Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 3: Recursion

Reading materials

Goodrich, Tamassia, Goldwasser, ch. 5
OpenDSA: 10
Liang (10): 18

Contents

Joanna Klukowska
joannakl@cs.nyu.edu

1 Recursion

Recursion is a powerful tool for solving certain kinds of problems. Recursion breaks a problem into smaller problems that are, in some sense, identical to the original, in such a way that solving the smaller problems provides a solution to the larger one.

Every recursive solution to a problem can be rewritten as an iterative solution and every iterative solution can be written as a recursive algorithm.

Every recursive function consists of two parts:

base case the case for which the solution can be stated non-recursively (this is the trivial case, but it is necessary for the recursion to terminate)

recursive case the case for which the solution is expressed in terms of a smaller version of itself.

In **direct recursion** the recursive function makes calls to itself. In **indirect recursion**, there is a chain of two or more function calls that eventually returns to the function that originated the chain.

Recursion comes with a price tag. In most case when recursion is implemented a function/method is called over and over again (by itself in direct recursion, or by another function/method in indirect recursion). As a programmer you need to worry about the *function call overhead* that this brings to your programs. Recursion is a powerful tool, but there are many problems (computing factorials, computing Fibonacci numbers) for which the iterative solution is as simple as the recursive solution. In all such cases, you should opt for iterative implementation. On the other hand, there are problems for which recursive solution is extremely simple (towers of Hanoi), and iterative solution is prohibitively complicated - these are the good candidates for recursive implementations.

2 First Recursive Methods that Everybody Learns

If you were ever introduced to recursion before this class, you probably have seen at least one of the algorithms discussed in this section. Be warned that some of them are examples of situations in which recursion should not be used.

2.1 Computing Factorials

Computation of factorials is a very intuitive example of recursive function:

$$n! = n \times (n-1)!$$

The factorial of number n is a product of all the integers between 1 and n. The special case is that factorial of zero is equal to 1, i.e., $0! = 1$.

The recursive algorithm that implements factorials is just Java implementation of the mathematical formula.

```java
public static long factorial ( long number ) {
  //base case
  if (number == 0 )
    return 1;
  //recursive case
  else
    return number * factorial ( number - 1 );
}
```

The iterative solution is equally simple and avoids the overhead of a function/method call:

```java
public static long factorial ( long number) {
  long tmpResult = 1;
  for ( ; number > 0; number--)
```

```
    tmpResult = tmpResult * number;
  return tmpResult;
}
```

2.2 Fibonacci Numbers

Computation of Fibonacci numbers is another mathematical concept that is defined recursively and tends to be among first recursive problems that are introduced. The Fibonacci numbers are defined as follows:

$$\begin{aligned} fib(0) &= 0, \\ fib(1) &= 1, \\ fib(n) &= fib(n-1) + fib(n-2). \end{aligned}$$

Note that some definitions start with $fib(0) = 1$ and $fib(1) = 1$. The interesting thing about this definition is that we have two base cases, not just one.

This definition, once again, translates trivially to Java implementation that uses recursion

```
public static long fibonacci ( long number ) {
  //base cases
  if (number == 0 )
    return 0;
  else if (number == 1 )
    return 1;
  //recursive case
  else
    return fibonacci( number - 1 ) + fibonacci(number - 2);
}
```

The iterative solution requires a bit more thinking and making sure that two previous values are stored, but it is only a bit longer than the above recursive solution

```
public static long fibonacci ( long number ) {
  //base cases
  if (number == 0 )
    return 0;
  else if (number == 1 )
    return 1;
  //recursive case
  else {
    long tmp1 = 0;
    long tmp2 = 1;
    long result = 0;
    int counter = 2;
    while ( counter <= number) {
      result = tmp1 + tmp2;
      tmp1 = tmp2;
      tmp2  = result;
      counter++;
    }
    return result;
  }
}
```

The computation time gained when running the iterative solution is very large. Particularly because it lets us avoid repeated computations of the same values.

WARNING: Recursive solution to the Fibonacci numbers problem is very inefficient and hence, you should always use the iterative solution (unless, of course you are teaching someone about inefficient uses of recursive solutions).

2.3 String Reversal

You probably have written code that, given a string, produces that string in reverse. For example:

- given a string ”Hello”, the new/reversed string should be ”olleH”,
- given a string ”recursion is fun”, the new/reversed string should be ”nuf si noisrucer”.

