Time Complexity, Recursion and Function Memory Allocation.

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

Microsoft Teams (search "song liu").

Previous Lecture

- In the previous lecture, we talked about:
    - Nested `if-else`
    - Nested Loops
    - Early Loop Break/Restart
        - `break;`
        - `continue;`

Previous Lab

- Finding the maximum number among `a`, `b` and `c`.
- Do pairwise comparison between `a`, `b` and `c`.
    - If `a>b` and `a>c`, `a` is the biggest.
    - If `a>b` and `c>=a`, `c` is the biggest.
    - If `a<=b` and `b<c`, `c` is the biggest.
    - If `a<=b` and `c<=b`, `b` is the biggest.

Previous Lab

- Write pseudo code.

```
Given three numbers: a, b and c
declare a variable: biggest
if a > b
    if a > c
        let biggest be a
    else
        let biggest be c
else
    if b < c
        let biggest be c
    else
        let biggest be b

return biggest
```

Previous Lab

Translate it into the actual c code.

```c
int max(int a, int b, int c){
    int biggest = 0;
    if(a < b){
        if(c > b){
            biggest = c;
        }else{
            biggest = b;
        }
    }else{
        if(c > a){
            biggest = c;
        }else{
            biggest = a;
        }
    }
    return biggest;
}
```

Today's Agenda

- Time Complexity of an Algorithm
    - Look back at prime finding algorithm.
- Recursion and Memory Layout

Finding Primes

Write a program that prints out all prime numbers from 1 to 100.

```
for(int i = 1; i <= 100; i++){
    int numfactor = 0;
    for (int j = 1; j <= i; j++){
        if(i % j ==0){
            numfactor = numfactor + 1;
        }
    }
    if(numfactor == 2){
        printf("%d ", i);
    }
}
```

How many iterations do we perform using the above code?

Making Code Faster!

1 + 2 + 3 + 4 ... 100 = 5050 iterations.

Can we make the code run faster? i.e., doing the same things but with less iterations?

Yes! We give up on checking factors when `numfactor > 2`.

```
for(int i = 1; i <= 100; i++){
    int numfactor = 0;
    for (int j = 1; j <= i; j++){
        if(i % j == 0){
            numfactor = numfactor + 1;
            if(numfactor > 2){
                break; // stop the current loop!
                       // go to check i+1!
            }
        }
    }
    if(numfactor == 2){
        printf("%d ", i);
    }
}
```

Making Code Even Faster!

The previous code runs only 1890 iterations!

Can we make the code even faster?

- For each `i`, there cannot be any factor bigger than `i/2` except itself.

```c
for(int i = 1; i <= 100; i++){
    int numfactor = 1; // start counting from 1!
    for (int j = 1; j <= i/2; j++){ // loop from 1 to i/2
        if(i % j == 0){
            numfactor = numfactor + 1;
            if(numfactor > 2){
                break;
            }
        }
    }
    if(numfactor == 2){
        printf("%d ", i);
    }
}
```

Time Complexity

The above code runs only for 715 iterations!

- You can write **different** algorithms that solves the **same** computational problem.
- Some are faster, some are slower.
- The number of elementary computing cycles (loop iterations) that are required for an algorithm is called **the time complexity of the algorithm**.
    - In our prime finding example, the time complexity of alg. 1 > alg. 2 > alg. 3.
    - Time complexity is an important metric of the algorithmic efficiency.

Unsolved Mystery

Given a computational problem, what is the lowest time complexity that one can achieve?

- For many problems, we know how fast we can get.
- Printing number from 1-100 requires 100 for loop iterations. These problems are called **easy** problems.
- Finding the solution of a sudoku game requires many more iterations. These problems are called **hard** problems.

Can we translate all **hard** problems into **simple** problems to achieve a lower time complexity?

- Nobody knows.

Recursion

- You **cannot** define a function inside another function.
- You can **call** a function inside another function.
  - A function can call itself!

