ECE 220

Lecture x000C - 10/07

Slides based on material originally by: Yuting Chen & Thomas Moon

Review

Review

- Last time we discussed sorting & searching

Review

- Last time we discussed sorting & searching
  - Selection sort

Review

- Last time we discussed sorting & searching
    - Selection sort
    - Insertion sort

Review

- Last time we discussed sorting & searching
  - Selection sort
  - Insertion sort
  - Quick sort

Review

- Last time we discussed sorting & searching
    - Selection sort
    - Insertion sort
    - Quick sort
    - Bubble sort

Review

- Last time we discussed sorting & searching
  - Selection sort
  - Insertion sort
  - Quick sort
  - Bubble sort

A. Select next element and move things to place it in proper spot

Review

- Last time we discussed sorting & searching
  - Selection sort
  - Insertion sort
  - Quick sort
  - Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps

Review

- Last time we discussed sorting & searching
  - Selection sort
  - Insertion sort
  - Quick sort
  - Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps
C. Find minimum in each pass and bring to appropriate spot

Review

- Last time we discussed sorting & searching
  - Selection sort
  - Insertion sort
  - Quick sort
  - Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps
C. Find minimum in each pass and bring to appropriate spot
D. Pick pivot & move elements to left (lesser) or right (greater) of pivot

Review

- Last time we discussed sorting & searching
  - Selection sort
  - Insertion sort
  - Quick sort
  - Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps
C. Find minimum in each pass and bring to appropriate spot
D. Pick pivot & move elements to left (lesser) or right (greater) of pivot

Review

- Last time we discussed sorting & searching
  - Selection sort

A
  - Insertion sort
  - Quick sort

B
  - Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps
C. Find minimum in each pass and bring to appropriate spot
D. Pick pivot & move elements to left (lesser) or right (greater) of pivot

Review

- Last time we discussed sorting & searching

C • Selection sort

A • Insertion sort

• Quick sort

B • Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps
C. Find minimum in each pass and bring to appropriate spot
D. Pick pivot & move elements to left (lesser) or right (greater) of pivot

Review

- Last time we discussed sorting & searching

C • Selection sort

A • Insertion sort

D • Quick sort

B • Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps
C. Find minimum in each pass and bring to appropriate spot
D. Pick pivot & move elements to left (lesser) or right (greater) of pivot

Review

- Last time we discussed sorting & searching

C • Selection sort

A • Insertion sort

D • Quick sort

B • Bubble sort

A. Select next element and move things to place it in proper spot
B. Keep comparing pairs and swapping them till no more swaps
C. Find minimum in each pass and bring to appropriate spot
D. Pick pivot & move elements to left (lesser) or right (greater) of pivot

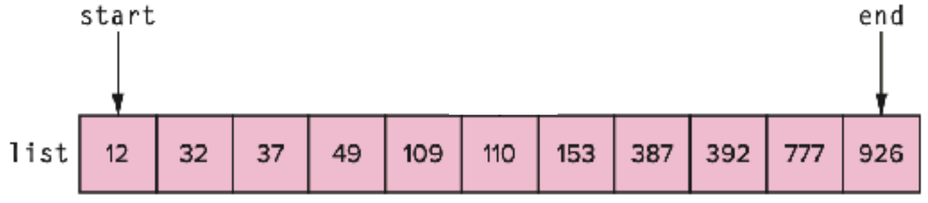
Complete codes for available on Github: https://github.com/iabraham/ece220-fa25/tree/main/lec11_1002/sorts

Lesson objectives

- Understand recursion as a *divide-and-conquer* programming technique.
    - Understand difference between recursive cases and terminating/ base cases.
    - Understand recursion implicitly uses stack data structure.
- Be able to implement recursive functions in C.
- Be (in principle) be able to convert recursive functions to iterative ones.
- Be able to lower recursive C functions to LC3 assembly.

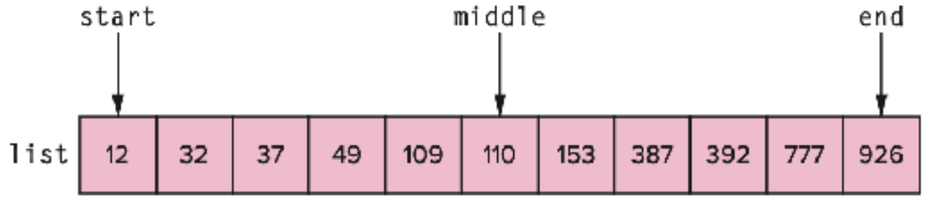
Recap binary search

Key = 109



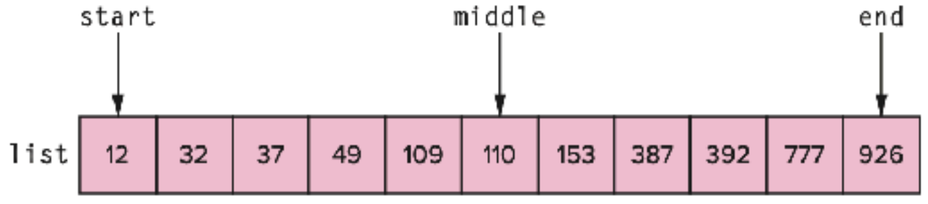
Recap binary search

Key = 109 **Pick middle**



Recap binary search

Key = 109 **Pick middle** **Is middle == key?**



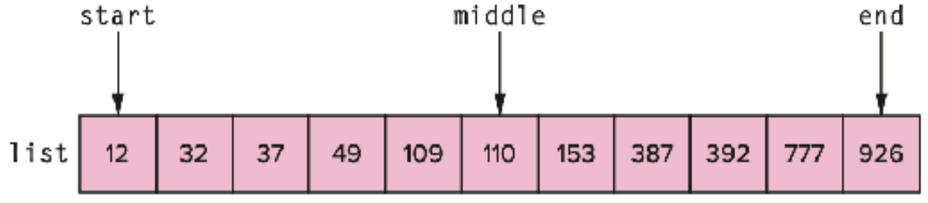
Recap binary search

Key = 109

Pick middle

Is middle > key?

Is middle == key?

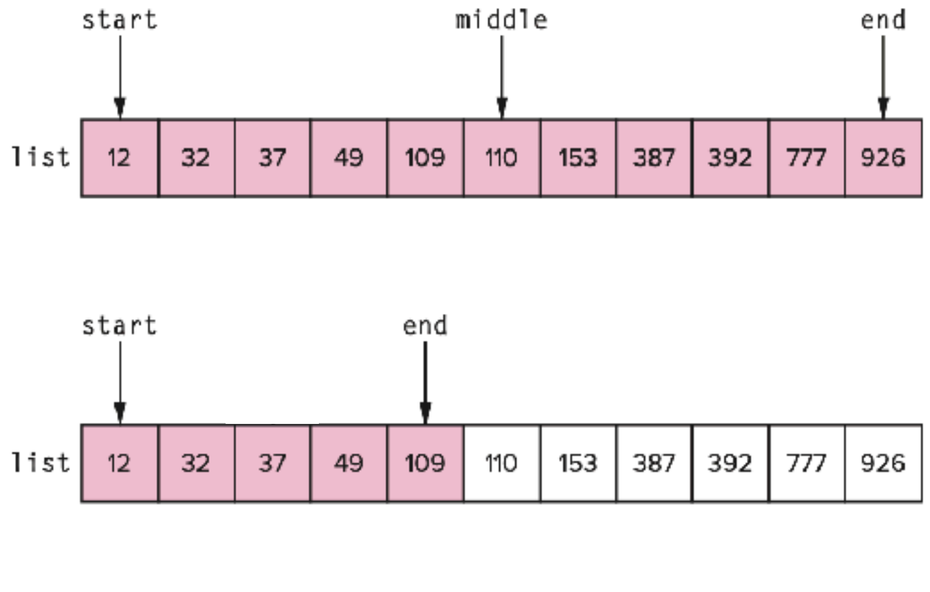


Recap binary search

Key = 109

Pick middle
Is middle > key?

