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

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

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

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

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

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

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).

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).

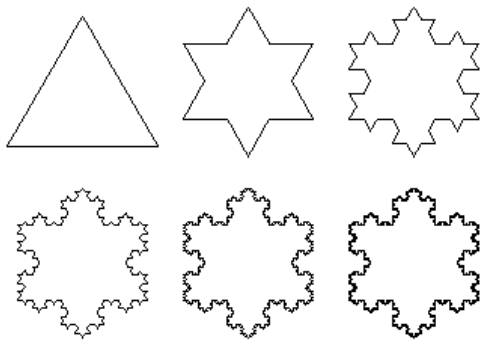


Figure 1: The Koch snowflake fractal, rendered at Level of Detail (LoD) 0 through 5.

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)

```
1  e_1, e_2, e_3 = edges of an equilateral triangle with side length 1
2  SNOWFLAKE-EDGE(e_1, n)
3  SNOWFLAKE-EDGE(e_2, n)
4  SNOWFLAKE-EDGE(e_3, n)
```

SNOWFLAKE-EDGE($edge, n$)

```
1  if n == 0
2      edge is an edge on the snowflake
3  else
4      e_1, e_2, e_3 = split edge in 3 equal parts
5      SNOWFLAKE-EDGE(e_1, n − 1)
6      f_2, g_2 = edges of an equilateral triangle whose 3rd edge is e_2, pointing outside the snowflake
7      Δ(f_2, g_2, e_2) is a triangle on the snowflake's surface
8      SNOWFLAKE-EDGE(f_2, n − 1)
9      SNOWFLAKE-EDGE(g_2, n − 1)
10     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

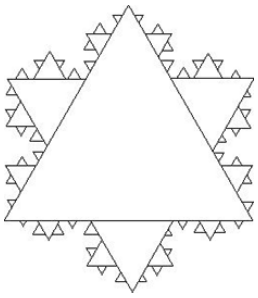


Figure 2: Koch snowflake drawn with triangles.

3. $3\,n$
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

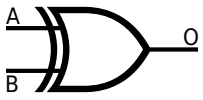


Figure 3: 2-input XOR gate; A and B supply the inputs, and O receives the output.

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.

Problem Set 3

Both theory and programming questions are due **Thursday, October 6** 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, 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 **Friday, October 7th, 11:59PM**. Your grade will be based on both your solutions and your critique of the solutions.

Problem 3-1. [45 points] **Range Queries**

Microware is preparing to launch a new database product, NoSQL Server, aimed at the real-time analytics market. Web application analytics record information (e.g., the times when users visit the site, or how much does it take the server to produce a HTTP response), and can display the information using a Pretty Graph™[1], so that CTOs can claim that they're using data to back their decisions.

NoSQL Server databases will support a special kind of index, called a *range index*, to speed up the operations needed to build a Pretty Graph™ out of data. Microware has interviewed you during the fall Career Fair, and immediately hired you as a consultant and asked you to help the NoSQL Server team design the range index.

The range index must support fast (sub-linear) insertions, to keep up with Web application traffic. The first step in the Pretty Graph™ algorithm is finding the minimum and maximum values to be plotted, to set up the graph's horizontal axis. So the range index must also be able to compute the minimum and maximum over all keys quickly (in sub-linear time).

(a) [1 point] Given the constraints above, what data structure covered in 6.006 lectures should be used for the range index? Microware engineers need to implement range indexes, so choose the simplest data struture that meets the requirements.

1. Min-Heap
2. Max-Heap
3. Binary Search Tree (BST)
4. AVL Trees
5. B-Trees

(b) [1 point] How much time will it take to insert a key in the range index?

1. $O(1)$

[1] U.S. patent pending, no. 9,999,999

2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

(c) [1 point] How much time will it take to find the minimum key in the range index?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

(d) [1 point] How much time will it take to find the maximum key in the range index?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

The main work of the Pretty Graph™ algorithm is drawing the bars in the graph. A bar shows how many data points there are between two values. For example, in order to produce the visitor graph that is the hallmark of Google Analytics, the range index would record each time that someone uses the site, and a bar would count the visiting times between the beginning and the ending of a day. Therefore, the range index needs to support a fast (sub-linear time) $\textsc{Count}(l, h)$ query that returns the number of keys in the index that are between l and h (formally, keys k such that $l \leq k \leq h$).

Your instinct (or 6.006 TA) tells you that $\textsc{Count}(l, h)$ can be easily implemented on top of a simpler query, $\textsc{Rank}(x)$, which returns the number of keys in the index that are smaller or equal to x (informally, if the keys were listed in ascending order, x's rank would indicate its position in the sorted array).

(e) [1 point] Assuming $l < h$, and both l and h exist in the index, $\textsc{Count}(l, h)$ is

1. $\textsc{Rank}(l) - \textsc{Rank}(h) - 1$
2. $\textsc{Rank}(l) - \textsc{Rank}(h)$
3. $\textsc{Rank}(l) - \textsc{Rank}(h) + 1$
4. $\textsc{Rank}(h) - \textsc{Rank}(l) - 1$
5. $\textsc{Rank}(h) - \textsc{Rank}(l)$
6. $\textsc{Rank}(h) - \textsc{Rank}(l) + 1$
7. $\textsc{Rank}(h) + \textsc{Rank}(l) - 1$
8. $\textsc{Rank}(h) + \textsc{Rank}(l)$
9. $\textsc{Rank}(h) + \textsc{Rank}(l) + 1$

(f) [1 point] Assuming $l < h$, and h exists in the index, but l does not exist in the index, $\text{COUNT}(l, h)$ is

1. $\text{RANK}(l) - \text{RANK}(h) - 1$
2. $\text{RANK}(l) - \text{RANK}(h)$
3. $\text{RANK}(l) - \text{RANK}(h) + 1$
4. $\text{RANK}(h) - \text{RANK}(l) - 1$
5. $\text{RANK}(h) - \text{RANK}(l)$
6. $\text{RANK}(h) - \text{RANK}(l) + 1$
7. $\text{RANK}(h) + \text{RANK}(l) - 1$
8. $\text{RANK}(h) + \text{RANK}(l)$
9. $\text{RANK}(h) + \text{RANK}(l) + 1$

(g) [1 point] Assuming $l < h$, and l exists in the index, but h does not exist in the index, $\text{COUNT}(l, h)$ is

1. $\text{RANK}(l) - \text{RANK}(h) - 1$
2. $\text{RANK}(l) - \text{RANK}(h)$
3. $\text{RANK}(l) - \text{RANK}(h) + 1$
4. $\text{RANK}(h) - \text{RANK}(l) - 1$
5. $\text{RANK}(h) - \text{RANK}(l)$
6. $\text{RANK}(h) - \text{RANK}(l) + 1$
7. $\text{RANK}(h) + \text{RANK}(l) - 1$
8. $\text{RANK}(h) + \text{RANK}(l)$
9. $\text{RANK}(h) + \text{RANK}(l) + 1$

(h) [1 point] Assuming $l < h$, and neither l nor h exist in the index, $\text{COUNT}(l, h)$ is

1. $\text{RANK}(l) - \text{RANK}(h) - 1$
2. $\text{RANK}(l) - \text{RANK}(h)$
3. $\text{RANK}(l) - \text{RANK}(h) + 1$
4. $\text{RANK}(h) - \text{RANK}(l) - 1$
5. $\text{RANK}(h) - \text{RANK}(l)$
6. $\text{RANK}(h) - \text{RANK}(l) + 1$
7. $\text{RANK}(h) + \text{RANK}(l) - 1$
8. $\text{RANK}(h) + \text{RANK}(l)$
9. $\text{RANK}(h) + \text{RANK}(l) + 1$

Now that you know how to reduce a $\text{COUNT}()$ query to a constant number of $\text{RANK}()$ queries, you want to figure out how to implement $\text{RANK}()$ in sub-linear time. None of the tree data structures that you studied in 6.006 supports optimized $\text{RANK}()$ out of the box, but you just remembered that tree data structures can respond to some queries faster if the nodes are cleverly augmented with some information.

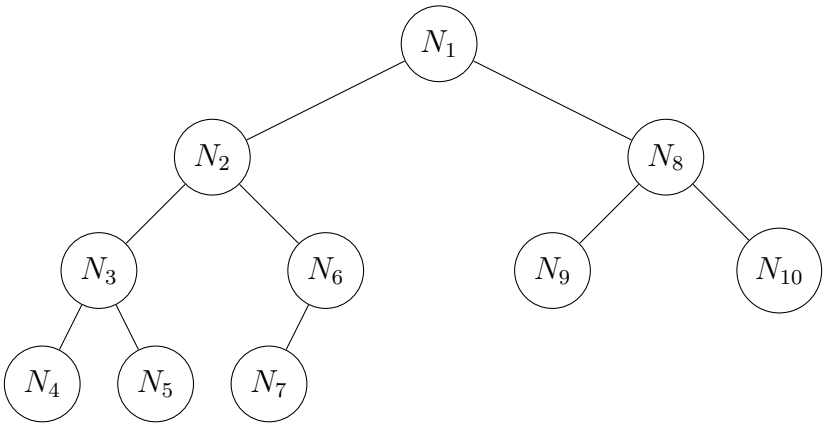
(i) [1 point] In order to respond to RANK() queries in sub-linear time, each node $node$ in the tree will be augmented with an extra field, $node.\gamma$. Keep in mind that for a good augmentation, the extra information for a node should be computed in $O(1)$ time, based on other properties of the node, and on the extra information stored in the node's subtree. The meaning of $node.\gamma$ is

1. the minimum key in the subtree rooted at $node$
2. the maximum key in the subtree rooted at $node$
3. the height of the subtree rooted at $node$
4. the number of nodes in the subtree rooted at $node$
5. the rank of $node$
6. the sum of keys in the subtree roted at $node$

(j) [1 point] How many extra *bits* of storage per node does the augmentation above require?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$
6. $O(N)$

The following questions refer to the tree below.



(k) [1 point] $N_4.\gamma$ is

1. 0
2. 1
3. 2
4. the key at N_4

(l) [1 point] $N_3.\gamma$ is

1. 1
2. 2
3. 3
4. the key at N_4
5. the key at N_5
6. the sum of keys at $N_3 \ldots N_5$

(m) [1 point] $N_2.\gamma$ is

1. 2
2. 3
3. 4
4. 6
5. the key at N_4
6. the key at N_7
7. the sum of keys at $N_3 \ldots N_5$

(n) [1 point] $N_1.\gamma$ is

1. 3
2. 6
3. 7
4. 10
5. the key at N_4
6. the key at N_{10}
7. the sum of keys at $N_1 \ldots N_{10}$

(o) [6 points] Which of the following functions need to be modified to update γ? If a function does not apply to the tree for the range index, it doesn't need to be modified. (True / False)

1. INSERT
2. DELETE
3. ROTATE-LEFT
4. ROTATE-RIGHT
5. REBALANCE
6. HEAPIFY

(p) [1 point] What is the running time of a COUNT() implementation based on RANK()?

1. $O(1)$
2. $O(\log(\log N))$

3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

After the analytics data is plotted using Pretty Graph™, the CEO can hover the mouse cursor over one of the bars, and the graph will show a tooltip with the information represented by that bar. To support this operation, the range index needs to support a LIST(l, h) operation that returns all the keys between l and h as quickly as possible.

LIST(l, h) cannot be sub-linear in the worst case, because LIST$(-\infty, +\infty)$ must return all the keys in the index, which takes $\Omega(n)$ time. However, if LIST only has to return a few elements, we would like it to run in sub-linear time. We formalize this by stating that LIST's running time should be $T(N) + \Theta(L)$, where L is the length of the list of keys output by LIST, and $T(N)$ is sub-linear.

Inspiration (or your 6.006 TA) strikes again, and you find yourself with the following pseudocode for LIST.

