Wesleyan University, Fall 2021, COMP 211
Homework 5: Recursion and the Game of Nim
Due by 11:59pm on October 12, 2021

---

1. Written problems (5 points)

Problem 1. *Consider the pseudocode in Program 1 and assume execution starts at the first line of* `main` *(i.e., assume you have a stack with a single empty binding table just before executing line 11). Describe how the environment changes as the program executes. Give the maximum number of binding tables that are ever on the the stack. Also give the program cost using Big-Oh notation.*

Solution: The program cost is $O(n)$ since the function `sum` will be called recursively n times. The maximum number of binding tables that are ever on the stack is 5, 3 of which belong to `sum` function calls, 1 of which belongs to `printf`, and 1 of which belongs to `main`. **Note: not including the binding table for `printf` and having 4 as the maximum number is also a valid answer, since you can argue that the arguments to `printf` would first be evaluated before `printf` is ever called, which would be after the recursion has finished.**

The execution semantics of the program are as follows.

(1) The program begins with the empty binding table for main, S^m, on the stack.

$$S^m = \{\}$$

(2) At line 11, the `printf` function is called and a new binding table is pushed on the stack. To evaluate the arguments being passed to `printf`, the function call to `sum` must be evaluated. We assume the first argument to `printf` is bound to the variable `str`, and the second argument is bound to the variable `val`. The stack of binding tables thus looks like:

$$S^m = \{\}; S^p = \{str \to \text{“}sum = \%d\backslash n\text{”}, val \to ?\}$$

(3) To evaluate the arguments being passed to `printf`, the function call `sum(2)` must be evaluated, which pushes a new binding table on the stack, and binds 2 to the variable n in the new binding table.

$$S^m = \{\}; S^p = \{str \to \text{“}sum = \%d\backslash n\text{”}, val \to ?\}; S^2 = \{n \to 2\}$$

(4) Since $S^2(n) = 2$, the else block in `sum` is executed. This requires evaluation of the function call `sum(n-1)`, with the argument to `sum` first being evaluated. The value of n is found to be 2 using S^2, and then 1 is subtracted giving 1. Then the function call `sum(1)` is evaluated, which pushes a new binding table, S^1 on the stack, with the value of 1 bound to the variable n in the new binding table, S^1.

$$S^m = \{\}; S^p = \{str \to \text{“}sum = \%d\backslash n\text{”}, val \to ?\}; S^2 = \{n \to 2\}; S^1 = \{n \to 1\}$$

(5) Since $S^1(n) = 1$, the else block in `sum` is again executed. This requires evaluation of the function call `sum(n-1)`, with the argument to `sum` first being evaluated. The value of n is found to be 1 using S^1, and then 1 is subtracted giving 0. Then the function call `sum(0)`

```
 1 int sum(int n) {
 2     if(n==0) {
 3         return 0;
 4     } else {
 5         return sum(n-1) + n;
 6
 7     }
 8 }
 9
10 int main(void) {
11     printf("sum = %d\n", sum(2));
12     return 0;
13 }
```

PROGRAM 1. A program that recursively sums the numbers from 1 to n.

is evaluated, which pushes a new binding table, S^0 on the stack, with the value of 0 bound to the variable n in the new binding table, S^0.

$$S^m = \{\}; S^p = \{str \to \text{``}sum = \%d\backslash n\text{''}, val \to ?\}; S^2 = \{n \to 2\}; S^1 = \{n \to 1\}; S^0 = \{n \to 0\}$$

(6) Since $S^0(n) = 0$, we have hit the base case of the recursion and the if block in `sum` is executed. This returns the value of 0, so the function call `sum(0)` evaluates to 0, and the binding table S^0 is popped from the stack.

$$S^m = \{\}; S^p = \{str \to \text{``}sum = \%d\backslash n\text{''}, val \to ?\}; S^2 = \{n \to 2\}; S^1 = \{n \to 1\}$$

(7) The evaluation of line 5 can now be completed, adding together the value 0 returned from the function call `sum(0)` to the value $S^1(n) = 1$, to return the value of 1. So the function call `sum(1)` evaluates to 1, and the binding table S^1 is popped from the stack.

$$S^m = \{\}; S^p = \{str \to \text{``}sum = \%d\backslash n\text{''}, val \to ?\}; S^2 = \{n \to 2\}$$

(8) The evaluation of line 5 can now again be completed, adding together the value 1 returned from the function call `sum(1)` to the value $S^2(n) = 2$, to return the value of 3. So the function call `sum(2)` evaluates to 3, and the binding table S^2 is popped from the stack.

$$S^m = \{\}; S^p = \{str \to \text{``}sum = \%d\backslash n\text{''}, val \to ?\}$$

(9) The evaluation of line 11 can now be completed, since `sum(2)` has been evaluated and found to equal 3, and the variable `val` in the `printf` binding table is updated.

$$S^m = \{\}; S^p = \{str \to \text{``}sum = \%d\backslash n\text{''}, val \to 3\}$$

(10) The instructions in the `printf` function are then executed, and when `printf` returns, the binding table S^m is popped from the stack.

$$S^m = \{\};$$

(11) When `main` returns, the binding table S^m is popped from the stack.

