Assignment 3: Recursion!

This assignment is all about recursive problem-solving. You've practiced writing recursive functions in Assignment 1 and in section, and now it's time to take those skills, combine them with the techniques we've covered over the past couple of lectures, and make some pretty impressive pieces of software.

We've chosen these problems because we think they're a great sampler of the different sorts of fundamental recursive techniques that we've explored. We hope that you find these problems interesting and get a better feel for what recursion can do!

Due Wednesday, February 6th at the start of class.

You are permitted to work on this assignment in pairs.

This assignment has four parts to it, and you have nine days to complete it. We recommend that you start this assignment early and make slow, steady progress on it throughout the week. Here's a recommended timetable for completing this assignment:

- Complete the Sierpinski Triangle within a day of this assignment going out.
- Complete Human Pyramids within three days of this assignment going out.
- Complete Shift Scheduling within six days of this assignment going out.
- Complete Riding Circuit within nine days of this assignment going out.

Recursive problem-solving can take a bit of time to get used to, and that's perfectly normal. Putting in an hour or two each day working on this assignment will give you plenty of time to adjust and gives you a comfortable buffer in case you get stuck somewhere.

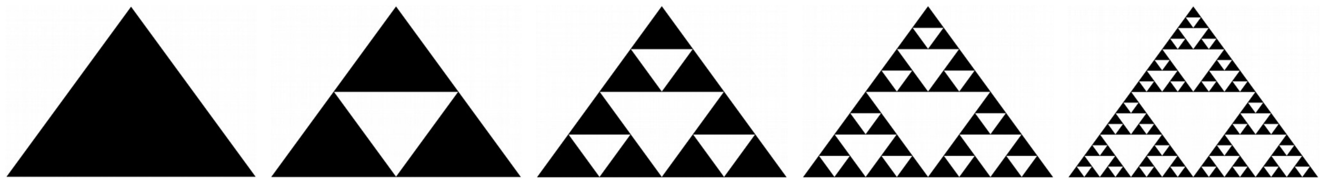
As always, feel free to get in touch with us if you need any assistance. We're happy to help out!

Problem One: The Sierpinski Triangle

One of the most famous self-similar fractals is the ***Sierpinski triangle***. As with most fractals, it's defined recursively:

- An order-0 Sierpinski triangle is a regular, filled triangle.
- An order-n Sierpinski triangle, where $n > 0$, consists of three Sierpinski triangles of order $n - 1$, each half as large as the main triangle, arranged so that they meet corner-to-corner.

For example, here are Sierpinski triangles of the first few orders:



Your task is to implement a function

```
void drawSierpinskiTriangle(GWindow& window,
                            GPoint p0, GPoint p1, GPoint p2,
                            int order)
```

that takes as input a window in which to draw the Sierpinski triangle, three points defining the corners of that triangle, and the order of that triangle, then draws that Sierpinski triangle of that order. We've provided you a helper function you can use to draw a triangle once you know what the coordinates of its corners; your job will mostly be to figure out where to draw those triangles.

The midpoint of a line segment whose endpoints are (x_1, y_1) and (x_2, y_2) is located at $\left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)$. You may find this useful when determining where the corners of the smaller triangles.

You can test your code by selecting the "Interactive Sierpinski" option from the starter files. That will let you drag around the corners of the triangle and adjust the order to see how your code behaves. Once you've gotten that working, click "Sierpinski Bungee Jump" to see just how deep the recursion rabbit hole goes.

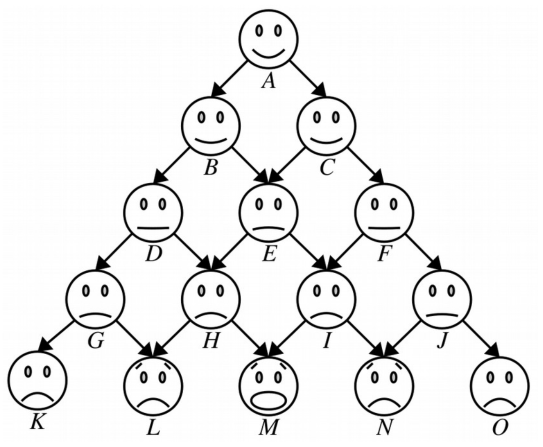
And yes, that's your code powering the bungee jump. All we're doing is specifying coordinates. 😃

Some notes on this problem:

- In the Crystals portion of Assignment 2, we used the `Point` type to represent integer coordinates in space. Here, we're using the `GPoint` type, which represents real-valued coordinates in space. Make sure not to mix the two up – if you do, you'll get a bunch of rounding errors that will make your triangle look kinda wonky.
- The Sierpinski triangle of order n is undefined when $n < 0$. In that case, use the `error()` function to report an error.
- *Draw pictures!* The meat of this part of the assignment is figuring out where each triangle's corners are, and that's much easier to do with a schematic in front of you.

