1. Eight Queens
    a. The backtracking approach is a powerful and widely applicable method of solving problems. It involves trying a solution and, if it fails, going back one step in the problem and changing something to see if it now works
    b. Many iterations (GOTO, 1D, 2D, Fancy, etc.) but all are similar

2. Backtracking OK Functions
    a. The OK function for backtracking problems changes depending on the data structure and the problem.
        i. For 8 Queens, a for loop checks the diagonals and row. Column does not need to be checked because the data structure (array) does not allow for multiple values in a column.
        ii. For 8 in a Cross, the row test is performed first. Then the helper array is consulted to see if number neighbors are placed in adjacent squares
3. 8 in a Cross
    a. In the 8 in a Cross problem, cell adjacency can be checked in the OK function using a helper array
    b. The helper array stores the indexes of the adjacent array boxes.
        i. For example, helper[7][] would list all of the boxes adjacent to the last box that gets a value in it
        ii. The last element in the helper array is a sentinel value, -1, which signifies the end of the list of adjacent boxes. It is used because the helper array is 2D, and different rows have different numbers of adjacencies. Box 0 has 0 since it is placed first, but Box 7 will have many adjacencies.

4. Stable Marriage
    a. How do you pair up n men and n women in a way that no one forms a rogue couple and runs off together? There is always a way.
    b. The OK function needs to test the pairs to see if they will last. If an issue arises, it backtracks. Otherwise it moves to the next couple.

5. Pointers - Symbols
    a. A pointer is a variable whose value is an address
    b. * is the symbol used to declare a pointer of a particular type, such as in int* ip;
    c. & is the symbol used to get the address of a variable, such as in ip = &i;
    d. * is also the symbol used to dereference pointers, such as in *ip = 5;
    e. The following code does the following: It initializes the integer i to the value 10. It creates an integer pointer named ip that stores the address of i. It dereferences the integer pointer ip to set the value of the address ip stores to 5. This has the same effect as saying i=5;
        i. int i = 10;
        ii. int* ip = &i;
        iii. *ip = 5;
    f. Note that *ip is another name/label for i;

6. Pointers - Addresses
    a. Given an array int a[3], if we cout << a, what prints is the address of the first element of the array, let's say 0x00001000. This is because the name of an array is actually a pointer to the first element of the array, since arrays store memory contiguously.
    b. cout << a+0 will print the same thing, 0x00001000.
    c. cout << a+1 will print the address of the element at index 1 of the array. For an integer array where each memory space is 4 bytes, this will print 0x00001004
    d. Since arrays are pointers, printing a[1] is really printing *(a+1). As such, if we cout << 1[a], this is really printing *(1+a), which is the same as *(a+1), so it will print the same as cout << a[1].
    e. When passing an array into a function, we always need to pass in the length of the array. This is because arrays are pointers, so we are passing an address into the function, which has no idea where the array ends. We need the length so we know where in memory the array ends.
    f. The same concept applies to 2D arrays. In memory, the values are stored contiguously, with one array following the next. Regardless,

they are still addresses and pointers. The pointer equivalent of a[i][j] is (*(*(a+i))+j).

i. Start at the beginning array a
ii. Add i to find the row
iii. Dereference to get a name, a[i]
iv. Add j to that name to find the column
v. Dereference to get a name, a[i][j]

g. Function names are also pointers, as they point to the location in memory where the function is stored, just like arrays.

7. Pointers - sizeOf
    a. The name of an array is pointers, and the compiler interprets the name of an array depending on the context it is used. These are just a few examples of how they work.

8. Fancy Print - typedef
    a. The fancy print uses ASCII characters to draw out a more realistic looking chess board.
    b. The most important feature of this is is the use of typedef, which allows the user to give an alias to an already existing data type
    c. In the case of Fancy Print, we use typedef char box[5][7]. What this means is we give the alias “box” to the already existing type char[5][7]. In other words, a box is defined as a 2D character array with 5 rows and 7 columns.
    d. Note: when using typedef, read right-left and bounce off barriers
        i. typedef bool (* comparator) (int, int);
            1. Comparator is a “)” pointer that points to “(“ a function that takes two integers and returns a “;” bool.
        ii. typedef double (*FUNC)(double);
            1. FUNC is a pointer to a function that takes a double and returns a double

9. Struct
    a. A struct is a way to create a new "type", similar to a class. However, structs are always public. The code above shows an example of creating a struct.
    b. Structs are created outside of main, and after closing the final bracket, a semicolon is used
    c. Structs, like classes, can have variables, constructors, member functions, etc.

10. Vectors
    a. A vector is a sort of dynamic array. The code above shows an example of creating and using a vector. vector<int>v;
    b. Vector elements are referenced just as arrays are, so v[0] == 1
    c. v.push_back(10) will extend the vector by one element, and assign the new element with a value of 10. It works similar to "append"
    d. v.pop_back() will remove the last element of the vector
    e. v.back() will return the last element of the vector

11. Towers of Hanoi
    a. The steps to solve the Towers of Hanoi problem with n rings follows:
        i. Initialize the three arrays in the vector (towers) so that they all contain the value "n+1" at the bottom and the first vector has the numbers 1->n+1 top to bottom.
        ii. Initialize the variables to solve the first move
        iii. Create a while loop that will only exit when the middle vector has all rings, numbers 1->n+1
            1. Inside the while loop, write out the instructions, taking ring "candidate" from vector "from" to vector "to"
            2. Transfer the value between the vectors using back() and pop_back()
            3. Get the next "from" vector; the vector with the smallest ring that wasn't just moved
            4. Get the next "to" vector; For an even number of rings, this is the first vector to the left. For odd, it's to the right.

