CSE-2050 – Data Structures and Object-Oriented Design

Fall 2022

Instructor	Office	Contact Details	Office Hours
Hasan Baig	305C	Email: hasan.baig@uconn.edu	TuThu: 4pm – 5pm
Joseph Steenhuisen (TA)	Virtual *(Contact for in-person meeting)*	Email: joseph.steenhuisen@uconn.edu	MoWed: 11–2:30, 3:30–4:30 Tu: 3:15–4:30 Th: 10–2, 3:15–4 Fri:12–2:30

Assignment - 1

Release Date: September 17, 2022	**Due by: September 27, 2022**
Total points: 300	**Points obtained:**
	Scaled marks (out of 12.5):

Students' Name(s):	**Students' ID(s):**

Purpose:
The purpose of this assignment is to help you strengthen your understandings about the topics related to Object-oriented design, timing analysis and linked-lists. In addition, programming tasks help you practice implementing these concepts.

Instructions:
1. This assignment should be submitted in a group of up to two students.
2. All theoretical questions should be answered in the space provided and the scanned copies be uploaded on GradeScope under **Assignment 1 – Theory** category. You can attach extra sheet if needed.
3. Source files for Programming exercises should be directly uploaded on GradeScope under **Assignment 1 – Programming** category.

Grading Criteria:
1. Your assignments will be checked by instructor/TA followed by a Viva (if needed).
2. During the viva, you will be judged whether you understood the question yourself or not. If you are unable to answer correctly to the question you have attempted right, you may lose your points. No excuses/justifications will be entertained.
3. Zero will be given if the assignment is found to be plagiarized.
4. Untidy work will result in reduction of your points.

Late submission penalty:
- 1-day late submission – 20% deduction of the maximum allowable marks.
- 2-days late submission – 40% deduction of the maximum allowable marks.
- No submission will be accepted after two days of the original deadline.

CLO Assessment:
This assignment assesses students for some portion of the following course learning outcomes.

Course Learning Outcomes		CLO Assessed
CLO 1	Write programs in python using imports, functions, and object-oriented programming.	✓
CLO 2	Compare data structures and algorithms based on time and space complexity and choose the correct ones for a given problem.	✓
CLO 3	Implement abstract data types (stacks, queues, dequeues, mappings, priority queues) using various data structures (lists, linked lists, doubly linked lists, heaps, trees, graphs) and algorithms	✓
CLO 4	Use recursive algorithms to solve problems.	

Questions
1. Define Class, Instance, Inheritance, Composition, and Polymorphism, with examples. (5x5 points)

2. Define encapsulation, subclass, superclass, method overloading and method overriding with examples. (5x5 points)

3. Calculate the time complexity of the following codes.
   (25x2 points)

```
#Program 1
def print_f(data):
    n = len(data)
    i=1
    while i <= n:
        i = i * 2

def calc_data(data):
    for i in range(1, len(data)):
        print_f(data)
```

```
#Program 2
for i in range(0,N):
    for j in range(N,i,-1):
        a=a + i+j

for j in range(0,N/2):
    b=b + i+j
```

Programming Exercises

a. Programming Exercise 1 - Running Time Analysis

We have two goals in this assignment:

- Use python to fit data with user-defined functions
- Plot raw data and a best-fit curve on the same figure

We will use the `scipy` module for the first and matplotlib (see lab 3) for the second. You should be able to install both in terminal using pip; e.g.:

```
pip install scipy
```

If you are struggling, you may consider using Anaconda - an all-in-one python install designed for data science. Anaconda includes many common packages, including the ones used here.

Fitting Data
The curve fitting functionality we are interested in is in scipy's optimize module. We can fit data to a function of our choice as follows:

```
from scipy import optimize
params, params_cov = optimize.curve_fit(func, xdata, ydata)
```

`optimize.curve_fit` returns two collections - the parameters that make `func` best fit our data, and the covariance of those parameters (which can be ignored for this assignment).

