Problem Set 1

Both theory and programming questions are due **Thursday, September 15** at **11:59PM**. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

We will provide the solutions to the problem set 10 hours after the problem set is due, which you will use to find any errors in the proof that you submitted. You will need to submit a critique of your solutions by **Tuesday, September 20th, 11:59PM**. Your grade will be based on both your solutions and your critique of the solutions.

Problem 1-1. [15 points] **Asymptotic Practice**

For each group of functions, sort the functions in increasing order of asymptotic (big-O) complexity:

(a) [5 points] **Group 1:**

$$\begin{aligned} f_1(n) &= n^{0.999999}\log n \\ f_2(n) &= 10000000n \\ f_3(n) &= 1.000001^n \\ f_4(n) &= n^2 \end{aligned}$$

(b) [5 points] **Group 2:**

$$\begin{aligned} f_1(n) &= 2^{2^{1000000}} \\ f_2(n) &= 2^{100000n} \\ f_3(n) &= \binom{n}{2} \\ f_4(n) &= n\sqrt{n} \end{aligned}$$

(c) [5 points] **Group 3:**

$$\begin{aligned} f_1(n) &= n^{\sqrt{n}} \\ f_2(n) &= 2^n \\ f_3(n) &= n^{10}\cdot 2^{n/2} \\ f_4(n) &= \sum_{i=1}^{n}(i+1) \end{aligned}$$

Problem 1-2. [15 points] **Recurrence Relation Resolution**

For each of the following recurrence relations, pick the correct asymptotic runtime:

(a) [5 points] Select the correct asymptotic complexity of an algorithm with runtime $T(n,n)$ where

$$\begin{aligned} T(x,c) &= \Theta(x) && \text{for } c \leq 2, \\ T(c,y) &= \Theta(y) && \text{for } c \leq 2, \text{ and} \\ T(x,y) &= \Theta(x+y) + T(x/2, y/2). \end{aligned}$$

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

(b) [5 points] Select the correct asymptotic complexity of an algorithm with runtime $T(n,n)$ where

$$\begin{aligned} T(x,c) &= \Theta(x) && \text{for } c \leq 2, \\ T(c,y) &= \Theta(y) && \text{for } c \leq 2, \text{ and} \\ T(x,y) &= \Theta(x) + T(x, y/2). \end{aligned}$$

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

(c) [5 points] Select the correct asymptotic complexity of an algorithm with runtime $T(n,n)$ where

$$\begin{aligned} T(x,c) &= \Theta(x) && \text{for } c \leq 2, \\ T(x,y) &= \Theta(x) + S(x, y/2), \\ S(c,y) &= \Theta(y) && \text{for } c \leq 2, \text{ and} \\ S(x,y) &= \Theta(y) + T(x/2, y). \end{aligned}$$

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Peak-Finding

In Lecture 1, you saw the peak-finding problem. As a reminder, a *peak* in a matrix is a location with the property that its four neighbors (north, south, east, and west) have value less than or equal to the value of the peak. We have posted Python code for solving this problem to the website in a file called `ps1.zip`. In the file `algorithms.py`, there are four different algorithms which have been written to solve the peak-finding problem, only some of which are correct. Your goal is to figure out which of these algorithms are correct and which are efficient.

Problem 1-3. [16 points] **Peak-Finding Correctness**

(a) [4 points] Is `algorithm1` correct?

1. Yes.
2. No.

(b) [4 points] Is `algorithm2` correct?

1. Yes.
2. No.

(c) [4 points] Is `algorithm3` correct?

1. Yes.
2. No.

(d) [4 points] Is `algorithm4` correct?

1. Yes.
2. No.

Problem 1-4. [16 points] **Peak-Finding Efficiency**

(a) [4 points] What is the worst-case runtime of `algorithm1` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

(b) [4 points] What is the worst-case runtime of `algorithm2` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.

3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

(c) [4 points] What is the worst-case runtime of `algorithm3` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

(d) [4 points] What is the worst-case runtime of `algorithm4` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Problem 1-5. [19 points] **Peak-Finding Proof**

Please modify the proof below to construct a proof of correctness for the *most efficient correct algorithm* among `algorithm2`, `algorithm3`, and `algorithm4`.

The following is the proof of correctness for `algorithm1`, which was sketched in Lecture 1.