5. Get the candidate, which is now on top of the “from” vector

12. Dynamic Memory Allocation
    a. If we want to allocate memory at runtime, such as reading in an integer and then declaring an array of that size, we need to use dynamic memory
    b. Dynamic memory can be accessed using “new” and “delete”. For every new, there needs to be a delete to avoid memory leaks.
        i. To request memory from the heap, use the “new” keyword, which returns a pointer to the memory that has been allocated
            1. int* ip = new int;
            2. int* a = new int[3];
        ii. To delete memory, use the”delete” keyword, which returns the allocated memory back to the heap.
            1. delete ip;
            2. delete [] a;

13. Ram Memory Diagram
    a.

Code Segment	main(), user written functions, other compiled lines The code segment contains the code written by the user
const static global	Stores any variables that are declared to be constants, static, or global, as their access is different from ordinary variables
Runtime Stack	Automatic variables - (scope variables) Stack access is top only - Last In First Out
Heap	Dynamic memory requested during runtime using new Must be returned using delete

14. Inline and Recursive Functions
    a. Inline functions are regular functions. Before the return type, you write “inline”, such as “inline int add_one(int x){”
    b. Inline is a request to the compiler to take the code located within the function and stick it in place of the function call
    c. Recursive functions are functions that call themselves with an easier value to work with. They have a base case and recursive case
    d. Recursive functions utilize activation records, which go on the activation stack, and act as screenshots of the current values of variables the compiler sees.

15. Memoization
    a. Memoization is the dynamic programming process of storing already known values in a cache, such as an array or vector, to look up later. In the case of Fibonacci, it takes a long time to make all the recursive calls, but memoization speeds this up dramatically.
    b. The steps for utilizing memoization are as follows:
        i. Check if the result for the given input parameters is already stored in the cache.
        ii. If the result is found in the cache, return it.
        iii. If the result is not in the cache, check for the base case.
        iv. If the base case is not met, compute the result and store it in the cache for future use.
        v. Return the computed result.

16. Towers of Hanoi (Recursive)
    a. Towers of Hanoi is truly a recursive program, as in order to move n rings, you must first move n-1 rings. In this program, rather than moving rings on towers, we just print out the moves that should be made.
    b. In main, we call the recursive function with the number of rings we have, the from tower, the to tower, and the extra tower.
    c. move(int n, char from, char to, char extra){
        i. Base Case: if the ring in question is the largest one, return; (will go to step iii.)

ii. Recursive Case 1: call the function with (n-1, from, extra, to)
iii. Print the instructions of the move to be made
iv. Recursive Case 2: call the function with (n-1, extra, to, from)

d. The reason for swapping the towers in the first function call is because of the odd/even problem. The reason for swapping the towers in the second function call is because we have already built the n-1 sized tower on the extra tower, so that is the new "from" tower, as we need to move the rings from there.
    i. In other words, before we move the largest ring, we are always trying to move rings from the first tower, since we need the largest ring which is at the bottom. After we move the largest ring, the rest of the rings are now on the third tower, so that becomes the new "from" tower and the process repeats itself in reverse order.

17. Robot Paths - Lattice Paths
    a. Robot paths are the same as lattice paths: how many ways can you reach a certain box on a grid given you can only move in the positive axis directions?
    b. Recursion is used to calculate the number of paths, since, in order to reach a box, you need to reach the box to the left of it and above it
    c. Memoization is handy for this problem, as shown in the code.

18. K Bishops on an NxN Board
    a. Dynamically create an array of size k. Each element of the array will store the bishop's position
    b. In order to check if bishops conflict with one another, you can find their row by taking their position and dividing and modding by n. For a 3x3 board with a bishop at position 6, 6/3 = 2 and 6%3 = 0, so a bishop with position 6 is in row 2 column 0
        i. Positions look like this:
        ii. 0 1 2
            3 4 5
            6 7 8

c. From there, conflict can be tested for by seeing if the absolute value of the difference in rows of two bishops equals the absolute value of the difference in columns of two bishops.
    i. 1 and 5 would conflict.
        1. 1/3 == 0 and 1%3 == 1
        2. 5/3 == 1 and 5%3 == 2
        3. |1-0| == |2-1|
    ii. 3 and 6 do not conflict.
        1. 3/3 == 1 and 3%3 == 0
        2. 6/3 == 2 and 6%3 == 0
        3. |2-1| != |0-0|

19. Class - Parameters
    a. Classes are private by default, so you need to declare what variables are private and public using private: and public:

20. Class - Functions
    a. Class functions are written in the same syntax as normal functions are
    b. Get and set functions should be used for private variables
    c. Constructors are similar to Java, except there is no privacy type. You just call the name of the class with the necessary parameters, such as Rat(){ or Rat(int a, int b){ or Rat(int a){

21. Class - Operator Overloads
    a. In order to change the meaning of an operator, write a function called operator+() that does what you want the sign to do. Just replace + with the sign you need, such as *, -, /, %, etc.

22. Class - IO Streams
    a. cout is an object called an output stream, and it tells the compiler what the << operator means
    b. cout << 5 translates to cout.operator<<(5)
    c. The cout object belongs to the ostream class, and it is used to print to the console

d. When we type cout << 5, the integer class's operator<< function takes the parameter of cout as well as the value of the integer
e. In the same way, cin is an input stream that works identically
f. IO streams must be declared as friends in order to give operator<<() access to the private sections of the class because it is not a function of the class. Furthermore, since we want to use the same IO streams each time, we pass them by reference.
    i. friend ostream& operator<< (ostream& os, Rat r){
    ii. friend istream& operator>> (istream& is, Rat r){