```
LIST(tree, l, h)
1  lca = LCA(tree, l, h)
2  result = []
3  NODE-LIST(lca, l, h, result)
4  return result

NODE-LIST(node, l, h, result)
1  if node == NIL
2      return
3  if l ≤ node.key and node.key ≤ h
4      ADD-KEY(result, node.key)
5  if node.key ≥ l
6      NODE-LIST(node.left, l, h, result)
7  if node.key ≤ h
8      NODE-LIST(node.right, l, h, result)

LCA(tree, l, h)
1  node = tree.root
2  until node == NIL or (l ≤ node.key and h ≥ node.key)
3      if l < node.key
4          node = node.left
5      else
6          node = node.right
7  return node
```

(q) [1 point] LCA most likely means

1. last common ancestor
2. lowest common ancestor
3. low cost airline
4. life cycle assessment
5. logic cell array

(r) [1 point] The running time of $\mathrm{LCA}(l, h)$ for the trees used by the range index is

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

(s) [1 point] Assuming that ADD-KEY runs in $O(1)$ time, and that LIST returns a list of L keys, the running time of the NODE-LIST call at line 3 of LIST is

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$
6. $O(1) + O(L)$
7. $O(\log(\log N)) + O(L)$
8. $O(\log N) + O(L)$
9. $O(\log^2 N) + O(L)$
10. $O(\sqrt{N}) + O(L)$

(t) [1 point] Assuming that ADD-KEY runs in $O(1)$ time, and that LIST returns a list of L keys, the running time of LIST is

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$
6. $O(1) + O(L)$
7. $O(\log(\log N)) + O(L)$
8. $O(\log N) + O(L)$
9. $O(\log^2 N) + O(L)$
10. $O(\sqrt{N}) + O(L)$

(u) [20 points] Prove that LCA is correct.

Problem 3-2. [55 points] **Digital Circuit Layout**

Your AMDtel internship is off to a great start! The optimized circuit simulator cemented your reputation as an algorithms whiz. Your manager capitalized on your success, and promised to deliver the Bullfield chip a few months ahead of schedule. Thanks to your simulator optimizations, the engineers have finished the logic-level design, and are currently working on laying out the gates on the chip. Unfortunately, the software that verifies the layout is taking too long to run on the preliminary Bullfield layouts, and this is making the engineers slow and unhappy. Your manager is confident in your abilities to speed it up, and promised that you'll "do your magic" again, in "one week, two weeks tops".

A chip consists of logic gates, whose input and output terminals are connected by wires (very thin conductive traces on the silicon substrate). AMDtel's high-yield manufacturing process only allows for horizontal or vertical wires. Wires must not cross each other, so that the circuit will function according to its specification. This constraint is checked by the software tool that you will optimize. The topologies required by complex circuits are accomplished by having dozens of layers of wires that do not touch each other, and the tool works on one layer at a time.

(a) [1 point] 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 circuit2.py < tests/10grid_s.in
```

Warning: the command above can take 15-60 minutes to complete, and bring the CPU usage to 100% on one of your cores. Plan accordingly. If you have installed PyPy successfully, you can replace `python` with `pypy` in the command above for a roughly 2x speed improvement.

