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:**

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

$f_3 > f_4 > f_2 > f_1$

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

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

$f_2 > f_3 > f_4 > f_1.$

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

$n^{\sqrt{n}}$ $\ln n.$

$\frac{n}{2}\cdot$

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

$\frac{\sqrt{n}\cdot \ln^{n}}{n\cdot \ln 2}$ $\frac{\ln n.}{\sqrt{n}.}$

$f_2 > f_1$ $n^{\sqrt{}}$

$f(2) > f(3) > f(1) > f(4).$

$\frac{2^{\frac{n}{2}}}{n^{10}.}$ $f_2 > f_3.$

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. $\Theta(2^n)$.

$T(n,n) = T(\frac{n}{2}, \frac{n}{2}) + \overset{\theta(n)}{\cancel{\theta(10\ 2n}})$

↙

$T(\frac{n}{4}, \frac{n}{4}) + \theta(\frac{n}{2})\ \cancel{\theta}$

$\vdots\ \frac{n}{4}$.

(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. $\Theta(2^n)$.

$T(n,n) = \theta(n) + T(n, \frac{n}{2})$

$n \cdot \log_2 n. + n.$

(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. $\Theta(2^n)$.

$T(n,n) = \theta(n) + S(n, \frac{n}{2})$

$S(n, \frac{n}{2}) = \theta(\frac{n}{2}) + T(\frac{n}{2}, \frac{n}{2}).$

$\therefore\ T.\ \theta(n) + \theta(\frac{n}{2}) + \theta(\frac{n}{2}) + \theta(\frac{n}{4}) + \theta(\frac{n}{4})$

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. $\Theta(2^n)$.

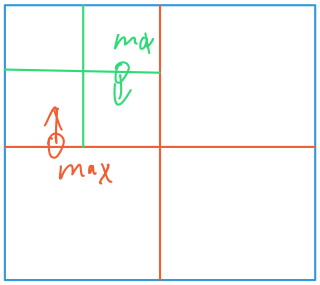
(b) [4 points] What is the worst-case runtime of `algorithm2` on a problem of size $n \times n$?

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

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

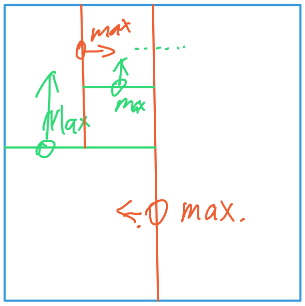
(c) [4 points] What is the worst-case runtime of `algorithm3` on a problem of size $n \times n$?

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).

Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Erik Demaine and Srini Devadas

September 15, 2011
Problem Set 2

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).

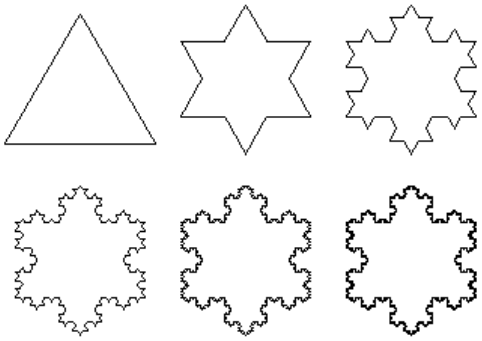


Figure 1: The Koch snowflake fractal, rendered at Level of Detail (LoD) 0 through 5.

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($n$)
$e_1, e_2, e_3$ = edges of an equilateral triangle with side length $1$
Snowflake-Edge($e_1, n$)
Snowflake-Edge($e_2, n$)
Snowflake-Edge($e_3, n$)
```

