EE466 Fall 2019 HW 2
Lewis Collum (0621539) EE/CE
Due: September 25, 2019

1.7.b

```
class Compiler:
    def __init__(self, instructions, executionTime):
        self.instructions = instructions
        self.executionTime = executionTime

    @property
    def instructionRate(self):
        return self.instructions / self.executionTime

compilers = {
    'A': Compiler(
        instructions = 1.0e9,
        executionTime = 1.1),
    'B': Compiler(
        instructions = 1.2e9,
        executionTime = 1.5)
}

instructionRateRatio = compilers['A'].instructionRate / compilers['B'].instructionRate
print((f"Compiler A's processor has a clock that is {instructionRateRatio:.2}"
       f" times faster than Compiler B's processor"))
```

```
Compiler A's processor has a clock that is 1.1 times faster than Compiler B's processor
```

1.8.2

Processor Name	Clock Rate (GHz)	Voltage (V)	Static Power (W)	Dynamic Power (W)
Pentium 4 Prescott	3.6	1.25	10	90
Core i5 Ivy Bridge	3.4	0.9	30	40

```
from collections import namedtuple
Processor = namedtuple('Processor', ['clockRate', 'voltage', 'staticPower', 'dynamicPower'])
processors = {row[0]: Processor(*row[1:]) for row in table}

for name, processor in processors.items():
    totalPower = processor.staticPower + processor.dynamicPower
    staticPowerPercentage = processor.staticPower / totalPower
    staticToDynamicRatio = processor.staticPower / processor.dynamicPower

    print((f"{name}:\n"
           f"  Static Power Percentage = {staticPowerPercentage*100:.3}%\n"
           f"  Static to Dynamic Ratio = {staticToDynamicRatio:.2}\n"))
```

```
Pentium 4 Prescott:
  Static Power Percentage = 10.0%
  Static to Dynamic Ratio = 0.11

Core i5 Ivy Bridge:
  Static Power Percentage = 42.9%
  Static to Dynamic Ratio = 0.75
```

1.9.1

Creating the Table

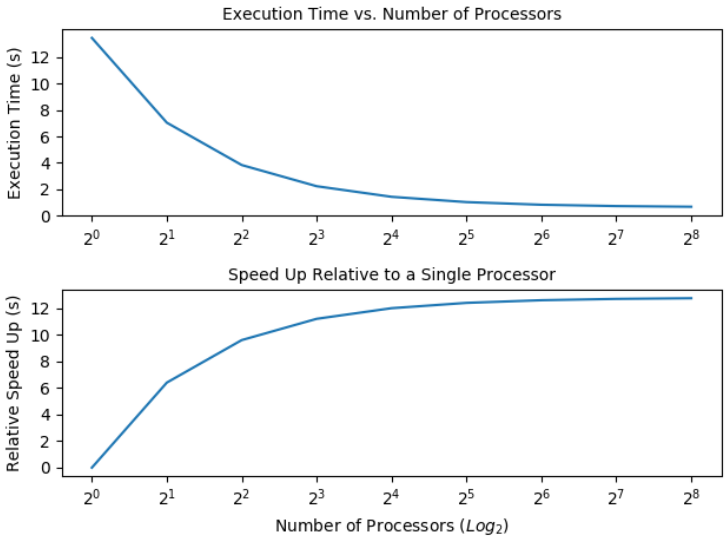
```
from __future__ import annotations
from collections import namedtuple

clockRate = 2e9

ProgramInstructions = namedtuple('ProgramInstructions', ['arithmetic', 'loadStore', 'branch'])
instructionCount = ProgramInstructions(
    arithmetic = 2.56e9,
    loadStore = 1.28e9,
    branch = 256e6)
cyclesPerInstruction = ProgramInstructions(
    arithmetic = 1,
    loadStore = 12,
    branch = 5)

singleProcesssorExecutionTime = (cyclesPerInstruction.arithmetic*instructionCount.arithmetic/0.7 +
          cyclesPerInstruction.loadStore*instructionCount.loadStore/0.7 +
          cyclesPerInstruction.branch*instructionCount.branch)/clockRate

table = [["Processors", "Execution Time (s)", "Speed Up (s)"],
         [1, singleProcesssorExecutionTime, 0.0]]

for i in [2**x for x in range(1, 9)]:
    executionTime = (cyclesPerInstruction.arithmetic*instructionCount.arithmetic/(0.7*i) +
              cyclesPerInstruction.loadStore*instructionCount.loadStore/(0.7*i) +
              cyclesPerInstruction.branch*instructionCount.branch)/clockRate
    speedUp = singleProcesssorExecutionTime - executionTime
    table.append([i, executionTime, round(speedUp, 3)])

print(table)
```