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

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

The method that has the performance bottleneck is called from the `CrossVerifier` class. Upon reading the class, it seems that the original author was planning to implement a *sweep-line* algorithm, but couldn't figure out the details, and bailed and implemented an inefficient method at the last minute. Fortunately, most of the infrastructure for a fast sweep-line algorithm is still in place. Furthermore, you notice that the source code contains a trace of the working sweep-line algorithm, in the `good_trace.jsonp` file.

Sweep-line algorithms are popular in computational geometry. Conceptually, such an algorithm sweeps a vertical line left to right over the plane containing the input data, and performs operations when the line "hits" point of interest in the input. This is implemented by generating an array containing all the points of interest, and then sorting them according to their position along the horizontal axis (x coordinate).

Read the source for `CrossVerifier` to get a feel for how the sweep-line infrastructure is supposed to work, and look at the good trace in the visualizer that we have provided for you. To see the good trace, copy `good_trace.jsonp` to `trace.jsonp`

```
cp good_trace.jsonp trace.jsonp
```

On Windows, use the following command instead.

```
copy good_trace.jsonp trace.jsonp
```

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

The questions below refer to the fast sweep-line algorithm shown in `good_trace.jsonp`, not to the slow algorithm hacked together in `circuit2.py`.

(c) [5 points] The x coordinates of points of interest in the input are (True / False)

1. the x coordinates of the left endpoints of horizontal wires
2. the x coordinates of the right endpoints of horizontal wires
3. the x coordinates of midpoints of horizontal wires
4. the x coordinates where horizontal wires cross vertical wires
5. the x coordinates of vertical wires

(d) [1 point] When the sweep line hits the x coordinate of the left endpoint of a horizontal wire

1. the wire is added to the range index
2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

(e) [1 point] When the sweep line hits the x coordinate of the right endpoint of a horizontal wire

1. the wire is added to the range index
2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

(f) [1 point] When the sweep line hits the x coordinate of the midpoint of a horizontal wire

1. the wire is added to the range index
2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

(g) [1 point] When the sweep line hits the x coordinate of a vertical wire

1. the wire is added to the range index

2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

(h) [1 point] What is a good invariant for the sweep-line algorithm?

1. the range index holds all the horizontal wires to the left of the sweep line
2. the range index holds all the horizontal wires "stabbed" by the sweep line
3. the range index holds all the horizontal wires to the right of the sweep line
4. the range index holds all the wires to the left of the sweep line
5. the range index holds all the wires to the right of the sweep line

(i) [1 point] When a wire is added to the range index, what is its corresponding key?

1. the x coordinate of the wire's midpoint
2. the y coordinate of the wire's midpoint
3. the segment's length
4. the x coordinate of the point of interest that will remove the wire from the index

Modify `CrossVerifier` in `circuit2.py` to implement the sweep-line algorithm discussed above. If you maintain the current code structure, you'll be able to use our visualizer to debug your implementation. To use our visualizer, first produce a trace.

```
TRACE=jsonp python circuit2.py < tests/5logo.in > trace.jsonp
```

On Windows, run the following command instead.

```
circuit2_jsonp.bat < tests/5logo.in > trace.jsonp
```

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

(j) [1 point] Run your modified code under the python profiler again, using the same test case as before, and identify the method that takes up the most CPU time.

What is the name of the method with the highest CPU usage? If two methods have similar CPU usage times, ignore the simpler one.

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

(l) [40 points] Modify `circuit2.py` to implement a data structure that has better asymptotic running time for the operation above. Keep in mind that the tool has two usage scenarios:

- Every time an engineer submits a change to one of the Bullhorn wire layers, the tool must analyze the layer and report the number of wire crossings. In this late stage of the project, the version control system will automatically reject the engineer's change if it causes the number of wire crossings to go up over the previous version.

- Engineers working on the wiring want to see the pairs of wires that intersect, so they know where to focus ther efforts. To activate this detailed output, run the tool using the following command.
  `TRACE=list python circuit2.py < tests/6list_logo.in`
  On Windows, run the following command instead.
  `circuit2_list.bat < tests/6list_logo.in`

When your code passes all tests, and runs reasonably fast (the tests should complete in less than 60 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 7 October, 2011
Professors Erik Demaine and Srini Devadas Problem Set 4

Problem Set 4

Both theory and programming questions are due **Friday, 14 October** at **11:59PM**.

Remember that for the written response question, 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, October 20th, 11:59PM**. Your grade will be based on both your solutions and your critique of the solutions.

Problem 4-1. [35 points] **Hash Functions and Load**

(a) Imagine that an algorithm requires us to hash strings containing English phrases. Knowing that strings are stored as sequences of characters, Alyssa P. Hacker decides to simply use the sum of those character values (modulo the size of her hash table) as the string's hash. Will the performance of her implementation match the expected value shown in lecture?

1. Yes, the sum operation will space strings out nicely by length.
2. Yes, the sum operation will space strings out nicely by the characters they contain.
3. No, because reordering the words in a string will not produce a different hash.
4. No, because the independence condition of the simple uniform hashing assumption is violated.

(b) Alyssa decides to implement both collision resolution and dynamic resizing for her hash table. However, she doesn't want to do more work than necessary, so she wonders if she needs both to maintain the correctness and performance she expects. After all, if she has dynamic resizing, she can resize to avoid collisions; and if she has collision resolution, collisions don't cause correctness issues. Which statement about these two properties true?

1. Dynamic resizing alone will preserve both properties.
2. Dynamic resizing alone will preserve correctness, but not performance.
3. Collision resolution alone will preserve performance, but not correctness.
4. Both are necessary to maintain performance and correctness.

(c) Suppose that Alyssa decides to implement resizing. If Alyssa is enlarging a table of size m into a table of size m, and the table contains n elements, what is the best time complexity she can achieve?