Problem Two: Human Pyramids

A human pyramid is a way of stacking people vertically. With the exception of the people in the bottom row, each person splits their weight evenly on the two people below them in the pyramid. For example, in the pyramid to the right, person *A* splits her weight across people *B* and *C*, and person *H* splits his weight – plus the accumulated weight of the people he's supporting – onto people *L* and *M*. It can be mighty uncomfortable to be in the bottom row, since you'll have a lot of weight on your back! In this question, you'll explore just how much weight that is. Just so we have nice round numbers here, let's assume that everyone in the pyramid weighs exactly 160 pounds.



Person *A* at the top of the pyramid has no weight on her back. People *B* and *C* each carry half of person *A*'s weight, so each shoulders 80 pounds. Uncomfortable, but not too bad.

Now, look at the people in the third row. Focus on person *E*. How much weight is she supporting? Well, she's directly supporting half the weight of person *B* (80 pounds) and half the weight of person *C* (80 pounds). On top of this, she's feeling some of the weight people *B* and *C* are carrying. Half of the weight that person *B* shoulders (40 pounds) gets transmitted to person *E* and half the weight person *C* shoulders (40 pounds) similarly gets sent down to person *E*, so person *E* ends up feeling an extra 80 pounds. That means she's supporting a net total of 240 pounds. That's going to be noticeable!

Not everyone in that third row is feeling the same amount, though. Look at person *D* for example. The only weight on person *D* comes from person *B*. Person *D* therefore ends up supporting

- half of person *B*'s body weight (80 pounds), plus
- half of the weight person *B* is holding up (40 pounds),

so person *D* ends up supporting 120 pounds, only half of what *E* is feeling! Going deeper in the pyramid, how much weight is person *H* feeling? Well, person *H* is supporting

- half of person *D*'s body weight (80 pounds),
- half of person *E*'s body weight (80 pounds), plus
- half of the weight person *D* is holding up (60 pounds), plus
- half of the weight person *E* is holding up (120) pounds.

The net effect is that person *H* is carrying 340 pounds – ouch! A similar calculation shows that person *I* is also carrying 340 pounds – can you see why? Compare this to person *G*. Person *G* is supporting

- half of person *D*'s body weight (80 pounds), plus
- half of the weight person *D* is holding up (60 pounds)

for a net total of 140 pounds. That's a lot, but it's not nearly as bad as what person *H* is feeling! Finally, let's look at poor person *M* in the middle of the bottom row. How is he doing? Well, he's supporting

- half of person *H*'s body weight (80 pounds),
- half of person *I*'s body weight (80 pounds),
- half of the weight person *H* is holding up (170 pounds), and
- half of the weight person *I* is holding up (170 pounds),

for a net total of 500 pounds. Yikes! No wonder he looks so unhappy.

There's a nice, general pattern here that lets us compute how much weight is on each person's back:

- Each person weighs exactly 160 pounds.
- Each person supports half the body weight of each of the people immediately above them, plus half of the weight that each of those people are supporting.

Using this general pattern, write a recursive function

```
int weightOnBackOf(int row, int col);
```

that takes as input the row and column number of a person in a human pyramid, then returns the total weight on that person's back. The row and column are each zero-indexed, so the person at row 0, column 0 is on top of the pyramid, and person *M* in the above picture is at row 4, column 2. For example, `weightOnBackOf(1, 1)` would return 80 pounds (since person *C* is shouldering 80 pounds), and `weightOnBackOf(4, 2)` should return 500 (since person *M* is shouldering a whopping 500 pounds).

If the input row and column are out of bounds – either because the row is negative or the column index is invalid – you should report an error by calling the `error()` function.

