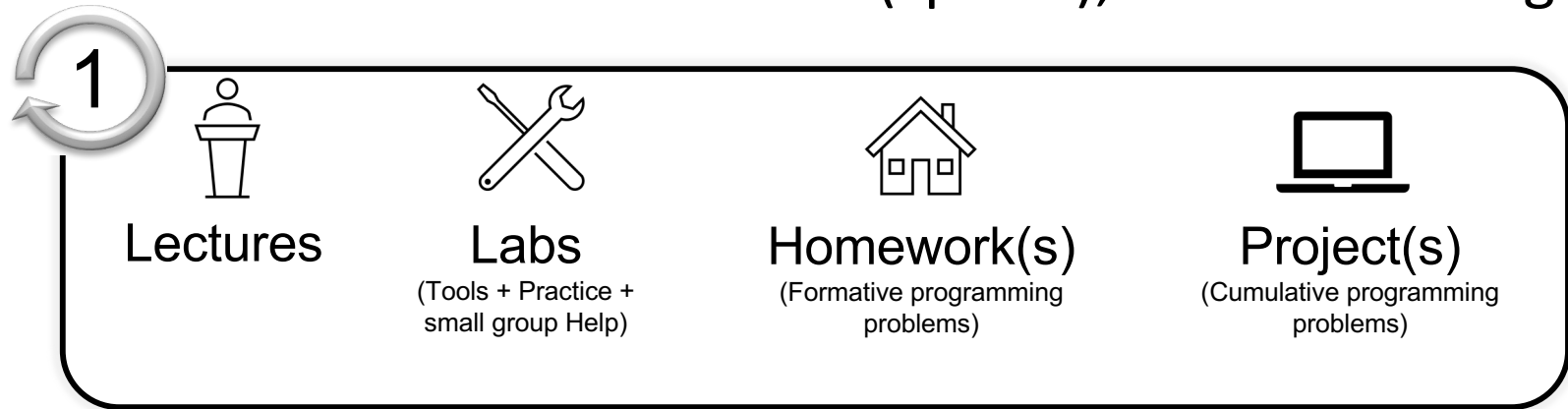


# CS103 Unit 6a - Recursion

CSCI 103L Teaching Team

# Unit 6 – Recursion

- The course is broken into 6 units (spirals), each consisting of:



**C++ Language  
Syntax**



**Algorithms and  
Computational Thinking**



**Memory, Objects,  
and I/O Pt. 1**



**Basic Data  
Structures**



**Memory, Objects,  
and I/O Pt. 2**



**Recursion**

# Recursion

- Defining an object, mathematical function, or computer function in terms of *itself*



GNU

- Makers of gedit, g++ compiler, etc.
- GNU = GNU is Not Unix

GNU is Not Unix

GNU is Not Unix

... is Not Unix is not Unix is Not Unix

# Recursion

- Problem in which the solution can be expressed in terms of *smaller versions of itself* and *a base/terminating case*
- Can often take the place of a loop and can lead to much more elegant coding solutions that if we were to use a loop
  - More guidance on this later
- Input to the problem must be categorized as a:
  - **Base case**: Small version of problem whose known beforehand or easily computable (no recursion needed)
  - **Recursive case**: Solution can be described using solutions to smaller problems of the same type
    - Keeping putting in terms of something smaller until we reach the base case
- Factorial:  $n! = n * (n-1) * (n-2) * \dots * 2 * 1$ 
  - $n! = n * (n-1)!$
  - **Base case**:  $n = 1$
  - **Recursive case**:  $n > 1 \Rightarrow n * (n-1)!$

# Recursive Function Mechanics

- Avoid infinite recursions by
  - Put the base case check first
  - Ensuring you recurse on a smaller problem [e.g.  $f(n)$  should NOT recurse on  $f(n)$ ] **but a smaller version of the problem**
- The system stack provides separate areas of memory for each **instance** of a recursive function
  - Thus each **local variable** and **actual parameter** of a function is separate and isolated from other recursive function instances [e.g. each  $\text{fact}(n)$  has it's own version of  $n$ ]

C Code:

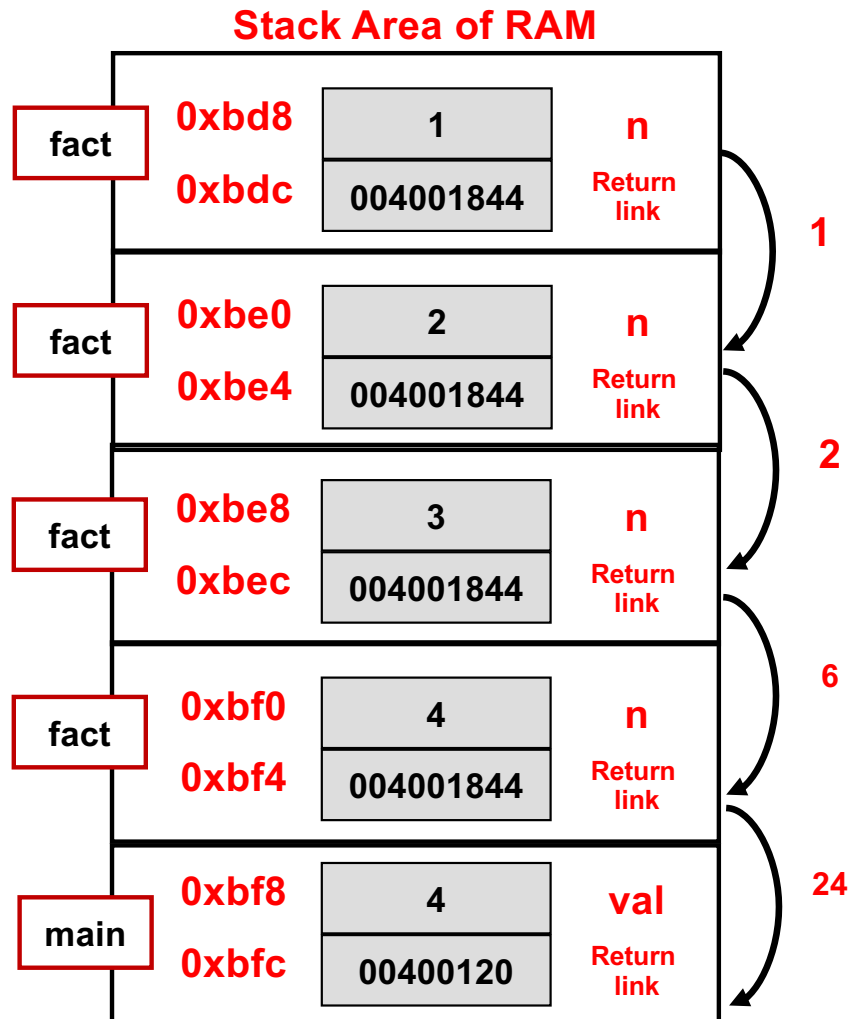
```
int fact(int n)
{
    if(n == 1){
        // base case
        return 1;
    }
    else {
        // recursive case
        return n * fact(n-1);
    }
}
```

Factorial:

- $n! = n * (n-1) * (n-2) * \dots * 2 * 1$
- $n! = n * (n-1)!$ 
  - Base case:  $n = 1$
  - Recursive case:  $n > 1 \Rightarrow n * (n-1)!$

# Recursion & the Stack

- Must return back through the each call



Value/version of n is implicitly “saved” and “restored” as we move from one instance of the ‘fact’ function to the next

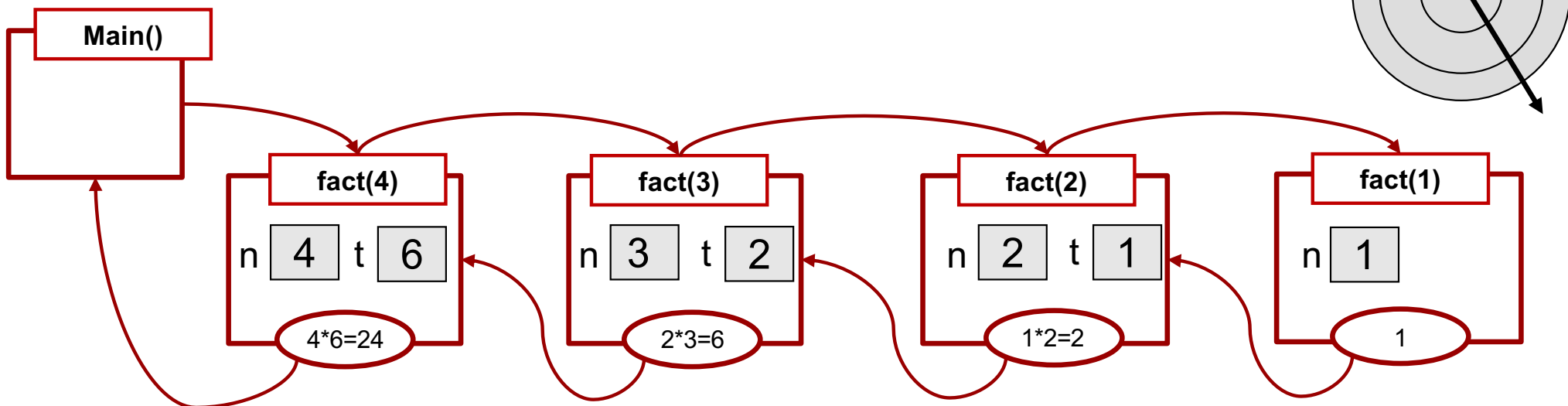
```
int fact(int n)
{
    if(n == 1){
        // base case
        return 1;
    }
    else {
        // recursive case
        return n * fact(n-1);
    }
}

int main()
{
    int val = 4;
    cout << fact(val) << endl;
}
```