Processors	Execution Time (s)	Speed Up (s)
1	13.44	0.0
2	7.04	6.4
4	3.84	9.6
8	2.24	11.2
16	1.44	12.0
32	1.04	12.4
64	0.84	12.6
128	0.74	12.7
256	0.69	12.75

Plotting the Table Results

```
import matplotlib.pyplot as pyplot
import numpy

table = numpy.asarray(table)
processors = table[1:, 0].astype(numpy.float)
executionTimes = table[1:, 1].astype(numpy.float)
speedUp = table[1:, 2].astype(numpy.float)

figure, axes = pyplot.subplots(2)
axes[0].set_title('Execution Time vs. Number of Processors', fontsize=10)
axes[0].set_ylabel('Execution Time (s)')
axes[0].plot(processors, executionTimes)
axes[0].set_xscale('log', basex=2)
axes[0].yaxis.set_ticks(numpy.arange(0, 14, 2))
```

```
axes[1].set_title('Speed Up Relative to a Single Processor', fontsize=10)
axes[1].set_ylabel('Relative Speed Up (s)')
axes[1].plot(processors, speedUp)
axes[1].set_xlabel(r'Number of Processors ($Log_2$)')
axes[1].set_xscale('log', basex=2)
axes[1].yaxis.set_ticks(numpy.arange(0, 14, 2))
figure.tight_layout()

fileName = f"figure/{homework}_{problem}.png"
figure.savefig(fileName)
return fileName
```



1.14.1

Creating the Table

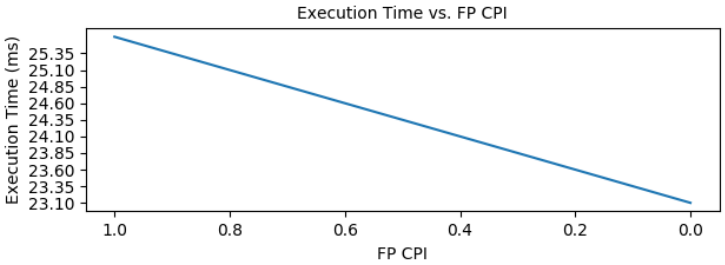
```
from collections import namedtuple
import numpy
table = numpy.asarray([["FP CPI", "Execution Time (s)", "Relative Speedup (s)"]])

Instructions = namedtuple('Instructions', ['FP', 'INT', 'LS', 'branch'])

class InstructionTimer:
    def __init__(self, clockRate):
        self.clockRate = clockRate

    @classmethod
    def fromClockRate(cls, clockRate):
        return InstructionTimer(clockRate)

    def executionTimeFromCpi(self, instructionCount: Instructions, cpi: Instructions) -> float:
        return sum(numpy.multiply(instructionCount, cpi))/self.clockRate

instructions = Instructions(
    FP = 50e6,
    INT = 110e6,
    LS = 80e6,
    branch = 16e6)

cpi = Instructions(
    FP = numpy.round(numpy.arange(1, -0.1, -0.1), 1),
    INT = 1,
```

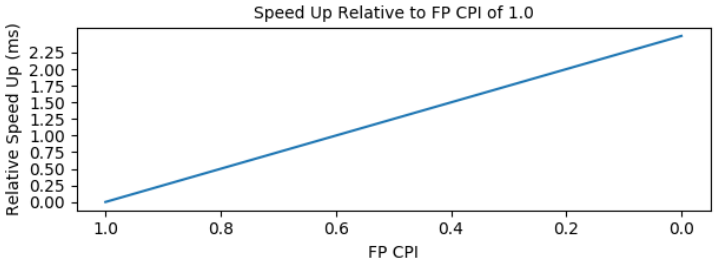
```
    LS = 4,
    branch = 2)

instructionTimer = InstructionTimer.fromClockRate(2e9)

executionTime = numpy.round(instructionTimer.executionTimeFromCpi(instructions, cpi), 4)

staticCpiExecutionTime = instructionTimer.executionTimeFromCpi(instructions, cpi = Instructions(1, 1, 4, 2))
speedUp = numpy.round(staticCpiExecutionTime - executionTime, 4)

values = numpy.transpose(numpy.asarray([
    cpi.FP,
    executionTime,
    speedUp]))

table = numpy.concatenate((table, values))
print(table)
```