1. $\Theta(m)$
2. $\Theta(m')$
3. $\Theta(n)$
4. $\Theta(nm')$
5. $\Theta(m + m')$
6. $\Theta(m + n)$
7. $\Theta(m' + n)$

(d) In lecture, we discussed doubling the size of our hash table. Ivy H. Crimson begins to implement this approach (that is, she lets $m' = 2m$) but stops when it occurs to her that she might be able to avoid wasting half of the memory the table occupies on empty space by letting $m' = m+k$ instead, where k is some constant. Does this work? If so, why do you think we don't do it? There is a good theoretical reason as well as several additional practical concerns; a complete answer will touch on both points.

Problem 4-2. [10 points] **Python Dictionaries**

We're going to get started by checking out a file from Python's Subversion repository at svn.python.org. The Python project operates a web frontend to their version control system, so we'll be able to do this using a browser.

Visit `http://svn.python.org/projects/python/trunk/Objects/dictnotes.txt`.

These are actual notes prepared by contributors to the Python project, as they currently exist in the Python source tree. (Cool! Actually, this document is a fascinating read—and you should be able to understand most of it.) Read over the seven use cases identified at the top of this document.

(a) Let's examine the "membership testing" use case. Which statement accurately describes this use case?

1. Many insertions right after creation, and then mostly lookups.
2. Many insertions right after creation, and then only lookups.
3. A workload of evenly-mixed insertions/deletions and lookups.
4. Alternating rounds of insertions/deletions and lookups.

(b) Now imagine that you have to pick a hash function, size, collision resolution strategy, and so forth (all of the characteristics of a hash table we've seen so far) in order to make a hash table perfectly suited to this use case alone. Pick the statement that best describes the choices you might make.

1. A large minimum size and a growth rate of 2.
2. A small minimum size and a growth rate of 2.

3. A large minimum size and a growth rate of 4.
4. A small minimum size and a growth rate of 4.

Problem 4-3. [55 points] **Matching DNA Sequences**

The code and data used in this problem are available on the course website. Please take a peek at the README.txt for some instructions.

Ben Bitdiddle has recently moved into the Kendall Square area, which is full of biotechnology companies and their shiny, window-laden office buildings. While mocking their dorky lab coats makes him feel slightly better about himself, he is secretly jealous, and so he sets out to earn one of his very own. To pick up the necessary geek cred, he begins experimenting with DNA-matching technologies.

Ben would like to create mutants to do his bidding, and to get started, he'd like to know how closely related the creatures he's collected are. If two sequences contain mostly the same subsequences in mostly the same places, then they're likely closely related; if they don't, they probably aren't. (This is, of course, a gross oversimplification.)

For our purposes, we'll represent a DNA sample as a sequence of characters. (These characters will all be upper-case. You can look at the Wikipedia page on nucleotides for a list of code characters and their meanings.) These sequences are very long, so comparing subsequences of them quickly is important. We've provided code in kfasta.py that reads the .fa files storing this data.

(a) Let's start with `subsequenceHashes`, which returns all length-k subsequences and their hashes (and perhaps other information, if there's anything else you might find useful).

Hint: There will likely be many of these matches; the DNA sequences are tens of millions of nucleotides long. To avoid keeping them all in memory at once, **implement your function as a generator**. See the Python reference materials available online for details if you aren't familiar with this important language construct.

(b) Implement `Multidict` and verify that your work passes the simple sanity tests provided.

`Multidict` should behave just like a Python dictionary, except that it can store multiple values per key. If no values exist for a key, it returns an empty list; otherwise, it returns the list of associated values. You may (and probably should) use the Python dictionary in your implementation.

(c) Now it's time to implement `getExactSubmatches`. Ignore the parameter m for the time being; we'll get to that in the next part. Again, implementing this function as a generator is probably a good idea. (You will probably have many, many matches– think about the combinatorics of the situation briefly.) As a hint, consider that much of the work has already been done by `Multidict` and `subsequenceHashes`; also take a peek at the `RollingHash` implementation we've given you. With these

building blocks, your solution probably does not need to be very complex (or more than a few lines).

This function should return pairs of offsets into the inputs. A tuple (x, y) being returned indicates that the k-length subsequence at position x in the first input matches the subsequence at position y in the second input.

We've provided a simple sanity test; your solution should be correct at this point (that is, `dnaseq.py` will produce the right output) but it'll probably be too slow to be useful. If you like, you can try running it on the first portion of two inputs; we've provided two such prefixes (the short files in the data directory) that might be helpful.

(d) The most significant reason why your solution is presently too slow to be useful is that you are hashing and inserting into your hash table tens of millions of elements, and then performing tens of millions of lookups into that hash table. Implement `intervalSubsequenceHashes`, which returns the same thing as `subsequenceHashes` except that it hashes only one in m subsequences. (A good implementation will not do more work than is necessary.) Modify your implementation of `getExactSubmatches` to honor m only for sequence A. Consider why we still see approximately the same result, and why we can't further improve performance by applying this technique to sequence B as well.

(e) Run comparisons between the two human samples (paternal and maternal) and between the paternal sample and each of the animal samples.

Feel free to take a peek at how the image-generation code works. Conceptually, what it's doing is keeping track of how many of your (x, y) match tuples land in each of a two-dimensional grid of bins, each of which corresponds to a pixel in the output image. At the end, it normalizes the counts so that the highest count observed is totally black and an empty bin is white.

Think for a second about what a perfect match (e.g., comparing a sequence to itself) should look like. Try comparing the two human samples you have (maternal and paternal), one of the humans against the chimp sample, and then against the dog sample. Make sure your results make sense!

We've posted what our reference solution produced for the human-human comparison, the human-chimp comparison, and the human-dog comparison.

Please submit the code that you wrote. (You should only have had to modify dnaseq.py, so that's all you need to submit.)

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	$\texttt{R1} + \texttt{R2}$
`SUB R1, R2`	8 / 16	8 / 16	16	$\texttt{R1} - \texttt{R2} \mod 65536$
`MUL R1, R2`	8	8	16	$\texttt{R1} \cdot \texttt{R2}$
`DIV R1, R2`	16	8	8	$\texttt{R1} \div \texttt{R2} \mod 256$
`MOD R1, R2`	16	8	8	$\texttt{R1} \,\%\, \texttt{R2}$
`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 ^ 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)
1  C = ZERO(N + 1)     ZERO(k) creates a k-digit number, with all digits set to 0s.
2  carry  = 0
3  for i = 1 to N
4        digit  = WORD(A[i]) + WORD(B[i]) + WORD(        )
5        C[ ] = LSB(d g  )
6        carry  = MSB(d g  )
7  C[N + 1] =
8  return C
```

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

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\quad)$
7. $\Theta(N\quad)$
8. $\Theta(N\ )$

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

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\quad)$
7. $\Theta(N\quad)$
8. $\Theta(N\ )$

(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.

```
MULTIPLY(A, B, N)
1  C = ZERO(2N)
2  for i = 1 to N
3       carry  = 0
4       for j = 1 to N
5            digit  = A[i] · B[j] + WORD(C[i + j − 1]) + WORD(        )
6            C[i + j − 1] = LSB(d g  )
7            carry  = MSB(d g  )
8       C[  +    ] = carry
9  return C
```

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

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

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

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

(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 $\text{COPY}(A, N)$ that creates a copy of an N-digit number A, using $\Theta(N)$ time.