# Recursive Analysis Tip: Box Diagrams

- Just for example's sake let's introduce a temporary local variable, *t*
- To analyze recursive functions draw a **box diagram** which is...
  - A simplified view of each function instance on the stack
- One box per function
  - Show arguments, pertinent local variables, and return values

```
int fact(int n)
{
    if(n == 1){
        return 1;
    }
    else {
        int t = fact(n-1);
        return n * t;
    }
}
```



# Head vs. Tail Recursion

- Head Recursion: Recursive call is made before the real work is performed in the function body
- Tail Recursion: Some work is performed and then the recursive call is made

## Tail Recursion

```
void doit(int n)
{
    if(n == 1) cout << "Stop" << endl;
    else {
        cout << "Go" << endl;
        doit(n-1);
    }
}
```

## Head Recursion

```
void doit(int n)
{
    if(n == 1) cout << "Stop" << endl;
    else {
        doit(n-1);
        cout << "Go" << endl;
    }
}
```

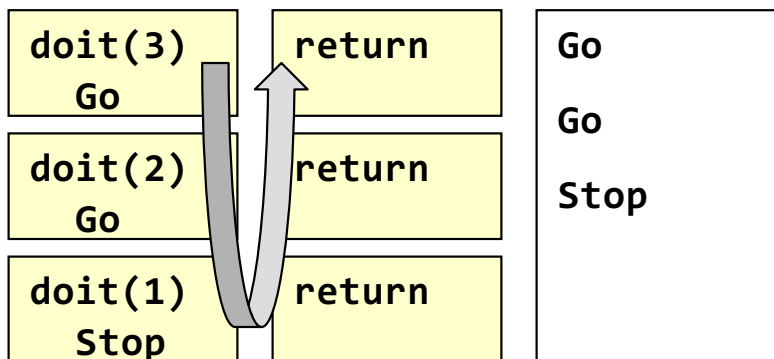


# Head vs. Tail Recursion

- Head Recursion: Recursive call is made before the real work is performed in the function body
- Tail Recursion: Some work is performed and then the recursive call is made

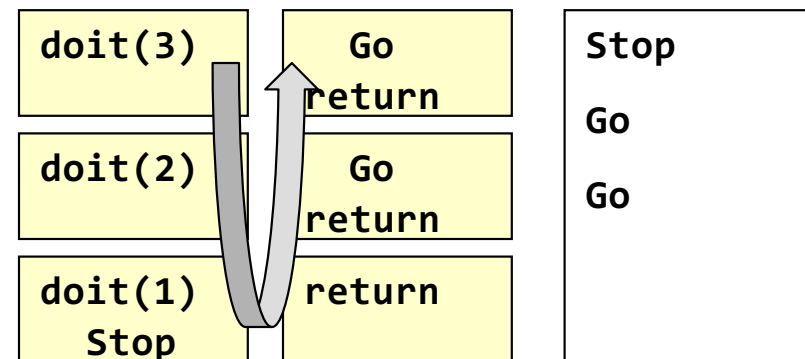
## Tail Recursion

```
Void doit(int n)
{
    if(n == 1) cout << "Stop";
    else {
        cout << "Go" << endl;
        doit(n-1);
    }
}
```



## Head Recursion

```
Void doit(int n)
{
    if(n == 1) cout << "Stop";
    else {
        doit(n-1);
        cout << "Go" << endl;
    }
}
```



# Exercise

- Exercises
  - Count-down
  - Count-up

# Steps to Formulating Recursive Solutions

1. Write out some solutions for a few input cases to discover how the problem can be decomposed into smaller problems of the same form
  - Does solving the problem on an input of smaller value or size help formulate the solution to the larger
2. Identify the base case
  - An input for which the answer is trivial
3. Identify how to combine the small solution(s) to solve the larger problem

Put another way, there are often 2 principles to finding recursive solutions

- Principle 1: Generally recursive functions are "responsible" for ONLY 1 value / element / item and use recursion to handle all remaining items
- Principle 2: Determine how to combine the 1 element you are responsible for and the answer returned by recursion
  - Assume via the "magic of recursion" you get the answer for all remaining elements. How can you combine that with your 1 element to form the large solution

# Recursive Functions

- **Many loop/iteration based approaches can be defined recursively as well**
- How could **summing an array** be implemented recursively?
  - Question to answer:
    - What 1 thing could each recursion be responsible for?
    - How could we combine that 1 thing with the solution from the recursive call?
- Once you have a recursive formulation all that's left is the mechanics of the implementation (i.e. arguments & return values)

800  
data[4]: 8 6 7 9  
0 1 2 3

C Code:

```
int main() {
    int data[4] = {8, 6, 7, 9};
    int size=4;
    int sum1 = isum_it(data, size);
    int sum2 = rsum_it(data, size);
}

int isum_it(int data[], int len)
{
    int sum = data[0];
    for(int i=1; i < len; i++){
        sum += data[i];
    }
    return sum;
}

int rsum_it(int data[], int len)
{
    if(len == 1)
        return _____;
    else
        return _____;
}
```

