# 601.220 Intermediate Programming

Recursion

# Recursion

- Most computation involves some form of repetition
  - Repeatedly make incremental progress until problem is solved
- You know how to use *loops* to expression repetition
  - `for` loops, `while` loops, etc.
- *Recursion* is another way to express repetition
  - Can be more succint than loops!
  - Leads to elegant solutions for some problems
  - Has some (potential) downsides

# How recursion works

Recursion works by

- dividing a large problem into one or more smaller problems, and then
- extending the solution to the smaller problem(s) to be a solution for the large problem

Recursion is implemented by having a function make a call *to itself* using different arguments. (This will seem really weird until you get the hang of it.)

A function calling itself is called a *recursive call*.

# Requirements for using recursion

When solving a problem using recursion:

1. The smaller problem(s) ("subproblems") must have the same form as the larger problem. Subproblems are solved recursively, by making another call to the same method using different parameter values.
2. There must be one or more instances of the problem that can be solved *without* recursion: these are known as the *base cases*.
3. All recursive calls must *make progress* towards a base case, so that the computation will eventually complete.

# Example problem

Let's say we want to compute the sum of the integers from 1 to $n$:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + (n-1) + n$$

Could write a loop:

```
unsigned sum_1_to_n(unsigned n) {
  unsigned sum = 0;
  for (unsigned i = 1; i <= n; i++) {
    sum = sum + i;
  }
  return sum;
}
```

# Solving using recursion

Key to solving a problem recursively is to find one or more *subproblems* of the same form within the larger problem.

In the case of summing integers 1 to $n$:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + (n-1) + n$$

Can you find a subproblem or subproblems?

# Solving using recursion

Key to solving a problem recursively is to find one or more *subproblems* of the same form within the larger problem.

In the case of summing integers 1 to $n$:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + (n-1) + n$$

Can you find a subproblem or subproblems?

This is the sum of integers from 1 to $n$ - 1! It has the same form as the overall problem, and can be solved recursively.

# Solving using recursion

Summing integers 1 to $n$ recursively:

$$\sum_{i=1}^{n} i = \underbrace{1 + 2 + \ldots + (n-1)}_{n-1} + n$$
$$= \sum_{i=1}^{n-1} i + n$$

Note that solving the subproblem *almost* solves the entire problem. All we need to do is add $n$ to the sum of integers 1 to $n-1$, and that gives us the sum of integers 1 to n. This is called *extending* the solution to the subproblem(s).

# Base cases

When solving a problem recursively, we need one or more *base cases* that can be solved without recursion. The recursion makes *progress* by working towards a base case.

Intuitively, base cases are "small" or "trivial" instances of the problem.

In the case of summing integers 1 to $n$, when $n = 0$, the sum is 0.

Another possibility: when $n = 1$, sum is 1.

# Implementing as a recursive function

```
unsigned sum_1_to_n(unsigned n) {
  if (n == 0) { // check base case
    return 0;
  }
  return sum_1_to_n(n - 1) + n;
}
```

# Does it work?

```c
// sumrec.c:
#include <stdio.h>

unsigned sum_1_to_n(unsigned n) {
  if (n == 0) { // check base case
    return 0;
  }
  return sum_1_to_n(n - 1) + n;
}

int main(void) {
  printf("sum 1..2 is %u\n", sum_1_to_n(2));
  printf("sum 1..10 is %u\n", sum_1_to_n(10));
  printf("sum 1..100 is %u\n", sum_1_to_n(100));
  return 0;
}
```

```
$ gcc sumrec.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
sum 1..2 is 3
sum 1..10 is 55
sum 1..100 is 5050
```

# Advice for using recursion successfully

Let's discuss a few techniques and principles that you should find useful when solving problems using recursion.

# Always check base case(s) first

A recursive function *must* check to see whether a base case was reached before attempting to solve a subproblem recursively.

Failing to do this could lead to an infinite recursion.