```
Snowflake-Edge($edge, n$)
if $n == 0$
    $edge$ is an edge on the snowflake
else
    $e_1, e_2, e_3$ = split $edge$ in 3 equal parts
    Snowflake-Edge($e_1, n - 1$)
    $f_2, g_2$ = edges of an equilateral triangle whose 3rd edge is $e_2$, pointing outside the snowflake
    $\Delta(f_2, g_2, e_2)$ is a triangle on the snowflake's surface
    Snowflake-Edge($f_2, n - 1$)
    Snowflake-Edge($g_2, n - 1$)
    Snowflake-Edge($e_3, n - 1$)
```

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.

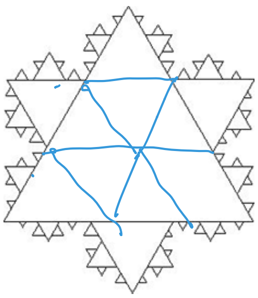
Gopple's engineers have prepared a prototype of the Koch fractal drawing software, which you can use to gain a better understanding of the problem. To use the prototype, download and unpack the problem set's `.zip` archive, and use Google Chrome to open `fractal/bin/fractal.html`.

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]



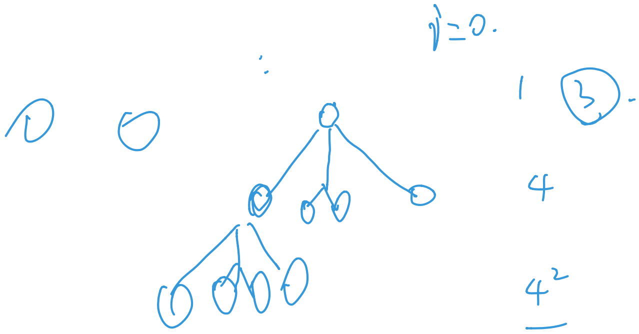


Figure 2: Koch snowflake drawn with triangles.

3. $3n$
4. $4n$

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

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 $0 \le i < n$?

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

i=1 i=2.

n=1, triangl $\boxed{3}$ + 12. 48.

3×4 3×4^2.

每个node ⇒ 会变为 4个Node: ⇒ 4个Node 画一个 △ ⇒ $\frac{\theta(1)}{4}$

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

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

b × c .

(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. [✓ $\Theta(4^n)$]

$$T_n = \begin{cases} \theta(1), & \text{if } (n=0). \\ 4 \cdot T_n(n-1) + \theta(1) \end{cases}$$

$$a^n = 4 \cdot a^{n-1} + c.$$

$a = 4$, 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. $3n$
4. $4n$

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

因为之前不用rendering line,
在最后一步render就可以了。

(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. [✓ $\Theta(4^i)$]

同上, last level render.

(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. [✓ 4. $\Theta(4^n)$]

$\underline{3 \cdot \theta(4^n)} \times \underline{\theta(1)}$

node 数 每个node画一条线。

(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. [✓ 4. $\Theta(4^n)$]

$j + h = I.$

0

0 0

- . ..

$3 \cdot \theta(4^n). = \theta(4^n).$

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. [✓ 2. n]
3. $3\,n$
4. $4\,n$

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

1. 3^i
2. 4^i
3. 4^{i+1}
4. [✓ 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 $0 \le i < n$? Assume that the sides of the initial triangle have length 1.

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. [✓ $\Theta(\frac{1}{3}^n)$] $\Longrightarrow$

$n-1 \to n$. length of node.

$1 \to 1/3$.

(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?

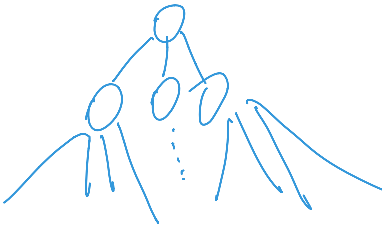
1. $\Theta(1)$
2. $\Theta(n)$
3. [✓ $\Theta(\frac{4}{3}^n)$]
4. $\Theta(4^n)$

(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.



Θ (1)

$(3 \cdot 4^i)$ ~~$\Theta($~~

$(\frac{1}{3})^2$

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

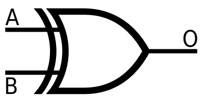


Figure 3: 2-input XOR gate; A and B supply the inputs, and O receives the output.

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? find_min.

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

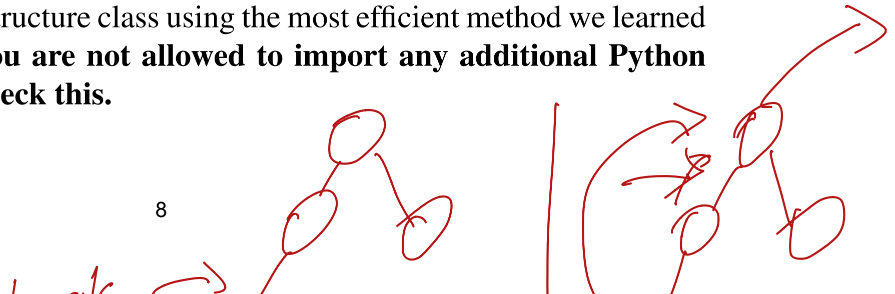