2. Coding problems (15 points)

Problem 2. *Write a recursive function* `unimodal_search_r` *that satisfies the following specification:*

- **Function header.** `int unimodal_search_r(int A[], int a, int b)`
- **Pre-condition.** *A has size n, $0 \le a < n$, $0 \le b < n$ and there is some i such that*

$$A[0] < A[1] < \cdots < A[i-1] < A[i] > A[i+1] > \cdots > A[n-1].$$

- **Function body.** *This should satisfy the following:*

$$\texttt{unimodal\_search\_r}(A, a, b) = i, \text{ where } A[i] = \max(A[a], \ldots, A[b]).$$

In other words, A consists of values that are strictly increasing up to some index i, and then strictly decreasing after that. `unimodal_search_r`(A, a, b) *will return i. Note that it could be that $i = 0$ or $i = n - 1$. You may choose to use as a starting point your code for the associated problem on Homework 4, or you may choose to use the Homework 4 solution code posted. Your implementation, however should have cost $O(\log_2 n)$ or $O(\log_3 n)$.*

Problem 3. *The game of* Nim *is the following game played by two players. To start, there are three piles of sticks: there can be any number of sticks greater than zero in each of the piles. The players then take turns removing sticks. In one turn, a player may remove any number of sticks greater than zero, but all the sticks must come out of the same pile. The goal is to force the other player to remove the last of the sticks (i.e., the player who removes the last stick loses).*

For this problem, you will implement the game of Nim. Your implementation will be a two-player game, where one player interacts and the other is the computer. The human player will be asked for the pile from which to remove sticks and how many to remove. The computer's strategy will be very straightforward (and not very intelligent): choose a pile at random, choose a number n at random such that $0 < n \le N$, where N is the number of sticks in that pile, and remove n sticks from that pile. Note that it is very easy to win when your opponent uses this strategy. Do not use recursion to implement the game of Nim.

Your implementation will be a self-contained program, with its own `main` *function and any other functions you choose to write. If you find that in your* `main` *function you perform a well-defined task more than once, you should encapsulate that task into a function that you call. Your* `main` *function will probably perform roughly the following tasks:*

1. *Initialize three piles of sticks to randomly-chosen values between 1 and 10 inclusive (see below on how to generate random numbers).*
2. *Repeat the game loop:*
   - (a) *Print out the number of sticks in each pile.*
   - (b) *Ask the user to choose a pile from which to draw sticks and how many sticks to draw. You may assume that the user will always enter a non-negative integer in both cases. Make sure to perform basic error-checking to ensure that the user chooses a legal pile,*

that the pile has some sticks in it, that the user chooses at least one stick, and that the user does not choose more sticks than are in the pile.

(c) *Remove the sticks from the pile.*
(d) *If there are no sticks left in any pile, report that the user has lost. The game should end at this point.*
(e) *Otherwise choose a pile at random for the computer player. The pile must not be empty. If you like, you can just repeatedly choose a pile until you choose one that is not empty.*
(f) *Choose a number at random between 1 and the number of sticks in the chosen pile.*
(g) *Remove the sticks from the pile.*
(h) *If there are no sticks left in any pile, report that the computer player has lost. The game should end at this point.*

Figures 1 and 2 are traces of our implementation. Your implementation need not be identical, but it should be similar. Credit will be given not just for a correct implementation, but a well-designed one. For example: have you made good use of functions? would it easy to change the number of piles? would it be easy to change the maximum number of sticks initially in any pile?

A few pointers on generating random numbers:

- *To use the libraries needed to generate random numbers you must include the following header files in your program:*

  ```
  #include <stdlib.h>
  #include <time.h>
  ```

- *To generate a random number, use the* `rand()` *function, which has the following signature:*

  ```
  unsigned long rand();
  ```

  This function is declared in the header file `stdlib.h`. *When called,* `rand()` *returns a randomly-generated number in the range* [0, `RAND_MAX`], *where* `RAND_MAX` *is a value that is also defined in* `stdlib.h`. *Of course, you don't want random numbers in the range* [0, `RAND_MAX`]*; usually you want numbers in the range* $[0, N)$ *for some other value of* N*. The modulus operator is your friend here:* `rand()%N` *returns a randomly-generated number in the range* [0, `N`).

