patients[mid]['age']:
                left = mid + 1
            elif target_age < sorted_patients[mid]['age']:
                right = mid - 1
            else:
                return mid
        return left

    start_index = above_equal_certain_age(sorted_patients, low_age)
    end_index = above_equal_certain_age(sorted_patients, high_age)

    # If no patients in the range, return 0
    if start_index >= len(sorted_patients) or sorted_patients[start_index]['age'] >=
    high_age:
        return 0, 0

    # Count the total number of male patients in the range
    num_patients_in_range = end_index - start_index
    num_males_in_range = sum(1 for p in sorted_patients[start_index:end_index] if p['gender'
    ] == 'male')#Use built-in function to count how many males

    return num_patients_in_range, num_males_in_range
```

Listing 11: Function for Age and Gender Range Query

We use a similar method as the previous question to do testing. The result matches the result of 1b 2.

```
min_age = sorted_patients[0]['age']
max_age = sorted_patients[-1]['age']

# 1. Test for total Male population in the range [min_age, max_age)
total_pop1 = age_and_gender_range(sorted_patients, min_age, max_age)
print(f"Total male population in the range [min_age, max_age): {total_pop1[-1]}")
print(f"Total population in the range [min_age, max_age): {total_pop1[0]}")
# 2. Test for total population in the range [min_age, max_age + 1)
total_pop2 = age_and_gender_range(sorted_patients, min_age, max_age + 1)
print(f"Total male population in the range [min_age, max_age + 1): {total_pop2[-1]}")
print(f"Total population in the range [min_age, max_age +1): {total_pop2[0]}")
# 3. Check total male population matches the length of sorted_patients
total_male_pop_check = sum(1 for patient in sorted_patients if patient['gender'] == 'male')
print(f"Total male population of the dataset: {total_male_pop_check}")
```

Listing 12: Function for Age and Gender Range Query testing

2 Exercise 2

2.1 2a

After we print the results 5, we can find the difference. 2e16 is a floating-point number in Python that follows the IEEE 754 standard, which can introduce precision issues for very large numbers. However, 2 * 10 ** 16 is an integer expression, which Python can handle exactly since it uses arbitrary precision for integers. Now, 2e16 = 20000000000000000 (17 digits), which means that the floating-point representation is already at the limit of its precision. This is why adding 1 to 2e16 won't change its value in floating-point arithmetic, because the difference between 2e16 and 2e16 + 1 is smaller than what the floating-point format can represent at that scale. On the other hand, when we compute 2 * 10 ** 16 + 1, we are dealing with integers in Python.

```
print(2e16 +1 == 2 * 10 ** 16+1)
print(2e16)
print(2e16+1)
print(2 * 10 ** 16+1)
```

✓ 0.0s