Your implementation of `weightOnBackOf` must be implemented recursively and must not use any loops. Before moving on, ***test your solution thoroughly!*** Feel free to write additional tests if you'd like.

Speeding Things Up

When you first code up this function, you'll find that it's pretty quick to tell you how much weight is on the back of the person in row 5, column 3, but that it takes a long time to say how much weight is on the back of the person in row 30, column 15. Why is this?

Think about what happens if we call `weightOnBackOf(30, 15)`. This makes two new recursive calls: one to `weightOnBackOf(29, 14)`, and one to `weightOnBackOf(29, 15)`. This first recursive call in turn fires off two more: one to `weightOnBackOf(28, 13)`, and another to `weightOnBackOf(28, 14)`. The second recursive call then calls `weightOnBackOf(28, 14)` and `weightOnBackOf(28, 15)`.

Notice that there are two calls to `weightOnBackOf(28, 14)` here. This means that there's a redundant call being made to `weightOnBackOf(28, 14)`, so all the work done to compute that intermediate answer is done twice. That call will in turn fire off its own redundant recursive calls, which in turn fire off their own redundant calls, etc. This might not seem like much, but the number of recursive calls can be huge. For example, calling `weightOnBackOf(30, 15)` makes over 600,000,000 recursive calls!

There are many techniques for eliminating redundant calls. One common approach is ***memoization*** (no, that's not a typo). Intuitively, memoization works by making an auxiliary table keeping track of all the recursive calls that have been made before and what value was returned for each of them. Then, whenever a recursive call is made, the function first checks the table before doing any work. If the the recursive call has already been made in the past, the function just returns that stored value. This prevents values from being computed multiple times, which can dramatically speed things up!

In pseudocode, memoization looks something like this:

```
// ========== Before ========== //
Ret recursiveFunction(Arg a) {
   if (base-case-holds) {
      return base-case-value;
   } else {
      do-some-work;
      return recursive-step-value;
   }
}
```

```
// ================ After ================ //
Ret recursiveFunction(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}
```

As a final step, modify `weightOnBackOf` so that it uses memoization to avoid recomputing values unnecessarily. The `weightOnBackOf` function must still take the same arguments as before, since our starter code expects to be able to call it with just two arguments, so you may need to make it a wrapper.

Once you've done that, try comparing how long it takes to evaluate `weightOnBackOf(40, 20)` both with and without memoization. Notice a difference? For fun, try computing `weightOnBackOf(200, 100)`. This will take a staggeringly long time to complete without memoization – the sun will probably burn out before you get an answer – but with memoization you should get back an answer instantly!

Problem Three: Shift Scheduling

You're in charge of a small business and have just hired a new part-time employee with flexible hours to help out. You'd like to schedule their hours in a way that gets them to produce the most value.

For each shift, you have an estimate of how much money the employee would produce if they worked that shift. Ideally, you'd have them work *every* shift, but alas, that isn't possible. There are two restrictions. First, the employee has a limited number of hours they're allowed to work each week. Second, you can't assign the employee to work concurrent shifts. (As great a hire as she was, she can only be in one place at one time). Given these restrictions, what shifts should you assign to your new hire?

For example, imagine that your new employee is available for twenty hours each week. There's a variety of tasks she could do, each of which has different shifts. Here's the different shifts she could take, as well as how much value (measured in dollars) she'll produce in each:

```
Mon  8AM - 12PM: $27          Tue  8AM - 12PM: $7           Wed  8AM - 12PM: $10
Mon 12PM -  4PM: $28          Tue 12PM -  4PM: $7           Wed 12PM -  4PM: $11
Mon  4PM -  8PM: $25          Tue  4PM -  8PM: $11          Wed  4PM -  8PM: $13
Mon  8AM -  2PM: $39          Tue  8AM -  2PM: $10          Wed  8AM -  2PM: $19
Mon  2PM -  8PM: $31          Tue  2PM -  8PM: $8           Wed  2PM -  8PM: $25
```

So, when should she come in to work? You might be eyeing the high-value Monday 8:00AM – 2:00PM and 2:00PM – 8:00PM time slots. That would bring in a full $70, using up twelve of her permitted twenty hours. You might then have her spend her remaining eight hours Wednesday afternoon in the 12:00PM – 4:00PM and 4:00PM – 8:00PM shifts, netting you another $24 for a total take-home of $94.