(c) [8 points] The class containing the troublesome method is implementing a familiar data structure. What is the tightest asymptotic bound for the worst-case running time of the method that contains the bottleneck? Express your answer in terms of n, the number of elements in the data structure.

1. $O(1)$.
2. $O(\log n)$.
3. [✓ $O(n)$.] 遍历 array
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?

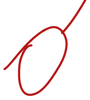
1. [✓ $O(1)$.] min-heap.
2. $O(\log n)$.
3. $O(n)$.
4. $O(n \log n)$.
5. $O(n \log^2 n)$.
6. $O(n^2)$.

(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.**



check.

">" "≤"



pop. ~~two~~ minimum

swap ~~th~~, then check.

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.

Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — October 24, 2011
Professors Erik Demaine and Srini Devadas — Problem Set 5

Problem Set 5

Both theory and programming questions are due **Monday, October 31** 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.

We will provide the solutions to the problem set 10 hours after the problem set is due. You will have to read the solutions, and write a brief **grading explanation** to help your grader understand your write-up. You will need to submit the grading explanation by **Thursday, November 3rd, 11:59PM**. Your grade will be based on both your solutions and the grading explanation.

Problem 5-1. [40 points] **The Knight's Shield**

The optimized circuit verifier that you developed on your Amdtel internship was a huge success and got you on a sure track to landing a sweet offer. You also got transferred to a research group that is working on the *Knight's Shield (KS)*[1], a high-stakes project to develop a massive multi-core chip aimed at the exploding secure cloud computing market.

The KS chip packs 16,384 cores in a die that's the same size as a regular CPU die. However, each core is very small, and can only do arithmetic operations using 8-bit or 16-bit unsigned integers (see Table 1). Encryption algorithms typically use 2,048-bit integers, so the KS chip will ship with software that supports arithmetic on large integers. Your job is to help the KS team assess the efficiency of their software.

Operation	R1 size	R2 size	Result size	Result
`ZERO`			8 / 16	0 (zero)
`ONE`			8 / 16	1 (one)
`LSB R1`	16		8	R1 % 256 (least significant byte)
`MSB R1`	16		8	R1 / 256 (most significant byte)
`WORD R1`	8		16	R1 (expanded to 16-bits)
`ADD R1, R2`	8 / 16	8 / 16	16	`R1` + `R2`
`SUB R1, R2`	8 / 16	8 / 16	16	`R1` − `R2` mod 65536
`MUL R1, R2`	8	8	16	`R1` · `R2`
`DIV R1, R2`	16	8	8	`R1` ÷ `R2` mod 256
`MOD R1, R2`	16	8	8	`R1` % `R2`
`AND R1, R2`	8 / 16	8 / 16	8 / 16	`R1` & `R2` (bitwise AND)
`OR R1, R2`	8 / 16	8 / 16	8 / 16	`R1` ∥ `R2` (bitwise OR)
`XOR R1, R2`	8 / 16	8 / 16	8 / 16	`R1` ˆ `R2` (bitwise XOR)

Table 1: Arithmetic operations supported by the KS chip. All sizes are in bits.

[1] The code name is Amdtel confidential information. Please refrain from leaking to TechCrunch.

The KS library supports arbitrarily large base-256 numbers. The base was chosen such that each digit is a byte, and two digits make up a 16-bit number. Numbers are stored as a little-endian sequence of bytes (the first byte of a number is the least significant digit, for example 65534 = 0xFFFE would be stored as [0xFE, 0xFF]). For the rest of the problem, assume all the input numbers have N digits.

Consider the following algorithm for computing $A + B$, assuming both inputs have N digits.