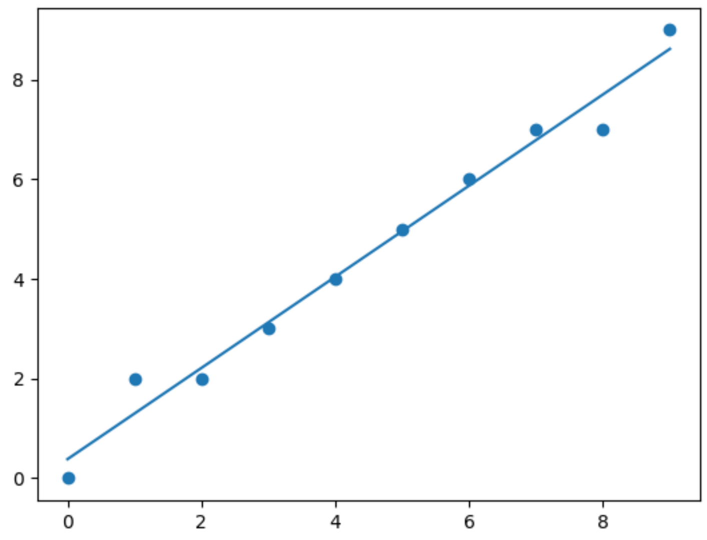
An example:

```
from scipy import optimize
```

```
from matplotlib import pyplot as plt

# linear fitting function
# the first parameter of a fitting funciton must be x
# subsequent parameters are automatically adjusted by curve_fit

def lin(x, m, b):
       return m*x + b

# generate data on y = 1*x + 0, then add some "noise"

xdata = [i for i in range(10)]

ydata = [i for i in range(10)]

y[1] = 2
y[8] = 7

# Find parameters for lin that best fit xdata and ydata

params, params_cov = optimize.curve_fit(lin, xdata, ydata)
```

```
# unpack the calculated parameters
m = params[0]
b = params[1]
# create an empty list for the line of best fit
y_fit = []
for x in xdata:
    y_fit.append(lin(x, m, b))
# plot the raw data and the line of best fit
plt.figure()
plt.scatter(xdata, ydata)
plt.plot(xdata, yfit)
plt.show()
```

See the included figure `example_data_fitting.png` below:



Task 1: `Fitting.py`

Create three fitting functions:

- `const` - tries to fit with $f(x) = c_0$
- `lin` – tries to fit with $f(x) = c_1 \cdot x + c_0$
- `quad` - tries to fit with $f(x) = c_2 \cdot x^2 + c_1 \cdot x + c_0$

Quantifying a Fit

We may ask ourselves “how good is that fit?” To answer that, we need to define the error of our fit relative to the provided data. There are many ways to do this, but we will use the “root mean square” method - we will calculate the square root of the mean of the squares of the error at all points.

The sum of the squares of the errors (se) is:

$$se = \sum_{i=0}^{N-1} (y_{data}[i] - y_{fit}[i])^2$$

The mean (average) square error (mse) is:

$$mse = \frac{1}{N} \cdot se$$

And the square root of the mean square error (RMS) is: $\sqrt{mse}$

Putting it all together:

$$RMS = \sqrt{\frac{\sum_{i=0}^{N-1}(y_{data}[i] - y_{fit}[i])^2}{N}}$$

For the data above, if our fit line was `y=1 ·x+0`, then the RMS can be calculated as follows:

$$se = (0-0)^2 - (1-2)^2 - (2-2)^2 - (3-3)^2 + \cdots + (8-7)^2 + (9-9)^2 = 2$$

$$mean_{sq_{err}} = \frac{1}{10} \cdot 2 = 0.2$$

$$RMS = \sqrt{0.2} \approx 0.4472$$

Task 2: `Fitting.py`

Write a function named fit_data that takes three inputs:

- fitting function
- some xdata
- some ydata

and returns 3 outputs (by returning a 3-tuple):

- parameters of best fit
- the root mean square error of that fit
- a list containing the y-data of that fit

The numbers you get back will likely be very long floats. If we truncated them to integers, your function's return might look something like:

```
>>> from Fitting import *
>>> x = [i for i in range(5)]
>>> y = [i for i in range(5)]
>>> fit_data(lin, x, y)
(array([1, 0]), 0, [0, 1, 2, 3, 4])
```

Notes:

- Do not actually truncate your return values. We did it here for demonstration purposes.
- array denotes a special class used in scipy - you do not need to "get rid of it", and it should not impact the rest of this assignment if you treat it like a typical list.

Algorithm Timing

We will look at sorting algorithms more in-depth in the coming weeks. For now, it is sufficient to know that these algorithms should take a (possibly) unsorted list as input, and modify that list until it is sorted. These algorithms do not need to return anything in Python - when a collection is passed as an argument to a function, any modifications to that collection are persistent outside of the function.

```
def bubble_sort(L):
    n = len(L) # no. of items in list
    for i in range(n): # for every item
        for j in range(n): # compare to every other item
            if L[i] < L[j]: # if out of order:
                L[i], L[j] = L[j], L[i] # swap items

L = [1, 5, 4]
print(L) # expect: [1, 5, 4]

bubble_sort(L)
print(L) # expect: [1, 4, 5]
```

Task 3 - `TimingPlot.py`

Generate a series of x (number of items to sort) and y (time to sort) data using the above algorithm. Use randomly sorted lists. For instance, to generate a list of 100 random integers, you might write:

```
import random
n = 100
L = [random.randint(0, n) for i in range(n)]
```

- Fit that data (n vs time) with the linear and quadratic functions. Choose n such that you can clearly see the expected shape of the curve.
- Generate a figure with three curves:
  - a scatter plot of the raw data
  - a line of best fit with `lin`
  - a line of best fit with `quad`
- Follow best practices when presenting data:
  - Scale your data so the axes numbers are between 1 and 1000
  - Include axis labels with units
  - Label each curve clearly, either with a textbox on the figure or a legend
  - Save your figure as "bestfit.png".