Is middle == key?
Go left

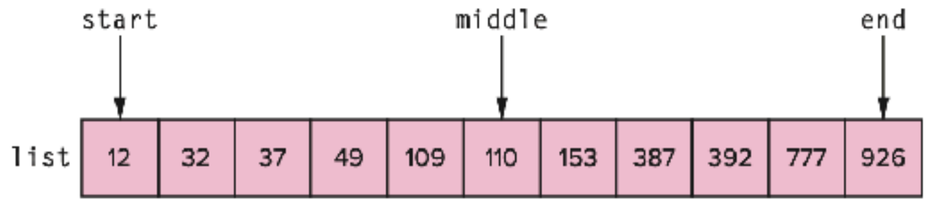


Recap binary search

Key = 109

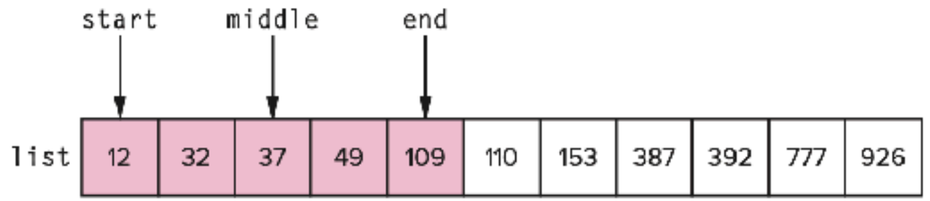
Pick middle
Is middle > key?

Is middle == key?
Go left



Pick middle

Is middle == key?



Recap binary search

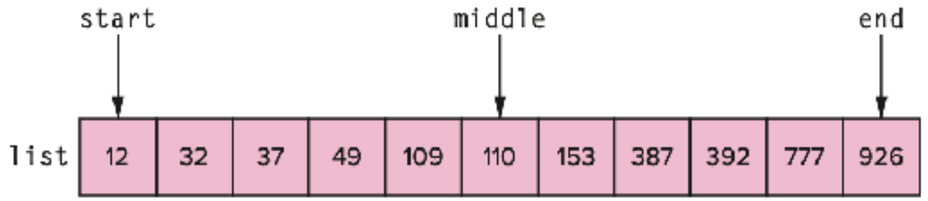
Key = 109

Pick middle

Is middle > key?

Is middle == key?

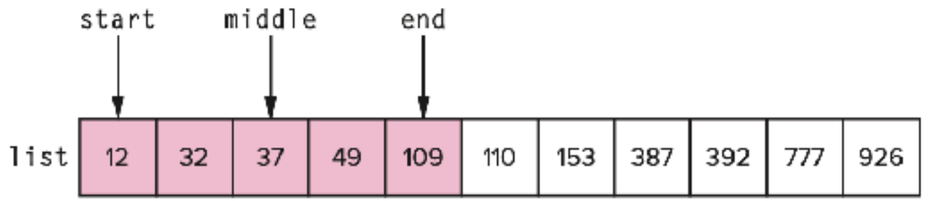
Go left



Pick middle

Is middle > key?

Is middle == key?



Recap binary search

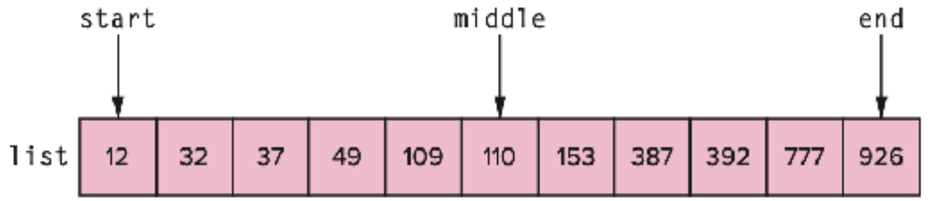
Key = 109

Pick middle

Is middle > key?

Is middle == key?

Go left

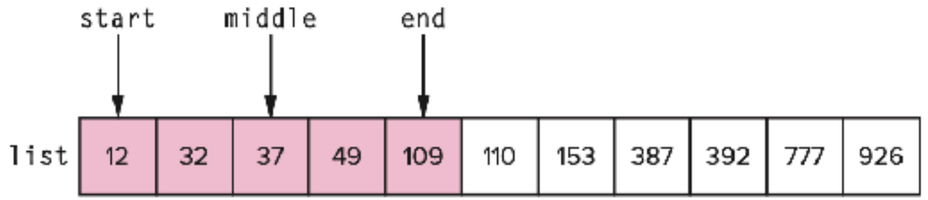


Pick middle

Is middle > key?

Is middle == key?

Is middle < key?



Recap binary search

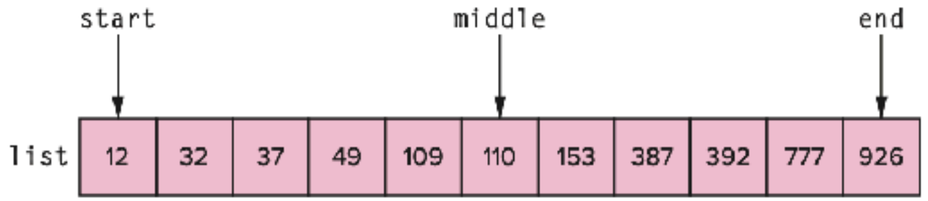
Key = 109

Pick middle

Is middle > key?

Is middle == key?

Go left



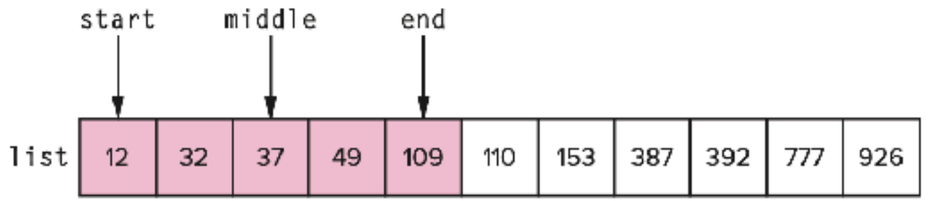
Pick middle

Is middle > key?

Is middle == key?

Is middle < key?

Go right



Recap binary search

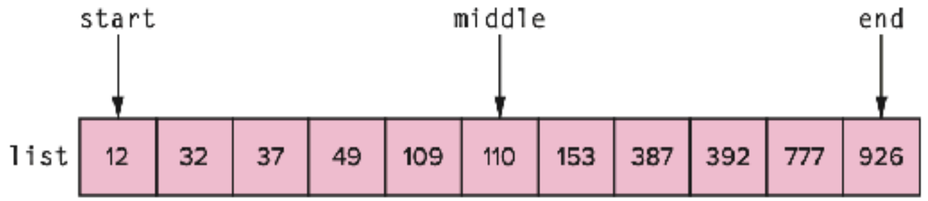
Key = 109

Pick middle

Is middle > key?

Is middle == key?

Go left



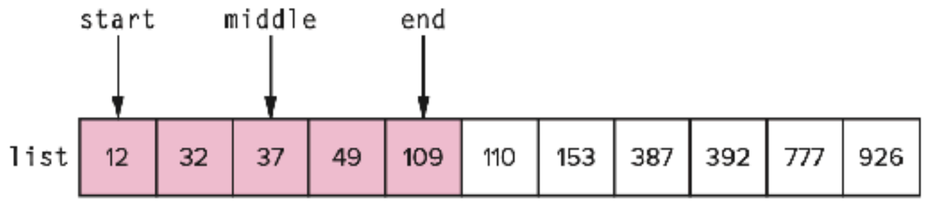
Pick middle

Is middle > key?

Is middle == key?