# Recursive Call Timeline

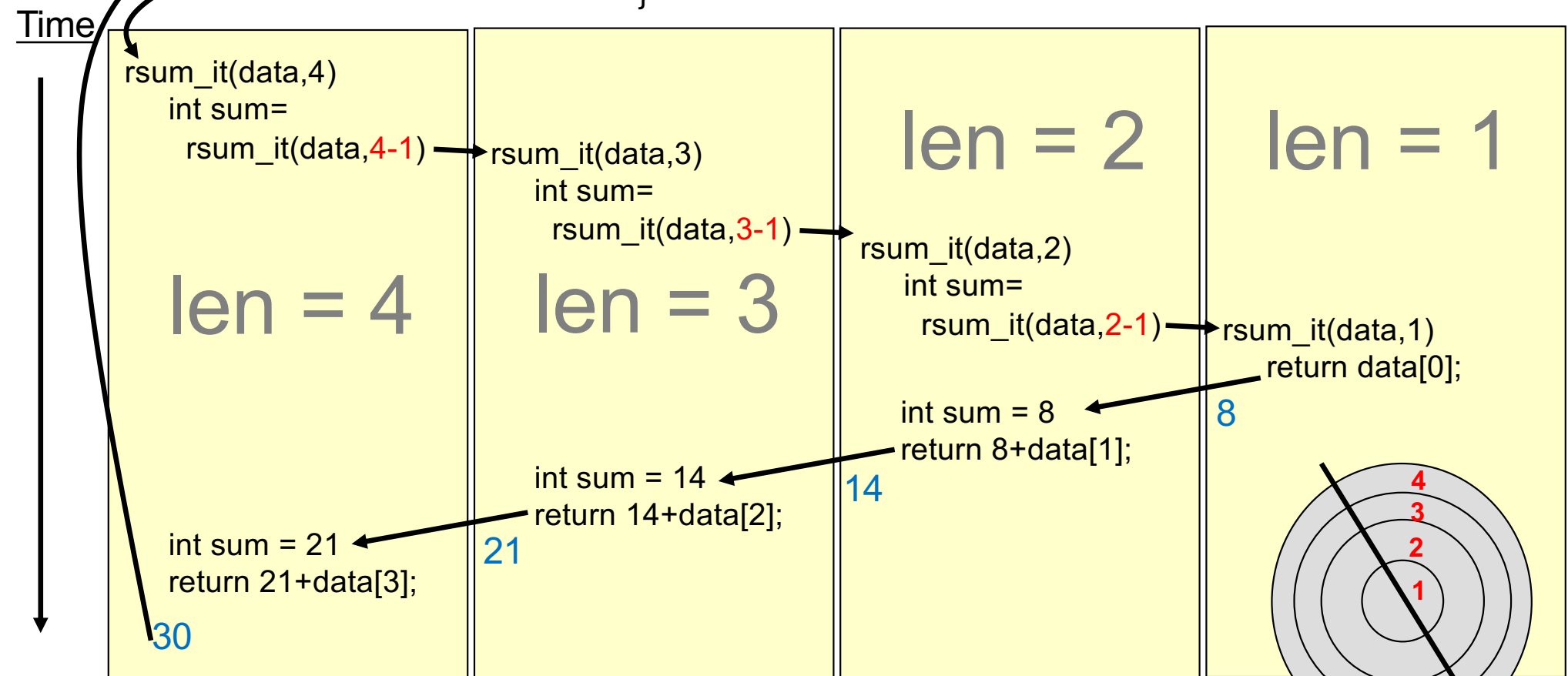
```
int main(){
    int data[4] = {8, 6, 7, 9};
    int size=4;
    cout << rsum_it(data, size);
    ...
}
```

```
int rsum_it(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int sum = rsum_it(data, len-1);
        return sum + data[len-1];
}
```

800

8	6	7	9
---	---	---	---

data[4]: 0 1 2 3



Each instance of rsum\_it has its own **len** argument and **sum** variable

Every instance of a function has its own copy of local variables

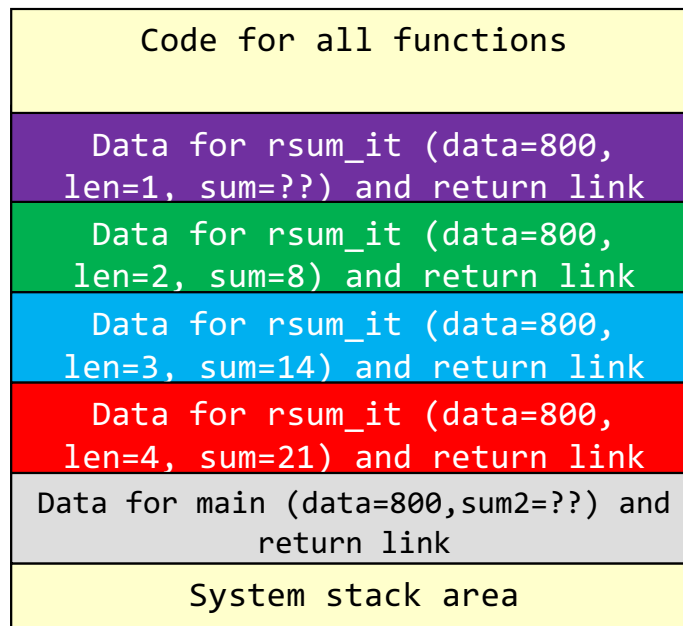
# System Stack & Recursion

- The system stack makes recursion possible by providing separate memory storage for the local variables of each running instance of the function

```
int main()
{
    int data[4] = {8, 6, 7, 9};
    int sum2 = rsum_it(data, 4);
}

int rsum_it(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int sum =
            rsum_it(data, len-1);
        return sum + data[len-1];
}
```