```
DIVMOD(A, B, N)
 1  Q = ZERO(N)      quotient
 2  R = COPY(A, N)      remainder
 3  S  = COPY(B, N)      S_i = B · 2
 4  i = 0
 5  repeat
 6      i = i + 1
 7      S_i = ADD(  1, S_i  , N)
 8  until S_i[N + 1] > 0 or CMP(S_i, A, N) == GREATER
 9  for j = i − 1 downto 0
10      Q = ADD(Q, Q, N)
11      if CMP(R, S_j, N) != SMALLER
12          R = SUBTRACT(R, S_j, N)
13          Q[0] = Q[0]||1      Faster version of Q = Q + 1
14  return (Q, R)
```

(g) [1 point] $\text{CMP}(A, B, N)$ 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. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\ \ \ )$
7. $\Theta(N\ \ \ )$
8. $\Theta(N\ )$

(h) [1 point] $\text{SUBTRACT}(A, B, N)$ 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. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\ \ \ )$

7. $\Theta(N \quad)$
8. $\Theta(N\ )$

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

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N \quad)$
7. $\Theta(N \quad)$
8. $\Theta(N\ )$

The KS library does not use the DIVMOD implementation above. Instead, it uses Newton's method to implement $\text{DIV}(A, B, N)$ 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. $\text{MOD}(A, B, N)$ 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. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N \quad)$
7. $\Theta(N \quad)$
8. $\Theta(N\ )$

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

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N \quad)$
7. $\Theta(N \quad)$
8. $\Theta(N\ )$

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)
 1  R = ONE(N)      result
 2  X = COPY(B, N)      multiplier
 3  for i = 1 to N
 4       mask = 1
 5       for bit = 1 to 8
 6            if E[i] & mask != 0
 7                 R = MOD(MULTIPLY(R, X, N), M, 2N)
 8            X = MOD(MULTIPLY(X, X, N), M, 2N)
 9            mask = LSB(m      · 2)
10  return R
```

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

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

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. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\ \ \ )$
7. $\Theta(N\ \ \ )$
8. $\Theta(N\ )$

(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. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\quad)$
7. $\Theta(N\quad)$
8. $\Theta(N\ )$

(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. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\quad)$
7. $\Theta(N\quad)$
8. $\Theta(N\ )$

(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 $\Theta(N\quad)$.

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 $d = e \mod (p-1)(q-1)$. 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. $\Theta(N\ )$
5. $\Theta(N\ \log N)$
6. $\Theta(N\quad)$
7. $\Theta(N\quad)$
8. $\Theta(N\ )$

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

(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. $\Theta(\text{—})$
4. $\Theta(\text{—})$
5. $\Theta(\text{—})$

(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. $\Theta(N\ )$
3. $\Theta(N\ \log N)$
4. $\Theta(N\quad)$
5. $\Theta(N\quad)$
6. $\Theta(RCN)$
7. $\Theta(RCN\ )$
8. $\Theta(RCN\ \log N)$
9. $\Theta(RCN\quad)$
10. $\Theta(RCN\quad)$
11. $\Theta(RN)$
12. $\Theta(RN\ )$
13. $\Theta(RN\ \log N)$
14. $\Theta(RN\quad)$
15. $\Theta(RN\quad)$

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

(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 $(e\ ,\ m\ )$. 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. $\Theta(N \quad)$
5. $\Theta(N \ )$
6. $\Theta(N \quad)$
7. $\Theta(N \ )$

(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. $\Theta(N \quad)$
5. $\Theta(N\ )$
6. $\Theta(N \quad)$
7. $\Theta(N\ )$

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. $\Theta(—)$
4. $\Theta(—)$
5. $\Theta(—)$

(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.

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

November 1, 2011
Problem Set 6

Problem Set 6

Both theory and programming questions are due on **Tuesday, November 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.

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 guide by **Tuesday, November 22, 11:59PM**. Your grade will be based on both your solutions and the grading explanation.

Problem 6-1. [30 points] **I Can Haz Moar Frendz?**

Alyssa P. Hacker is interning at RenBook (人书 / 人書 in Chinese), a burgeoning social network website. She needs to implement a new friend suggestion feature. For two friends u and v (friendship is undirected), the ***EdgeRank*** $ER(u, v)$ can be computed in constant time based on the interest u shows in v (for example, how frequently u views v's profile or comments on v's wall). Assume that EdgeRank is directional and asymmetric, and that its value falls in the range $(0, 1)$. A user u is ***connected*** to a user v if they are connected through some mutual friends, i.e., $u = u_0$ has a friend u_1, who has a friend u_2, ..., who has a friend $u_k = v$. The integer k is the ***vagueness*** of the connection. Define the ***strength*** of such a connection to be

$$S(p) = \prod_{i=1}^{k} ER(u_{i-1}, u_i).$$

For a given user s, Alyssa wants to have a way to rank potential friend suggestions according to the strength of the connections s has with them. In addition, the vagueness of those connections should not be more than k, a value Alyssa will decide later.

Help Alyssa by designing an algorithm that computes the **strength** of the strongest connection between a given user s and **every other user** v to whom s is connected with vagueness at most k, in $O(kE + V)$ time (i.e., for every pair of s and v for $v \in V \backslash \{s\}$, compute the strength of the strongest connection between them with vagueness at most k). Assume that the network has $|V|$ users and $|E|$ friend pairs. Analyze the running time of your algorithm. For partial credit, give a slower but correct algorithm. You can describe your algorithm in words, pseudocode or both.

Problem 6-2. [30 points] **RenBook Competitor**

Having experienced RenBook and its alternatives, you've come to the conclusion that the social networking websites currently out there lack focus. You've decided to create a social network that caters specifically to the algorithms-in-Python crowd. Excited about this idea, you've pitched to various investors, a few of whom have traded you a check in return for convertible debt. You're ready to go! Now you have to implement your idea.

The first step is to rent a machine on the web, a task which you've easily accomplished using your newly-acquired funding. The next step, however, is harder to complete: you must install a web server library and all of its dependencies. Each library that you wish to install can depend on a number of other libraries, which you will have to install first. Each of those libraries can in turn have its own dependencies.

You can try to install the libraries one-by-one and resolve the dependencies by hand, but you'd like to write a script that will do the work for you. Fortunately, having studied graphs in 6.006, you're confident that you will be able to accomplish this task. You will need to determine which libraries need to be installed and then generate the order in which the libraries will be installed so that there will be no dependency problems.

Examining the software library repository, you see that there are V total libraries, which together have a total of E dependencies. The repositories enforces the rule that dependencies cannot be cyclic. Libraries rarely all depend on one another, so you can safely assume that $E \ll V^2$.

(a) An ***installation order*** is an ordering of all the libraries such that each library's dependencies appear prior to it in the sequence. If we install each library in this sequence in order, we are guaranteed to avoid dependency problems. Describe in detail how to generate an installation order for the entire repository in $O(V + E)$ time.