String reversal has a trivial iterative solution. (If you are not sure how to write such an iterative solution, you should definitely try to do it!) It also can be solved in a recursive way. Here is the outline of the though process that leads to such a solution.

- What should the base case be? There are two options here: an empty string and a one character string.
  - If the method is called with an empty string, its reverse is simply the empty string.
  - If the method is called with a one character string, its reverse is that same string.

  So which one should we use? The first option covers one special case that we will have to handle separately, if we opt to go with the second base case. On the other hand, the second base case allows us to stop a bit sooner and save one recursive call to the method. We could also include both. For simplicity sake, we will go with the first recursive case.

```java
String reverseString ( String phrase ) {
  //base case
  if ( 0 == phrase.length() )
    return phrase;
  //recursive case
  else {
    //to be implemented
  }
}
```

- What should the recursive step be? Given an arbitrary length string, we need to take its last letter and place it at the beginning, then reverse the remaining letters (all, but the last one, which has been moved to the beginning). Reversing only the remaining letters, guarantees that the problem is getting smaller and smaller each time we make the recursive call. Placing the last character at the beginning means that each time we do just a little bit of work (the simplest possible thing) without worrying about the complexity of the whole problem.

  In code this may look as follows:

```java
String reverseString ( String phrase ) {
  int length = phrase.length();
  //base case
  if ( 0 == length )
    return phrase;
  //recursive case
  else {
    return phrase.charAt(length - 1) +    //the last character is moved to the front
           reverseSring ( phrase.substring(0, length-1) ) ; //reverse the rest of it
  }
}
```

You may, quiet correctly, observe that the above method has some issues. It does not work (crashes) when called with a `null` argument - to make this code as robust as possible, this case should be handled. It is also very inefficient in memory allocation since `String` objects are immutable. A much better solution would be to use `StringBuilder` in place of `String` class.

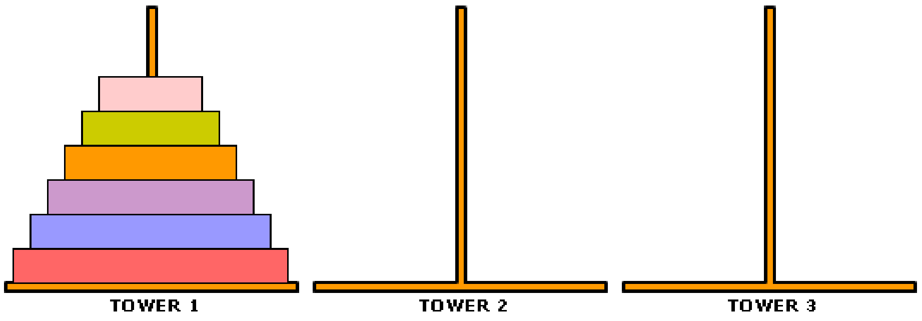
Joanna Klukowska
joannakl@cs.nyu.edu

Something to Think About:
Rewrite the `reverseString` method, so that it has exactly the same signature as the one above, but internally, uses a `StringBuilder` object to manipulate the string.
Hint: You may need a helper/wrapper method to do so.

2.4 Towers of Hanoi

Towers of Hanoi can be a game for a kid or a challenging computer science problem. It is an example of a problem for which the recursive solution is very short and clear, but the iterative solutions can be stated relatively simply but the implementations of them is not trivial (see the Wikipedia page for description of an iterative solution to the Towers of Hanoi problem:
`http://en.wikipedia.org/wiki/Tower_of_Hanoi#Iterative_solution`).

Try to solve the towers of Hanoi problem by hand with 4 or more disks at
`http://www.mathsisfun.com/games/towerofhanoi.html`.



Doing it might be confusing and it will most likely take you many more steps that are necessary for solving the problem.

Imagine that you could

- magically move the top $n - 1$ disks from Tower 1 to Tower 2,
- move the one disk from Tower 1 to Tower 3.
- magically move all the disks from Tower 2 to Tower 3.

This is, believe it or not, the recursive algorithm. The ”magic” is in recursive calls.

Something to Think About:
How would you write a code that solves the problem? How can you provide visual feedback/instructions for the user to follow the steps.

3 Searching

We will consider searching for an element in an unsorted and in a sorted array.

