Week 7 Recitation Exercise

CS 261 – 10 points

Recursion is a method of solving a programming problem by writing a function that calls itself. A function that does this is called a ***recursive function***. Conceptually, recursion is just a different mechanism for repeating a piece of code multiple times, just like a loop, and in many cases loops and recursion can be used interchangeably to solve the same problems.

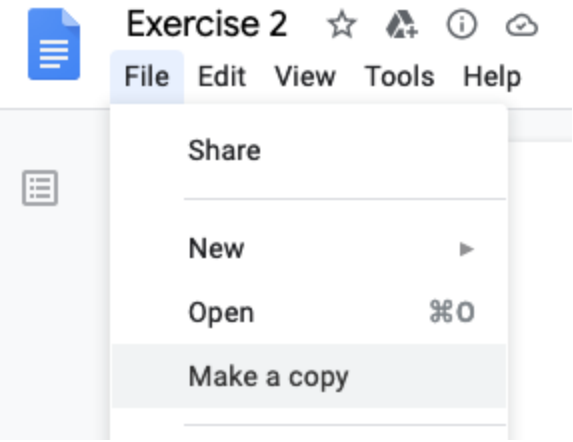
Often, solving a problem recursively results in code that's cleaner and easier to understand than when that same problem is solved using a loop, and recursive solutions are sometimes preferred for this reason. In particular, problems whose solutions

It's important to understand how recursion works, so you can recognize situations where a recursive solution might be preferred. For this reason, we'll explore recursion in this recitation exercise.

Follow the instructions below, and when you're finished, make sure to submit your completed exercise on Canvas. Please submit one copy per group (as long as all group members' names are listed below, it's fine if just one person submits). Remember that this exercise will be graded based on effort, not correctness, so don't worry if you don't get all the right answers. Do make sure you *try* to get the right answers though.

Step #1: Organize your group and create your submission document

Your TA will assign you to a small group of students. Among your group, select one group member to record your group's answers for this exercise. This group member should make a personal copy of this document so they are able to edit it:



Step #2: Add your names

Next, your group's recorder should add the names and OSU email addresses of all of your group's members in the table below:

Group member name	OSU email address
HyunTaek, Oh	ohhyun@oregonstate.edu

Step #3: Grab code for this recitation

Now, take a look at the code in the following GitHub repo:

https://github.com/osu-cs261-f23/recitation-7

Clone this repo onto your own machine or onto one of the ENGR servers, but note that you won't be able to push changes back to the repo, since your access to it is read-only. If you'd like to be able to push any changes you make to the code back to GitHub, you'll have to make a fork first and then clone and work with the fork.

This repo contains several C files along with a Makefile that compiles them. The contents of each of the files will be addressed below.

Step #4: Explore the basics of recursion

Recursion is most useful when a problem can be broken down into steps, where each step solves an increasingly smaller instance of the same problem. To see an example of how this works, consider the problem of computing the factorial of a number, i.e.

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

In the file factorial.c in the repo you cloned for this exercise, you can see two different implementations of the factorial function, one that iteratively computes the factorial and one that recursively computes the factorial. The recursive version of the factorial takes advantage of the fact that $n!$ can be solved in steps by using the solution of a smaller instance of the factorial problem. Specifically, the recursive factorial function operates based on the fact that $n! = (n-1)! \times n$. Indeed, you can see this formula encoded in the return statement of that function.

In general, to be able to implement a recursive function that solves a given problem, we must be able to satisfy two constraints:

- The recursive function must contain at least one ***base case***, which is an input for which the function can produce a trivial output without recursing.
- The recursive calls to the function must simplify or reduce the problem in such a way that a base case will *always* eventually be reached. In other words, each recursive function call must "move towards" the base case.

A recursive function that does not satisfy these constraints is at risk of entering an infinite recursion, i.e. a never-ending series of recursive calls (similar to an infinite loop).

It should be clear that both of the above constraints are satisfied by the recursive factorial function in factorial.c. In that function, the base case occurs when $n \leq 1$, in which case the function directly returns 1 instead of making a recursive call. In addition, the recursive call in that function obviously reduces the problem towards the base case by subtracting 1 from the current value of n. In other words, each recursive call decrements n by 1, which moves it closer to the base case of $n = 1$.

