Recursion

Recursion is one of the most difficult but powerful computer science topics (no sh**).
Use it for things like AI for games, solving sudoku, and cracking codes.
Job interviews love to ask you to write recursive functions.

Idea Behind Recursion

- Break the problem into two or more simpler sub-problems
- Solve each sub-problem j by calling ourself!
- Collect all the solutions to the sub-problems
- Use the sub-soutions to construct a solution to the complete problem

Lazy Person's Sort
Take a bunch of index cards with numbers, split the cards into two roughly-equal piles. Hand one pile to nerdy student A and ask them to sort it. Hand the other pile to nerdy student B and ask them to sort it. Take the two sorted piles and merge them into a single sorted pile.

If you're handed just one card, then just give it right back. Split the cards into two roughly-equal piles. Hand one pile to student A and say "do the Lazy Person's sort". Hand the other pile to student B and say "do the Lazy Person's sort". Take the two sorted piles and merge them into a single sorted pile.

The Lazy Person's sort = Merge Sort

```
void MergeSort(an array)
{
    if(array's size == 1)
    {
        return;    //array has just one item, all done!
    }
    MergeSort(first half of an array); //process the 1st half of the array
    MergeSort(second half of an array);    //process the 2nd half of the array
    Merge(the two array halves); //merge the two sorted halves
    //Now the complete array is sorted
}
```

The Two Rules of Recursion

> RULE ONE: Every recursive function must have a "stopping condition"!

The Stopping Condition (aka Base Case): Your recursive function must be able to solve the simplest, most basic problem without usig recursion. A recursive function calls itself, so every recursive function must have

some mechanism to allow it to stop calling itself.

> RULE TWO: Every recursive function must have a "simplifying step"!

Every time a recursive function calls itself, it must pass in a smaller sub-problem that ensures the algorithm will eventually reach its stopping condition. Remember, a recursive function must eventually reach its stopping condition or it'll run forever.

```
void eatCandy(int layer)
{
    if(layer == 0)
    {
        cout << "Eat center!";
        return;
    }
    cout << "Lick layer " << layer << endl;
    eatCandy(layer - 1);
}
```

Writing Recursive Functions in 6 Steps

1. Write the function header
2. Define your magic function
3. Add your base case code
4. Solve the problem with the magic function
5. Remove the magic
6. Validate your function

Adding base case code:
Identify the simplest possible input(s) to our function, and then have our function process those inputs without caling itself. Some problems may require more than one check.
Recursive method to calculate a factorial

```
int fact(int n)
{
    if(n == 1)
    {
        return 1;
    }
```

```
    return n * fact(n - 1);
}
```

Recursion on an array:
To sum up all of the items in an array, we need a pointer to the array and its size.

```
int sumArr(int arr[], int n)
{
    if(n == 0)
        return 0;
    if(n == 1)
        return arr[0];
    return arr[0] + sumArr(arr + 1, n - 1);
}
```

OR

```
int sumArr(int arr[], int n)
{
    if(n == 0)
        return 0;
    return a[n - 1] + sumArr(arr, n - 1);
}
```

Recursion Challenge:
Write a recursive function called `printArr` that prints out an array of integers in reverse from bottom to top.

```
void printArr(int arr[], int n)
{
    if(n == 0)
        return;
    cout << arr[0] << endl;
    printArr(arr + 1, n - 1);
}
```

Recursion on a Linked List:
Instead of passing in a pointer to an array element, you pass in a pointer to a node. You don't need to pass in a size value for your list.
Example:
Write a function that finds the biggest number in a NON-EMPTY linked list.

```
struct Node
{
    int val;
    Node *next;
}
int biggest(Node *cur)
{
    if(cur->next == nullptr)
        return cur->val;
    int value = biggest(cur->next);
    if(cur->val > val)
    {
        return cur->val;
    }
    else
        return val;
}
```

**Critical tip:
Your recursive function should generally only access the current node/array cell passed into it! It should rarely access the values in the nodes/cells below it.

Recursive Challenge #2:
Write a recursive function called `count` that counts the number of times a number appears in an array.
My solution:

```
int count(int arr[], int n, int num)
{
    if(n == 0)
        return 0;
```

```
    if(arr[n - 1] == num)
        return 1 + count(arr, n - 1, num);
    else
        return count(arr, n - 1, num);
}
```

Their solution:

```
int count(int arr[], int n, int num)
{
    if(n == 0)
        return 0;
    int total = count(arr + 1, n - 1, num);
    if(arr[0] == num)
        total++;
    return total;
}
```

Recursive Challenge #3
Write a function that finds and returns the earliest position of a number in a linked list. If the number is not in the list or the list is empty, your function should return -1 to indicate this.
My solution:

```
struct Node
{
    int val;
    Node *next;
}


int findPos(Node *cur, int num)
{
    if(cur == nullptr)
        return -1;
    if(cur->val == num)
```

```
    {
        return 0;
    }
    else
    {
        return 1 + findPos(cur->next, num); //what if findPos(cur->next, num) returns -1???
    }
}
```

Their solution:

```
int findPos(Node *cur, int num)
{
    if(cur == nullptr)
        return -1;
    if(cur->val == num)
        return 0;
    int posInRestOfList = findPos(cur->next, num);
    if(posInRestOfList == -1)
        return -1;
    else
        return posInRestOfList + 1;
}
```

Looking at their solution, I can see where my solution failed to consider the case where calling `findPos()` would return -1, thus returning a wrong answer.

Recursion: Binary Search
Goal: search a sorted array of data for a particular item
Idea: use recursion to quickly find an item within a sorted array
Algorithm:

```
Search(sortedWordList, findWord)
{
    if(there are no words in the list)
```

```
        We're done: NOT FOUND!
    Select middle word in the word list.
    if(findWord == middle word)
        We're done: FOUND!
    if(findWord < middle word)
        search(first half of sortedWordList);
    else
        search(second half of sortedWordList);
}
```

```
int binarySearch(string A[], int top, int bot, string f)
{
    if(top > bot)
        return -1;
    else
    {
        int mid = (top + bot) / 2;
        if(f == A[mid])
            return mid;
        else if(f < A[mid])
            return binarySearch(A, top, mid - 1, f);
        else
            return binarySearch(A, mid + 1, bot, f);
    }
}
```

Solving a maze:
The recursive solution works in the same basic way as the stack-based solution we saw earlier. The algorithm uses recursion to keep moving down paths until it hits a dead end. Once it hits a dead end, the function returns until it finds another path to try.

```
char m[11][11] = {.....};
```

```
bool solve(int sx, int sy, int dx, int dy)
{
    m[sy][sx] = '#';
    if(sx == dx && sy == dy)
        return true;
    if(m[sy - 1][sx] == ' ')
        solve(sx, sy - 1);
    if(m[sy + 1][sx] == ' ')
        solve(sx, sy + 1);
    if(m[sy][sx - 1] == ' ')
        solve(sx - 1, sy);
    if(m[sy][sx + 1] == ' ')
        solve(sx + 1, sy);
}
```

Co-recursion is when two functions recursively call each other! This (probably) won't be on the exam???????