With a little creativity, though, you might note that you actually would make more if instead of having her work eight hours on Wednesday in two four-hour time slots, you could instead let her work six hours from 2:00PM – 8:00PM and bring in $25 of value, simultaneously upping your total take-home to $95 and giving her two free hours.

Even this isn't optimal – as enticing as those $39 and $31 time slots on Monday are, it's actually better to have your employee work the three four-hour shifts on Monday, bringing in $27, $28, and $25, respectively, for a total of $80 on Monday. Overall, if you pick the three four-hour shifts on Monday and the closing six-hour shift on Wednesday, your new hire is bringing in $105 and clocking in only eighteen hours, below the twenty-hour limit.

Your task is to write a function

```
Set<Shift> highestValueScheduleFor(const Set<Shift>& shifts, int maxHours);
```

that takes as input a `Set` of all the possible shifts, along with the maximum number of hours the employee is allowed to work for, then returns which shifts the employee should take in order to generate the maximum amount of value. Here, `Shift` is a `struct` containing information about when the shift is (what day of the week it is, when it starts, and when it ends). It's defined in the header file `Shift.h`. Chances are that you won't need to read any of the fields of this `struct`, and that instead you'll want to use these functions from `Shift.h`:

```
int lengthOf(const Shift& shift);     // Returns the length of a shift.

int valueOf(const Shift& shift);      // Returns the value of this shift.

bool overlapsWith(const Shift& one,   // Returns whether two shifts overlap.
                  const Shift& two);
```

(Continued on the next page...)

Before you write any code for this problem, think about the answers to the following questions. The biggest challenge in solving this problem is making sure that you've picked the right problem to solve.

- In lecture, we saw how to list subsets, permutations, and combinations. Of those options, which one looks like the closest fit for this problem? What recursive insights are useful for working with those types of objects?
- Is this an *enumeration* problem (list all objects of a certain type), an *optimization* problem (find the best object of some type), or a *backtracking* problem (see whether an object of some type exists?) Based on that, think about what the shape of your recursion might look like.
- This will be your first time doing any kind of recursive exploration. How exciting! If you need some suggestions for how to do this, take a look at this week's section handout, which includes a couple of problems along these lines.

Once you've answered those questions, keep the following in mind as you start writing up your solution:

- ***Test as you go!*** We've included a few test cases in the `ShiftScheduling.cpp` file, but as usual we haven't considered every possible case. Add in a few of your own test cases as well, and proceed slowly! Get out a sheet of paper if needed and work through a few examples.
- If multiple schedules are tied for producing the most value, you're free to choose any of them.
- The employee cannot work two shifts that have any amount of overlap, though you are allowed to assign one shift that starts as soon as another ends.
- The best schedule might not actually use up all of the employee's available hours. That's fine.
- The values assigned to shifts don't necessarily need to follow any patterns. For example, it might be the case that one shift from 12:00PM to 4:00PM produces 10 units of value while another shift from 10:00AM to 6:00PM on the same day only produces 4. (There are many real-life scenarios where this could happen. Maybe those shifts are spent performing different tasks. Maybe they're the same task, but the cost of paying the employee to stay around longer eats into profits.)
- The `maxHours` parameter should be nonnegative. It's fine if the worker is allotted zero available hours – that just means that they're really busy that week – but if `maxHours` is negative you should call `error()` to report an error.

Once you've finished writing your solution, try running our demo program, which will let you generate random weights on some standard shift sets and see what schedules end up getting chosen.

Here's something to think about. Some companies assign different shifts to workers each week, and they do so by running code that's pretty similar to the code that you wrote here based on weekly forecasts. As you can see from the demo program, this can easily lead to schedules that vary wildly from week to week. If the person you're hiring needs to arrange for child care each week, this rapid swing in shift selection can be extremely stressful. In fact, this has been so rough on workers that it made national news.

So take a look at the code that you wrote. We assume that nothing in your code says something like this:

```
if (isSingleMother()) {
    chooseShiftsArbitrarilyAndCapriciously();
} else {
    chooseStableShifts();
}
```

Instead, what's happening here is that your code is designed to purely optimize a single quantity – the amount of revenue generated by the employee – and doesn't take anything else into account. Your code isn't malicious. It just isn't aware of the surrounding context.