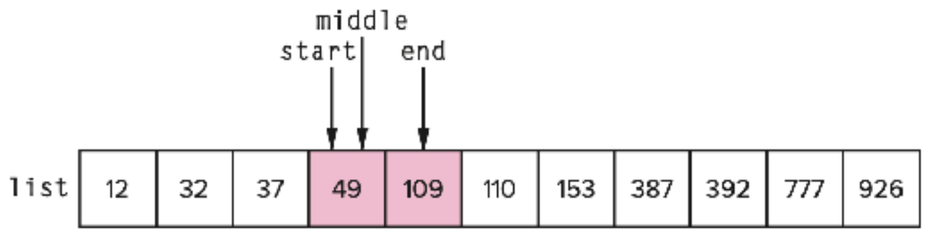
Is middle < key?

Go right



Pick middle

Is middle == key?



Recap binary search

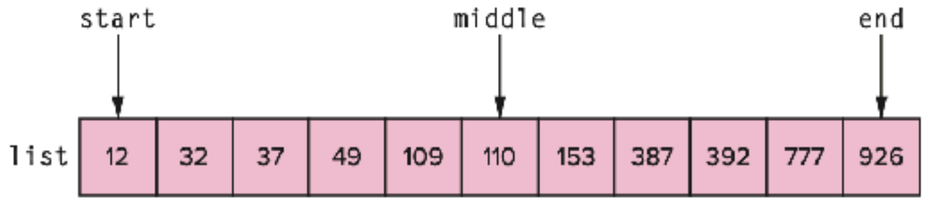
Key = 109

Pick middle

Is middle > key?

Is middle == key?

Go left



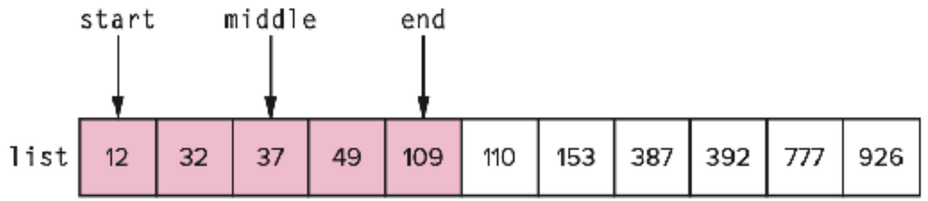
Pick middle

Is middle > key?

Is middle == key?

Is middle < key?

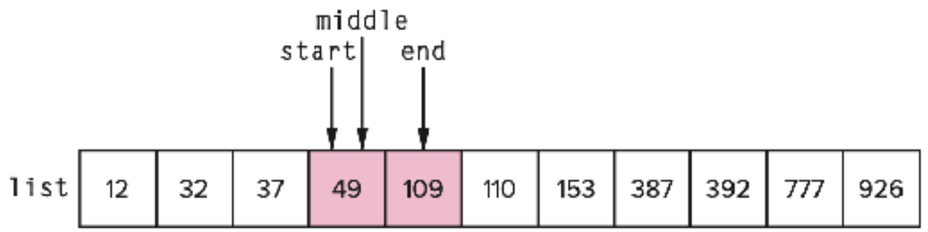
Go right



Pick middle

Is middle > key?

Is middle == key?



Recap binary search

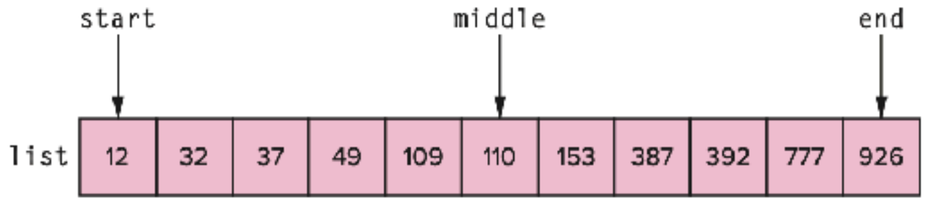
Key = 109

Pick middle

Is middle > key?

Is middle == key?

Go left



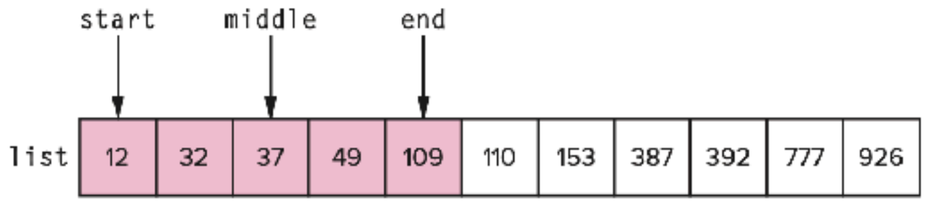
Pick middle

Is middle > key?

Is middle == key?

Is middle < key?

Go right

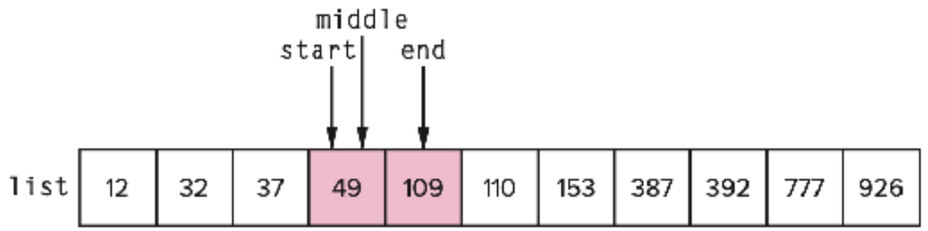


Pick middle

Is middle > key?

Is middle == key?

Is middle < key?

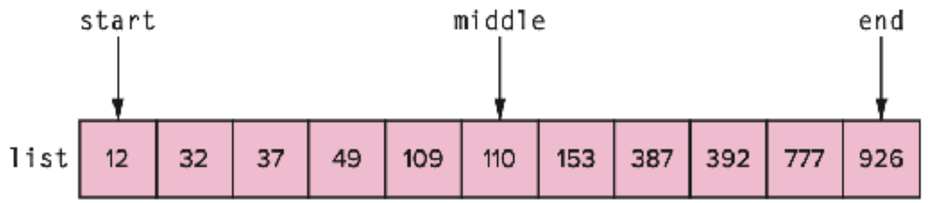


Recap binary search

Key = 109

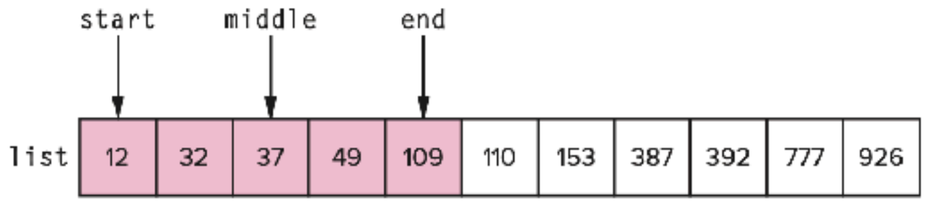
Pick middle
Is middle > key?

Is middle == key?
Go left



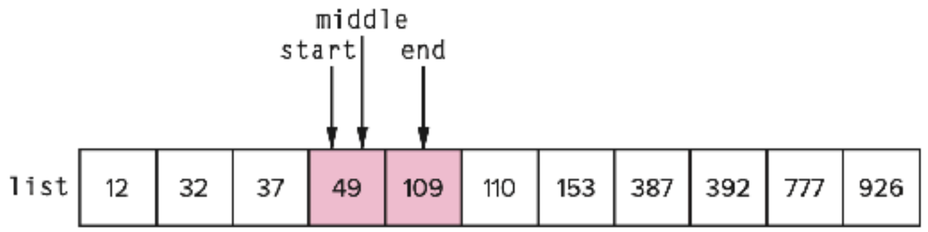
Pick middle
Is middle > key?