- *Before your first call to* `rand()`, *execute the following instruction once:*

  ```
  srand(time(NULL));
  ```

  The function `srand()` *is declared in* `stdlib.h` *and the function* `time()` *is declared in* `time.h`. *The reason to do this function call is the following. The* `rand()` *function described above actually generates a sequence of what are called* pseudo-random numbers, *using a specific formula; each call to* `rand()` *returns the next number in that sequence. The sequence itself is determined by what is called the* seed *of the pseudo-random number generator. The* `srand()` *function sets the seed. If you never call the* `srand()` *function, then the seed defaults to 1 every time. That means that if you never call* `srand()`, *then calls to* `rand()` *in your program will return the same sequence of numbers every time you run the program. The smaller the seed is typically, the less randomness there is in the sequence of random numbers generated.*

```
$ ./nim
Pile 0:  2 sticks.
Pile 1:  2 sticks.
Pile 2:  10 sticks.
From which pile (0-2) do you want to remove sticks? 0
How many sticks (1-2) do you want to remove? 2
Pile 0:  0 sticks.
Pile 1:  2 sticks.
Pile 2:  10 sticks.
I choose to remove 1 sticks from pile 1.
Pile 0:  0 sticks.
Pile 1:  1 sticks.
Pile 2:  10 sticks.
From which pile (0-2) do you want to remove sticks? 1
How many sticks (1-1) do you want to remove? 0
You must choose a number between 1 and 1.
How many sticks (1-1) do you want to remove? 87
You must choose a number between 1 and 1.
How many sticks (1-1) do you want to remove? 1
Pile 0:  0 sticks.
Pile 1:  0 sticks.
Pile 2:  10 sticks.
I choose to remove 5 sticks from pile 2.
Pile 0:  0 sticks.
Pile 1:  0 sticks.
Pile 2:  5 sticks.
From which pile (0-2) do you want to remove sticks? 2
How many sticks (1-5) do you want to remove? 4
Pile 0:  0 sticks.
Pile 1:  0 sticks.
Pile 2:  1 sticks.
I choose to remove 1 sticks from pile 2.
Oh no!  I lose!
```

Figure 1. A winning trace of our `nim` implementation.

The `srand()` *invocation given above sets the seed value to the value of* `time(NULL)`*, which in turn is the number of seconds that have elapsed since midnight of 01 January 1970 (or possibly 1900, depending on your operating system). The point is that the value of* `time(NULL)` *is different every time you run your program, and so every time you run your program, you will get a different sequence of numbers when you repeatedly call* `rand()`*.*

3. Code distribution

This assignment comes with a *code distribution*, comprising files that you will need to complete Problem 2 of this assignment:

- `hw5a.h`: header file for the code you will write. This file declares the function that you must implement. Do not change the contents of this file; if it appears to be causing problems

```
$ ./nim
Pile 0:  3 sticks.
Pile 1:  8 sticks.
Pile 2:  9 sticks.
From which pile (0-2) do you want to remove sticks? 0
How many sticks (1-3) do you want to remove? 3
Pile 0:  0 sticks.
Pile 1:  8 sticks.
Pile 2:  9 sticks.
I choose to remove 7 sticks from pile 1.
Pile 0:  0 sticks.
Pile 1:  1 sticks.
Pile 2:  9 sticks.
From which pile (0-2) do you want to remove sticks? 1
How many sticks (1-1) do you want to remove? 1
Pile 0:  0 sticks.
Pile 1:  0 sticks.
Pile 2:  9 sticks.
I choose to remove 1 sticks from pile 2.
Pile 0:  0 sticks.
Pile 1:  0 sticks.
Pile 2:  8 sticks.
From which pile (0-2) do you want to remove sticks? 2
How many sticks (1-8) do you want to remove? 6
Pile 0:  0 sticks.
Pile 1:  0 sticks.
Pile 2:  2 sticks.
I choose to remove 1 sticks from pile 2.
Pile 0:  0 sticks.
Pile 1:  0 sticks.
Pile 2:  1 sticks.
From which pile (0-2) do you want to remove sticks? 2
How many sticks (1-1) do you want to remove? 1
You lose!
```

FIGURE 2. A losing trace of our `nim` implementation.

with compilation, the problem is with your solution.

- `hw5a.c`: A function stubs matching hw5a.h has been implemented for you to get you started.
- `comp211.h`: a header file that defines `dotest`, which is used in `tests.c`.
- `tests.c`: a small testing program. This program provides just a few tests. You should certainly add more. To compile the testing program, use the command

  ```
  gcc --std=c99 -o tests tests.c hw5a.c
  ```

- `driver.c`: a small driver program. This program provides a simple interactive program that uses your `unimodal_search_r` function. You may modify it however you like. To compile the driver program, use the command

```
gcc --std=c99 -o driver driver.c hw5a.c
```

- `Makefile`: a Makefile for this assignment. Instead of using the above commands for compiling your code, you can use the commands `make tests` and `make driver`.

4. Submission

Submit your written work as `hw5.pdf`, Problem 2 as `hw5a.c`, and Problem 3 as `hw5b.c` to the Google Drive directory I have created for you named `comp211-f21-USERNAME/hw5/`. You should replace `USERNAME` with your Wesleyan username.

Do not forget that your written work must be submitted as a PDF! And make sure that at the top of each file you have put your name! Do not, however, change the names of the files.

You should *not* submit `hw5a.h` or any test or driver programs. When we test your code, we will add in our copy of `hw5a.h` and our own testing program. In particular, if you change `hw5a.h` in order to make your code compile, then your code will probably fail to compile with our `hw5a.h`, and hence you will receive little to no credit for the coding portion of this assignment.