We wish to install a web server library along with its dependencies. Suppose that some libraries are already installed on your system, and that only P libraries remain to be installed (you can determine whether a library has already been installed by performing a dictionary lookup in $O(1)$ time). Assume that the maximum number of dependencies for any given library is D.

(b) Give pseudocode for an algorithm that generates an installation order for the non-installed libraries that are needed for installing the web server library in $O(P + PD)$ time. Describe your algorithm. You may use any routine given in CLRS as a subroutine in your pseudocode, and you can use a textual description, a clarifying example, or a correctness proof for the description.

Problem 6-3. [30 points] **Rubik's Cube**

In this problem, you will develop algorithms for solving the $2 \times 2 \times 2$ Rubik's Cube, known as the *Pocket Cube*. Call a configuration of the cube "k levels from the solved position" if it can reach the solved configuration in exactly k twists, but cannot reach the solved configuration in any fewer twists.

The `rubik` directory in the problem set package contains the Rubik's Cube library and a graphical user interface to visualize your algorithm.

We will solve the Rubik's Cube puzzle by finding the shortest path between two configurations (the start and goal) using BFS.

A BFS that goes as deep as 14 levels (the diameter of the pocket cube) will take a few minutes (not to mention the memory needed). A few minutes is too slow for us: we want to solve the cube very quickly!

Instead, we will take advantage of a fact that we saw in lecture: the number of nodes at level 7 (half the diameter) is much smaller than half the total number of nodes.

With this in mind, we can instead do a two-way BFS, starting from each end at the same time, and meeting in the middle. At each step, expand one level from the start position, and one level from the end position, and then check to see whether any of the new nodes have been discovered in both searches. If there is such a node, we can read off parent pointers (in the correct order) to return the shortest path.

Write a function `shortest_path` in `solver.py` that takes two positions, and returns a list of moves that is a shortest path between the two positions.

Test your code using `test_solver.py`. Check that your code runs in less than 5 seconds.

Problem 6-4. [30 points] **From Berklee to Berkeley**

Jack Florey and his fellow hackers are planning to put a TARDIS[1] on Berkeley's most symbolic tower. However, the company responsible for transporting the construction material mistook the destination as Berklee College of Music. In order to save the extra cost of transportation back to Berkeley, Jack wants to help them to find the fastest route from Berklee to Berkeley. He downloaded the data from the National Highway Planning Network (NHPN)[2]. However, as he did not take 6.006 and does not know how to implement the Dijkstra's algoritm, he turns to you for help.

The partial code is in the `dijkstra` directory in the zip file for this problem set. It contains a `data` directory which includes node and link text files from the NHPN. Open `nhpn.nod` and `nhpn.lnk` in a text editor to get a sense of how the data is stored (`datadict.txt` and `format.txt` have more precise descriptions of the data fields and their meanings). The Python module `nhpn.py` provided contains code to load the text files into `Node` and `Link` objects. Read `nhpn.py` to understand the format of the `Node` and `Link` objects.

In `dijkstra.py`, the `PathFinder` object contains a source node, a destination node, and a `Network` object which represents the highway network with a list of `Node` objects. For each `node`, `node.adj` contains a list of all nodes adjacent to `node`.

Your task is implementing the method

```
PathFinder.dijkstra(weight, nodes, source, destination)
```

using Dijkstra's algorithm. It is given a function `weight(node1, node2)` that returns the weight of the link between `node1` and `node2`, a list of all the `nodes`, a `source` node and a `destination` node in the network. The method should return a tuple of the shortest path from the `source` to the `destination` as a list of nodes, and the number of nodes visited during the execution of the algorithm. A `node` is visited if the shortest path of it from the `source` is computed. You should stop the search as soon as the shortest path to the `destination` is found.

Function `distance(node1, node2)` is a `weight` function used by the main program to call the `dijkstra` method. It returns the distance between two NHPN nodes. Nodes come with

[1] Doctor Who's Time And Relative Dimension In Space

[2] http://www.fhwa.dot.gov/planning/nhpn/

latitude and longitude (in millionths of degrees). For simplicity, we treat these as (x, y) coordinates on a flat surface, where the distance between two points can be calculated using the Pythagorean Theorem.

The `NodeDistancePair` object wraps a given node and its distance which can be used as a key in the priority queue.

The Python module `priority_queue.py` contains a min-heap based implementation of priority queue. It is augmented with a map of keys to their indices in the heap, so that the `decrease_key(key)` method takes $O(1)$ time to lookup the key in the priority queue.

After implementing `PathFinder.dijkstra()`, you can run

```
python dijkstra.py < tests/0boston_berkeley.in
```

to see the shortest distance from Boston, MA to Berkeley, CA.

To visualize the result, you can run the following command.

```
TRACE=kml python dijkstra.py < tests/0boston_berkeley.in
```

On Windows, use the following command instead:

```
dijkstra_kml.bat < tests/0boston_berkeley.in
```

This will create two files, `path_flat.kml` and `path_curved.kml`. Both should be paths from Boston, MA to Berkeley, CA. `path_flat.kml` is created using the distance function described earlier, and `path_curved.kml` is created using a distance function that does not assume that the Earth is flat. `.kml` files can be viewed using Google Maps, by putting the file in a web-accessible location (such as your *Athena Public directory), going to `http://maps.google.com` and putting the URL (`http://...`) in the search box. Try asking Google Maps for driving directions from Berklee to Berkeley to get a sense of how similar their answer is. Two sample `.kml` files `path_flat_sol.kml` and `path_curved_sol.kml` are provided in the same directory and you can check your results against the samples.

You can use the following command to run all the tests on your Dijkstra's implementation:

```
python dijkstra_test.py
```

When your code passes all the tests and runs reasonably fast (the tests should complete in less than 40s when using CPython), upload your modified `dijkstra.py` to the course submission site. Our automated grading code will use our versions of `dijkstra_test.py`, `nhpn.py`, and `priority_queue.py`, so please do not modify these files.

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

Problem Set 7

Both theory and programming questions are due on **Tuesday, December 6** at **11:59PM**. Please download the .zip archive for this problem set. Instructions for submitting your answers will be posted to the course website by Tuesday, November 29.

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 guide by **Thursday, December 8, 11:59PM**. Your grade will be based on both your solutions and the grading explanation.

Problem 7-1. [30 points] **Seam Carving**

In a recent paper, "Seam Carving for Content-Aware Image Resizing", Shai Avidan and Ariel Shamir describe a novel method of resizing images. You are welcome to read the paper, but we recommend starting with the YouTube video:

`http://www.youtube.com/watch?v=vIFCV2spKtg`

Both are linked from the Problem Sets page on the class website. After you've watched the video, the terminology in the rest of this problem will make sense.

If you were paying attention around time 1:50 of the video, then you can probably guess what you're going to have to do. You are given an image, and your task is to calculate the best vertical seam to remove. A *vertical seam* is a connected path of pixels, one pixel in each row. We call two pixels *connected* if they are vertically or diagonally adjacent. The *best* vertical seam is the one that minimizes the total "energy" of pixels in the seam.