Is middle == key?
Is middle < key?
Go right



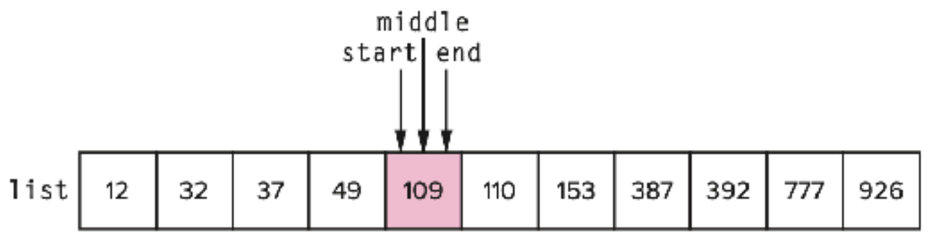
Pick middle
Is middle > key?

Is middle == key?
Is middle < key?
Go right

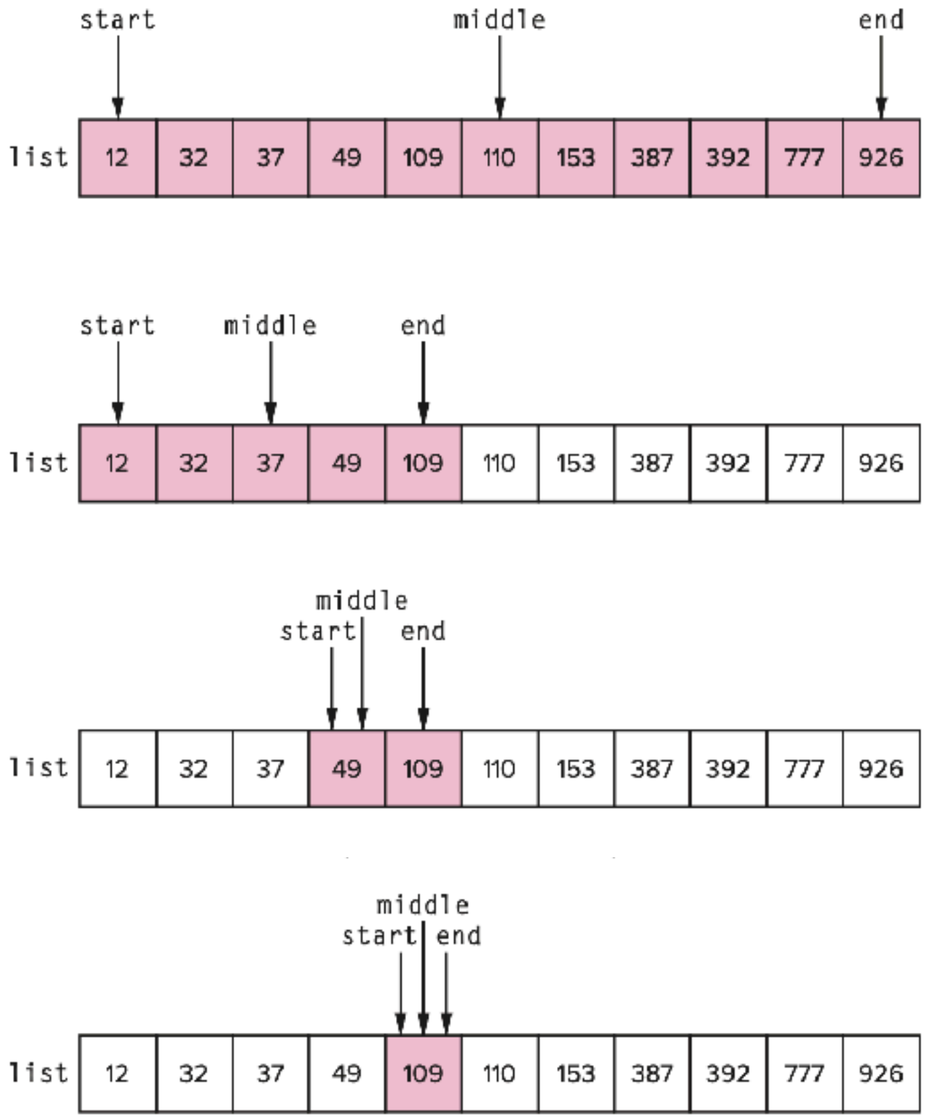


Pick middle

Is middle == key?

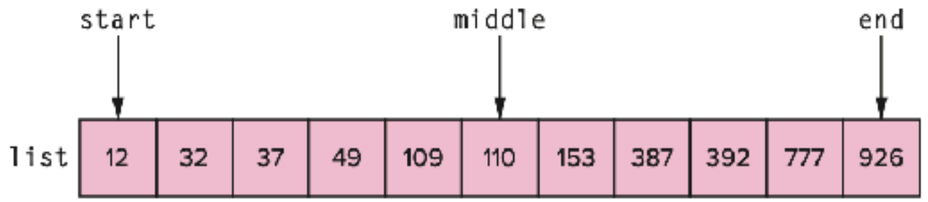


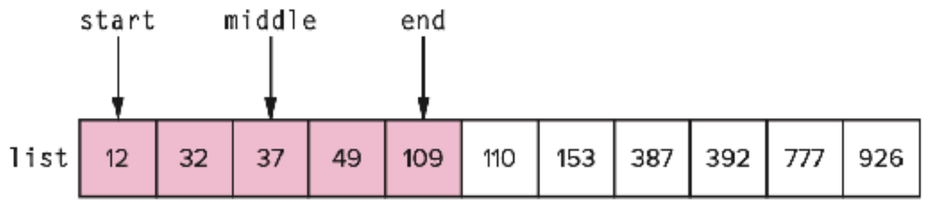
Recap binary search

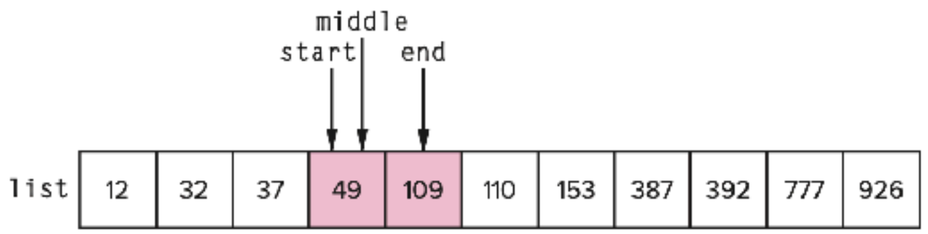


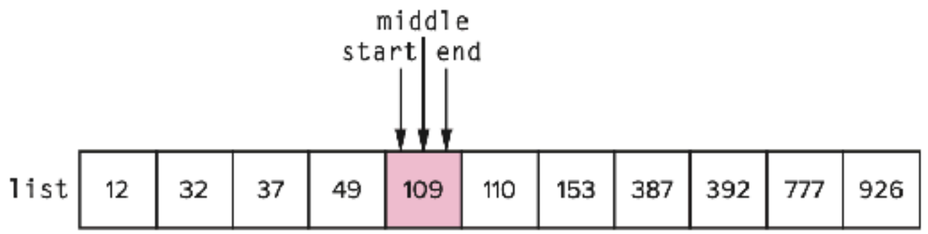
Recap binary search

```c
int binary(int arr[], int n, int key){
  int start = 0;       // Left pointer
  int end = _____; // Right pointer

  while (end >= start){
    int mid = (____________) / 2; // Pick middle element

    // Logic to focus search on left or right of mid
    if (key == arr[mid])
      return mid;
    else if (key < arr[mid])
      end = _______;
    else
      start = ________;
  }
  return -1; // Loop exited, element not present.
}
```

