YEAH! Hours - Recursion

Part 1 - Sierpinski Triangle

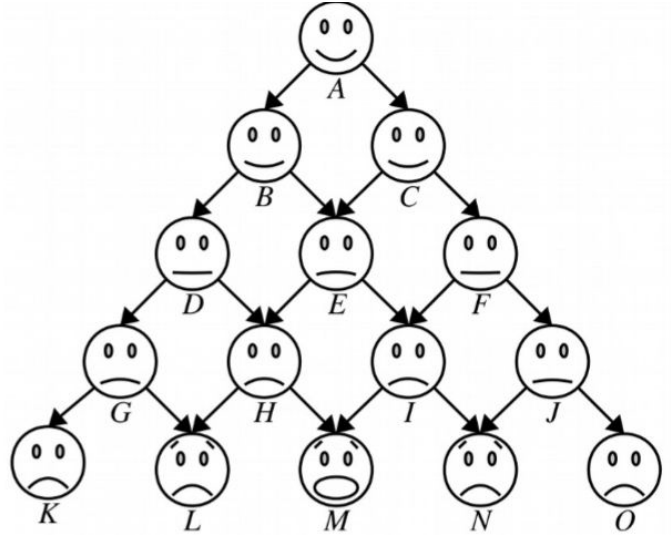
- Order 0 triangle is just a filled in triangle
- Order n triangle consists of three n-1 triangle, half as large as the main triangle, each

Part 1 - Sierpinski Triangle

- You are given the helper function to actually draw the triangle once you know the coordinates
- Your job: recursively determine the coordinates for the triangles in an order n Sierpinski triangle
- Tips: Draw pictures to help you determine how to calculate the coordinates, don’t guess until you get the coordinates right!

Part 2 - Human Pyramids

- Each person splits their weight with evenly with the two people below them on the pyramid.
- Person A (Coordinates 0,0): 0 pounds
- Person B (1,0): ½ of A
- Person E (2,1): ½ of B + ½ of C (which includes weight added from person A)
- Notice those on the edges will have less total weight than those in the middle
    - For example, person D is only holding up person B, while person E is holding up people C and B.



Part 2 - Human Pyramids

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

- Each Person weighs 160 pounds
- If the given row/col is out of bounds, throw an error
- Tips: Write additional tests!

- Write out what some of the support weights would be, it will help you find the recursive pattern!

Part 2 - Human Pyramids: A (mandatory) optimization

- Calculating the weight supported by each person will take a *long* time without memoization; your code **will** timeout
- Memoization: Allows you to avoid redundant recursive calls and significantly speeds up your algorithm, especially when the size of the pyramid increases.
- Use a table to keep track of what weights you have already calculated, your new function will look something like this:

```
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;
    }
}
```

Part 3 - Shift Scheduling

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

- Your Job: given a set of possible shifts accompanied by value they produce, optimize the business's revenue
- Shifts cannot overlap; the worker cannot work more than maxHours, but can work less.
- Some functions of the struct Shift (as defined in struct.h) that you may find useful

```
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);
```

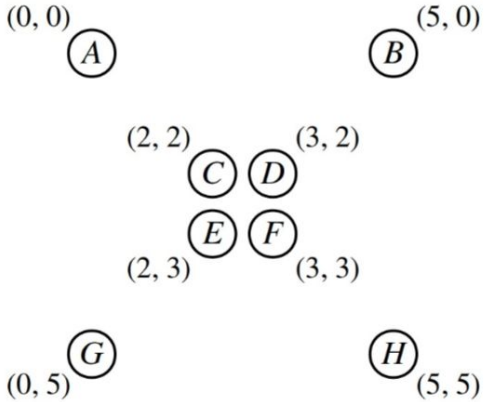
Part 3 - Shift Scheduling

- What type of recursive problem is this? Think about what type of problem it is *before* you try to solve it
- Look to lecture and section examples for some good *suggestions* on how to tackle this problem
- **Test as you go**
- The optimal shift schedule does not necessarily have the worker working for maxHours or working the shift with the highest values

Part 4 - Riding Circuit

- Your job: Minimize travel time using Manhattan distance

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



Part 4 - Riding Circuit

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

- Find the path with the smallest total Manhattan distance
- You must visit every point in the points Vector, but you can visit them in any order
- You can start at any point
- Your return Vector should only contain one copy of each point, even though you will technically be returning to the starting point, do not add it twice. Keep this in mind while designing your recursive solution
- There will be multiple optimal paths- choose any one of them.

In General:

- You **must** use recursion for every problem
- Test often, add tests of your own, be wary of edge cases and how optimizations will affect your solution
- Start early and use the LAIR!