**System  
Memory  
(RAM)**



800

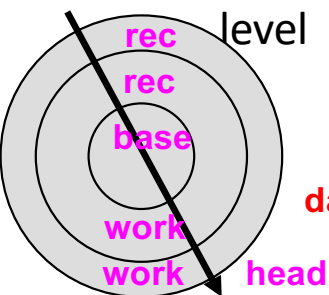
8	6	7	9
---	---	---	---

data[4]:    0   1   2   3

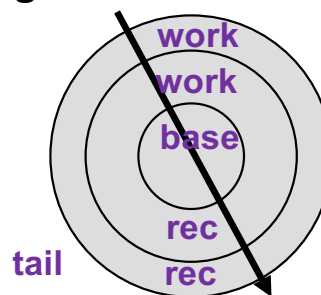
# Head and Tail Recursion Revisited

C Code:

- Both head and tail recursion methods are shown
- Head recursion:**
  - Recurses first and combines the answer received in the return value with the one element it is responsible for.
- Tail recursion:**
  - The recursive case is usually just what would have been in the body of a loop (of an iterative approach) BUT WITH a recursive call at the end to start the next "iteration/recursion".
  - Usually requires additional arguments to track the progress.
  - The final result is usually "complete and ready" when the base case is hit and just needs to be returned through each recursive level



800  
**data[4]:**    8   6   7  
               0   1   2



```
int main()
{
    int data[4] = {8, 6, 7};
    int size=4;
    int sumh = rsumh(data, size);
    int sumt = rsumt(data, size, 0, 0);
}

int rsumh(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int prevsum = rsum_it(data, len-1);
        return prevsum + data[len-1];
}

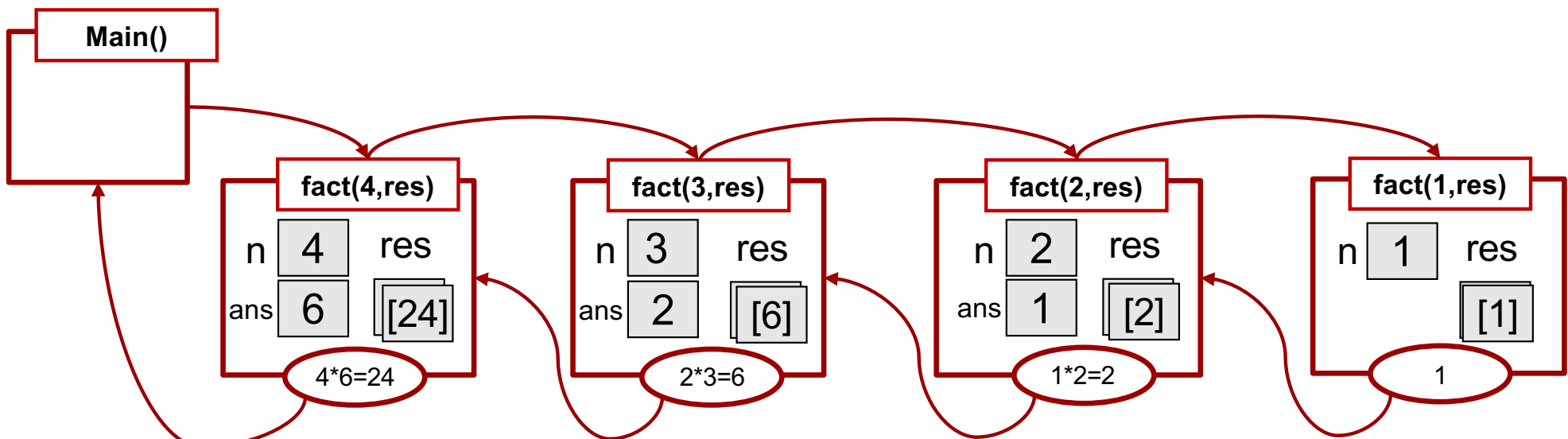
int rsumt(int data[], int len,
          int i, int prevsum)
{
    if(i == len) return prevsum;

    prevsum += data[i];
    return rsumt(data, len, i+1, prevsum);
}
```