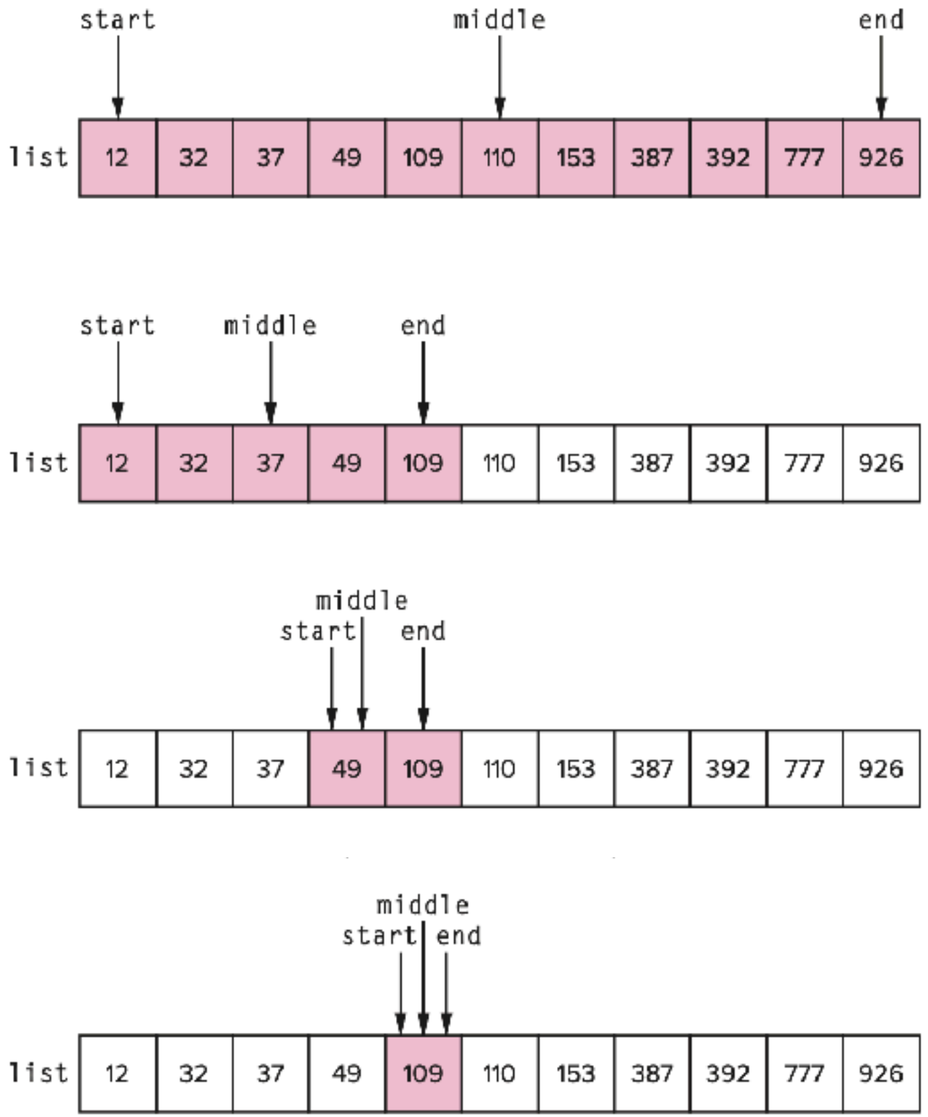






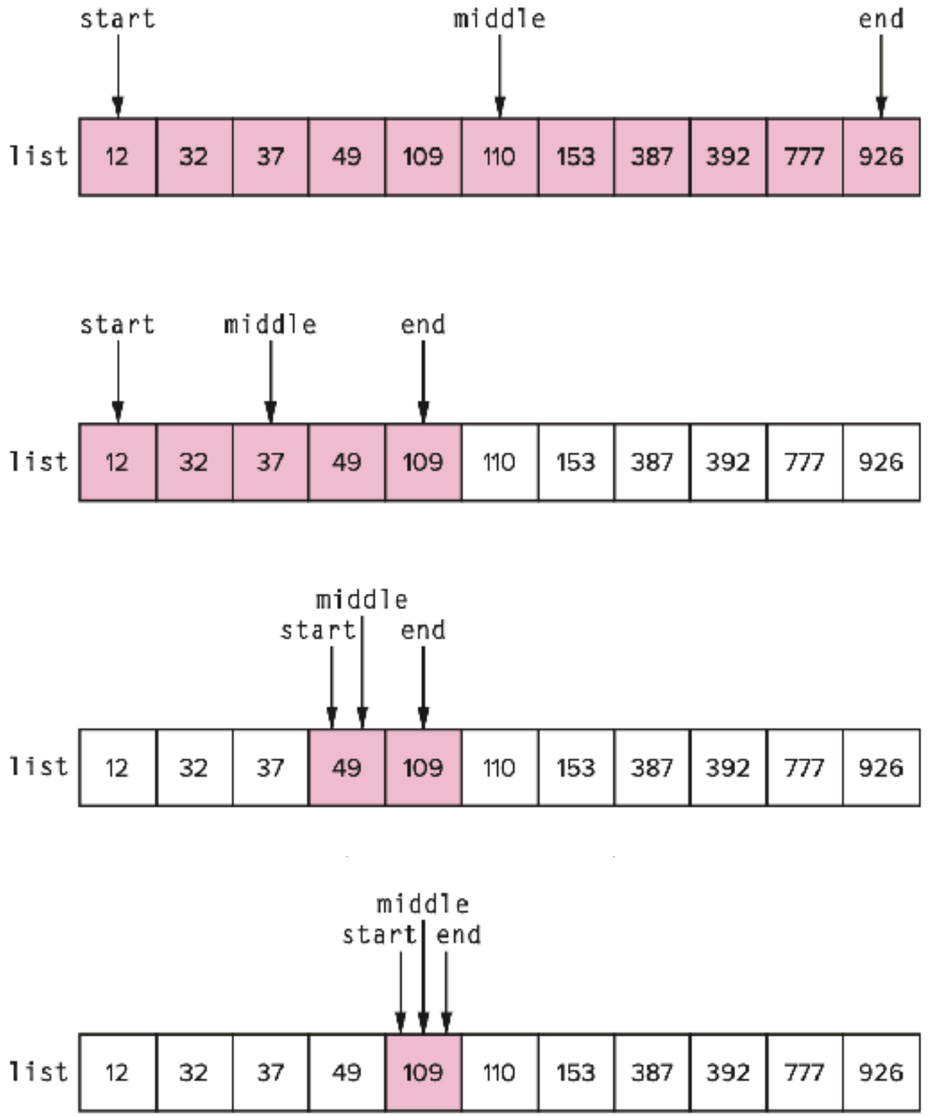


Binary search



Binary search

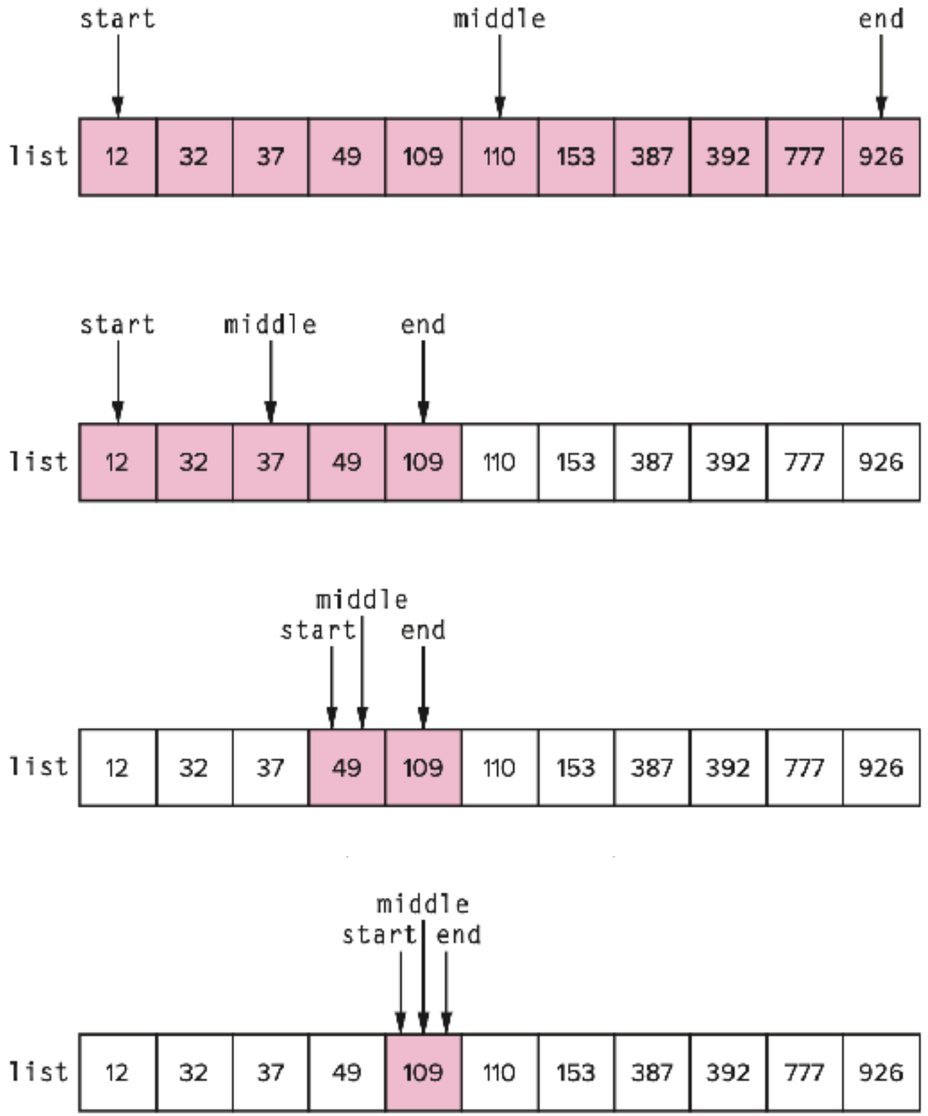
- We are repeating the same process of finding `mid` and going left or right of `mid` on each *subarray.*



Binary search

- We are repeating the same process of finding `mid` and going left or right of `mid` on each *subarray.*

- Can we apply