The video didn't spend much time on dynamic programming, so here's the algorithm:

Subproblems: For each pixel (i, j), what is the lower-energy seam that starts at the top row of the image, but ends at (i, j)?

Relation: Let `dp[i,j]` be the solution to subproblem (i, j). Then

```
dp[i,j] = min(dp[i,j-1],dp[i-1,j-1],dp[i+1,j-1]) + energy(i,j)
```

Analysis: Solving each subproblem takes $O(1)$ time: there are three smaller subproblems to look up, and one call to `energy()`, which all take $O(1)$ time. There is one subproblem for each pixel, so the running time is $\Theta(A)$, where A is the number of pixels, i.e., the area of the image.

Download `ps7_code.zip` and unpack it. To solve this problem set, you will need the Python Imaging Library (PIL), which you should have installed for Problem Set 4. If you wish to view your results, you will additionally need the Tkinter library.

In `resizeable_image.py`, write a function `best_seam(self)` that returns a list of coordinates corresponding to the cheapest vertical seam to remove, e.g., $[(5, 0), (5, 1), (4, 2), (5, 3), (6, 4)]$. You should implement the dynamic program described above in a bottom-up manner.

The class `ResizeableImage` inherits from `ImageMatrix`. You should use the following components of `ImageMatrix` in your dynamic program:

- `self.energy(i,j)` returns the energy of a pixel. This takes $O(1)$ time, but the constant factor is sizeable. If you call it more than once, you might want to cache the results.
- `self.width` and `self.height` are the width and height of the image, respectively.

Test your code using `test_resizable_image.py`, and submit `ResizeableImage.py` to the class website. You can also view your code in action by running `gui.py`. Included with the problem set are two differently sized versions of the same sunset image. If you remove enough seams from the sunset image, it should center the sun.

Also, please try out your own pictures (most file formats should work), and send us any interesting before/after shots.

Problem 7-2. [70 points] **HG Fargo**

You have been given an internship at the extremely profitable and secretive bank HG Fargo. Your immediate supervisor tells you that higher-ups in the bank are very interested in learning from the past. In particular, they want to know how much money they *could* have made if they had invested optimally.

Your supervisor gives you the following data on the prices[1] of select stocks in 1991 and in 2011:

Company	Price in 1991	Price in 2011
Dale, Inc.	\$12	\$39
JCN Corp.	\$10	\$13
Macroware, Inc.	\$18	\$47
Pear, Inc.	\$15	\$45

As a first step, you decide to examine what the optimal decision is for a couple of small examples:

(a) [5 points] If you had \$20 available to purchase stocks in 1991, how much of each stock should you have bought to maximize profits when you sell everything in 2011? Note that you do not need to invest all of your money — if it is more profitable to keep some as cash, you do not need to invest it.

(b) [5 points] If you had \$30 available to purchase stocks in 1991, how much of each stock should you have bought?

(c) [5 points] If you had \$120 available to purchase stocks in 1991, how much of each stock should you have bought?

Your supervisor asks you to write an algorithm for computing the best way to purchase stocks, given the initial money $total$, the number $count$ of companies with stock available, an array $start$ containing the prices of each stock in 1991, and an array end containing the prices of each stock in 2011. All prices are assumed to be positive integers.

There is a strong relationship between this problem and the knapsack problem. The knapsack problem takes four inputs: the number of different items $items$, the item sizes $size$ (all of which are integers), the item values $value$ (which may not be integers), and the size $capacity$ of the knapsack. The goal is to pick a subset of the items that fits inside the knapsack and maximizes the total value.

(d) [1 point] Which input to the knapsack problem corresponds to the input $total$ in the stock purchasing problem?

1. $items$ 2. $size$ 3. $value$ 4. $capacity$

[1] Note that for the purposes of this problem, you should ignore some of the intricacies of the real stock market. The only income you can make is from purchasing stocks in 1991, then selling those same stocks at market value in 2011.

(e) [1 point] Which input to the knapsack problem corresponds to the input $count$ in the stock purchasing problem?

1. $items$ 2. $size$ 3. $value$ 4. $capacity$

(f) [1 point] Which input to the knapsack problem corresponds to the input $start$ in the stock purchasing problem?

1. $items$ 2. $size$ 3. $value$ 4. $capacity$

(g) [1 point] Which input to the knapsack problem corresponds to the input end in the stock purchasing problem?

1. $items$ 2. $size$ 3. $value$ 4. $capacity$

(h) [6 points] Unfortunately, the algorithm for the knapsack problem cannot be directly applied to the stock purchasing problem. For each of the following potential reasons, state whether it's a valid reason not to use the knapsack algorithm. (In other words, if the difference mentioned were the only difference between the problems, would you still be able to use the knapsack algorithm to solve the stock purchasing problem?)

1. In the stock purchasing problem, there is a time delay between the selection and the reward.
2. All of the numbers in the stock purchasing problem are integers. The $value$ array in the knapsack problem is not.
3. In the stock purchasing problem, the money left over from your purchases is kept as cash, which contributes to your ultimate profit. The knapsack problem has no equivalent concept.
4. In the knapsack problem, there are some variables representing sizes of objects. There are no such variables in the stock purchasing problem.
5. In the stock purchasing problem, you can buy more than one share in each stock.
6. In the stock purchasing problem, you sell all the items at the end. In the knapsack problem, you don't do anything with the items.

Despite these differences, you decide that the knapsack algorithm is a good starting point for the problem you are trying to solve. So you dig up some pseudocode for the knapsack problem, relabel the variables to suit the stock purchasing problem, and then start modifying things. After a long night of work, you end up with a couple of feasible solutions. Unfortunately, there is a bit of a hard-drive error the next morning, and the files are all mixed up. You have recovered six different functions, from various states in your development process. The first function is the following:

```
STOCK(total, count, start, end)
1  purchase = STOCK-TABLE(total, count, start, end)
2  return STOCK-RESULT(total, count, start, end, purchase)
```

This is the function that you ran to get your results. The STOCK-TABLE function generates the table of subproblem solutions. The STOCK-RESULT function uses that to figure out which stocks to purchase, and in what quantities. Unfortunately, you have two copies of the STOCK-TABLE function and three copies of the STOCK-RESULT function. You know that there's a way to take one of each function to get the pseudocode for the original knapsack problem (with the names changed). You also know that there's a way to take one of each function to get the pseudocode for the stock purchases problem. You just don't know which functions do what.

Analyze each of the other five procedures, and select the correct running time. Recall that $total$ and $count$ are positive integers, as are each of the values $start[stock]$ and $end[stock]$. To make the code simpler, the arrays $start$, end, and $result$ are assumed to be indexed starting at 1, while the tables $profit$ and $purchase$ are assumed to be indexed starting at $(0, 0)$. You may assume that entries in a table can be accessed and modified in $\Theta(1)$ time.