We are purposefully giving you flexibility here - how should you generate the lists as n grows? How should you time the functions? You solved a very similar problem in lab; try to apply those techniques here.
Note that we will manually grade the final plot, and that you produced the data appropriately. Bad-faith attempts (like arbitrarily generating numbers for your times) will not receive any credit on this assignment. **Include all code you use to generate data and the figures.**

Submission
At a minimum, submit the following files:

- Fitting.py
  – const()
  – lin()
  – quad()
  – fit_data()

- TimingPlot.py
  – bubble_sort()
  – whatever functions you use to generate timing data

- bestfit.png

Also submit any other files you may use to generate data or the final figure.

Grading

Auto Graded

- 40 - data fitting functionality

Manually Graded

- 30 - TimingPlot.py generates time vs n data appropriately and times functions correctly
- 30 - "bestfit.png" figure

b. Programming Exercise 2 - Circular Linked Data Structure

Not all linked data structures are linear - it is often helpful to have nodes that link to multiple nodes (as in Trees and Graphs), or nodes that link together in other non-linear ways.

In this assignment, we build a circular linked data structure - a list that loops around on itself. This is useful when we need to repeatedly loop through a series of nodes. For instance, an operating system may keep such a circular list of tasks that need CPU time, and continually cycle through the list of tasks.

We will implement two classes:

- `Task` - a task for the operating system to complete. Each task has the following attributes, in addition to any "linking" attributes (like `next` or `prev`) you decide are necessary:
  - `id` - a unique identifier (an int)
  - `time_left` - the amount of time necessary to complete this task
  - `reduce_time()` - reduces `time_left` by the appropriate amount

- TaskQueue - A circular data structure with attributes:
  - current - the current task to process
  - time_per_task - the amount of time to dedicate to each task before moving to the next one. This has a default value of 1, but should be a parameter a user can specify when creating a TaskQueue.
  - `add_task()` - adds a task **immediately before current**. Should be *O*(1).
  - `remove_task()` - removes the task with a given id. Should be *O*(n).
  - `len` - the number of tasks in the `TaskQueue`. *O*(1).
  - `is_empty()` - returns True if the `TaskQueue` is empty; false otherwise. *O*(1).
  - `execute_tasks()`
    - ∗ executes tasks cyclically. Each task will run for `time_per_task` or the amount of time it has remaining, whichever is lower. This should print out information whenever a task is finished (see examples for string formatting). At the end, return the total time it took to execute all tasks. You do not need to test the print outs but should test the return value. See next page for class diagrams.

Special Behavior

- If a task is reduced to 0 seconds by `reduce_time`, then the `TaskQueue` should remove that task from the queue and print out info - the task id and the time it finished (see Examples below for formatting)

- If a user tries to remove a task with an id that is not in the `TaskQueue`, you should raise a `RuntimeError` with an appropriate string.

- This check should run in $O(1)$. Think - what data structure can you use that allows $O(1)$ membership testing?
- **Test this behavior.** (The error, not the running time)

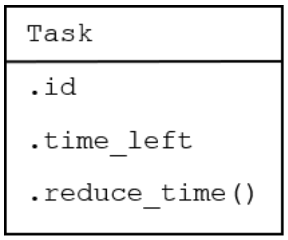


Figure 1: Class diagram for a Task.