The lesson here is to *be careful about what you want to optimize*. Computers can help you find strategies that maximize lots of different quantities. Make sure that you pick a quantity that takes in the perspectives of all the stakeholders in a situation. Get it right and you end up with schedules where workers are happy and companies make money. Get it wrong and you can significantly impair quality of life.

Problem Four: Riding Circuit

Federal courts in the United States are organized into a hierarchy. Cases are initially heard in district courts; Stanford's is the District Court for the Northern District of California. Should the parties wish to appeal, they can do so, and the case is escalated to a court of appeals. These courts are divided up into different geographic regions called *circuits*; Stanford sits in the Ninth Circuit. Should different courts of appeals reach differing conclusions on similar cases, it's called a *circuit split* and the Supreme Court often then takes up the case to resolve the inconsistency.

It might seem odd to call a geographical region of the US a "circuit." The reason is historical. It used to be the case that justices from the Supreme Court were assigned a particular geographical region of the US and would physically travel around that region hearing cases. This is called *riding circuit* and is still continued by judges in rural areas, where the population density isn't high enough to sustain fixed courthouses, to this day.

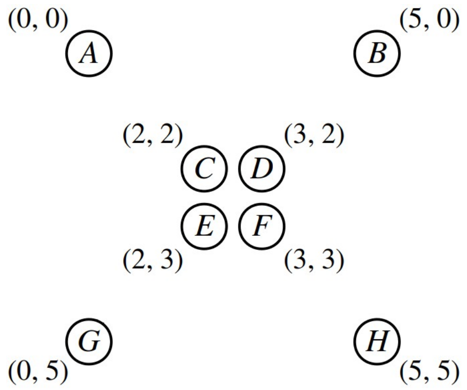
Let's imagine that you've been assigned to ride circuit through a collection of cities. You start off in one of the cities, then need to take a trip that will route you through each of the other cities, returning back to where you started. Depending on the order in which you visit the other cities, you might spend different amounts of time traveling. The question you'll want to answer is this one: what route should you take to minimize the total travel time?

For the purposes of this problem, we'll imagine that each city has a designated (x, y) coordinate, and the further apart two cities are, the longer it will take to travel between them. To keep the math simple, we'll measure the distances between two cities using what's called the ***Manhattan distance***. The Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is defined as

$$|x_1 - x_2| + |y_1 - y_2|.$$

That is, it's the sum of the change in x coordinates and the change in y coordinates as you move from one city to the next. (The name "Manhattan distance" arises from moving between locations in New York City, where you can't travel in a straight line and must instead follow streets and avenues.)

As an example, suppose you want to visit these eight cities:



If you start at city A, one option would be to visit the cities in the order A, C, D, B, H, F, E, G, and then back to A. That would require traveling a total distance of 4 + 1 + 4 + 5 + 4 + 1 + 4 + 5 = 28. But you can do better than this: taking the route A, B, H, G, E, F, D, C, and then back to A travels a total distance of 5 + 5 + 5 + 4 + 1 + 1 + 1 + 4 = 26, and that's the best you can do.

Your task is to write a function

```
Vector<Point> bestCircuitThrough(const Vector<Point>& points);
```

that takes as input a list of all the cities you need to visit (represented here as `Point` objects) and returns a `Vector<Point>` containing the sequence in which you should visit those cities to minimize your total travel time. You're allowed to start anywhere you'd like; since you're traveling a closed loop, it doesn't actually matter where you begin – do you see why?

Before you start off on this problem, we recommend thinking through the following:

- In lecture, we saw how to list subsets, permutations, and combinations. Of those options, which one looks like the closest fit for this problem? What recursive insights are useful for working with those types of objects?
- Is this an *enumeration* problem (list all objects of a certain type), an *optimization* problem (find the best object of some type), or a *backtracking* problem (see whether an object of some type exists?) Based on that, think about what the shape of your recursion might look like.

