National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester II, 2022/2023

Mission 6
Diagnostics

Release date: 12th February 2023
Due: 18th February 2023, 23:59

Required Files

- mission06-template.py
- hi_graph_connect_ends.py
- diagnostic.py
- fibtrace.py

Background:

After days of training, the disciples are now able to freely conjure up curves. However, simple curves do not strengthen one's mindpower enough to manipulate real-life objects.

Grandmaster Ben explains that there is an interesting way to make the curves much more potent. Through Gosperization, images of these curves can be multiplied to greatly increase the strength of defence. However, he cautions that the process of Gosperization should be done quickly, so as not to waste the efficiency of the mind.

"Good. The technique of the mind comes in many forms, not just from a single source. Now multiply your image in your head, with the images coming in from different angles. Make sure the images come quickly into your mind..."

Information:

The Python source file has been renamed to **`hi_graph_connect_ends.py`** and modified to include the function `connect_ends` which should have been previously coded. You may now use the function directly from this source file.

To perform timing and tracing analysis, a new Python source file **`diagnostic.py`** has been provided to you. It contains various functions to help analyse the performance of your code.

For your convenience, the template file **`mission06-template.py`** contains a line to import the Python source file **`hi_graph_connect_ends.py`**, as well as **`diagnostic.py`**. Use the template file to answer the questions.

This mission has **three** tasks.

Task 1: (4 marks)

We now have three functions to compute gosper curves:

- `gosper_curve`
- `gosper_curve_with_angle` with argument `lambda lvl: pi/4` using the "hand-crafted" definition of `gosperize_with_angle` above
- `your_gosper_curve_with_angle` in Mission 5 Task 3 that uses `your_gosperize_with_angle` based on `put_in_standard_position`.

Your task is to compare the time measurements of these functions for computing selected points on the curve at a **significant level**.[1] The function `profile_fn(fn, n)` has been defined for you to help you with this task.

The `profile_fn(fn, n)` function defined in **`diagnostic.py`** will report the time in milliseconds required to evaluate a function `fn` for `n` times. For example, evaluating

```
profile_fn(lambda: gosper_curve(10)(0.1), 50)
```

will print out the time to compute the point at .1 on the level 10 `gosper_curve` 50 times.

An example of a sample run is as follows:

```
>>> print(profile_fn(lambda: gosper_curve(10)(0.1), 50))
2.2289029999997823
>>> print(profile_fn(lambda: gosper_curve_with_angle(
        10, lambda lvl: pi/4)(0.1), 50))
2.4584490000005843
```

In the example above, the difference in time taken to profile both `gosper_curve` and `gosper_curve_with_angle` is very small (~0.2ms). This is insufficient to show whether there may be any difference in time taken to compute the two functions. You need to adjust the parameters well in order to obtain a significant difference among the two functions.

Similar to the sample run above, time **each of your three gosper curve functions for at least 5 times** to report the average time measurements for more accurate results, and present your findings in a neat and organized manner, as provided by the example in `mission06-template.py`.

Use your results from the three functions used to conclude, in general, if functions that are more customized (`gosper_curve` and `gosper_curve_with_angle`) have an advantage (in speed) as compared to functions which are more customizable (in this case, `your_gosper_curve_with_angle`).

Think of customizable functions as functions that use higher-level operations compared to customized functions. For example, the identity function is more customizable than the function that always returns 1. Another example is that a function that uses a user-defined higher-order function is more customized than a function that only uses basic operations to achieve the same result.

[1] Significant level refers to one that does not take a ridiculous amount of time to perform the function, but yet enough to show the difference in time measurements between the different functions.

Task 2: (4 marks)

One of your fellow disciples, Joe, isn't entirely happy with the style of several of the definitions found in this training. In particular, he feels the code goes overboard in inventing names for values that are used infrequently, and this lengthens the code and burdens someone reading the code with remembering the invented names. For example, he thinks the definition

```
def rotate(angle):
    def transform(curve):
        def rotated_curve(t):
            pt = curve(t)
            x, y = x_of(pt), y_of(pt)
            cos_a, sin_a = cos(angle), sin(angle)
            return make_point(cos_a*x - sin_a*y, sin_a*x + cos_a*y)
        return rotated_curve
    return transform
```

would be a bit more readable if the name `pt` for the value of `curve(t)` was dropped. He proposes instead:

```
def joe_rotate(angle):
    def transform(curve):
        def rotated_curve(t):
            x, y = x_of(curve(t)), y_of(curve(t))
            cos_a, sin_a = cos(angle), sin(angle)
            return make_point(cos_a*x - sin_a*y, sin_a*x + cos_a*y)
        return rotated_curve
    return transform
```

The experienced curve instructor, Junwei, warns Joe that the substitution of `pt` with `curve(t)` he uses in `joe_rotate` is actually more computationally expensive compared to just using the abbreviation `pt`.