```
ADD($A, B, N$)
$C = \text{ZERO}(N+1)$ // ZERO($k$) creates a $k$-digit number, with all digits set to 0s.
$carry = 0$
for $i = 1$ to $N$
    $digit = \text{WORD}(A[i]) + \text{WORD}(B[i]) + \text{WORD}(carry)$
    $C[i] = \text{LSB}(digit)$
    $carry = \text{MSB}(digit)$
$C[N+1] = carry$
return $C$
```

(a) [1 point] What is the running time of ADD?

1. $\Theta(1)$
2. $\Theta(\log N)$
[✓ 3. $\Theta(N)$]
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(b) [1 point] What is the size of ADD's output?

1. $\Theta(1)$
2. $\Theta(\log N)$
[✓ 3. $\Theta(N)$]
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(c) [1 point] ADD's output size suggests an easy lower bound for the subroutine. Does the running time of ADD match this lower bound?

[✓ 1. Yes]

2. No

Consider the following brute-force algorithm for computing $A \cdot B$, assuming both inputs have N digits.

$a_1 \quad a_2 \quad a_3$

$b_1 \quad b_2 \quad b_3.$

```
MULTIPLY$(A, B, N)$
$C = \text{ZERO}(2N)$
for $i = 1$ to $N$        $n.$
    $carry = 0$
    for $j = 1$ to $N$        $n.$
        $digit = \underline{A[i] \cdot B[j] + \text{WORD}(C[i + j - 1]) + \text{WORD}(carry)}$
        $C[i + j - 1] = \text{LSB}(digit)$
        $carry = \text{MSB}(digit)$
    $C[i + N] = carry$
return $C$
```

(d) [1 point] What is the running time of MULTIPLY?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. [✓ $\Theta(N^2)$]
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(e) [1 point] What is the size of MULTIPLY's output?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. [✓ $\Theta(N)$]
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(f) [1 point] MULTIPLY's output size suggests an easy lower bound for the subroutine. Does the running time of MULTIPLY match this lower bound?

1. Yes
2. [✓ No]

Consider the following brute-force algorithm for computing $A \div B$ and $A \mod B$, assuming both inputs have N digits. The algorithm uses a procedure COPY(A, N) that creates a copy of an N-digit number A, using $\Theta(N)$ time.

```
DIVMOD$(A, B, N)$
$Q = $ ZERO$(N)$ // quotient
$R = $ COPY$(A, N)$ // remainder
$S_0 = $ COPY$(B, N)$ // $S_i = B \cdot 2^i$
$i = 0$
repeat
    $i = i + 1$
    $S_i = $ ADD$(S_{i-1}, S_{i-1}, N)$
until $S_i[N + 1] > 0$ or CMP$(S_i, A, N)$ == GREATER
for $j = i - 1$ downto 0
    $Q = $ ADD$(Q, Q, N)$
    if CMP$(R, S_j, N)$ != SMALLER
        $R = $ SUBTRACT$(R, S_j, N)$
        $Q[0] = Q[0] || 1$ // Faster version of $Q = Q + 1$
return $(Q, R)$
```

64 . 5 , 10 20 40

n· lg N

lg n.

~~log~~ N. n

n.

Θ(N), n.

1 , 24. R=24.

Q=0 Q[0]=1

Q=2, 6, 12. R=4.

(g) [1 point] CMP(A, B, N) returns GREATER if $A > B$, EQUAL if $A = B$, and SMALLER if $A < B$, assuming both A and B are N-digit numbers. What is the running time for an optimal CMP implementation?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. [✓ $\Theta(N)$]
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(h) [1 point] SUBTRACT(A, B, N) computes $A - B$, assuming A and B are N-digit numbers. What is the running time for an optimal SUBTRACT implementation?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. [✓ $\Theta(N)$]
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$

7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(i) [1 point] What is the running time of DIVMOD?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. [✓ $\Theta(N^2)$]
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

N bit, the most number.

$2^n - 1$.

$\log_2(2^n-1) \approx n$.

so loop is n. not log n.

The KS library does not use the DIVMOD implementation above. Instead, it uses Newton's method to implement DIV(A, B, N) which computes the division quotient $A \div B$, assuming both inputs have N digits. DIV relies on the subroutines defined above. For example, it uses MULTIPLY to perform large-number multiplication and ADD for large-number addition. MOD(A, B, N) is implemented using the identity $A \mod B = A - (A \div B) \cdot B$.

(j) [2 points] How many times does DIV call MULTIPLY?

1. $\Theta(1)$
2. [✓ $\Theta(\log N)$]
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(k) [2 points] What is the running time of MOD?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. [✓ $\Theta(N^2)$]
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Consider the following brute-force algorithm for computing $B^E \mod M$, assuming all the input numbers have N digits.