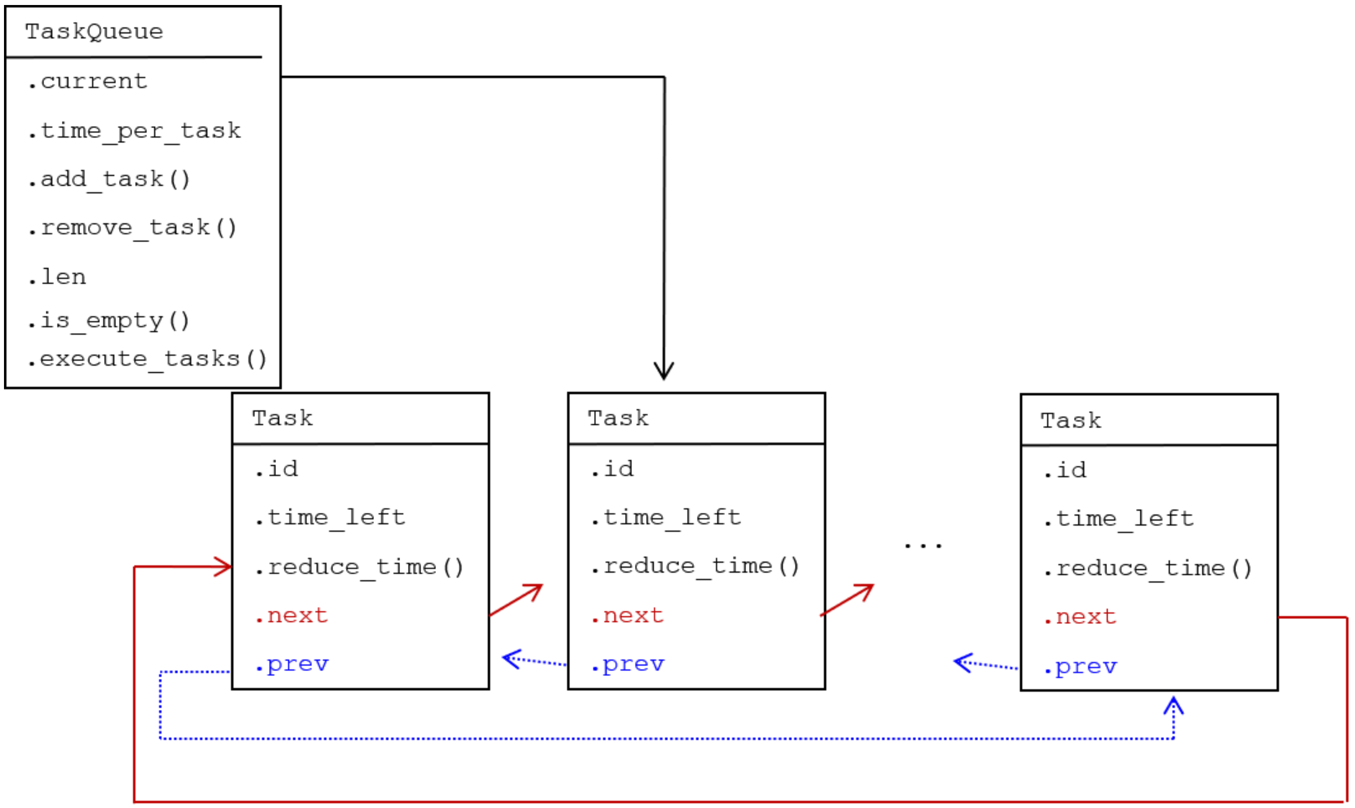


Figure 2: Class diagram for a `TaskQueue`. We use red solid arrows for `next` and blue dotted arrows for `prev` pointers. Note that `TaskQueue.current` is expected to cycle through Tasks as the program runs. Note - this class uses `next` and `prev` pointers. Decided for yourself if you need both.

Examples

Any examples below are intended to be illustrative, not exhaustive. Your code may have bugs even if it behaves as below. Write your own tests, and think carefully about edge cases.

```
>>> from TaskQueue import *
>>> t1 = Task(id=1, time_left=3)
>>> t2 = Task(id=2, time_left=1)
>>> t3 = Task(id=3, time_left=5)
>>> tasks = [t1, t2, t3]
>>> TQ = TaskQueue(time_per_task=1)
>>> for task in tasks:
```

```
...     TQ.add_task(task)
>>> TQ.remove_task(17) # should raise an error

... # info truncated to not give away the answer
RuntimeError: id 17 not in TaskQueue
>>> time = TQ.execute_tasks() # note that this prints info *and* returns a
time Finished task 2 at t = 2 seconds
Finished task 1 at t = 6 seconds
Finished task 3 at t = 9 seconds
>>> print(f"time = {time}")
time = 9
```

External Modules
Do not use any imported modules (`math, collections,...`) when implementing functionality. It is okay to use imported modules for testing; e.g. the `random` and `string` modules for generating random items.
It is okay to import modules you write yourself for this assignment; e.g. any data structures you write yourself.

Submission
At a minimum, submit the following files:
- `TaskQueue.py`
- `TestTaskQueue.py`

Grading
Code readability matters. Code that is difficult to read (no comments, poor variable/function names, not enough whitespace) will incur a penalty of 10 points.
Most of this assignment is manually graded, including tests. Use test driven development - write tests for functionality, run those tests to verify they fail, then implement the functionality.

Auto-graded
- 10 - `Task` class
- 10 - `TaskQueue` class

Manually-graded
- 10 - `Task` class (including tests)
- 70 - `TaskQueue` class (including tests)