Assignment 3: Recursion!

Parts of this handout were written by Julie Zelenski, Jerry Cain, and Marty Stepp.

Thanks to Anton Apostolatos for building out the starter files for this assignment and to the CS106B Section Leaders for preflighting this assignment!

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

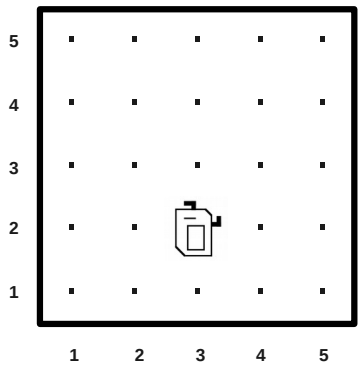
We've chosen these problems because we think they're a great sampler of the different sorts of fundamental recursive techniques that we've explored. There are six total problems here, two of which are warm-up assignments that we'll release solutions to on Wednesday, and four of which are the actual assignment that needs to be submitted for credit. We hope that you find these problems interesting and get a better feel for what recursion can do – it's a powerful and beautiful technique once you get the hang of it!

Due Monday, February 6 at the start of class.

You are permitted to work on this assignment in pairs.

Warm-Up Problem 0A: Karel Goes Home

As most of you know, Karel the Robot lives in a world composed of streets and avenues laid out in a regular rectangular grid that looks like this:



Suppose that Karel is sitting on the intersection of 2nd Street and 3rd Avenue as shown in the diagram and wants to get back to the origin at 1st Street and 1st Avenue. Even if Karel wants to avoid going out of the way, there are still several equally short paths. For example, in this diagram there are three possible routes, as follows:

- Move left, then left, then down.
- Move left, then down, then left.
- Move down, then left, then left.

Your job in this problem is to write a recursive function

```
int numPathsHome(int street, int avenue)
```

that returns the number of paths Karel could take back to the origin from the specified starting position, subject to the condition that Karel doesn't want to take any unnecessary steps and can therefore only move west or south (left or down in the diagram). Watch out for the edge case where the street or avenue numbers are zero or below!

Warm-Up Problem 0B: Subsequences

If *S* and *T* are strings, we say that *S* is a subsequence of *T* if all of the letters of *S* appear in *T* in the same relative order that they appear in *S*. For example, `pin` is a subsequence of the **`p`**`rogramm`**`in`**`g`, and the string `singe` is a subsequence of **`s`**`pr`**`ing`**`tim`**`e`**. However, `steal` is not a subsequence of `least`, since the letters are in the wrong order, and `i` is not a subsequence of `team` because there is no `i` in `team`. The empty string is a subsequence of every string, since all 0 characters of the empty string appear in the same relative order in any arbitrary string.

Write a function

```
bool hasSubsequence(const string& text, const string& subseq)
```

that accepts as input two strings and returns whether the second string is a subsequence of the first. Your solution should be recursive and must not use any loops (e.g. while, for). This problem has a beautiful recursive decomposition. As a hint, try thinking about the following:

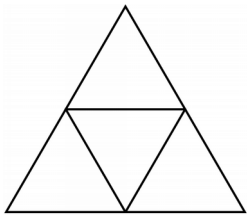
- What strings are subsequences of the empty string?
- What happens if the first character of the subsequence matches the first character of the text? What happens if it doesn't?

You can assume that the strings are case-sensitive, so `AGREE` is not a subsequence of `agreeable`.

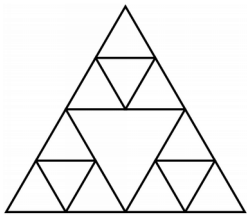
The remaining problems in this assignment should be submitted for credit.

Problem One: The Sierpinski Triangle

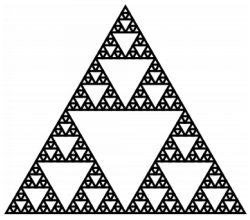
If you search the web for fractal designs, you will find many intricate wonders beyond the fractal tree we coded up in lecture. One of these is the ***Sierpinski Triangle***, named after its inventor, the Polish mathematician Wacław Sierpiński. The order-0 Sierpinski Triangle is just a regular equilateral triangle. An order-*n* Sierpinski triangle consists of three order-(*n*-1) Sierpinski triangles, each half as large as the original, arranged in the corners of a larger triangle. For example, the order-1 Sierpinski triangle is shown below. It consists of three order-0 Sierpinski triangles (each of which is an equilateral triangle) arranged in the corners of a larger triangle.