```
PowMod$(B, E, M, N)$
$R = \textsc{One}(N)$ // result
$X = \textsc{Copy}(B, N)$ // multiplier
for $i = 1$ to $N$
    $mask = 1$
    for $bit = 1$ to $8$
        if $E[i]$ & $mask$ != 0
            $R = \textsc{Mod}(\overset{N^2}{\underline{\textsc{Multiply}(R, X, N)}}, M, 2N)$      $N^2$
        $X = \textsc{Mod}(\textsc{Multiply}(X, X, N), M, 2N)$
        $mask = \textsc{Lsb}(mask \cdot 2)$
return $R$
```

(l) [2 points] What is the running time for PowMod?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. [✓ $\Theta(N^3)$]

Assume the KS library swaps out the brute-force Multiply with an implementation of Karatsuba's algorithm.

(m) [1 point] What will the running time for Multiply be after the optimization?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. [✓ $\Theta(N^{\log_2 3})$]
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(n) [2 points] What will the running time for Mod be after the optimization?

1. $\Theta(1)$

2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
[✓ 6. $\Theta(N^{\log_2 3})$]
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

(o) [2 points] What will the running time for POWMOD be after the optimization?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
[✓ 7. $\Theta(N^{\log_2 6})$]
8. $\Theta(N^3)$

(p) [20 points] Write pseudo-code for KTHROOT(A, K, N), which computes $\lfloor \sqrt[K]{A} \rfloor$ using binary search, assuming that A and K are both N-digit numbers. The running time for KTHROOT(A, K, N) should be $\Theta(N^{2+\log_2 3})$.

Problem 5-2. [18 points] **RSA Public-Key Encryption**

The RSA (Rivest-Shamir-Adelman) public-key cryptosystem is a cornerstone of Internet security. It provides the "S" (security) in the HTTPS sessions used for e-commerce and cloud services that handle private information, such as e-mail. RSA secures SSH sessions (used to connect to Athena,* for example), and MIT certificates used to log into Stellar. You figure that the KS chip must perform RSA efficiently, since RSA plays such an important role in cloud security. This problem will acquaint you with the encryption and decryption algorithms in RSA.

RSA works as follows. Each user generates two large random primes p and q, and sets his public modulus $m = p \cdot q$. The user then chooses a small number[2] e that is co-prime with $(p-1)(q-1)$, and computes $d = e^{-1} \mod (p-1)(q-1)$. The user announces his public key (e, m) to the world, and keeps d private. In order to send an encrypted message to our user, another user would encode the message as a number smaller than n, and encrypt it as $c = E(n) = n^e \mod m$. Our user would decode the message using $D(c) = c^d \mod m$. Assume that keys can be generated reasonably fast and that $D(E(n)) = n$, for all but a negligible fraction of values of n.

(a) [1 point] What is the running time of an implementation of $D(n)$ that uses the KS library in Problem 1, with the optimized version of MULTIPLY (Karatsuba's algorithm), assuming that n, d and m are N-byte numbers?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

You're thinking of using RSA to encrypt important sensitive images, such as last night's picture of you doing a Keg stand. Formally, a picture has $R \times C$ pixels (R rows, C columns), and each pixel is represented as 3 bytes that are RGB color space coordinates[3]. The RSA key is (e, m), where m is an N-byte number. An inefficient encryption method would process each row of pixel data as follows:

1.Break the $3C$ bytes of pixel data into groups of $N-1$ bytes

2.Pad the last group with 0 bytes up to $N-1$ bytes

3.Encrypt each group of $N-1$ bytes to obtain an N-byte output

4.Concatenate the N-byte outputs

[2] 65,537 is a popular choice nowadays

[3] see http://en.wikipedia.org/wiki/RGB_color_space

*Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

(b) [1 point] How many calls to the RSA encryption function $E(n)$ are necessary to encrypt an $R \times C$-pixel image?

1. $\Theta(1)$
2. $\Theta(RC)$
3. $\Theta(\frac{RC}{N})$
4. $\Theta(\frac{RN}{C})$
5. $\Theta(\frac{CN}{R})$

(c) [1 point] What is the running time for decrypting an $R \times C$-pixel image that was encrypted using the method above, using the KS library in Problem 1, with the optimized version of MULTIPLY (Karatsuba's algorithm)?

1. $\Theta(N)$
2. $\Theta(N^2)$
3. $\Theta(N^2 \log N)$
4. $\Theta(N^{\log_2 3})$
5. $\Theta(N^{\log_2 6})$
6. $\Theta(RCN)$
7. $\Theta(RCN^2)$
8. $\Theta(RCN^2 \log N)$
9. $\Theta(RCN^{\log_2 3})$
10. $\Theta(RCN^{\log_2 6})$
11. $\Theta(RN)$
12. $\Theta(RN^2)$
13. $\Theta(RN^2 \log N)$
14. $\Theta(RN^{\log_2 3})$
15. $\Theta(RN^{\log_2 6})$

(d) [5 points] A fixed point under RSA is a number n such that $E(n) \equiv n \mod m$, so RSA does not encrypt the number at all. Which of the following numbers are fixed points under RSA? (True / False)

1. 0
2. 1
3. 2
4. 3
5. $m - 2$
6. $m - 1$

(e) [5 points] What other weaknesses does the RSA algorithm have? (True / False)

1. $E(-n) \equiv -E(n) \mod m$

2. $E(n_1) + E(n_2) \equiv E(n_1 + n_2) \mod m$
3. $E(n_1) - E(n_2) \equiv E(n_1 - n_2) \mod m$
4. $E(n_1) \cdot E(n_2) \equiv E(n_1 \cdot n_2) \mod m$
5. $E(n_1)^{n_2} \equiv E(n_1^{n_2}) \mod m$

(f) [5 points] Amdtel plans to use RSA encryption to secretly tell Gopple when its latest smartphone CPU is ready to ship. Amdtel will send one message every day to Gopple, using Gopple's public key (e_G, m_G). The message will be NO (the number 20079 when using ASCII), until the day the CPU is ready, then the message will change to YES (the number 5858675 when using ASCII). You pointed out to your manager that this security scheme is broken, because an attacker could look at the encrypted messages, and know that the CPU is ready when the daily encrypted message changes. This is a problem of deterministic encryption. If $E(20079)$ always takes the same value, an attacker can distinguish $E(20079)$ from $E(5858675)$. How can the problem of deterministic encryption be fixed? (True / False)

1. Append the same long number (the equivalent of a string such as 'XXXPADDINGXXX') to each message, so the messages are bigger.
2. Append a random number to each message. All random numbers will have the same size, so the receiver can recognize and discard them.
3. Use a different encryption key to encrypt each message, and use Gopple's public exponent and modulus to encrypt the decryption key for each message.
4. Use an uncommon encoding, such as UTF-7, so that the attacker will not know the contents of the original messages.
5. Share a "secret" key with Gopple, so that the attacker can't use the knowledge on Gopple's public exponent and modulus.

Problem 5-3. [42 points] **Image Decryption**

Your manager wants to show off the power of the Knight's Shield chip by decrypting a live video stream directly using the RSA public-key crypto-system. RSA is quite resource-intensive, so most systems only use it to encrypt the key of a faster algorithm. Decrypting live video would be an impressive technical feat!

Unfortunately, the performance of the KS chip on RSA decryption doesn't come even close to what's needed for streaming video. The hardware engineers said the chip definitely has enough computing power, and blamed the problem on the RSA implementation. Your new manager has heard about your algorithmic chops, and has high hopes that you'll get the project back on track. The software engineers suggested that you benchmark the software using images because, after all, video is just a sequence of frames.

The code is in the `rsa` directory in the zip file for this problem set.

(a) [2 points] Run the code under the python profiler with the command below, and identify the method inside `bignum.py` that is most suitable for optimization. Look at the methods that take up the most CPU time, and choose the first method whose running time isn't proportional to the size of its output.