```
False
2e+16
2e+16
20000000000000001
```

Figure 5: 2a Output

2.2 2b

Figure 6 shows the result of the trial. We use the log-log axis to plot because we aim to visualize the power-law relationship in a linear-line manner and have a more straightforward interpretation. We can observe that as we run the code and h becomes smaller, the absolute error initially decreases. When h reaches around 1e-8, it has a slight fluctuation, and the error finally continually increases. One hypothesis of this fluctuation is the 'conflict' between truncation and round-off errors.

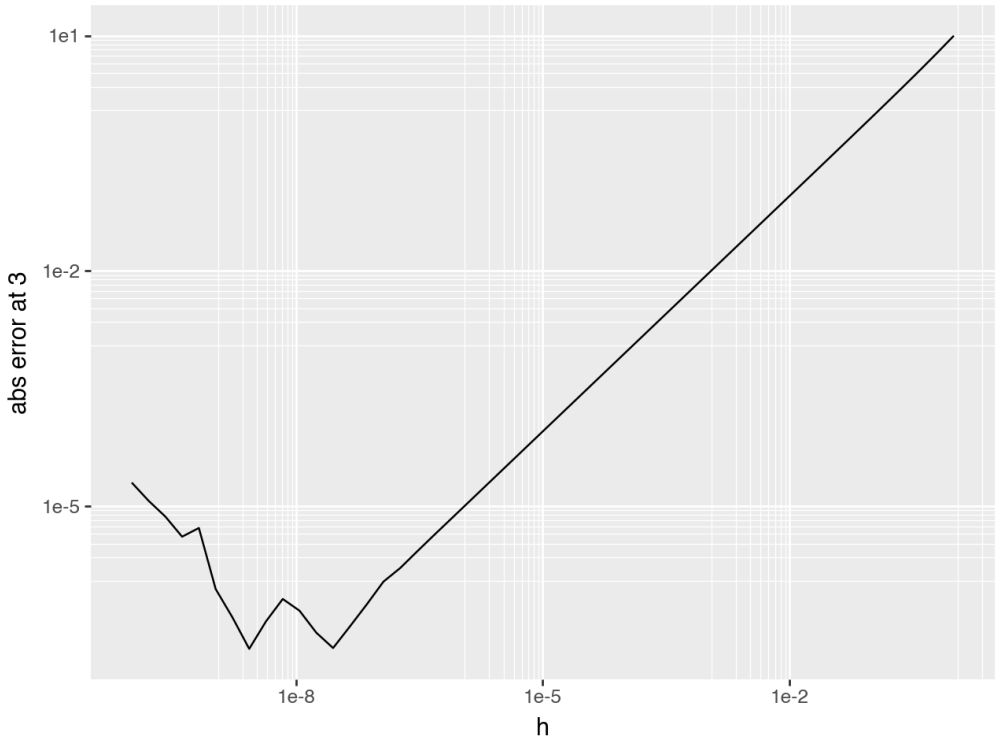


Figure 6: 2b Output

2.2.1 2b Hypothesis testing

We are aiming to theoretically analyze the fluctuation in the error of the finite difference approximation of the derivative of the function $f(x) = x^3$ at $x_0 = 3$. The error is defined as:

$$\text{Error} = \left| \frac{f(x_0 + h) - f(x_0)}{h} - 3x_0^2 \right|$$

where h is a small incrementing number.

1. Truncation Error

The finite difference approximation of the derivative is:

$$D(h) = \frac{f(x_0 + h) - f(x_0)}{h}$$

Using the Taylor series expansion of $f(x_0 + h)$ around x_0:

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + \frac{1}{6}f'''(x_0)h^3 + \mathcal{O}(h^4)$$

For $f(x) = x^3$:

$$\begin{aligned} f(x_0) &= x_0^3 \\ f'(x_0) &= 3x_0^2 \\ f''(x_0) &= 6x_0 \\ f'''(x_0) &= 6 \end{aligned}$$

Thus, we have:

$$f(x_0 + h) = x_0^3 + 3x_0^2 h + 3x_0 h^2 + h^3 + \mathcal{O}(h^4)$$

We can compute the truncation error as below:

$$\begin{aligned} E_{\text{trunc}} &= D(h) - f'(x_0) \\ &= \left(\frac{f(x_0 + h) - f(x_0)}{h} \right) - 3x_0^2 \\ &= \left(\frac{3x_0^2 h + 3x_0 h^2 + h^3}{h} \right) - 3x_0^2 \\ &= \left(3x_0^2 + 3x_0 h + h^2\right) - 3x_0^2 \\ &= 3x_0 h + h^2 \end{aligned}$$

Thus, the truncation error is:

$$E_{\text{trunc}} = 3x_0 h + h^2$$

2.2.2 2. Round-Off Error Analysis

Round-off errors arise due to finite precision in floating-point arithmetic. When subtracting two nearly equal numbers, significant digits can be lost (catastrophic cancellation).
Let ϵ be the machine epsilon (the relative error due to floating-point representation, approximately $\epsilon \approx 1.11 \times 10^{-16}$ for double-precision arithmetic)[1].
In the finite difference approximation of the derivative, the round-off error arises due to the limitations of floating-point arithmetic, especially when subtracting nearly equal numbers.
Any real number x in floating-point arithmetic is represented as:

$$\tilde{x} = x(1 + \delta), \quad |\delta| \leq \epsilon$$

The computed values of $f(x_0)$ and $f(x_0 + h)$ are:

$$\tilde{f}(x_0) = f(x_0)(1 + \delta_1), \quad \tilde{f}(x_0 + h) = f(x_0 + h)(1 + \delta_2)$$

where $|\delta_1|, |\delta_2| \leq \epsilon$.

[1] Check `https://en.wikipedia.org/wiki/Machine_epsilon` for more information

The finite difference approximation is:

$$D(h) = \frac{\tilde{f}(x_0 + h) - \tilde{f}(x_0)}{h}$$

The true difference is:

$$f(x_0 + h) - f(x_0) = \Delta f$$

The error in the numerator due to round-off is:

$$\begin{aligned} \text{Error in Numerator} &= [\tilde{f}(x_0 + h) - \tilde{f}(x_0)] - [f(x_0 + h) - f(x_0)] \\ &= f(x_0 + h)\delta_2 - f(x_0)\delta_1 \end{aligned}$$

Taking absolute values and using $|\delta_1|, |\delta_2| \leq \epsilon$:

$$|\text{Error in Numerator}| \leq \epsilon \left(|f(x_0 + h)| + |f(x_0)| \right)$$

For small h, $f(x_0 + h) \approx f(x_0)$, so:

$$|\text{Error in Numerator}| \leq 2\epsilon |f(x_0)|$$

The round-off error in the derivative approximation is:

$$E_{\text{round}} = \left| \frac{\text{Error in Numerator}}{h} \right| \leq \frac{2\epsilon |f(x_0)|}{h}$$

The absolute round-off error in the finite difference approximation can be approximated as:

$$E_{\text{round}} \approx \epsilon \left(\frac{2|f(x_0)|}{h} \right)$$

This expression accounts for the amplification of the relative error when dividing by a small h.
For $f(x_0) = x_0^3$, this becomes:

$$E_{\text{round}} \approx \epsilon \left(\frac{2|x_0^3|}{h} \right)$$

3. Total Error

The total error is the sum of the truncation error and the round-off error:

$$E_{\text{total}} = E_{\text{trunc}} + E_{\text{round}} = \left(3x_0 h + h^2 \right) + \epsilon \left(\frac{2x_0^3}{h} \right)$$

Substituting $x_0 = 3$:

$$E_{\text{total}} = \left(9h + h^2 \right) + \epsilon \left(\frac{54}{h} \right)$$

Therefore, the behavior of the fluctuation can be understood as follows: Around an optimal h, the truncation error and round-off error are of comparable magnitude, causing the total error to fluctuate.
The mathematical proof above is the proof of the previous hypothesis. By theoretically modeling both error components and analyzing their contributions, we can explain the fluctuation observed in the error as h varies in a certain range.

3 Exercise 3

3.1 3a

The two algorithms are doing sorting on a list (data). As shown in figure 7, we can find them successfully sorting all the data we created using the given data generation functions.