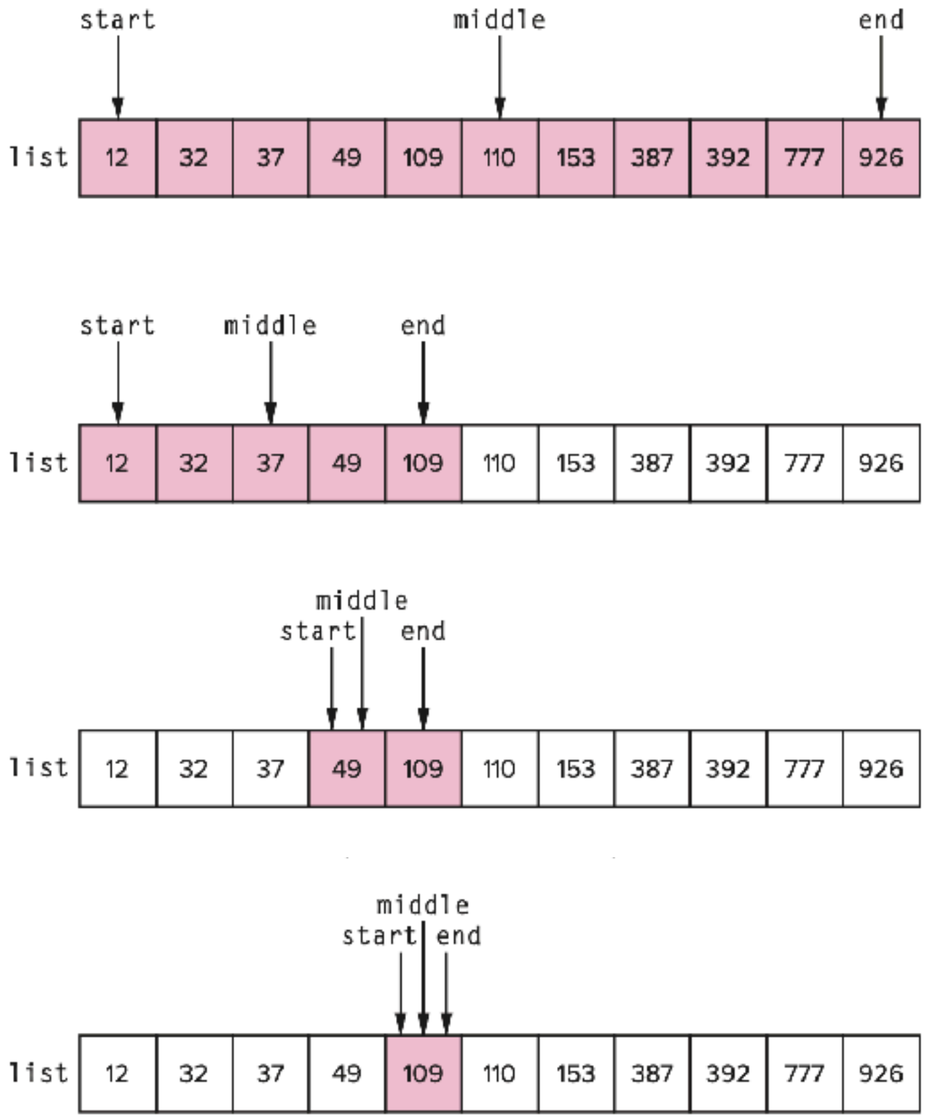


Binary search

- We are repeating the same process of finding `mid` and going left or right of `mid` on each *subarray.*

- Can we apply

```
binary(arr[], n, key)
```



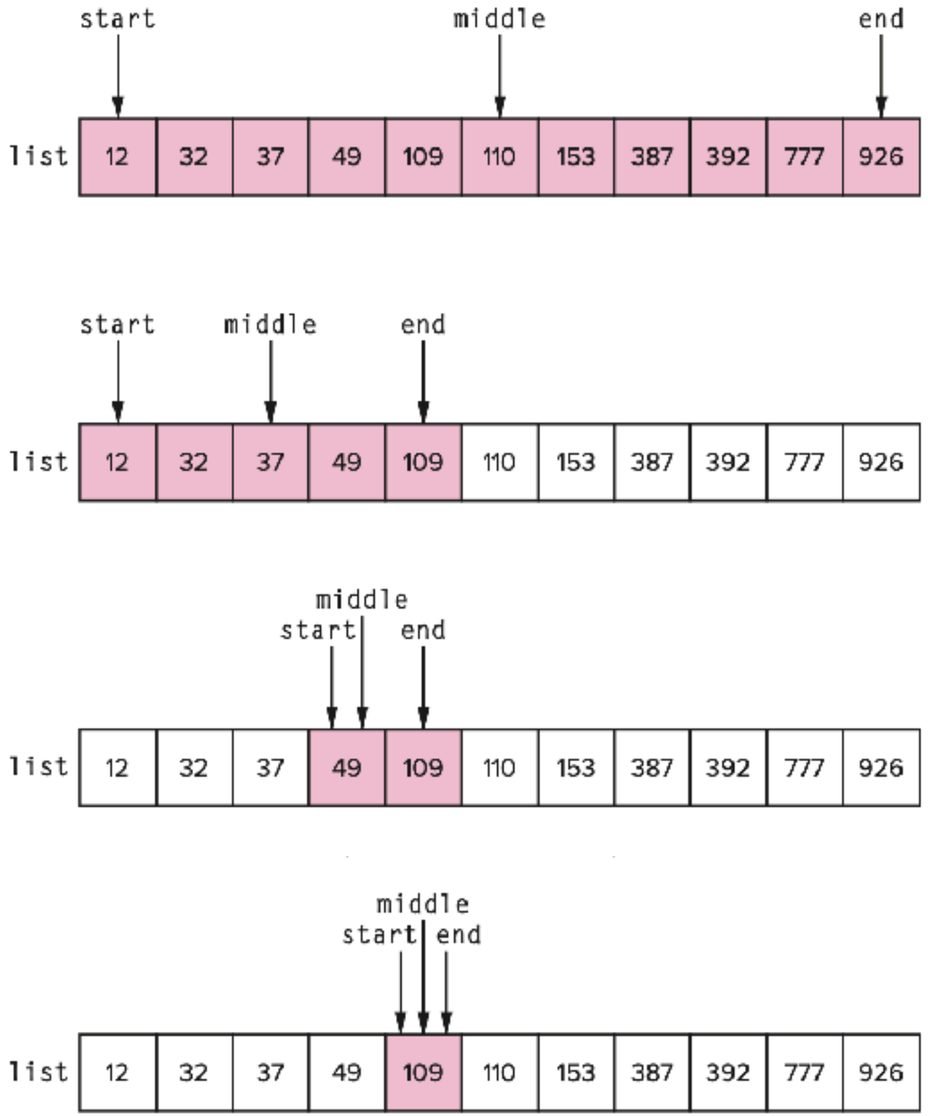
Binary search

- We are repeating the same process of finding `mid` and going left or right of `mid` on each *subarray.*

- Can we apply

```
binary(arr[], n, key)
```

on each subarray?