```
python -m cProfile -s time rsa.py < tests/1verdict_32.in
```

Warning: the command above can take 1-10 minutes to complete, and bring the CPU usage to 100% on one of your cores. Plan accordingly. If you have installed PyPy successfully, you should replace `python` with `pypy` in the command above for a 2-10x speed improvement.

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

(b) [1 point] How many times is the method called?

(c) [1 point] The troublesome method is implementing a familiar arithmetic operation. 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 digits in the input numbers.

1. $\Theta(N)$.
2. $\Theta(N \log n)$
3. $\Theta(N \log^2 n)$
4. $\Theta(N^{\log_2 3})$
5. $\Theta(N^2)$
6. $\Theta(N^{\log_2 7})$
7. $\Theta(N^3)$

(d) [1 point] What is the tightest asymptotic bound for the worst-case running time of division? Express your answer in terms of N, the number of digits in the input numbers.

1. $\Theta(N)$.

2. $\Theta(N \log n)$
3. $\Theta(N \log^2 n)$
4. $\Theta(N^{\log_2 3})$
5. $\Theta(N^2)$
6. $\Theta(N^{\log_2 7})$
7. $\Theta(N^3)$

We have implemented a visualizer for your image decryption output, to help you debug your code. The visualizer will also come in handy for answering the question below. To use the visualizer, first produce a trace.