3.1 Searching in an Unsorted Array

When we do not know anything about organization of the data in the array, it is hard to predict where we should start the search in order to find the elements as fast as possible. Moreover, if the element is not in the array, we cannot decide that until we check every single potential location.

The algorithm that sequentially checks every location in the array is called **linear search**: it starts at the beginning of the array and proceeds through the the array one element at a time until either it finds what we are looking for, or reaches the end of the array.

Joanna Klukowska
joannakl@cs.nyu.edu

3.1.1 Linear Search in Arrays of Primitive Types

Here is the Java source code for linear search algorithm in an array of type int:

```java
/** Find the position in A that holds value K, if any does.
 *  Note, the location of the first occurance of K is returned.
 *  @param A  an array of integers
 *  @param K  value to look for
 *  @return the index at which K is located in array A, or
 *     -1 if K is not found
 */
public static  <E>  int linSearch(E [] A, int K) {
  for (int i = 0; i < A.length; i++)
    if ( A[] != null && A[i].equals(K) )          // if we found it return this position
       return i;
  return -1;                 // otherwise, return -1
}
```

Something to Think About:
Could this method be written using recursion?
What are some disadvantages of using the recursive solution? (HINT: read about recursion depth in the textbook.)

3.1.2 Linear Search in Arrays of Objects

If we are searching for an element in an unsorted array of objects (not primitive type variables), then we can no longer use == operator for comparing elements. The type either has to provide the definition of the .equals() method, or the definition of the .compareTo() method. Depending on those definitions the results of searching might be different.

3.2 Searching in a Sorted Array

If we know that the array is sorted, we can take advantage of it in the search for an element. There is a particular location where the any item is expected to be. If it is not there, then it cannot be anywhere else. Having a sorted array, allows us to find the item much faster if it is located in the array, and to quickly declare that it is not there, if that's the case. (Well, assuming that we use an algorithm that takes advantage of that fact.)

You most likely have seen an algorithm called **binary search**. At each step it eliminates half of the remaining items in the array (unlike linear search which eliminated a single item in each step). We will look at the iterative and recursive implementation of the binary search algorithm.

The general outline of the binary search algorithm follows. Assume that key is the element that we are searching for.

1. If the array is not empty, pick an element in the middle of the current array. Otherwise, go to step 5.
2. If that element is smaller than the key, discard all elements in the left half of the array and go back to step 1.
3. If that element is greater than the key, discard all elements in the right half of the array and go back to step 1.
4. If that element is greater than the key, return its index.
5. If the array is empty, the key is not there.

3.2.1 Iterative Implementation of Binary Search

The following code provides iterative solution to the binary search algorithm. It is designed for the Point class.

```java
int binSearchIterative( Point[] points, Point p) {
  int left = 0;
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
  int right = points.length - 1;
  int mid;

  while ( left != right ) { //array is not empty
    mid = (left+right) / 2;
    if (0 > points[mid].compareTo(p) )        //mid element is smaller than p
      left = mid;                             //discard left half
    else if ( 0 < points[mid].compareTo(p) ) //mid element is larger than p
      right = mid;                            //discard right half
    else return mid;                          //found it, return the index
    }

  return -1;           //did not find it, return -1
}
```

3.2.2 Recursive Implementation of Binary Search

Something to Think About:
Write a recursive implementation of a binary search.
HINT: this method will require you to provide a public wrapper method and the private recursive method that performs the search.

3.3 Performance of Search Algorithms

If the array has **n** items how many locations will be accessed by ...	linear search	binary search
in the **best** case?	1	1
in the **worst** case?	n	$\log_2 n$
on **average***?	$< n$	$< \log_2 n$

* the average values depend on the actual data distribution

In computer science, when we analyze problems in this way, we do not like to need to worry about the constant multiples. The reason for it is that for really really large values of n multiplying it by $1/2$ does not change the value that much. We use an order notation that only indicates the highest power of n in the number of operations performed. The notation is ocalled Big-O notation, and we write $\mathrm{O}(n)$ to indicate that the number of operations performed is proportional to n (that may mean exactly n, or $n/2$, or $10n$, or $2n + 15$ - it does not matter which, because for large values of n, those are all similar.