```
def alg1(data):
  data = list(data)
  changes = True
  while changes:
    changes = False
    for i in range(len(data) - 1):
      if data[i + 1] < data[i]:
        data[i], data[i + 1] = data[i + 1], data[i]
        changes = True
  return data

def alg2(data):
  if len(data) <= 1:
    return data
  else:
    split = len(data) // 2
    left = iter(alg2(data[:split]))
    right = iter(alg2(data[split:]))
    result = []
    # note: this takes the top items off the left and right piles
    left_top = next(left)
    right_top = next(right)
    while True:
      if left_top < right_top:
        result.append(left_top)
        try:
          left_top = next(left)
        except StopIteration:
          # nothing remains on the left; add the right + return
          return result + [right_top] + list(right)
      else:
        result.append(right_top)
        try:
          right_top = next(right)
        except StopIteration:
          # nothing remains on the right; add the left + return
          return result + [left_top] + list(left)

def data1(n, sigma=10, rho=28, beta=8/3, dt=0.01, x=1, y=1, z=1):
    import numpy
    state = numpy.array([x, y, z], dtype=float)
    result = []
    for _ in range(n):
        x, y, z = state
        state += dt * numpy.array([
            sigma * (y - x),
            x * (rho - z) - y,
            x * y - beta * z
        ])
        result.append(float(state[0] + 30))
    return result