Binary search

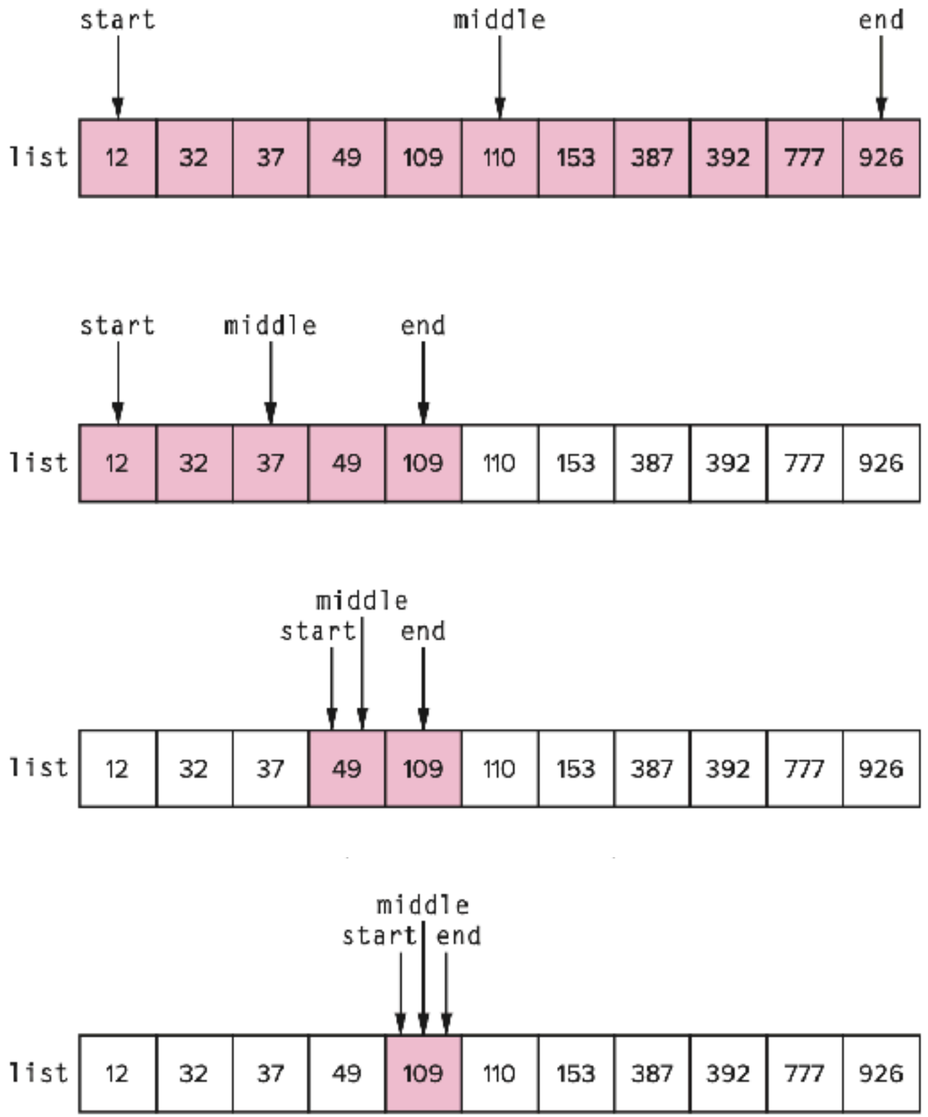
- We are repeating the same process of finding `mid` and going left or right of `mid` on each *subarray.*

- Can we apply

```
binary(arr[], n, key)
```

on each subarray?

- Idea is called *recursion*.



Binary search

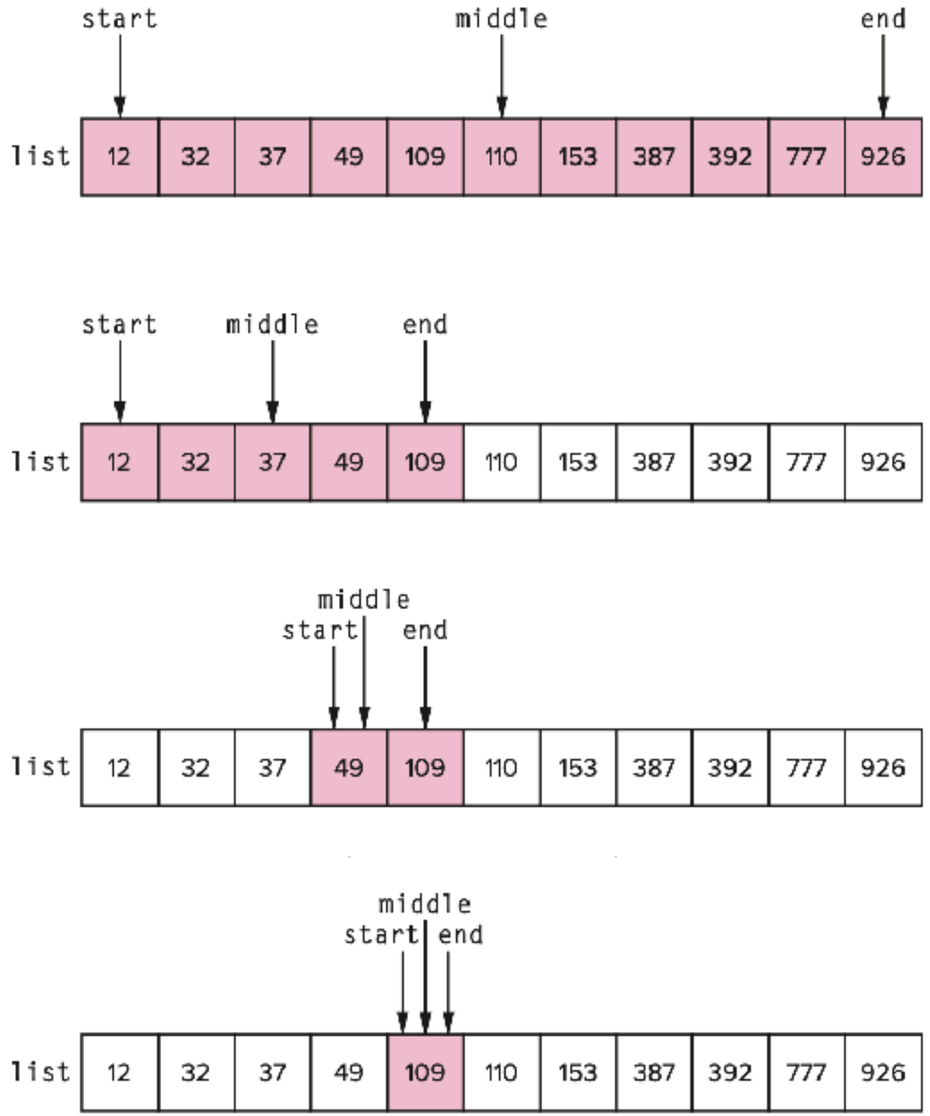
- We are repeating the same process of finding `mid` and going left or right of `mid` on each *subarray.*

- Can we apply

```
binary(arr[], n, key)
```

on each subarray?