Using that notation the table above would look as follows:

If the array has **n** items how many locations will be accessed by ...	linear search	binary search
in the **best** case?	$\mathrm{O}(1)$	$\mathrm{O}(1)$
in the **worst** case?	$\mathrm{O}(n)$	$\mathrm{O}(\log_2 n)$
on **average**?	$\mathrm{O}(n)$	$\mathrm{O}(\log_2 n)$

4 Backtracking

Backtracking is a method of solving computation problems that is often used to enumerate all solutions that satisfy some constraints (if any) and is able to eliminate the paths that lead to invalid solutions (it *backtracks* from the paths leading to invalid solutions).

4.1 Generating All Sequences

Consider a problem of generating all possible binary sequences of length n. Remember that a binary sequence is a string consisting of 0s and 1s. For a small value of n this problem can be trivially solved by hand. For example, when $n = 3$, we have eight such sequences: 000, 001, 010, 011, 100, 101, 110, 111. In general, there are 2^n such sequences. But as n gets larger, it gets harder to produce such sequences by hand. Instead, we will come up with a method (using simple backtracking) of producing all such sequences.

There are two constraints in this problem:

- each character of the string that we generate has to be either zero or one,
- the string has length n.

For simplicity, our method will print each sequence that it computes to the screen.

```
 1 void getAllBinarySequences ( int length, String seq ) {
 2   if (seq.length() == length ) {//reached the desired length
 3     System.out.printf("%s %n", seq.toString() );
 4   }
 5   else { //add the next bits to the sequence (two possibilities)
 6     String seq0 = seq + "0"; //add zero to the current sequence
 7     getAllBinarySequences( length, seq0);
 8     String seq1 = seq + "1"; //replace the zero with one
 9     getAllBinarySequences( length, seq1);
10   }
11 }
```

Since we want all of the possible sequences, we never take any paths that lead to invalid solutions. The backtracking here can be seen when we finish with recursive call on line 7 and are ready to continue by adding bit one and making the next recursive call.

Another version of this problem is one that adds a constrain regarding certain bit positions. Let's say that we want all binary sequences of a given length, except for the ones in which the third bit (counting from the left) is one. One possibility is to generate all possible sequences and right before printing them, test if the third bit is equal to one. But that means we are doing unnecessary work and then throwing out these sequences. Instead, we will add another condition to our method that prevent such sequences from being generated in the first place.

```
 1 void getSomeBinarySequences ( int length, String seq  ) {
 2   if (seq.length() == length ) {//reached the desired length
 3     System.out.printf("%s %n", seq.toString() );
 4   }
 5   else if (seq.length() == 2 ) {
 6     String seq0 = seq + "0"; //add zero to the current sequence
 7     getSomeBinarySequences( length, seq0);
 8   }
 9   else { //add the next bits to the sequence (two possibilities)
10     String seq0 = seq + "0"; //add zero to the current sequence
11     getSomeBinarySequences( length, seq0);
12     String seq1 = seq + "1"; //replace the zero with one
13     getSomeBinarySequences( length, seq1);
14   }
15 }
```

Something to Think About:
How would you write a method that generates all sequences of a given length that contain digits 0 through 9 (all ten digits)?

Something to Think About:
How would you write a method that generates all possible sequences of a given length that contain letters *a* through *z*?

Joanna Klukowska
joannakl@cs.nyu.edu

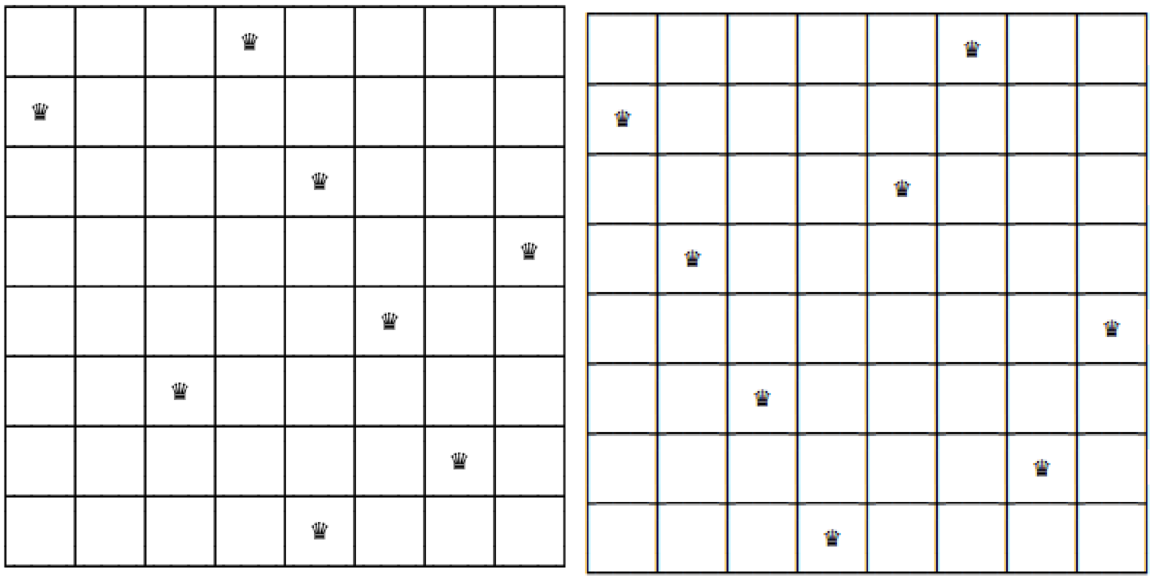
Something to Think About:
How would you write a method that generates all possible English language words of a given length that contain letters *a* through *z*?