(i) [1 point] What is the worst-case asymptotic running time of STOCK-TABLE-A (from Figure 1) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

(j) [1 point] What is the worst-case asymptotic running time of STOCK-TABLE-B (from Figure 2) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

(k) [1 point] What is the worst-case asymptotic running time of STOCK-RESULT-A (from Figure 3) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

```
STOCK-TABLE-A(total, count, start, end)
 1  create a table profit
 2  create a table purchase
 3  for cash = 0 to total
 4       profit[cash, 0] = cash
 5       purchase[cash, 0] = FALSE
 6       for stock = 1 to count
 7            profit[cash, stock] = profit[cash, stock − 1]
 8            purchase[cash, stock] = FALSE
 9            if start[stock] ≤ cash
10                 leftover = cash − start[stock]
11                 current = end[stock] + profit[leftover, stock]
12                 if profit[cash, stock] < current
13                      profit[cash, stock] = current
14                      purchase[cash, stock] = TRUE
15  return purchase
```

Figure 1: The pseudocode for STOCK-TABLE-A.

```
STOCK-TABLE-B(total, count, start, end)
 1  create a table profit
 2  create a table purchase
 3  for cash = 0 to total
 4       profit[cash, 0] = 0
 5       purchase[cash, 0] = FALSE
 6       for stock = 1 to count
 7            profit[cash, stock] = profit[cash, stock − 1]
 8            purchase[cash, stock] = FALSE
 9            if start[stock] ≤ cash
10                 leftover = cash − start[stock]
11                 current = end[stock] + profit[leftover, stock − 1]
12                 if profit[cash, stock] < current
13                      profit[cash, stock] = current
14                      purchase[cash, stock] = TRUE
15  return purchase
```

Figure 2: The pseudocode for STOCK-TABLE-B.

```
STOCK-RESULT-A(total, count, start, end, purchase)
 1  create a table result
 2  for stock = 1 to count
 3       result[stock] = 0
 4
 5  cash = total
 6  stock = count
 7  while stock > 0
 8       quantity = purchase[cash, stock]
 9       result[stock] = quantity
10       cash = cash − quantity · start[stock]
11       stock = stock − 1
12
13  return result
```

Figure 3: The pseudocode for STOCK-RESULT-A.

```
STOCK-RESULT-B(total, count, start, end, purchase)
 1  create a table result
 2  for stock = 1 to count
 3       result[stock] = FALSE
 4
 5  cash = total
 6  stock = count
 7  while stock > 0
 8       if purchase[cash, stock]
 9            result[stock] = TRUE
10            cash = cash − start[stock]
11       stock = stock − 1
12
13  return result
```

Figure 4: The pseudocode for STOCK-RESULT-B.

```
STOCK-RESULT-C(total, count, start, end, purchase)
 1  create a table result
 2  for stock = 1 to count
 3      result[stock] = 0
 4
 5  cash = total
 6  stock = count
 7  while stock > 0
 8      if purchase[cash, stock]
 9          result[stock] = result[stock] + 1
10          cash = cash − start[stock]
11      else
12          stock = stock − 1
13
14  return result
```

Figure 5: The pseudocode for STOCK-RESULT-C.

(l) [1 point] What is the worst-case asymptotic running time of STOCK-RESULT-B (from Figure 4) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

(m) [1 point] What is the worst-case asymptotic running time of STOCK-RESULT-C (from Figure 5) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

(n) [2 points] The recurrence relation computed by the STOCK-TABLE-A function is:

1. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s-1]\}$
2. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s-1] + end[s]\}$
3. $profit[c, s] = \max_q\{profit[c - q \cdot start[s], s-1] + q \cdot end[s]\}$
4. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s]\}$
5. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s] + end[s]\}$
6. $profit[c, s] = \max_q\{profit[c - q \cdot start[s], s] + q \cdot end[s]\}$

(o) [2 points] The recurrence relation computed by the STOCK-TABLE-B function is:

1. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s-1]\}$
2. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s-1] + end[s]\}$
3. $profit[c, s] = \max_q\{profit[c - q \cdot start[s], s-1] + q \cdot end[s]\}$
4. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s]\}$
5. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s] + end[s]\}$
6. $profit[c, s] = \max_q\{profit[c - q \cdot start[s], s] + q \cdot end[s]\}$

With this information, you should be able to figure out whether STOCK-TABLE-A or STOCK-TABLE-B is useful for the knapsack problem, and similarly for the stock purchasing problem. From there, you can figure out which of STOCK-RESULT-A, STOCK-RESULT-B, and STOCK-RESULT-C is best for piecing together the optimal distribution of stocks and/or items.

(p) [3 points] Which two methods, when combined, let you compute the answer to the knapsack problem?

1. STOCK-TABLE-A and STOCK-RESULT-A
2. STOCK-TABLE-A and STOCK-RESULT-B
3. STOCK-TABLE-A and STOCK-RESULT-C
4. STOCK-TABLE-B and STOCK-RESULT-A
5. STOCK-TABLE-B and STOCK-RESULT-B
6. STOCK-TABLE-B and STOCK-RESULT-C

(q) [3 points] Which two methods, when combined, let you compute the answer to the stock purchases problem?

1. STOCK-TABLE-A and STOCK-RESULT-A
2. STOCK-TABLE-A and STOCK-RESULT-B
3. STOCK-TABLE-A and STOCK-RESULT-C
4. STOCK-TABLE-B and STOCK-RESULT-A
5. STOCK-TABLE-B and STOCK-RESULT-B
6. STOCK-TABLE-B and STOCK-RESULT-C

With all that sorted out, you submit the code to your supervisor and pat yourself on the back for a job well done. Unfortunately, your supervisor comes back a few days later with a complaint from the higher-ups. They've been playing with your program, and were very upset to discover that when they ask what to do with \$1,000,000,000 in the year 1991, it tells them to buy tens of millions of shares in Dale, Inc. According to them, there weren't that many shares of Dale available to purchase. They want a new feature: the ability to pass in limits on the number of stocks purchaseable.

You choose to begin, as always, with a small example:

Company	**Price in 1991**	**Price in 2011**	**Limit**
Dale, Inc.	\$12	\$39	3
JCN Corp.	\$10	\$13	∞
Macroware, Inc.	\$18	\$47	2
Pear, Inc.	\$15	\$45	1

(r) [5 points] If you had \$30 available to purchase stocks in 1991, how much of each stock should you have bought, given the limits imposed above?

(s) [5 points] If you had \$120 available to purchase stocks in 1991, how much of each stock should you have bought, given the limits imposed above?

(t) [20 points] Give pseudocode for an algorithm STOCKLIMITED that computes the maximum profit achievable given a starting amount $total$, a number $count$ of companies with stock available, an array of initial prices $start$, an array of final prices end, and an array of quantities $limit$. The value stored at $limit[stock]$ will be equal to ∞ in cases where there is no known limit on the number of stocks. The algorithm need only output the resulting quantity of money, not the purchases necessary to get that quantity.

Remember to analyze the runtime of your pseudocode, and provide a brief justification for its correctness. It is sufficient to give the recurrence relation that your algorithm implements, and talk about why the recurrence relation solves the problem at hand.