def data2(n):
    return list(range(n))

def data3(n):
    return list(range(n, 0, -1))
data1 = data1(7)
data2 = data2(7)
data3 = data3(7)
```

Listing 13: Sorting Algorithms

Figure 7: 3a Output

3.2 3b

The first algorithm is bubble sort, while the second one is merge sort. The first algorithm uses two loops (here we use a while loop as the outer loop) to iterate through the list, compare adjacent elements, and swap them if they are in reverse order.
The second algorithm uses the idea of divide and conquer which recursively divides the entire list into two halves and sorts the sublist separately. Specifically, we divide the list into two halves, sort each half recursively, and then merge the sorted halves together. The entire process can be done in two parts. One is taking charge of merging the sorted lists, the other takes charge of recursively running this divide-merge process.

3.3 3c

As shown in the code below, we can plot the comparison of two algorithms over time. Repeatedly, we can also plot for data2 and data3.

```
import time
n_values = np.logspace(1, 4, num=10, dtype=int)
alg1_times_data1 = []
alg2_times_data1 = []

for n in n_values:
    data = data1(n)

    # Time alg1
    data_alg1 = data.copy()
    start_time = time.perf_counter()
    alg1(data_alg1)
    end_time = time.perf_counter()
    alg1_times_data1.append(end_time - start_time)

    # Time alg2
    data_alg2 = data.copy()
    start_time = time.perf_counter()
    alg2(data_alg2)
    end_time = time.perf_counter()
    alg2_times_data1.append(end_time - start_time)

plt.figure(figsize=(8, 5))
plt.loglog(n_values, alg1_times_data1, label='alg1 (Bubble Sort)')
plt.loglog(n_values, alg2_times_data1, label='alg2 (Merge Sort)')
plt.xlabel('n (Number of Elements)')
plt.ylabel('Time (seconds)')
plt.title('Performance of alg1 and alg2 using data1')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```

Listing 14: Algorithms comparisons

It is apparent to see that, for `data1`,

- **alg1** shows a curve that is approximately linear with a slope of about 2, indicating an apparent $O(n^2)$ scaling.
- **alg2** shows a curve that is approximately linear with a slope less than 2, closer to 1, indicating an apparent $O(n \log n)$ scaling.

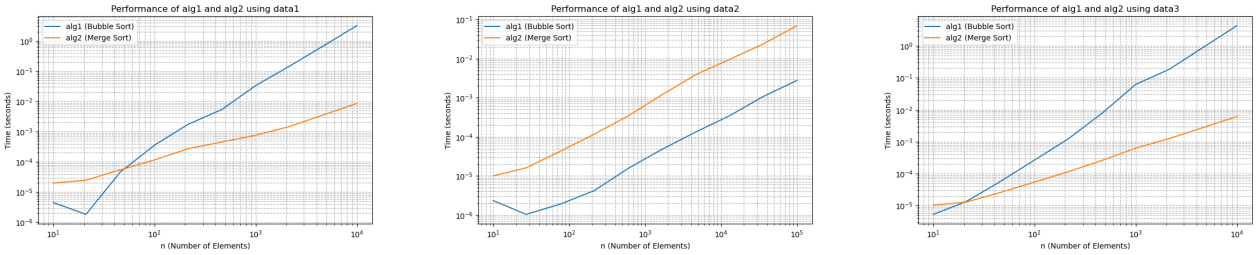
For `data2` (sorted data):

- **alg1** performs significantly better than in the previous case, showing nearly linear scaling $O(n)$. This is because bubble sort detects the sorted list quickly and terminates early.
- **alg2** maintains its $O(n \log n)$ performance.

For `data3` (reverse-sorted data):

- **alg1** shows $O(n^2)$ scaling, as it needs to perform the maximum number of swaps.
- **alg2** maintains its $O(n \log n)$ performance.

Figure 8 shows the comparison results of three different data.



(a) Data1 (b) Data2 (c) Data3

Figure 8: Comparison Results of two sorting algorithms under different data.