4.2 Eight Queens Problem

The eight queen problem is a chess related problem of how to place eight queens on a chess board so that all of them are ”safe” - not under attack by another queen on the board. This requires you to know something about the queen and her powers in chess:

- queen attacks every position in its own column
- queen attacks every position in its own row
- queen attacks every position on its two diagonals

Given the above rules, there are many ways of arranging eight queens on the board without any attacks. But coming up with such arrangements may not be easy. The figure below has one possible solution and one invalid solution:



Solving the problem

First approach of solving the problem is as follows

- Place the first queen anywhere in the top row.
- Then find a position in the second row that is not under attack and put your second queen there.
- Then find a position in the third row that is not under attack and put your third queen there.
- ...
- Then find a position in the eighth row that is not under attack and put your eighth queen there.

PROBLEM: There is a major problem with this approach: you may not be able to find any space in a given row that is not under attack (and unless we can place one queen in each row, we cannot place all the queens since the chess board is eight by eight).

So how can the above solution be changed to handle situation in which we get to a row where all spaces are under attack?

- Whenever you get to a row in which all squares are under attack, go back to a previous row and move the queen in that row to an alternative unattacked position.

PROBLEM: This may solve some of the problem situations of the original approach, but it ignores two things:

1. There might not be another unattacked square in a previous row.
2. Even if there are alternative squares and we move the queen there, this does not guarantee that we can find an unattacked square in the current row.

Fortunately, the solution mentioned above can be applied multiple times, so if we find ourselves in the situation of 1 or 2 above, we go back to the row before the previous row and try to move the queen there. If that still does not work, we go back again, and so on. Eventually we might be moving the queen on the very first row.

This approach of solving problems is called backtracking: we keep going as long as we can, and if we run into problems, we just go back to a previous place where we had a choice to make and pick an alternative way. It seems that it might be hard to keep track of all the steps forwards and step backwards (especially if we go back more than one step). Recursion, can keep track of all such steps.

Pseudo code:

```
placeEightQueens( chessboard )
  placeQueen ( chessboard,  row = 0 )

placeQueen( chessboard, row )
  if row is greater than 8,
    return true (problem is solved)

  for each column from 0 to 8
    try to add queen to that column,
    - if the row, column position is valid for the new queen
    (i.e., it is not under attack)
    then move on to the next row of the chessboard
    - if placeQueen ( chessboard, row + 1) is successfull
    then return true to stop the for loop from checking
    remaining columns

  return false, no position in the current row is valid
```

If we implement the code according to this specification, you'll see that it finds always the same solution. That's because we always start checking from column 0 and advance one column at a time.

An alternative solution, randomizes which columns are used in the for loop.

5 Fractals

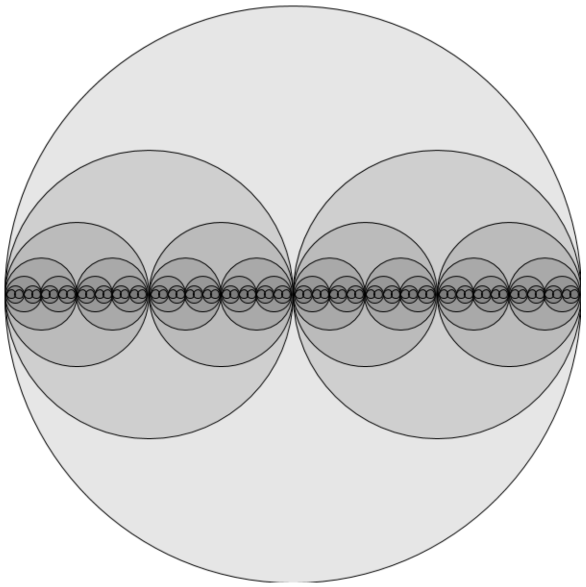
A fractal is a geometric figure that can be divided into parts, each of which is a smaller copy of the whole (if you could zoom in on a fractal, it still would look like the original shape).