FP CPI	Execution Time (s)	Relative Speedup (s)
1.0	0.256	0.0
0.9	0.2535	0.0025
0.8	0.251	0.005
0.7	0.2485	0.0075
0.6	0.246	0.01
0.5	0.2435	0.0125
0.4	0.241	0.015
0.3	0.2385	0.0175
0.2	0.236	0.02
0.1	0.2335	0.0225
0.0	0.231	0.025

Plotting the Table

```
import matplotlib.pyplot as pyplot
import numpy

table = numpy.asarray(table)[1:, :].astype(numpy.float)
cpi = table[:, 0]
executionTime = table[:, 1]*100
speedUp = table[:, 2]*100

figure, axes = pyplot.subplots(2)
axes[0].set_title('Execution Time vs. FP CPI', fontsize=10)
axes[0].set_ylabel('Execution Time (ms)')
axes[0].yaxis.set_ticks(numpy.arange(numpy.min(executionTime), numpy.max(executionTime), .25))
axes[0].plot(cpi, executionTime)

axes[1].set_title('Speed Up Relative to FP CPI of 1.0', fontsize=10)
axes[1].set_ylabel('Relative Speed Up (ms)')
axes[1].plot(cpi, speedUp)
axes[1].yaxis.set_ticks(numpy.arange(numpy.min(speedUp), numpy.max(speedUp), .25))

for i in axes:
    i.set_xlabel('FP CPI')
    i.invert_xaxis()

figure.tight_layout()
figure.savefig(fileName)
return fileName
```





It is not possible to make the program run two times faster by improving the CPI of FP instructions.

This one is not from the textbook

```
from collections import namedtuple
import math

class Circle:
    @classmethod
    def areaFromDiameter(cls, diameter: float) -> float:
        radius = diameter/2
        return math.pi*radius**2

    @classmethod
    def circumferenceFromDiameter(cls, diameter: float) -> float:
        return math.pi*diameter


class WaferResults:
    def __init__(self, dieCount, costPerDie):
        self.dieCount = dieCount
        self.costPerDie = costPerDie


class Wafer:
    dieCountStrategy = None

    def __init__(self, diameter, cost, dieArea, defectRatio):
        self.diameter = diameter
        self.cost = cost
        self.dieArea = dieArea
        self.defectRatio = defectRatio

        self.area = Circle.areaFromDiameter(self.diameter)

    @property
    def goodDieYield(self):
        return 1/(1+(self.defectRatio * self.dieArea/2))**2

    def generateWaferResults(self):
        dieCount = Wafer.dieCountStrategy(self.diameter, self.dieArea)
        costPerDie = self.cost / (dieCount * self.goodDieYield)
        return WaferResults(dieCount, costPerDie)
```

```
def simpleDieCountStrategy(waferDiameter: float, dieArea: float) -> float:
    return Circle.areaFromDiameter(waferDiameter)/dieArea

def accurateDieCountStrategy(waferDiameter: float, dieArea: float) -> float:
    return simpleDieCountStrategy(waferDiameter, dieArea) - Circle.circumferenceFromDiameter(waferDiameter)/ma

wafer = Wafer(
    diameter = 30,
    cost = 400,
    dieArea = 2.07*1.05,
    defectRatio = 0.020)

print(f"(a) Yield = {wafer.goodDieYield:.3}\n")

Wafer.dieCountStrategy = simpleDieCountStrategy
waferResults = wafer.generateWaferResults()
print(("Simple Approximation: \n"
       f"  (b) Dies per Wafer = {round(waferResults.dieCount)}\n"
       f"  (c) Cost per Die = {waferResults.costPerDie:.3}\n"))

Wafer.dieCountStrategy = accurateDieCountStrategy
waferResults = wafer.generateWaferResults()
print(("More Precise Approximation: \n"
       f"  (b) Dies per Wafer = {round(waferResults.dieCount)}\n"
       f"  (c) Cost per Die = {waferResults.costPerDie:.3}"))
```

```
(a) Yield = 0.958

Simple Approximation:
  (b) Dies per Wafer = 325
  (c) Cost per Die = 1.28

More Precise Approximation:
  (b) Dies per Wafer = 280
  (c) Cost per Die = 1.49
```