```
TRACE=jsonp python rsa.py < tests/1verdict_32.in > trace.jsonp
```

On Windows, use the following command instead.

```
rsa_jsonp.bat < tests/1verdict_32.in > trace.jsonp
```

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

(e) [6 points] The test cases that we supply highlight the problems of RSA that we discussed above. Which of the following is true? (True / False)

1. Test `1verdict_32` shows that RSA has fixed points.
2. Test `1verdict_32` shows that RSA is deterministic.
3. Test `2logo_32` shows that RSA has fixed points.
4. Test `2logo_32` shows that RSA is deterministic.
5. Test `5future_1024` shows that RSA has fixed points.
6. Test `5future_1024` shows that RSA is deterministic.

(f) [1 point] Read the code in `rsa.py`. Given a decrypted image of $R \times C$ pixels (R rows, C columns), where all the pixels are white (all the image data bytes are 255), how many times will `powmod` be called during the decryption operation in `decrypt_image`?

1. $\Theta(1)$
2. $\Theta(RC)$
3. $\Theta(\frac{RC}{N})$
4. $\Theta(\frac{RN}{C})$
5. $\Theta(\frac{CN}{R})$

(g) [30 points] The multiplication and division operations in `big_num.py` are implemented using asymptotically efficient algorithms that we have discussed in class. However, the sizes of the numbers involved in RSA for typical key sizes aren't suitable for complex algorithms with high constant factors. Add new methods to `BigNum` implementing multiplication and division using straight-forward algorithms with low constant factors, and modify the main multiplication and division methods to use the

> simple algorithms if at least one of the inputs has 64 digits (bytes) or less. Please note that you are not allowed to import any additional Python libraries and our test will check this.

The KS software testing team has put together a few tests to help you check your code's correctness and speed. `big_num_test.py` contains unit tests with small inputs for all `BigNum` public methods. `rsa_test.py` runs the image decryption code on the test cases in the `tests/` directory.

You can use the following command to run all the image decryption tests.

```
python rsa_test.py
```

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

```
python rsa.py < tests/1verdict_32.in > out
```

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

```
diff out tests/1verdict_32.gold
```

For Windows, use `fc` to compare files.

```
fc out tests/1verdict_32.gold
```

While debugging your code, you should open a new Terminal window (Command Prompt in Windows), and set the `KS_DEBUG` environment variable (`export KS_DEBUG=true`; on Windows, use `set KS_DEBUG=true`) to use a slower version of our code that has more consistency checks.

When your code passes all tests, and runs reasonably fast (the tests should complete in less than 90 seconds on any reasonably recent computer using PyPy, or less than 600 seconds when using CPython), upload your modified `big_num.py` to the course submission site. Our automated grading code will use our versions of `test_rsa.py`, `rsa.py` and `ks_primitives.py` / `ks_primitives_unchecked.py`, so please do not modify these files.