Displaying a fractal is an ideal task for recursion, because fractals are inherently recursive.

5.1 Simple Circle Fractal

(You might have seen this example if you took the class with Craig Kapp.)

Joanna Klukowska
joannakl@cs.nyu.edu



This figure may look complicated, but in fact it is very simple to produce if you know about recursion. There are very few steps required to produce this fractal. Assume that the point (0,0) in the image above is in the upper left corner.

1. for a given center (X, Y) and radius R, draw a circle
2. pick the center and a radius for the inner two circles as follows
   (a) circle 1: center = (X - R/2, Y), radius = R/2
   (b) circle 1: center = (X + R/2, Y), radius = R/2

   for each of (a) and (b) go back to step 1

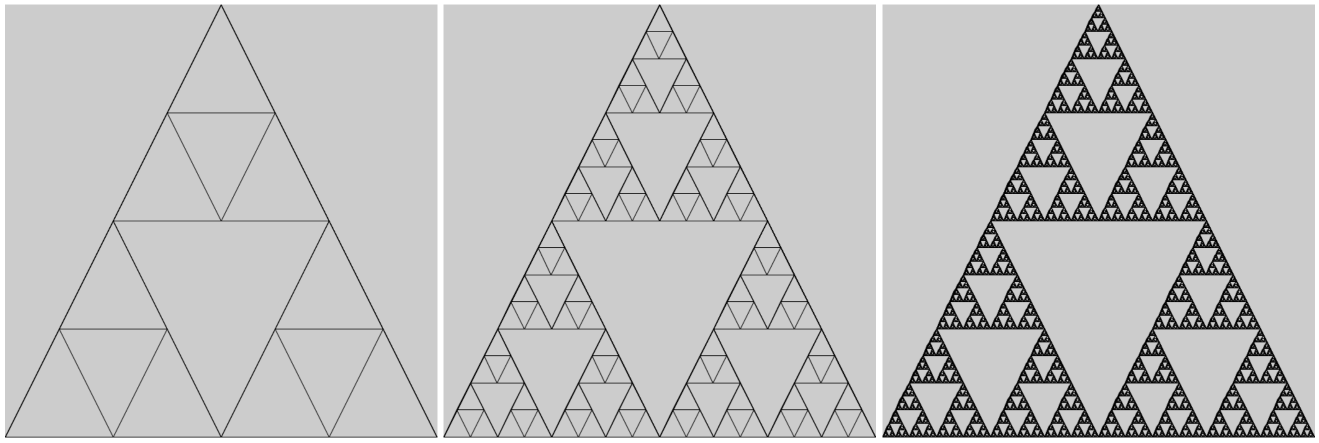
In practice the program cannot recurse indefinitely, so we need to end somewhere. In the following code, we end when the radius becomes smaller than 10 (whatever units we operate in).

```

public void drawCircle(int x, int y, int radius)
{
  // draw a circle of the desired size
  ellipse(x,y,2*radius,2*radius);

  //base case is to stop when radius is < 10
  if (radius >= 10)
  {
    int newRadius = radius/2;
    int offset = radius/2;
    drawCircle(x - offset, y, newRadius);
    drawCircle(x + offset, y, newRadius);
  }
}
```

5.2 Sierpinski Triangle Fractal

This fractal is probably one of the most famous ones. You may not know its name (or not know how to pronounce it), but you most likely have seen it.

Joanna Klukowska
joannakl@cs.nyu.edu



In this fractal, each triangle has three triangles inside of it: one attached to each corner (that's why the middle looks empty). For each triangle we need to know three points in order to draw it. Then, we need to figure out the three points for each of the three triangles inside of it.

Assuming a "big" triangle has its vertices at points (x1, y1), (x2, y2), (x3,y3), what are the vertices of the three triangles inside?