Step #5: Identify recursive properties

Now, look at the code in the file fibonacci.c in the repo you cloned for this exercise. This file contains two different versions—one iterative and one recursive—of a function to

compute the n^{th} number in the Fibonacci sequence, which is a sequence of integers defined by this mathematical recurrence:

$$F_n = F_{n-1} + F_{n-2} \text{ where } F_0 = 0 \text{ and } F_1 = 1$$

Focus on the recursive version of the Fibonacci function. Among your group, discuss the following two questions, and then write your answers to them below:

- What is the base case for this recursive function? Is there more than one base case?
- How does the function guarantee its recursive calls will eventually reach the base case(s)?

[**1. What is the base case for this recursive function?**

- The base case can be when n==0 and n==1 in the both the Fibonacci functions. When the count 'n' of the iteration is equal to 0 or 1, the case would return 0 or 1 for base cases.

2. Is there more than one base case?

- Maybe not. If there is a possibility to make more than one base case, It would also end the base case above.

3. How does the function guarantee its recursive calls will eventually reach the base case(s)?

- It depends on the problem that can be a smaller part and has base case(s). The case the problem can be a smaller part and this part also can be a much smaller part of it, and then finally reaches the smallest part, which is base case. So, if the problem cannot be a smaller and even much smaller part of it, and there is no base case the problem finally reaches, which are the most important thing to guarantee stable recursive calls.]

Step #6: Implement a very simple recursive function

Now that you have an idea how recursion works, apply your knowledge to implement a recursive function that solves a very simple problem. Specifically, the file simple_sum.c in the repo you cloned for this exercise contains an iterative version of a function to compute the sum of all integers between 1 and n, i.e.

$$\sum_{i=1}^{n} \quad i$$

That same file also contains the start of the definition of a recursive function to compute the same value. Finish implementing that recursive function, and copy/paste your complete recursive function definition below. Don't forget to make sure your recursive function has a base case that your recursive calls always move towards.

Hint: Your recursive sum function can be implemented in a very similar way to the recursive version of the factorial function.

```
[ int recursive_sum(int n) {
        If (n==1) {
                Return 1;
        } else if (n==0) {
                Return 0;
        }

      Return n + recursive_sum(n-1);       ]
```

Step #7: Implement a (very) slightly more complex recursive function

Next, look at the file digit_sum.c in the repo you cloned for this exercise. Again, this file contains an iterative version of a function and the start of a definition for a recursive version of the same function, which you must implement. This time, the function you'll implement must compute the sum of the digits in an integer value n. For example, the sum of the digits in the integer value 123 is 1 + 2 + 3 = 6.

As before, complete the recursive function implementation, and copy/paste your complete recursive function definition below. Again, don't forget to make sure your recursive function has a base case that your recursive calls always move towards.

Hint: Can you recursively mimic the behavior of the loop in the iterative version of the function?

```
[ int recursive_digit_sum(int n) {
        If (n==0) {
                Return 0;
        }
        Return n%10 + recursive_digit_sum(n/10);
  } ]
```

Step #8: Implement a recursive version of binary search

Finally, turn your attention to the file binary_search.c in the repo you cloned for this exercise. This file contains an iterative version of the binary search algorithm. Your last task for this exercise is to complete the implementation of the recursive version of binary search.

The recursive implementation of binary search will follow a different recursive pattern than the recursive numerical computations you explored above. However, it will still have a base case, and the recursive call you make in your function will still always reduce the problem towards the base case. Once you figure out how to implement the recursive version of binary search, copy/paste your entire recursive function definition below. If you're not able to figure out how to implement binary search recursively, instead of copy/pasting your function definition below, describe how you tried to implement it and where you got hung up.

Note: It is possible to implement a recursive version of binary search using the signature of recursive_binary_search() in binary_search.c. However, if you find it challenging to implement the recursive function with this signature, feel free to implement a recursive "helper" function with a different signature and to simply call that recursive function from recursive_binary_search(). This is a common pattern when implementing recursive functions.

```
[ int helper (int q, int* array, int low, int high) {
      Int mid, l=low, h=high;
      Mid = (l+h)/2;

      If (array[mid]==q) {
         Return mid;
      }
      Else if ( array[mid] > q) {
         Return helper(q, array, l, mid-1);
      }
      Else {
         Return helper(q, array, mid+1, h);


      Return -1;
  }
```

```
Int recursive_binary_search(int q, int* array, int n) {
    Int low=0, high=n-1;

    Return helper(q, array, low, high);
}]
```

Submission and wrap-up

Once you've completed this exercise, download your completed version of this worksheet as a PDF and submit that PDF on Canvas (refer to the image below to help find how to download a PDF from Google Docs). Again, please submit one copy of your completed worksheet for your entire group. It's OK if just one member submits on Canvas. All group members will receive the same grade as long as their names and email addresses are included above.