Here are some specific notes on this problem:

- ***Test as you go!*** We've included a few test cases in the `RidingCircuit.cpp` file, but as usual we haven't considered every possible case. Add in a few of your own test cases as well, and proceed slowly! Get out a sheet of paper if needed and work through a few examples.
- The `Vector<Point>` you return should contain exactly one copy of each city from the list. We'll implicitly assume that after visiting all the cities in the list in that order, you'll return back to the first city in the list. For example, the `Vector` [A, B, C, D, E] corresponds to starting at A, going from A to B, going from B to C, going from C to D, going from D to E, and then returning from E back to A, even though there isn't a second copy of A at the end of the list.
- It's possible that there won't be any cities to visit, in which case you should hand back an empty list. It's also possible that there's exactly one city to visit, in which case you should hand back a list consisting of just that city.
- If there are multiple equally good paths to choose from, you can return any one of them. In fact, there will almost *always* be multiple equally good paths to choose from, since given any cycle you can start anywhere in that cycle, or reverse that cycle, and the total length won't change.
- There are a few tricky bits of edge-case logic you'll need to address because you're working with a closed circuit. You might find it easier to first solve this problem under the assumption that you don't need to return back to where you started at the end of your journey. Then, once that's working, add in some modifications to close the cycle.
- As you're designing test cases, keep in mind that as the number of cities increases, the number of possible circuits goes up really, really quickly. We recommend keeping the number of cities small – say, at most 10.

Our provided demo for this part of the assignment lets you load in collections of cities and to plot out the route that your code generates. It can take a short while for your code to compute the optimal route – there are a *very* large number of routes to pick from! – so don't worry too much if that's the case. However, if you find that it's taking more than a minute to plot a route through any of the sample demos, that might indicate that your solution is doing more work than is necessary. In that case, feel free to stop by the LaIR to ask questions.

Problem Five: (Optional) Extensions!

Want more to explore? Here are a few suggestions to help you get started. As always, if you're planning on doing extensions, please submit two versions of your assignment – one with extensions and one without – so that grading goes a bit more smoothly.

- ***Sierpinski Triangle:*** The Sierpinski triangle is only one of many self-similar images; consider coding up another!

  Although this assignment is all about recursion, if you're up for a challenge, try replacing the recursive version of your code to draw an order-*n* Sierpinski triangle with an iterative function that draws an order-*n* Sierpinski triangle.
- ***Human Pyramids:*** There's another way to avoid making multiple recursive calls called *dynamic programming* that's equivalent to memoization, but uses iteration rather that recursion. Look up dynamic programming and try coding this function up both ways. Which one do you find easier?
- ***Shift Scheduling:*** The discussion at the end of the shift scheduling problem points out a weakness in how that algorithm works. What changes might you make to the shift scheduler to make it produce more stable schedules from week to week? How do you balance workers' needs for predictability with management's goal of increasing profits?

  If you're careful with how you set up the recursion, you can actually add memoization into this problem and dramatically speed things up. This will only work if you find yourself encountering the exact same subproblems over and over again. See if you can find a way to make this work!
- ***Riding Circuit:*** This problem is a variation on a famous problem called the *traveling salesperson problem* (TSP) that comes up all the time in operations research and CS theory. Read up on some of the techniques used to solve TSP efficiently. There's a very beautiful approach that uses memoization to dramatically speed up a solution compared to brute-force. For a real challenge, look up the Christofides algorithm, which gives an approximate answer that's always within a factor of $^3/_2$ of the optimal solution but does so extremely quickly.

General Notes

Here's some clarifications, notifications, expectations, and recommendations for this assignment.

- ***Your functions must actually use recursion***; it defeats the point of the assignment to solve these problems iteratively! That being said, as you've seen in lecture, it's perfectly alright for recursive functions to contain loops. The main question is whether your solution fundamentally works by breaking problems down into smaller copies of themselves. If so, great! If not, you may want to revisit your solution.
- ***Test your code thoroughly!*** We've included some test cases with the starter files, but they aren't exhaustive. Be sure to add tests as you go!
- ***Recursion is tricky***, so don't be dismayed if you can't immediately sit down and solve these problems. We've allowed you to work in pairs on this assignment so that you can discuss these problems with a partner, which can be a great way to help build an intuition for the concepts. Please feel free ask for advice and guidance if you need it. Once everything clicks, you'll have a much deeper understanding of just how cool a technique this is. We're here to help you get there!

Submission Instructions

Once you've worked through the Assignment Submission Checklist to make sure your code is ready to go, please submit `Sierpinski.cpp`, `HumanPyramids.cpp`, `ShiftScheduling.cpp`, and `RidingCircuit.cpp` at https://paperless.stanford.edu. If you edited any other files, please include them as well, but otherwise please just submit these four files. And that's it! You're done!

Good luck, and have fun!