> We wish to show that `algorithm1` will always return a peak, as long as the problem is not empty. To that end, we wish to prove the following two statements:
>
> **1. If the peak problem is not empty, then `algorithm1` will always return a location.** Say that we start with a problem of size $m \times n$. The recursive subproblem examined by `algorithm1` will have dimensions $m \times \lfloor n/2 \rfloor$ or $m \times (n - \lfloor n/2 \rfloor - 1)$. Therefore, the number of columns in the problem strictly decreases with each recursive call as long as $n > 0$. So `algorithm1` either returns a location at some point, or eventually examines a subproblem with a non-positive number of columns. The only way for the number of columns to become strictly negative, according to the formulas that determine the size of the subproblem, is to have $n = 0$ at some point. So if `algorithm1` doesn't return a location, it must eventually examine an empty subproblem.
>
> We wish to show that there is no way that this can occur. Assume, to the contrary, that `algorithm1` does examine an empty subproblem. Just prior to this, it must examine

a subproblem of size $m \times 1$ or $m \times 2$. If the problem is of size $m \times 1$, then calculating the maximum of the central column is equivalent to calculating the maximum of the entire problem. Hence, the maximum that the algorithm finds must be a peak, and it will halt and return the location. If the problem has dimensions $m \times 2$, then there are two possibilities: either the maximum of the central column is a peak (in which case the algorithm will halt and return the location), or it has a strictly better neighbor in the other column (in which case the algorithm will recurse on the non-empty subproblem with dimensions $m \times 1$, thus reducing to the previous case). So `algorithm1` can never recurse into an empty subproblem, and therefore `algorithm1` must eventually return a location.

2. If `algorithm1` returns a location, it will be a peak in the original problem. If `algorithm1` returns a location (r_1, c_1), then that location must have the best value in column c_1, and must have been a peak within some recursive subproblem. Assume, for the sake of contradiction, that (r_1, c_1) is not also a peak within the original problem. Then as the location (r_1, c_1) is passed up the chain of recursive calls, it must eventually reach a level where it stops being a peak. At that level, the location (r_1, c_1) must be adjacent to the dividing column c_2 (where $|c_1 - c_2| = 1$), and the values must satisfy the inequality $val(r_1, c_1) < val(r_1, c_2)$.

Let (r_2, c_2) be the location of the maximum value found by `algorithm1` in the dividing column. As a result, it must be that $val(r_1, c_2) \leq val(r_2, c_2)$. Because the algorithm chose to recurse on the half containing (r_1, c_1), we know that $val(r_2, c_2) < val(r_2, c_1)$. Hence, we have the following chain of inequalities:

$$val(r_1, c_1) < val(r_1, c_2) \leq val(r_2, c_2) < val(r_2, c_1)$$

But in order for `algorithm1` to return (r_1, c_1) as a peak, the value at (r_1, c_1) must have been the greatest in its column, making $val(r_1, c_1) \geq val(r_2, c_1)$. Hence, we have a contradiction.

Problem 1-6. [19 points] **Peak-Finding Counterexamples**

For each incorrect algorithm, upload a Python file giving a counterexample (i.e. a matrix for which the algorithm returns a location that is not a peak).

6.006 Introduction to Algorithms
Fall 2011

Problem Set 2

Both theory and programming questions are due **Tuesday, September 27** at **11:59PM**. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions.

Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

We will provide the solutions to the problem set 10 hours after the problem set is due, which you will use to find any errors in the proof that you submitted. You will need to submit a critique of your solutions by **Thursday, September 29th, 11:59PM**. Your grade will be based on both your solutions and your critique of the solutions.

Problem 2-1. [40 points] **Fractal Rendering**

You landed a consulting gig with Gopple, who is about to introduce a new line of mobile phones with Retina HD displays, which are based on unicorn e-ink and thus have infinite resolution. The high-level executives heard that fractals have infinite levels of detail, and decreed that the new phones' background will be the ***Koch snowflake*** (Figure 1).

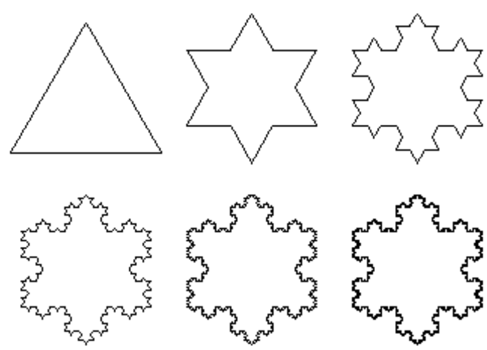

Figure 1: The Koch snowflake fractal, rendered at Level of Detail (LoD) 0 through 5.