# Subproblems must be smaller than the overall problem

Recursive calls *must* make progress towards the base case. This means that the arguments passed to a recursive call must describe a subproblem that is *smaller* (simpler) than the overall problem.

Trying to solve the same problem recursively is, by definition, an infinite recursion.

# Make subproblem(s) as large as possible

When thinking about how to find one or more subproblems in the overall problem, keep in mind that you want the subproblem(s) to be as large as possible.

This ensures that only a small amount of work is required to extend the solution to the subproblem(s) to be a solution to the overall problem.

# The downside to recursion

Every function call requires an *stack frame*.

The stack frame is an area of memory where variables and temporary values for a function call are stored, as well as the *return address* indicating the code location where the function call will return to.

Every recursive function call creates a *new* stack frame.

- The *call stack*, where stack frames are allocated, is limited in size.
- "Deep" recursion may fail because the call stack size is exceeded.

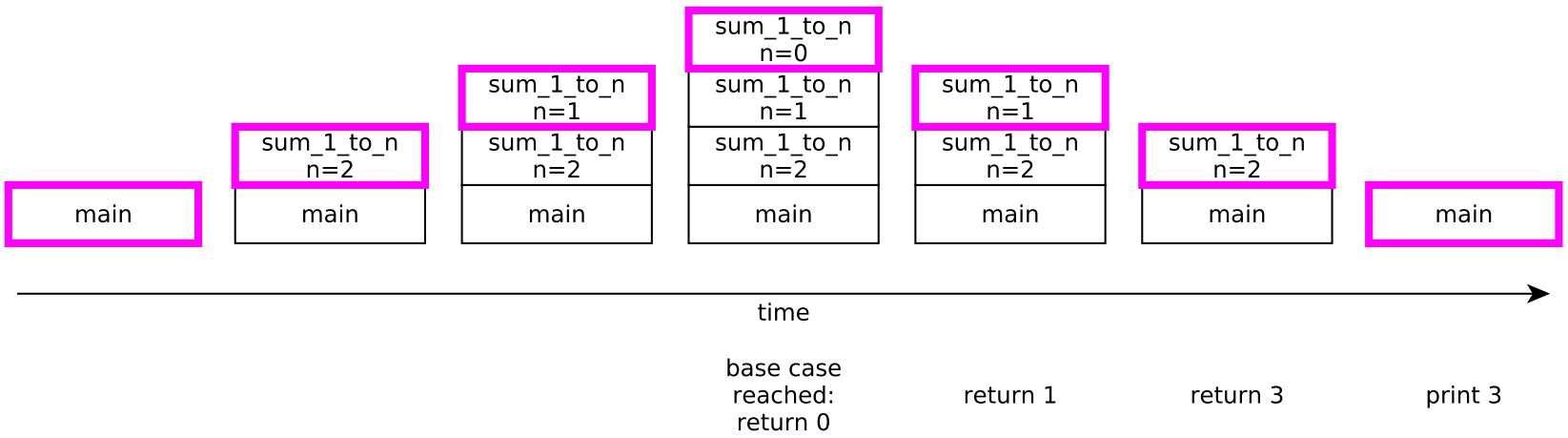
# Tracing a recursive computation

Consider this program:

```
unsigned sum_1_to_n(unsigned n) {
  if (n == 0) { // check base case
    return 0;
  }
  return sum_1_to_n(n - 1) + n;
}

int main(void) {
  printf("%u\n", sum_1_to_n(2));
  return 0;
}
```

# Tracing a recursive computation




Note that the maximum number of stack frames at the "deepest" point is proportional to the value of `n` passed to `sum_1_to_n`

# Recursion vs. loop

- Avoid deep recursion
- Recursion is great for computations on inherently recursive structures
  - Many data structures are recursive (e.g., trees)
- Recursion is great for "divide and conquer" algorithms
  - E.g., merge sort, quicksort