# Tail Recursion w/ Reference Parameter

- What if we didn't want to JUST compute  $n!$  but compute AND store all the factorial values between  $1!$  and  $n!$  in an array/vector
- What do you think of the code on the right to achieve that goal?
  - Hint: Consider how the vector is being passed.

```
// res is short for "results"
int fact(int n, vector<int> res)
{
    if(n == 1){
        res.push_back(1);
        return 1;
    }
    else {
        int ans = fact(n-1, res) * n;
        res.push_back( ans );
        return ans;
    }
}
```



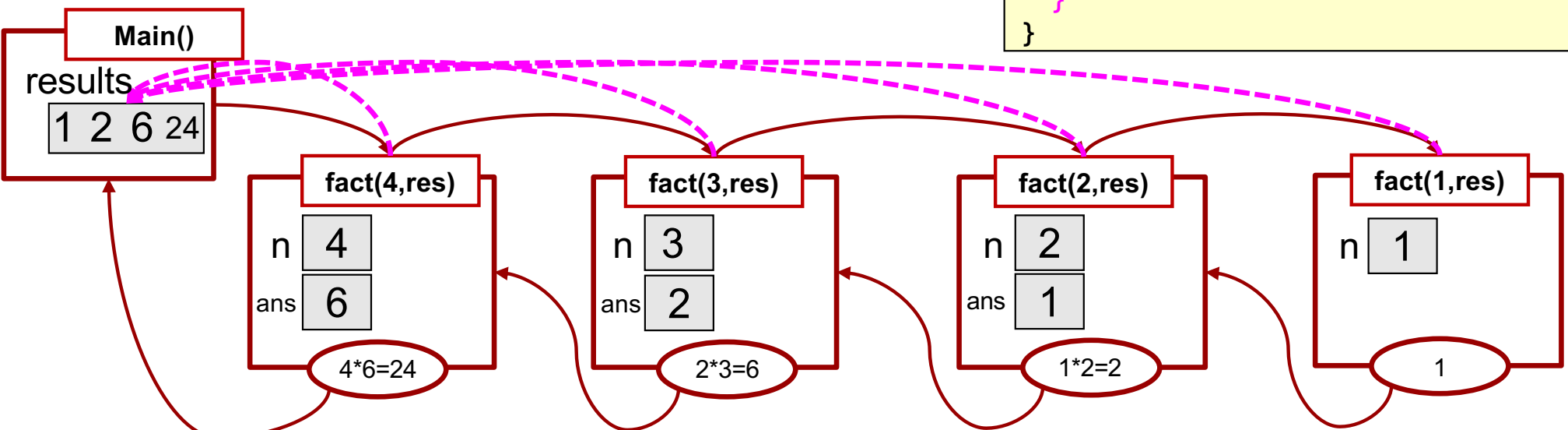


# Tail Recursion w/ Reference Parameter

- What if we didn't want to JUST compute  $n!$  but compute AND store all the factorial values between  $1!$  and  $n!$  in an array/vector
- Often this is a better way:
  - Have all recursions take in a **REFERENCE argument** so that all recursive calls see/share one object/variable that can collect their results.

```
int main() {
    vector<int> results;
    fact(4, results);
    // use results
}

int fact(int n, vector<int>& res )
{
    if(n == 1){
        res.push_back(1);
        return 1;
    }
    else {
        int ans = fact(n-1, res) * n;
        res.push_back( ans );
        return ans;
    }
}
```



# Recursion Double Check

- When you write a recursive routine:
  - Check that you have appropriate base cases
    - Need to check for these first before recursive cases
  - Check that each recursive call makes progress toward the base case
    - Otherwise you'll get an infinite loop and stack overflow
  - Check that you use a 'return' statement at each level to return appropriate values back to each recursive call
    - You have to return back up through every level of recursion, so make sure you are returning something (the appropriate thing)

# Loops & Recursion

- Is it better to use recursion or iteration?
  - ANY problem that can be solved using recursion can also be solved with iteration and other appropriate data structures
    - Often 1 recursive call per function can be implemented with a loop
- Why use recursion?
  - Usually clean & elegant.
  - For some problems an iterative approach would be very difficult to formulate but the recursive approach is small and elegant
    - Often occurs when the function instance makes *multiple (more than 1) recursive calls*
  - Can lead to stack overflow if recursive depth is too large
- How do you choose?
  - Iteration is usually faster and uses less memory and should be used when possible
  - However, if iteration produces a very complex solution, consider recursion

```
void rfunc1(int n)
{
    ...
    rfunc1(n-1);
    ...
}
```