3.4 3d

For the first dataset which is generated chaotically (in one specific manner), when the number of elements is larger than 50, using the second algorithm will be better since the time complexity is O($nlog(n)$) which will be more efficient than O(n^2). For the second dataset, which is already created in sorted order, using bubble sort with a time complexity of O(n) will be more efficient than using merge sort with O($nlog(n)$) time complexity. For the third dataset, when the number of elements is larger than 20, the merge sort will be more efficient.

All in all, the best case for the first algorithm (bubble sort) reaches the time complexity with O(n), and the worst(average case) case reaches the time complexity with O(n^2). All the cases for merge sort will reach an O($nlog(n)$) time complexity. We can decide on a better algorithm based on the number of elements.

However, sometimes there will be a trade-off between time complexity and space complexity. Since merge sort requires additional space for storing the split lists, it may require more memory than bubble sort. But it depends on the application scenarios and what data we are using.

4 Exercise 4

4.1 4a, 4b, 4c

I combine the answers of 4a, 4b, and 4c together. By using the code shown in Listing 16, we can test the functions we created for the Tree class.

```
class Tree:
    def __init__(self):
        self._value = None
        self._data = None
        self.left = None
        self.right = None

    def add(self, patient_id, id):
        if self._value is None:
            self._value = patient_id
            self._data = id
        elif patient_id < self._value:
            # Left subtree
            if self.left is None:
                self.left = Tree()
            self.left.add(patient_id, id)  # Recursive call to left subtree
        elif patient_id > self._value:
            # Right Tree
            if self.right is None:
                self.right = Tree()
            self.right.add(patient_id, id)  # Recursive call to right subtree
        else:
            # Value already exists
            self._data = id

    def __contains__(self, patient_id):
        if self._value == patient_id:
            return True
        elif self.left and patient_id < self._value:
            return patient_id in self.left
        elif self.right and patient_id > self._value:
            return patient_id in self.right
        else:
            return False

    def has_data(self, data):
        if self._data == data:
            return True
        if self.left and self.left.has_data(data):
            return True
        if self.right and self.right.has_data(data):
            return True
        return False
```

Listing 15: BST generation

```
my_tree = Tree()
for patient_id, initials in [(24601, "JV"), (42, "DA"), (7, "JB"), (143, "FR"), (8675309, "
    JNY")]:
    my_tree.add(patient_id, initials)
# Test cases
test_ids = [24601, 42, 1492, 8675309, 12345]
for test_id in test_ids:
    if test_id in my_tree:
        print(f"Patient ID {test_id} is in the tree.")
    else:
        print(f"Patient ID {test_id} is not in the tree.")

# Test the has_data method
test_data_values = ["JV", 24601, "XYZ", "FR", "JNY", "AB"]
for data_value in test_data_values:
    if my_tree.has_data(data_value):
        print(f"Data '{data_value}' is present in the tree.")
    else:
        print(f"Data '{data_value}' is NOT present in the tree.")
```

Listing 16: Code for testing

Figure 9 shows the test results for the functions in the Tree Class.