Recursion

- How do you countdown from 10 to 1?
    - i.e., printing out number `10 9 8 ... 1`?
- You can use a `for` loop.

```
for(int i = 10; i >= 1; i = i - 1) {printf("%d", i);}
```

- Or, you can use the following process:

    i. initiate the countdown from i=10

    ii. If `i>=1`, then

        a. print the current number `i`

        b. continue the countdown from `i-1`.

- Weirdly, this procedure is defined in terms of itself, i.e., the countdown process refers to itself.

Recursion

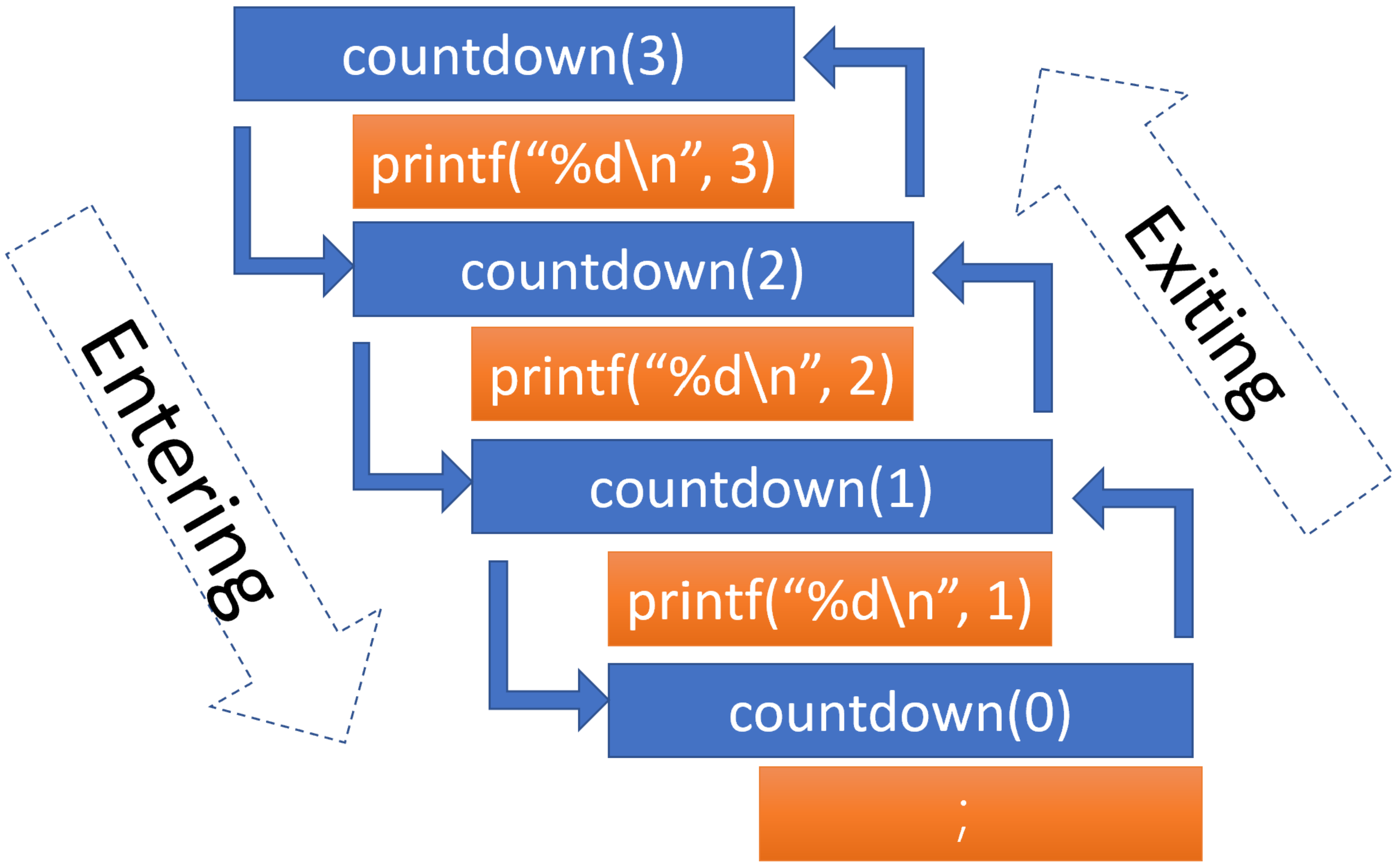
- A self-referential definition is called a **recursion**.
- Recursion is natural in math.
- Define $\mathbb{N}$, the set of all Natural Numbers.
    - 0 is in $\mathbb{N}$.
    - If i is in $\mathbb{N}$, $i + 1$ is in $\mathbb{N}$ too.
    - The smallest set that satisfies the above two criteria is the set of all natural numbers.
- Some mathematical process does not have a closed form, and can only be defined via a recursive process.