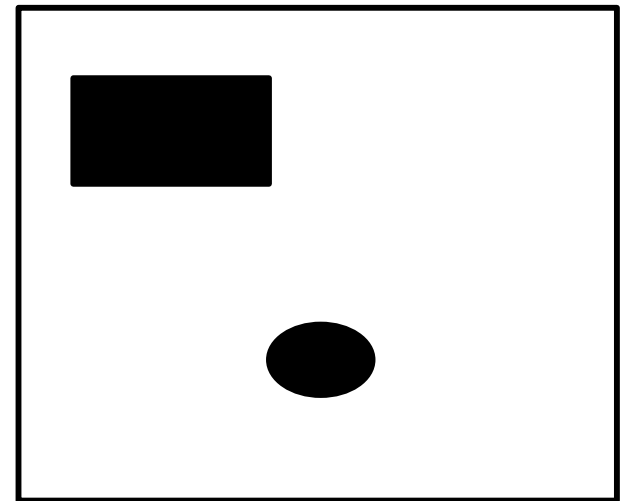
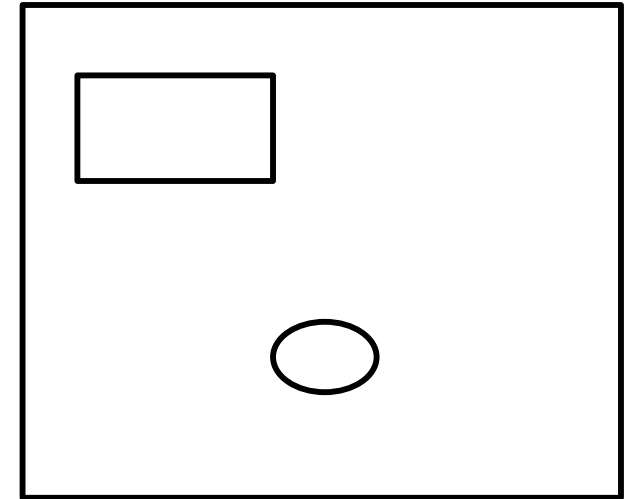
```
void rfunc2(int n)
{
    ...
    t = rfunc2(n-1);
    s = rfunc2(n-2);
    ...
}
```

# Flood Fill Exercise

- Codio Recursion Exercise

# Flood Fill

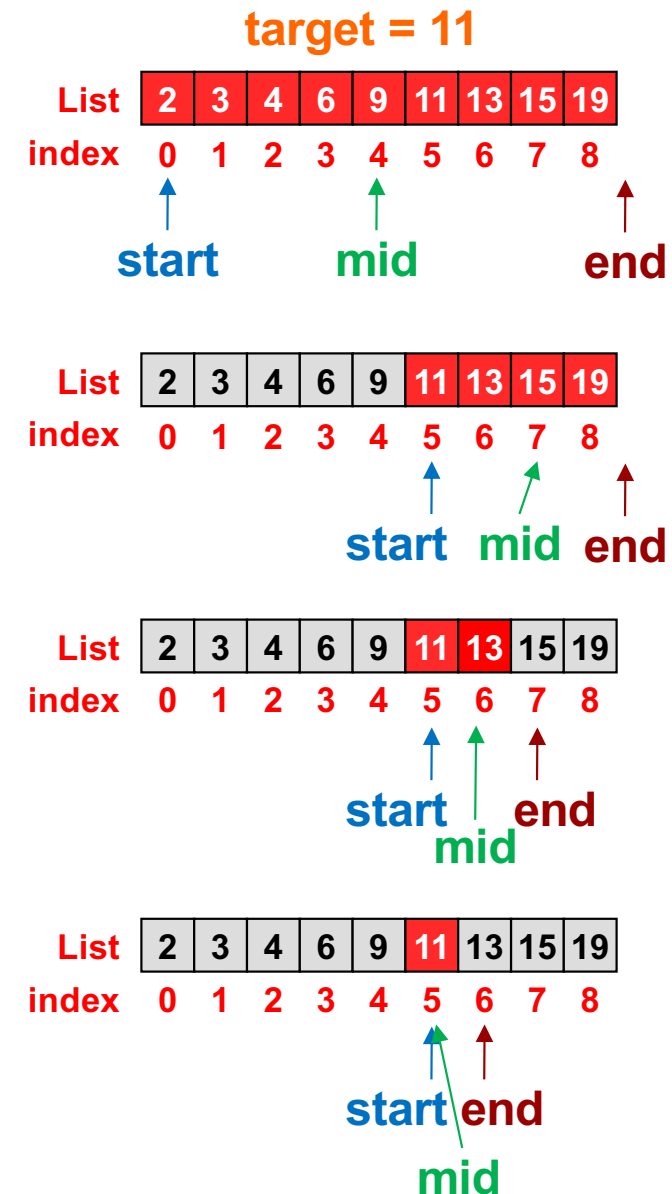
- Imagine you are given an image with outlines of shapes (boxes and circles) and you had to write a program to shade (make black) the inside of one of the shapes. How would you do it?
- Flood fill is a recursive approach
- Given a pixel
  - Base case: If it is black already, stop!
  - Recursive case: Call floodfill on each neighbor pixel
  - Hidden base case: If pixel out of bounds, stop!