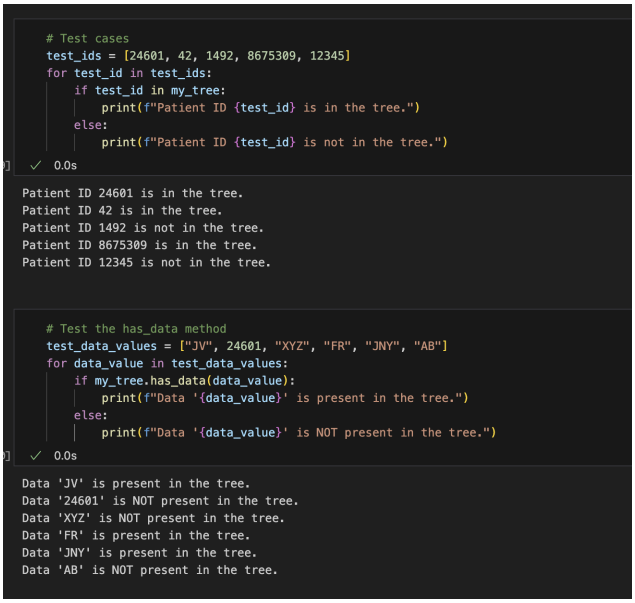


Figure 9: 4a visualization

4.2 4d

With the help of CHATGPT, I used the function data1 to generate the random data. Then, I use the Tree class before to build up trees using the random data. Then, using the same method as the previous question's visualization, we can observe the comparison between two algorithms.

```
import random
def data1(n): ### This function is generated by CHATGPT
    patient_ids = random.sample(range(1, n * 10), n)  # Unique patient IDs
    patient_data = [''.join(random.choices('ABCDEFGHIJKLMNOPQRSTUVWXYZ', k=2)) for _ in
    range(n)]
    return list(zip(patient_ids, patient_data))
## I use alg1 and alg2 to represent two search functions. It helps me to directly use the
    previous visualization functions in previous exercise questions
def alg1(tree, test_ids):
    for test_id in test_ids:
        _ = test_id in tree

def alg2(tree, test_data_list):
    for data in test_data_list:
        _ = tree.has_data(data)

n_values = np.logspace(1, 4, num=10, dtype=int)
alg1_times_data1 = []
alg2_times_data1 = []

for n in n_values:
    data = data1(n)
    tree = Tree()
    for patient_id, patient_data in data:
        tree.add(patient_id, patient_data)

    # Prepare test data (50% present, 50% absent) This step is inspired by CHATGPT
    # At the very beginning, I was stuck at the stage where I did not know how to generate
    the test data,
    # and Chatgpt inspired me to randomly sample from the original data
    num_tests = 100
    num_present = num_tests // 2
    num_absent = num_tests - num_present

    # alg1 (__contains__)
    present_ids = random.sample([pid for pid, _ in data], k=min(num_present, len(data)))
    absent_ids = random.sample(range(n * 10, n * 20), k=num_absent)
    test_ids = present_ids + absent_ids

    # alg2 (has_data)
    present_data = random.sample([pdata for _, pdata in data], k=min(num_present, len(data))
    )
```

```
    absent_data = [''.join(random.choices('ABCDEFGHIJKLMNOPQRSTUVWXYZ', k=2)) for _ in range
    (num_absent)]
    test_data_list = present_data + absent_data
### ------------Here below is simlar as the previous exercise ---------

    data_alg1 = test_ids.copy()
    start_time = time.perf_counter()
    alg1(tree, data_alg1)
    end_time = time.perf_counter()
    alg1_times_data1.append(end_time - start_time)


    data_alg2 = test_data_list.copy()
    start_time = time.perf_counter()
    alg2(tree, data_alg2)
    end_time = time.perf_counter()
    alg2_times_data1.append(end_time - start_time)
```

Listing 17: Performance Analysis of two search methods

We can plot the results as shown in figure 10. We can observe that the contains method shows an $O(log(n))$ time complexity while the has data method shows a $O(n)$ time complexity. To visualize the tree construction