1. Does `joe_rotate` work and achieve the same purpose as `rotate`? (1 mark)
2. Briefly explain why using `joe_rotate` as a function in place of the original `rotate` in the definition of `gosper_curve` will turn a process whose time is linear in the level into one which is exponential in the level. (3 marks)

Task 3: (4 marks)

In this exercise we will learn how to use `trace`. Open **`fibtrace.py`** in IDLE:

```
from diagnostic import *
from hi_graph_connect_ends import *
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
trace(fib)
fib(3)
untrace(fib)
fib(3)
```

Run this Python file. You should get the output as shown in Figure 1. When a function is `traced`, information on every entry and exit of that function is printed onto the screen. Upon entry, the function call is printed. Upon exit, the return value is printed after the arrow -->. Thus, we can see from the trace that `fib` has been called five times (there are five function calls, and five corresponding returned values). This information can be used to debug a function. `untrace` will need to be called to stop receiving information on function entry and exit.

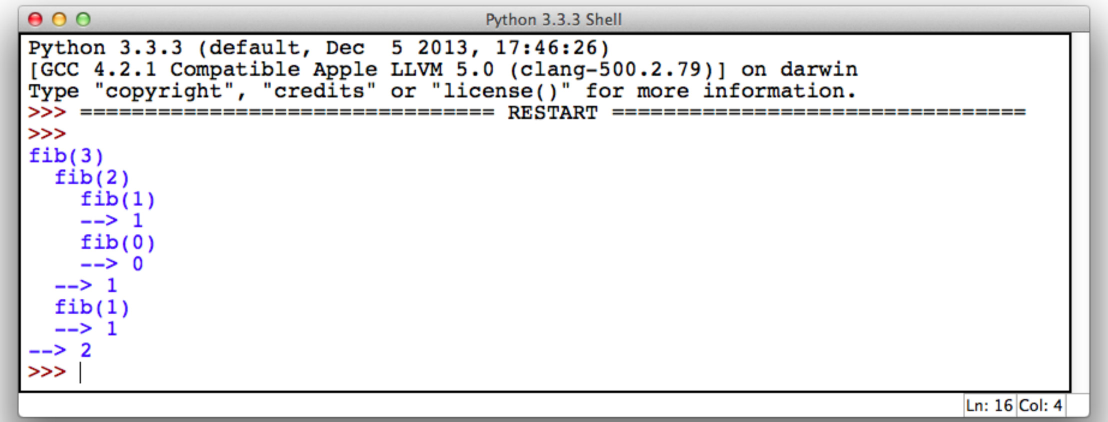

Figure 1: Sample call to `trace`.

Trace `x_of` to show how dramatically Junwei's warning is confirmed when computing points on the gosper curve using `joe_rotate` as a subfunction in place of the original `rotate`.

Submit a table summarizing the number of calls to `x_of` by `gosper_curve` using the two different rotating functions for 5 illustrative levels.

Hints

1. A way to use `joe_rotate` in place of `rotate` is to use `replace_fn`. This is how to use it:

```
>>> original_rotate = rotate
```

Now both `original_rotate` and `rotate` refer to the same function. We do this so that we will have a 'handle' on the original `rotate` function.

```
>>> replace_fn(rotate, joe_rotate)
```

Both `rotate` and `joe_rotate` now refer to Joe's version of the rotate function. If we called `gosperize` now, it would utilize Joe's version of rotate. In this way, we are able to take timing for `gosper_curve` that uses `joe_rotate` without having to edit `gosperize`. If we did not define `original_rotate`, none of the variables would refer to the original rotate function and we would have 'lost' the function. To restore the original rotate function, do

```
>>> replace_fn(rotate, original_rotate)
```

2. Do not call a drawing function during tracing, otherwise Python may overflow your Python buffer by generating large amounts of trace output. Instead, trace the number of calls to `x_of` by obtaining the mid-point of `gosper_curve`, e.g. for level 3, call `gosper_curve(3)(0.5)`.