# MORE EXAMPLES

# Recursive Binary Search

- Assume remaining items = [start, end)
  - start** is the **inclusive** index of start item in remaining list
  - end** is the **exclusive** index of end item (1 beyond the end) in remaining list
  - Put another way, consider items at index  $i$  such that:  $start \leq i < end$
- `binSearch(target, list[], start, end)`
  - Perform base check (empty range/list)
    - Return NOT FOUND (-1)
  - Pick **mid** item
  - Based on comparison of **target** with `list[mid]`
    - EQ => Found => return **mid**
    - LT => return answer to `binSearch[start, mid)`
    - GT => return answer to `binSearch[mid+1, end)`



# Sorting

- If we have an unordered list, sequential search becomes our only choice
- If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
- Many sorting algorithms of differing complexity (i.e. faster or slower)
- Bubble or Selection Sort
  - Simple though not terribly efficient
  - On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size \_\_\_\_\_ (i.e. w/o the newly placed maximum value)

**List** 7 3 8 6 5 1  
**index** 0 1 2 3 4 5  
**Original**

**List** 3 7 6 5 1 8  
**index** 0 1 2 3 4 5  
**After Pass 1**

**List** 3 6 5 1 7 8  
**index** 0 1 2 3 4 5  
**After Pass 2**

**List** 3 5 1 6 7 8  
**index** 0 1 2 3 4 5  
**After Pass 3**

**List** 3 1 5 6 7 8  
**index** 0 1 2 3 4 5  
**After Pass 4**

**List** 1 3 5 6 7 8  
**index** 0 1 2 3 4 5  
**After Pass 5**



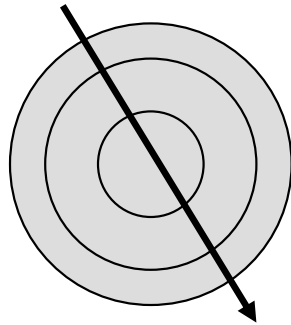
# Exercise

- Exercises
  - Text-based fractal

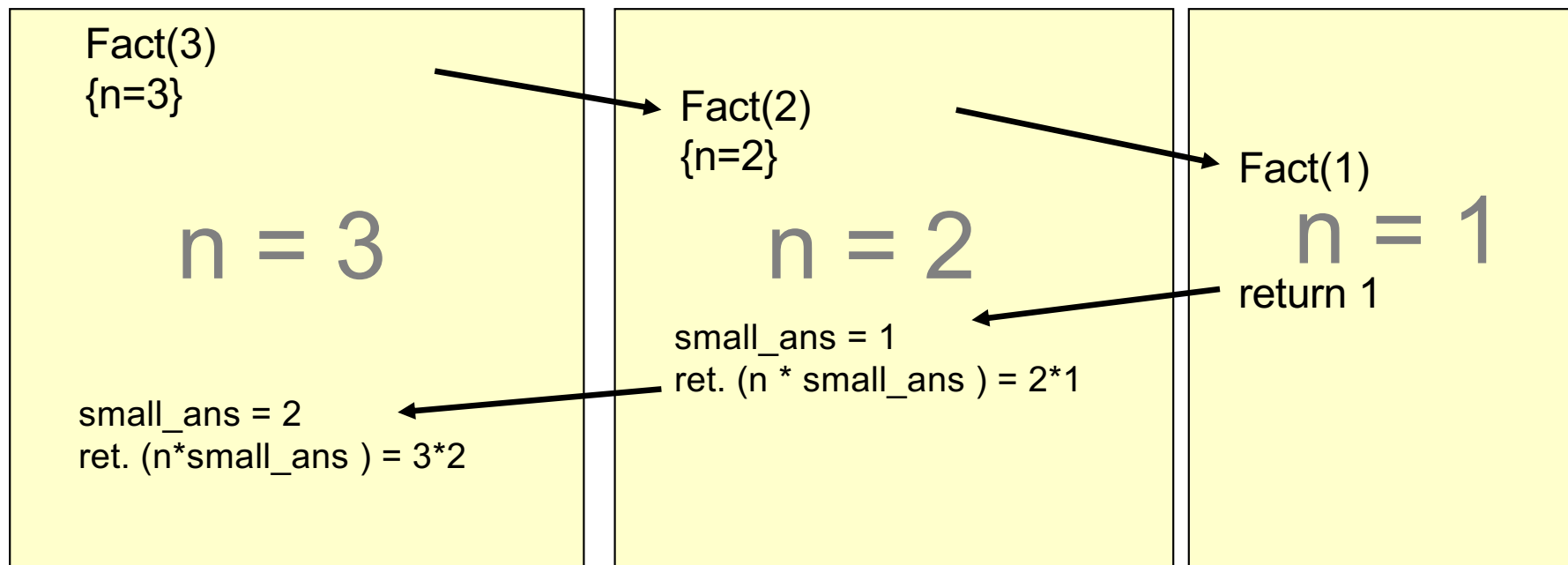
**OLD/UNUSED**

# Recursive Call Timeline

Time



```
int fact(int n)
{
    if(n == 1)
        return 1;
    else {
        int small_ans = fact(n-1);
        return n * small_ans ;
    }
}
```



- Value/version of  $n$  is implicitly “saved” and “restored” as we move from one instance of the ‘fact’ function to the next