The downward-pointing triangle in the middle of this figure is not drawn explicitly, but is instead formed by the sides of the other three triangles. That area is not recursively subdivided and remains unchanged at every level of the fractal, so the order-2 Sierpinski Triangle has the same open area in the middle:



The order-5 Sierpinski triangle is shown here:



Your task is to implement the function

```
void drawSierpinskiTriangle(GWindow& window, double x, double y,
                            double sideLength, int order)
```

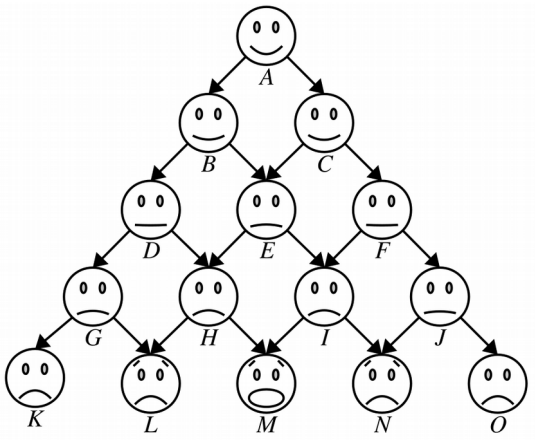
that takes as input the (*x*, *y*) coordinates of the *bottom-left corner* of the triangle, the length of one side of the triangle, and the order of the fractal, then draws a Sierpinski triangle of that order in the window.

Although you might expect this assignment is going to require a lot of math with sines and cosines, if you structure your program correctly you won't need to do any explicit trigonometry. Take advantage of the fact that the the `drawPolarLine` member function of `GWindow` returns a `GPoint` indicating where the line ends (see the fractal tree example from lecture for an example). Consider having your recursive function not only draw the Sierpinski triangle, but also return the coordinate of the topmost point of the triangle.

Also, don't forget that in our coordinate system, *x* values increase from left to right (as in the Cartesian plane), but *y* values increase from top to bottom (the opposite of the regular Cartesian plane). The arguments to `drawPolarLine` are specified in *degrees*, not radians, and although *y* increases top-to-bottom in our coordinate system, an angle of 60° points upward toward the top of the window, not downward.

Problem Two: Human Pyramids

A human pyramid is a way of stacking people vertically in a triangle. 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 200 pounds.



Person *A* at the top of the pyramid has no weight on her back. People *B* and *C* are each carrying half of person *A*'s weight. That means that each of them are shouldering 100 pounds. Slightly uncomfortable, but not too bad.

Now, let's look at the people in the third row. Let's begin by focusing on person *E*. How much weight is she supporting? Well, she's directly supporting half the weight of person *B* (100 pounds) and half the weight of person *E* (100 pounds), so she's supporting at least 200 pounds. On top of this, she's feeling some of the weight that people *B* and *C* are carrying. Half of the weight that person *B* is shouldering (50 pounds) gets transmitted down onto person *E* and half the weight that person *C* is shouldering (50 pounds) similarly gets sent down to person *E*, so person *E* ends up feeling an extra 100 pounds. That means she's supporting a net total of 300 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 (100 pounds), plus
- half of the weight person *B* is holding up (50 pounds),

so person *D* ends up supporting 150 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 (100 pounds),
- half of person *E*'s body weight (100 pounds), plus
- half of the weight person *D* is holding up (75 pounds), plus
- half of the weight person *E* is holding up (150) pounds.

The net effect is that person *H* is carrying 425 pounds – ouch! A similar calculation shows that person *I* is also carrying 425 pounds – can you see why? Compare this to person *G*. Person *G* is supporting

- half of person *D*'s body weight (100 pounds), plus
- half of the weight person *D* is holding up (75 pounds)

for a net total of 175 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 she doing? Well, she's supporting

- half of person *H*'s body weight (100 pounds),
- half of person *I*'s body weight (100 pounds),
- half of the weight person *H* is holding up (212.5 pounds), and
- half of the weight person *I* is holding up (215.5 pounds),

for a net total of 625 pounds. Yikes! No wonder she 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 200 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

```
double 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 100 pounds (since person *C* is shouldering 100 pounds on her back), and `weightOnBackOf(4, 2)` should return 625 (since person *M* is shouldering a whopping 625 pounds on her back).

Your implementation of `weightOnBackOf` must be implemented recursively and must not use any loops (`for`, `while`, or `do ... while`). We hope that you'll be pleasantly surprised how little code is required!

Speeding Things Up

When you first code up this function, you'll likely 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 tell you how much weight is on the back of the person in row 30, column 15. Why is this?

Think about what happens if we make a call to `weightOnBackOf(30, 15)`. This will make two new recursive calls: one to `weightOnBackOf(29, 14)`, and one to `weightOnBackOf(29, 15)`. This first recursive call will in turn fire off two of its own calls: one to `weightOnBackOf(28, 13)`, and another to `weightOnBackOf(28, 14)`. The second recursive call fires of two calls: a first recursive call to `weightOnBackOf(28, 14)`, and second one `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 a whopping 601,080,389 recursive calls!

There are a number of techniques for eliminating redundant calls. One common approach is a technique called ***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;
   }
}
```