Unfortunately, the phone's processor (CPU) and the graphics chip (GPU) powering the display do not have infinite processing power, so the Koch fractal cannot be rendered in infinite detail. Gopple engineers will stop the recursion at a fixed depth n in order to cap the processing requirement. For example, at $n = 0$, the fractal is just a triangle. Because higher depths result in more detailed drawing, this depth is usually called the ***Level of Detail (LoD)***.

The Koch snowflake at LoD n can be drawn using an algorithm following the sketch below:

```
SNOWFLAKE(n)
e1, e2, e3 = edges of an equilateral triangle with side length 1
SNOWFLAKE-EDGE(e1, n)
SNOWFLAKE-EDGE(e2, n)
SNOWFLAKE-EDGE(e3, n)
```

```
SNOWFLAKE-EDGE(edge, n)
if n == 0
    edge is an edge on the snowflake
else
    e1, e2, e3 = split edge in 3 equal parts
    SNOWFLAKE-EDGE(e1, n − 1)
    f2, g2 = edges of an equilateral triangle whose 3rd edge is e2, pointing outside the snowflake
    Δ(f2, g2, e2) is a triangle on the snowflake's surface
    SNOWFLAKE-EDGE(f2, n − 1)
    SNOWFLAKE-EDGE(g2, n − 1)
    SNOWFLAKE-EDGE(e3, n − 1)
```

The sketch above should be sufficient for solving this problem. If you are curious about the missing details, you may download and unpack the problem set's `.zip` archive, and read the CoffeeScript implementation in `fractal/src/fractal.coffee`.

In this problem, you will explore the computational requirements of four different methods for rendering the fractal, as a function of the LoD n. For the purpose of the analysis, consider the recursive calls to SNOWFLAKE-EDGE; do not count the main call to SNOWFLAKE as part of the recursion tree. (You can think of it as a super-root node at a special level -1, but it behaves differently from all other levels, so we do not include it in the tree.) Thus, the recursion tree is actually a forest of trees, though we still refer to the entire forest as the "recursion tree". The root calls to SNOWFLAKE-EDGE are all at level 0.

Gopple's engineers have prepared a prototype of the Koch fractal drawing software, which you can use to gain a better understanding of the problem. To use the prototype, download and unpack the problem set's `.zip` archive, and use Google Chrome to open `fractal/bin/fractal.html`.

First, in 3D hardware-accelerated rendering (e.g., iPhone), surfaces are broken down into triangles (Figure 2). The CPU compiles a list of coordinates for the triangles' vertices, and the GPU is responsible for producing the final image. So, from the CPU's perspective, rendering a triangle costs the same, no matter what its surface area is, and the time for rendering the snowflake fractal is proportional to the number of triangles in its decomposition.

(a) [1 point] What is the height of the recursion tree for rendering a snowflake of LoD n?

1. $\log n$
2. n

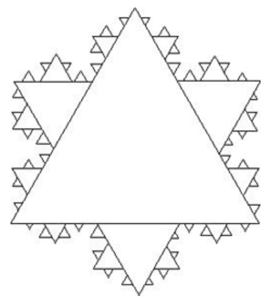

Figure 2: Koch snowflake drawn with triangles.

3. $3\,n$
4. $4\,n$

(b) [2 points] How many nodes are there in the recursion tree at level i, for $0 \leq i \leq n$?

1. 3^i
2. 4^i
3. 4^{i+1}
4. $3 \cdot 4^i$

(c) [1 point] What is the asymptotic rendering time (triangle count) for a node in the recursion tree at level i, for $0 \leq i < n$?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^i)$
4. $\Theta(\frac{1}{3}^i)$

(d) [1 point] What is the asymptotic rendering time (triangle count) at each level i of the recursion tree, for $0 \leq i < n$?

1. 0
2. $\Theta(\frac{4}{9}^i)$
3. $\Theta(3^i)$
4. $\Theta(4^i)$

(e) [2 points] What is the total asymptotic cost for the CPU, when rendering a snowflake with LoD n using 3D hardware-accelerated rendering?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

Second, when using 2D hardware-accelerated rendering, the surfaces' outlines are broken down into open or closed paths (list of connected line segments). For example, our snowflake is one closed path composed of straight lines. The CPU compiles the list of cooordinates in each path to be drawn, and sends it to the GPU, which renders the final image. This approach is also used for talking to high-end toys such as laser cutters and plotters.