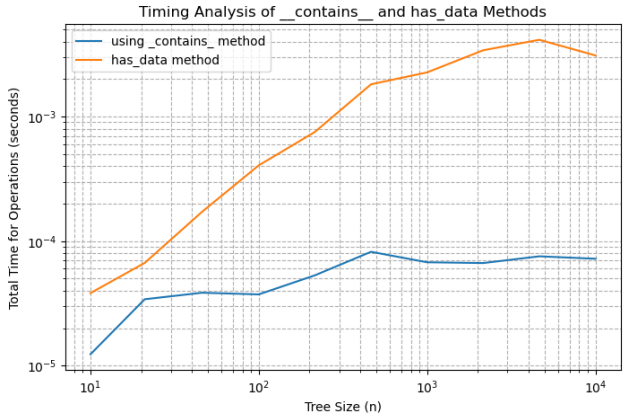


Figure 10: Time analysis of two methods

time complexity (in between $O(n)$ and $O(n^2)$), we enlarge the number of data up to 10^7, the result is shown in 11.

```

n_values = np.logspace(4, 7, num=10, dtype=int)  # From 10^4 to 10^7 patients, we enlarge
    the number of n to expect a balanced tree
construction_times = []
#Similar operation
for n in n_values:
    patients = data1(n)
    start_time = time.perf_counter()
    tree = Tree()
    for patient_id, data in patients:
        tree.add(patient_id, data)
    end_time = time.perf_counter()
    construction_times.append(end_time - start_time)
    print(f"Constructed tree with {n} elements in {end_time - start_time:.4f} seconds.")

# Theoretical times for O(n) and O(n^2), this is helped by CHATGPT
n_values_float = n_values.astype(float)
on_times = construction_times[0] * (n_values_float / n_values_float[0])  # Scale O(n) curve
on2_times = construction_times[0] * (n_values_float / n_values_float[0]) ** 2  # Scale O(n
    ^2) curve

plt.figure(figsize=(10, 6))
plt.loglog(n_values, construction_times, marker='o', label='Measured Construction Time')
plt.loglog(n_values, on_times, linestyle='--', label='$O(n)$')
plt.loglog(n_values, on2_times, linestyle='--', label='$O(n^2)$')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time to Construct Tree (seconds)')
plt.title('Tree Construction Time Analysis')
plt.legend()
plt.grid(True, which="both", linestyle='--')
```

```
29 plt.show()
```

Listing 18: Tree construction time complexity

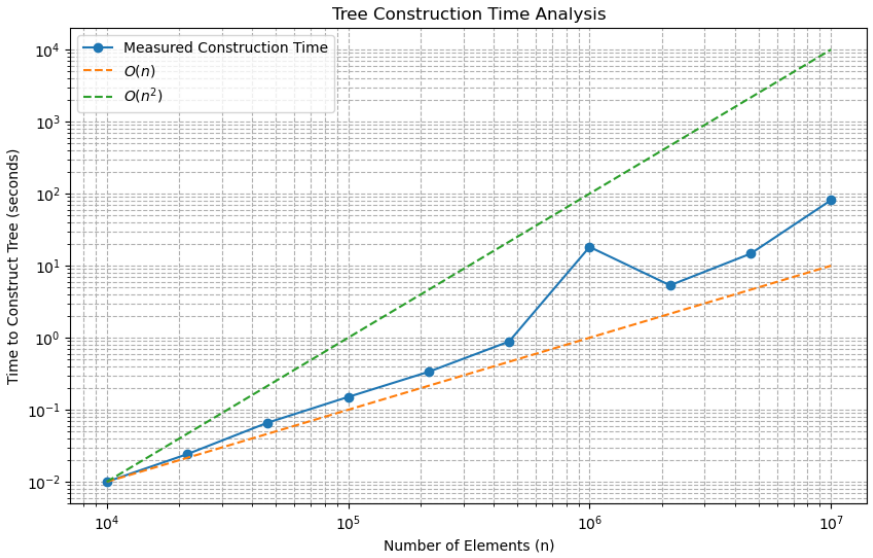


Figure 11: Time Complexity of tree construction

4.3 4e

If a specific value like patient id = 1 is used consistently, the performance analysis may only reflect the behavior of the system for that particular input. For example, if we are continually inserting sorted values, then the search process will lead to the worst-case scenario. Also, when the number of data inputs is small, it is much more likely that the tree is imbalanced. Therefore, a single test data can not evaluate the time complexity or does not allow for observing trends or patterns in performance as the input size or complexity changes.

5 Exercise 5

The data I want to use is called Weibo-COV [1], a dataset with a great amount of Chinese social media platform data on the topic of COVID-19. I find this data on the official website of the EMNLP conference and on GitHub. This dataset is licensed under the MIT License. I think this dataset is interesting because it provides tons of data that contains people's moods and reactions during the pandemic. I can do different analyses and visualizations with the data, for example, Choropleth, Violin plot, and other sentiment analysis (machine learning based) or topic analysis like LDA (Latent Dirichlet allocation).

References

[1] Y Hu, H Huang, A Chen, and XL Mao. 2020. Weibo-COV: a large-scale COVID-19 social media dataset from Weibo. Arxiv. *Preprint posted online May* 19 (2020).