Recursion

-
  ```c
  void countdown(int i){
      if(i >= 1 ){
          printf("%d\n", i); // print
          countdown(i - 1); //countdown from n-1
      }
  }

  void main(){
      countdown(3); //initiate the countdown
  }
  ```
- Prints out `3 2 1.`
- For each `i >0` , it prints out `i` and continues to `countdown` with a smaller number `i-1` .

How does Recursion Work?



How does Recursion Work?

```c
void countdown(int i){
    printf("enter countdown(%d)\n", i); //new!
    if(i >= 1 ){
        printf("%d\n", i);
        countdown(i - 1);
    }
    printf("exit countdown(%d)\n", i); //new!
}
```

```
enter countdown(3)
3
enter countdown(2)
2
enter countdown(1)
1
enter countdown(0)
exit countdown(0)
exit countdown(1)
exit countdown(2)
exit countdown(3)
```

Memory Allocation for Functions

- When the function is being executed on the CPU, its data (such as variables declared in the function) and code are temporarily stored in the memory.
- The memory region for storing function data in the current program is called "stack".

Stack

- Stack is a data structure that the newest element is placed on top of the old elements.

Stack Memory Allocation

1. When a function is called, its data and code is added to the top of the stack.
2. CPU can only access data and code from **the top stack**.
    - It means, CPU can only access variables from the current function that is being called!
3. When a function finishes its execution, its data is removed from the stack and the space it occupies is freed for future calls of functions.