(f) [1 point] What is the height of the recursion tree for rendering a snowflake of LoD n using 2D hardware-accelerated rendering?

1. $\log n$
2. n
3. $3\,n$
4. $4\,n$

(g) [1 point] How many nodes are there in the recursion tree at level i, for $0 \leq i \leq n$?

1. 3^i
2. 4^i
3. 4^{i+1}
4. $3 \cdot 4^i$

(h) [1 point] What is the asymptotic rendering time (line segment count) for a node in the recursion tree at level i, for $0 \leq i < n$?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^i)$
4. $\Theta(\frac{1}{3}^i)$

(i) [1 point] What is the asymptotic rendering time (line segment count) for a node in the last level n of the recursion tree?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^n)$
4. $\Theta(\frac{1}{3}^n)$

(j) [1 point] What is the asymptotic rendering time (line segment count) at each level i of the recursion tree, for $0 \leq i < n$?

1. 0
2. $\Theta(\frac{4}{9}^i)$
3. $\Theta(3^i)$
4. $\Theta(4^i)$

(k) [1 point] What is the asymptotic rendering time (line segment count) at the last level n in the recursion tree?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

(l) [1 point] What is the total asymptotic cost for the CPU, when rendering a snowflake with LoD n using 2D hardware-accelerated rendering?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

Third, in 2D rendering without a hardware accelerator (also called software rendering), the CPU compiles a list of line segments for each path like in the previous part, but then it is also responsible for "rasterizing" each line segment. Rasterizing takes the coordinates of the segment's endpoints and computes the coordinates of all the pixels that lie on the line segment. Changing the colors of these pixels effectively draws the line segment on the display. We know an algorithm to rasterize a line segment in time proportional to the length of the segment. It is easy to see that this algorithm is optimal, because the number of pixels on the segment is proportional to the segment's length. Throughout this problem, assume that all line segments have length at least one pixel, so that the cost of rasterizing is greater than the cost of compiling the line segments.

It might be interesting to note that the cost of 2D software rendering is proportional to the total length of the path, which is also the power required to cut the path with a laser cutter, or the amount of ink needed to print the path on paper.

(m) [1 point] What is the height of the recursion tree for rendering a snowflake of LoD n?

1. $\log n$
2. n
3. $3\,n$
4. $4\,n$

(n) [1 point] How many nodes are there in the recursion tree at level i, for $0 \leq i \leq n$?

1. 3^i
2. 4^i
3. 4^{i+1}
4. $3 \cdot 4^i$

(o) [1 point] What is the asymptotic rendering time (line segment length) for a node in the recursion tree at level i, for $0 \leq i < n$? Assume that the sides of the initial triangle have length 1.

1. 0

2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^i)$
4. $\Theta(\frac{1}{3}^i)$

(p) [1 point] What is the asymptotic rendering time (line segment length) for a node in the last level n of the recursion tree?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^n)$
4. $\Theta(\frac{1}{3}^n)$

(q) [1 point] What is the asymptotic rendering time (line segment length) at each level i of the recursion tree, for $0 \leq i < n$?

1. 0
2. $\Theta(\frac{4}{9}^i)$
3. $\Theta(3^i)$
4. $\Theta(4^i)$

(r) [1 point] What is the asymptotic rendering time (line segment length) at the last level n in the recursion tree?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

(s) [1 point] What is the total asymptotic cost for the CPU, when rendering a snowflake with LoD n using 2D software (not hardware-accelerated) rendering?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

The fourth and last case we consider is 3D rendering without hardware acceleration. In this case, the CPU compiles a list of triangles, and then rasterizes each triangle. We know an algorithm to rasterize a triangle that runs in time proportional to the triangle's surface area. This algorithm is optimal, because the number of pixels inside a triangle is proportional to the triangle's area. For the purpose of this problem, you can assume that the area of a triangle with side length l is $\Theta(l^2)$. We also assume that the cost of rasterizing is greater than the cost of compiling the line segments.

(t) [4 points] What is the total asymptotic cost of rendering a snowflake with LoD n? Assume that initial triangle's side length is 1.

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

(u) [15 points] Write a succinct proof for your answer using the recursion-tree method.

Problem 2-2. [60 points] **Digital Circuit Simulation**

Your 6.006 skills landed you a nice internship at the chip manufacturer AMDtel. Their hardware verification team has been complaining that their circuit simulator is slow, and your manager decided that your algorithmic chops make you the perfect candidate for optimizing the simulator.