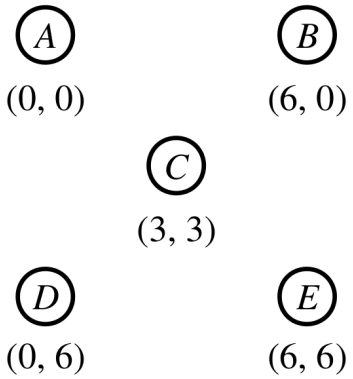
```
// ================ After ================ //
Ret recursiveFunction(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;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

As a final step in this part of the assignment, once you've gotten everything working, modify your function so that it uses memoization to avoid recomputing values unnecessarily. However, 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. You will likely need to make `weightOnBackOf` a wrapper function in order to do this.

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 – so long, in fact, that the sun will probably burn out before you get an answer – but with memoization you should get back an answer extremely quickly!

Problem Three: Drill, Baby, Drill!

There's a robot on an assembly line whose job is to take a steel plate and drill a number of holes in it at specific (*x*, *y*) coordinates. For example, consider the plate given here, where the units are in centimeters:



Imagine the drill starts at position (0, 0). If the robot drills the holes in the order *A*, *B*, *C*, *D*, *E*, then the drill will

- drill a hole at *A*, then move 6cm from *A* to *B*;
- drill a hole at *B*, then move roughly 4.24cm from *B* to *C*;
- drill a hole at *C*, then move roughly 4.24cm from *C* to *D*;
- drill a hole at *D*, then move 6cm from *D* to *E*; then
- drill a hole at *E*, then move roughly 8.49cm to reset the drill to the starting point of *A*.

This ends up moving the drill a total of (about) 29cm in the course of a single cycle. On the other hand, if the robot drills the holes in the order *C*, *A*, *B*, *E*, *D*, then the drill will

- drill a hole at *C*, then move roughly 4.24cm from *C* to *A*;
- drill a hole at *A*, then move roughly 6cm from *A* to *B*;
- drill a hole at *B*, then move roughly 6cm from *B* to *E*;
- drill a hole at *E*, then move 6cm from *E* to *D*; then
- drill a hole at *D*, then move roughly 4.24cm to reset the drill to the starting point of *C*.

This ends up moving the drill a total of (about) 26.5cm. If drilling holes is fast and moving the robot is slow, then this approach is faster than the initial approach and could save a lot of time on the shop floor.

Imagine there's a `struct` representing a drill site:

```
struct DrillSite {
    string name;    // The name of the drill site
    GPoint pt;      // Where it is
};
```

Your task is to write a function

```
Vector<DrillSite> bestDrillRouteFor(const Vector<DrillSite>& sites);
```

that takes as input a list of the drill sites for a job, then returns the optimal order in which the robot should drill the holes, as measured by the total distance the robot travels. (The drill will automatically move back to the first drill site after it finishes drilling the last hole, so you don't need to explicitly include the first drill site in the `Vector` you return.)

Some notes on this problem:

- 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 without changing the length.
- As a corollary of that above point, because you're coming up with a way to visit all the sites once and return back to the starting point, it doesn't matter which drill site you pick as the first location.
- In lecture, we saw two ways to write a recursive function that looked at all possible subsets of a set, one that built a gigantic list of all the subsets and returned it, and one that built up a single subset, processed it, then discarded it. This latter approach is substantially more memory-efficient. In the course of solving this problem, model your approach on the second of these two strategies, not the first, because the first strategy is extremely memory-inefficient.
- Our provided starter code contains some helpful functions to compute the total distance the drill would travel along a given route and to compute the distance between two points. Feel free to use these functions rather than implementing your own versions from scratch!

Testing Your Solution

We've included as part of the starter files a number of test cases that you can use to check your work. Each test case also includes what the optimal answer should be. While these test cases are good for helping to make sure that your code works correctly, they're not exhaustive and don't cover all possible cases.

In addition to coming up with code for this part of the assignment, you must design at least two custom test cases. When you design those test cases, include within each test case

- why you chose that particular test case,
- what the test case is specifically testing for, and
- what the optimal answer is for that test case.

You should submit your test cases along with the rest of your assignment. We've included four files with the starter code where you should place those test cases. Your first test case should be in the file `drill_your_test_1.txt`, with its solution stored in the file `drill_your_test_1.txt.analyzed`, and your second test case should be in `drill_your_test_2.txt` with its solution stored in the associated file `drill_your_test_2.txt.analyzed`. The starter code contains information about how to edit these files, and you're encouraged to look at the other test files to get a sense of how they're structured as well.

Problem Four: Universal Health Care

You have just been hired as the Minister of Health for the small country of Recursia (it's one of those countries most Americans can't find on a map). Recursia has recently had major economic growth and wants to build a network of national hospitals. They've convened a governmental Recursian Hospital Task Force to investigate possible locations for these new hospitals. The task force has done its job and come back with a list of potential sites. Each of these hospitals has an associated cost and will provide coverage to some number of nearby cities. We can represent each potential hospital site as a `struct`, like this:

```
struct Hospital {
    string name;              // Name of the hospital, for testing purposes
    int cost;                 // How much it costs to build
    Set<string> citiesServed; // Which cities it would cover
};
```

Although you are interested in providing health care to all, you do not have funds to build an unlimited number of hospitals. Your goal is to provide coverage to as many cities as possible using the funds you have available. For example, suppose that Recursia has these cities:

- Bazekas
- Baktrak Ing
- Leapofayt
- Permutation City
- Frak Tell
- Suburb Setz
- Hanoi Towers
- Hooman Pyramids
- Jenuratif
- Cambinashun

Suppose that these are the possible hospital sites:

- Site 1: Covers Bazekas, Hanoi Towers, and Cambinashun. Price: $10,000,000.
- Site 2: Covers Bazekas, Frak Tell, Suburb Setz, and Perumutation City. Price: $50,000,000.
- Site 3: Covers Hanoi Towers, Jenuratif, and Hooman Pyramids. Price: $10,000,000.
- Site 4: Covers Permutation City and Baktrak Ing. Price: $10,000,000.
- Site 5: Covers Frak Tell, Cambination, and Permutation City. Price: $50,000,000.
- Site 6: Covers Jenuratif, Leapofaty, Bazekas, and Hanoi Towers. Price: $50,000,000.

If you build Site 1 and Site 2, you've provided coverage to Bazekas, Frak Tell, Hanoi Towers, Permutation City, Suburb Setz, and Cambinashun, a total of six cities, at a cost of $60,000,000. (Notice that both Site 1 and Site 2 provide coverage to Bazekas, so even though Site 1 covers three cities and Site 2 covers four cities, only six total cities end up covered). Adding in Site 5 would add in a cost of $50,000,000 without actually providing any new coverage, since all three of the cities it services are already covered.

Your task is to write a function

```
Vector<Hospital> bestCoverageFor(const Vector<Hospital>& sites,
                                 int fundsAvailable)
```

that takes as input a list of possible hospital sites and an amount of funds available, then returns a list of hospitals that should be built to provide coverage to the greatest number of cities. Some notes:

- If there's a tie for multiple approaches that each provide the best coverage, your function can return any one of them, and it doesn't necessarily have to be the cheapest one.
- The order in which you return the hospitals is irrelevant.
- It's okay if multiple hospitals provide coverage to the same city. Make sure that when you count up the total number of cities covered, though, that you don't double-count a city.

Testing Your Solution

As in Problem Three of this assignment, you should be sure to test your solution thoroughly before submitting. We've included some sample test cases along with the assignment, but these test cases don't cover every possible case. In addition to running the test cases we've provided, you must create ***two*** of your own custom test cases. When you design those test cases, include within each test case

- why you chose that particular test case,
- what the test case is specifically testing for, and
- what the optimal answer(s) are for that test case.

You should submit your test cases along with the rest of your assignment. There are four files for you to update here as well (`hospital_your_test_1.txt` and `hospital_your_test_1.txt.analyzed`, plus versions for the second test case). As with Problem 3, there's instructions in those files about how to specify your test case and we recommend looking at the other test files as a reference.

Problem Five: (Optional) Extensions!

Want more to explore? Here are a few suggestions to help you get started:

- ***Sierpinski Triangle:*** 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.

  There are a ton of other cool fractal designs that you can create using recursion. Go investigate and see if you can find anything else cool to code up!

- ***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?

- ***Drill, Baby, Drill:*** 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.

- ***Universal Health Coverage:*** Suppose your objective now is to provide coverage to the maximum number of people, given that each city has a different population. Update your function so that each city is annotated with a total population, then have your function find the maximum number of people that can be covered.

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!

- ***Do not change any of the function prototypes.*** We have given you function prototypes for each of the problems in this assignment, and the functions you write must match those prototypes exactly – the same names, the same arguments, the same return types. This means that you may need to treat your functions as wrappers and define your own functions that actually do the recursion.

- ***Test your code thoroughly!*** We'll be running a battery of automated tests on your code, and it would be a shame if you turned in something that almost worked but failed due to a lack of proper testing.

- ***Recursion is a tricky topic***, 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

Submit your `Recursion.cpp` file on paperless.stanford.edu, along with the four test cases and solutions you developed for the last two parts of the assignment. That should be a grand total of nine files (one `.cpp` file, four test files, and four test case solutions). And that's it! You're done!

Good luck, and have fun!