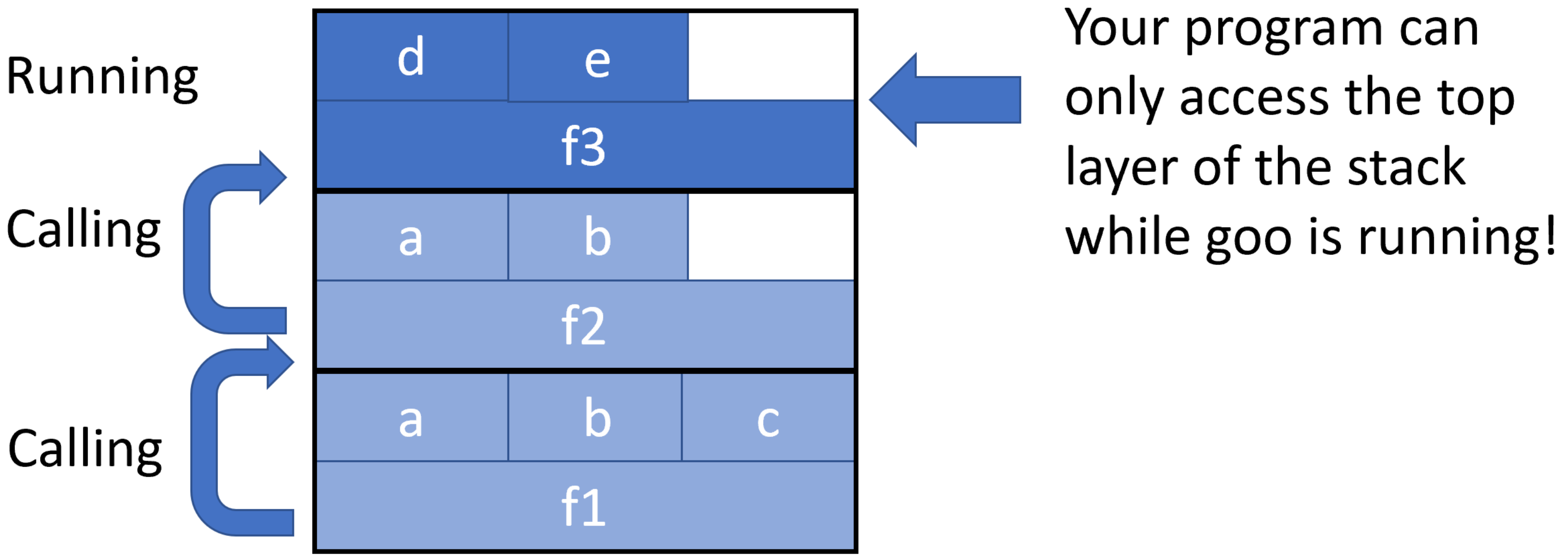
Stack Memory Allocation

Consider a situation where function `f1` calls `f2` and `f2` calls `f3`.

```c
void f3(){
    int d = 6, e = 7;
};
void f2(){
    int a = 4, b = 5;
    f3();
}
void f1(){
    int a = 1, b = 2, c = 3;
    f2();
};
void main(){
    f1();
}
```

Stack Memory Allocation

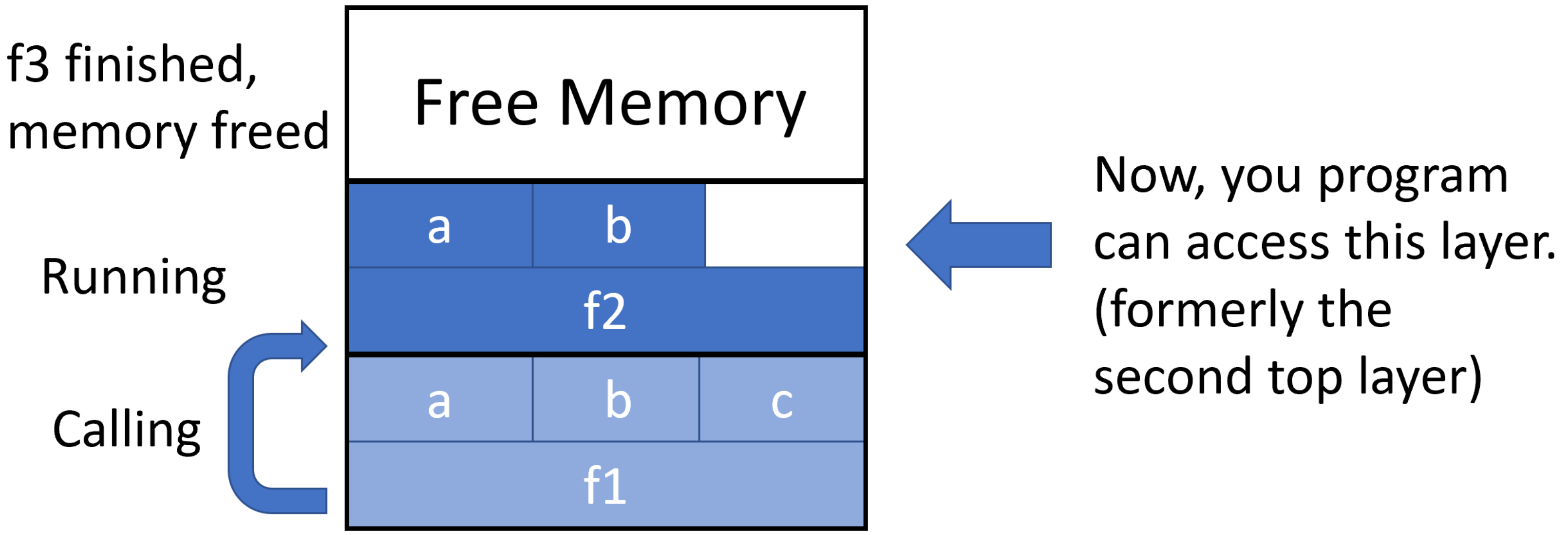
Below is the stack while `f3` is running.