A ***combinational circuit*** is made up of ***gates***, which are devices that take Boolean (True / 1 and False / 0) input signals, and output a signal that is a function of the input signals. Gates take some time to compute their functions, so a gate's output at time τ reflects the gate's inputs at time $\tau - \delta$, where δ is the gate's delay. For the purposes of this simulator, a gate's output transitions between 0 and 1 instantly. Gates' output terminals are connected to other gates' inputs terminals by ***wires*** that propagate the signal instantly without altering it.

For example, a 2-input XOR gate with inputs A and B (Figure 3) with a 2 nanosecond (ns) delay works as follows:

Time (ns)	Input A	Input B	Output O	Explanation
0	0	0		Reflects inputs at time -2
1	0	1		Reflects inputs at time -1
2	1	0	0	0 XOR 0, given at time 0
3	1	1	1	0 XOR 1, given at time 1
4			1	1 XOR 0, given at time 2
5			0	1 XOR 1, given at time 3

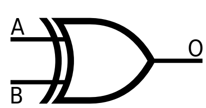

Figure 3: 2-input XOR gate; A and B supply the inputs, and O receives the output.

The circuit simulator takes an input file that describes a circuit layout, including gates' delays, probes (indicating the gates that we want to monitor the output), and external inputs. It then simulates the transitions at the output terminals of all the gates as time progresses. It also outputs transitions at the probed gates in the order of the timing of those transitions.

This problem will walk you through the best known approach for fixing performance issues in a system. You will profile the code, find the performance bottleneck, understand the reason behind it, and remove the bottleneck by optimizing the code.

To start working with AMDtel's circuit simulation source code, download and unpack the problem set's `.zip` archive, and go to the `circuit/` directory.

The circuit simulator is in `circuit.py`. The AMDtel engineers pointed out that the simulation input in `tests/5devadas13.in` takes too long to run. We have also provided an automated test suite at `test-circuit.py`, together with other simulation inputs. You can ignore these files until you get to the last part of the problem set.

(a) [8 points] Run the code under the python profiler with the command below, and identify the method that takes up most of the CPU time. If two methods have similar CPU usage times, ignore the simpler one.

```
python -m cProfile -s time circuit.py < tests/5devadas13.in
```

Warning: the command above can take 15-30 minutes to complete, and bring the CPU usage to 100% on one of your cores. Plan accordingly.

What is the name of the method with the highest CPU usage?

(b) [6 points] How many times is the method called?

(c) [8 points] The class containing the troublesome method is implementing a familiar data structure. What is the tightest asymptotic bound for the worst-case running time of the method that contains the bottleneck? Express your answer in terms of n, the number of elements in the data structure.

1. $O(1)$.
2. $O(\log n)$.
3. $O(n)$.
4. $O(n \log n)$.
5. $O(n \log^2 n)$.
6. $O(n^2)$.

(d) [8 points] If the data structure were implemented using the most efficient method we learned in class, what would be the tightest asymptotic bound for the worst-case running time of the method discussed in the questions above?

1. $O(1)$.
2. $O(\log n)$.
3. $O(n)$.
4. $O(n \log n)$.
5. $O(n \log^2 n)$.
6. $O(n^2)$.

(e) [30 points] Rewrite the data structure class using the most efficient method we learned in class. **Please note that you are not allowed to import any additional Python libraries and our test will check this.**

We have provided a few tests to help you check your code's correctness and speed. The test cases are in the `tests/` directory. `tests/README.txt` explains the syntax of the simulator input files. You can use the following command to run all the tests.

```
python circuit_test.py
```

To work on a single test case, run the simulator on the test case with the following command.

```
python circuit.py < tests/1gate.in > out
```

Then compare your output with the correct output for the test case.

```
diff out tests/1gate.gold
```

For Windows, use `fc` to compare files.

```
fc out tests/1gate.gold
```

We have implemented a visualizer for your output, to help you debug your code. To use the visualizer, first produce a simulation trace.

```
TRACE=jsonp python circuit.py < tests/1gate.in > circuit.jsonp
```

On Windows, use the following command instead.

```
circuit_jsonp.bat < tests/1gate.in > circuit.jsonp
```

Then use Google Chrome to open `visualizer/bin/visualizer.html`

We recommend using the small test cases numbered 1 through 4 to check your implementation's correctness, and then use test case 5 to check the code's speed.

When your code passes all tests, and runs reasonably fast (the tests should complete in less than 30 seconds on any reasonably recent computer), upload your modified `circuit.py` to the course submission site.