Running
Calling
Calling
d
e
f3
a
b
f2
a
b
c
f1
Your program can only access the top layer of the stack while goo is running!

The only variables are accessible to us are `d` and `e`.

Stack Memory Allocation

When `f3` finishes running, its memory is freed.


f3 finished,
memory freed
Running
Calling
Free Memory
a
b
f2
a
b
c
f1
Now, you program can access this layer. (formerly the second top layer)

The only variables are accessible to us are `a` and `b` in `f2`.

When `f2` finishes running, its memory will be freed too and only variables in `f1` will be accessible.

Local Variables

Variables declared inside the function are called **local variables** (This includes all input argument variables!).

- They can only be accessed by the function where it is defined.
    - They cannot be accessed by other functions.
- **Why?** The program can only access the top layer of the stack, which stores variables of the **running function**.
- In the "f1-f2-f3" example, your program cannot access `a` and `b` defined in `f1` while `f2` is running.

Stack Memory Allocation

Stack is a highly efficient memory allocation/release mechanism.

- The memory allocation and release are all automatically handled by the OS.
- However, there is only a limited stack space for each program (determined by the OS). If a single function occupies a large memory space, or the call stack gets too "tall", we may run out of stack memory and an "runtime error" will be raised by the OS.
    - This out-of-memory error is called "Stack Overflow".

Conclusion

1. There can be multiple algorithms that solves the same computational task.
2. The computing cycles required for each algorithm is called time complexity.
3. The function definition that refers to itself is called recursion.
4. Function data and code are stored in stack memory.
    i. Local variables can only be accessed by the function where it is defined.

Labs

1. Download the lab file and unzip it to your labpack folder.

Homework 1

1. We are going to verify the time-complexity for different prime finding algorithms I introduced in the lecture.
2. Open `prime1.c`.
3. Slightly modify the code, so it prints out the number of elementary computing cycles (loop iterations) it has gone through in order to find all prime numbers <= 100.
   - For example, in the **pseudo code** below, the outer loop repeats 10 times and the inner loop repeats 11 times, so the program will go through 10*11 elementary computing cycles before it stops.

     ```
     for i from 1 to 10
         for j from 1 to 11
             print out i*j
     ```

Homework 2

1. Modify `prime2.c` and `prime3.c` and print out the number of elementary computing cycles.
2. Can you find an algorithm that requires even smaller number of computing cycles than the one described `prime3.c`?
3. Imagine a program that finds all prime numbers which are smaller than `n` by using the algorithm in `prime1.c`
   - Express the number of cycles executed by this program in terms of `n`.

Homework 3 (21-22 Exam Question)

```
#include <stdio.h>
void g(){
    int b = 4, c = 5;
    printf("g is being called!\n");
}
void f(){
    int a = 1, b = 2;
    g();
    printf("g has been called!\n");
}
void main(){
    f();
}
```

1. **Without running the program**, draw the stack memory layout (see lecture slides) when the program is printing out `g is being called!`.
2. Verify your answer using VSCode debugger (see video).

Homework 4 (submit)

0. Open `prime4.c`.
1. Write a function `num_factor` that takes input `i` and `j`, and output an integer.
   - For any input `i > j > 1`,
   - The function returns the number of `i`'s factors from 2 to `j`.
   - e.g. `i = 5, j = 4`, the output is 0.
   - e.g. `i = 16, j = 5`, the output is 2.
   - You are not allowed to use any loop. Use recursion.
2. Test it in `main()` with different input `i,j`.

Homework 4 (submit)

3. Now, make use of `num_factor` function you just wrote, write some additional code in `main` (don't modify `num_factor`), so it prints out all prime numbers from 1 to 100. You can use **one** loop here.
4. Rename `prime4.c` according to your student ID as you have done before, submit it to the blackboard.