- Idea is called *recursion*.
  - We already saw this in QuickSort!



Recursion

Recursion

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

Recursion

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics

Recursion

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics
- Like iteration — can be used interchangeably; sometimes recursion results in simpler solution … but not always!

Recursion

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics
- Like iteration — can be used interchangeably; sometimes recursion results in simpler solution … but not always!
- Must have **at least** *one* base case (terminal case) that ends the recursive process; similar to loop needing condition to exit.

Recursion

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics
- Like iteration — can be used interchangeably; sometimes recursion results in simpler solution … but not always!
- Must have **at least** *one* base case (terminal case) that ends the recursive process; similar to loop needing condition to exit.

Examples: Factorial function, Fibonacci series, binary search, etc.

Recursive function

Recursive function

- Base case (a.k.a terminating case)

Recursive function

- Base case (a.k.a terminating case)
  - This case is **required** so the recursion can terminate.

Recursive function

- Base case (a.k.a terminating case)
  - This case is **required** so the recursion can terminate.
  - The base case must provide a condition that will *eventually become true* and returns from the function. **Otherwise, the run-time stack will overflow.**

Recursive function

- Base case (a.k.a terminating case)
  - This case is **required** so the recursion can terminate.
  - The base case must provide a condition that will *eventually become true* and returns from the function. **Otherwise, the run-time stack will overflow**.
- Recursive case (a.k.a inductive case)

Recursive function

- Base case (a.k.a terminating case)
  - This case is **required** so the recursion can terminate.
  - The base case must provide a condition that will *eventually become true* and returns from the function. **Otherwise, the run-time stack will overflow**.
- Recursive case (a.k.a inductive case)
  - This case returns a (recursive) call to the function itself. It breaks down the problem into ***smaller*** chunks that can be solved by the *same* function.

Recursive function

- Base case (a.k.a terminating case)
  - This case is **required** so the recursion can terminate.
  - The base case must provide a condition that will *eventually become true* and returns from the function. **Otherwise, the run-time stack will overflow**.
- Recursive case (a.k.a inductive case)
  - This case returns a (recursive) call to the function itself. It breaks down the problem into ***smaller*** chunks that can be solved by the *same* function.
  - The input to the next call gets reduced gradually to the terminating case.

Example: Factorial

Example: Factorial

- Mathematical definition

Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

- Recursive form

Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

- Recursive form

$$n! = \begin{cases} 1, & \text{if} \quad n = 1 \\ n \cdot (n-1)!, & \text{else} \end{cases}$$

Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

- Recursive form

$$n! = \begin{cases} 1, & \text{if} \quad n = 1 \\ n \cdot (n-1)!, & \text{else} \end{cases}$$

$4!$

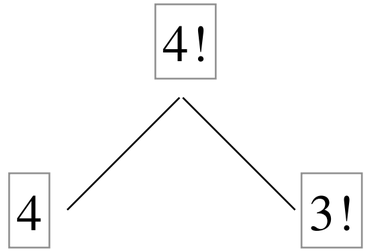
Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

- Recursive form

$$n! = \begin{cases} 1, & \text{if} \quad n = 1 \\ n \cdot (n-1)!, & \text{else} \end{cases}$$



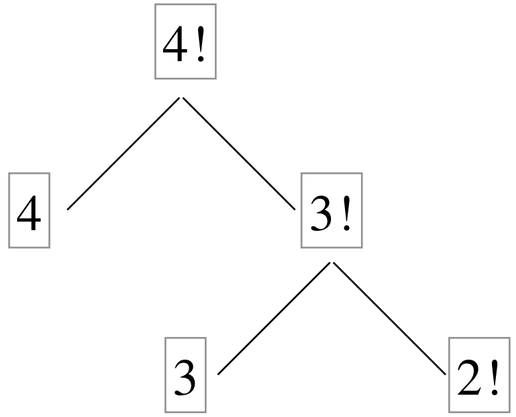
Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

- Recursive form

$$n! = \begin{cases} 1, & \text{if} \quad n = 1 \\ n \cdot (n-1)!, & \text{else} \end{cases}$$



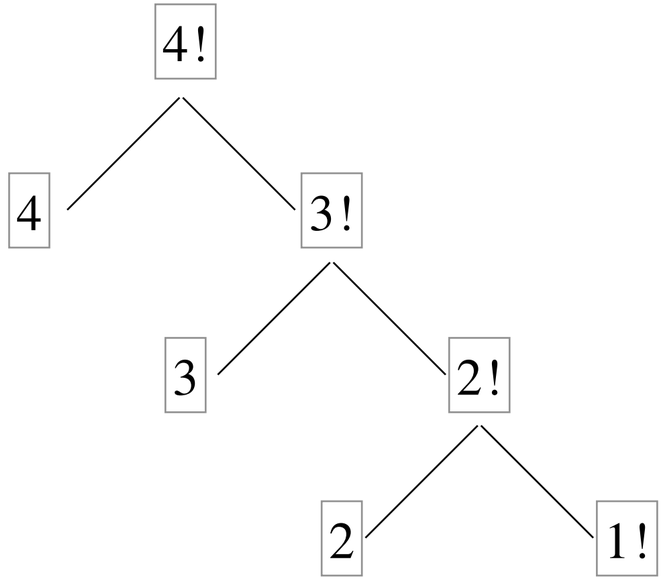
Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

- Recursive form

$$n! = \begin{cases} 1, & \text{if} \quad n = 1 \\ n \cdot (n-1)!, & \text{else} \end{cases}$$



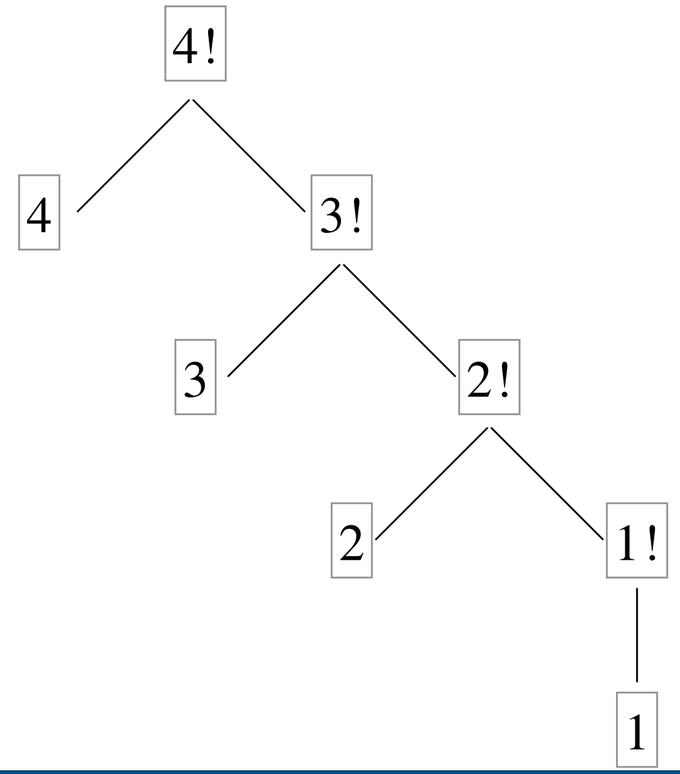
Example: Factorial

- Mathematical definition

$$n! = n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 \qquad for \quad n > 0$$

- Recursive form

$$n! = \begin{cases} 1, & \text{if} \quad n = 1 \\ n \cdot (n-1)!, & \text{else} \end{cases}$$



Example: Factorial

Example: Factorial

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

Example: Factorial

```
fn = Factorial(4);
```

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

Example: Factorial

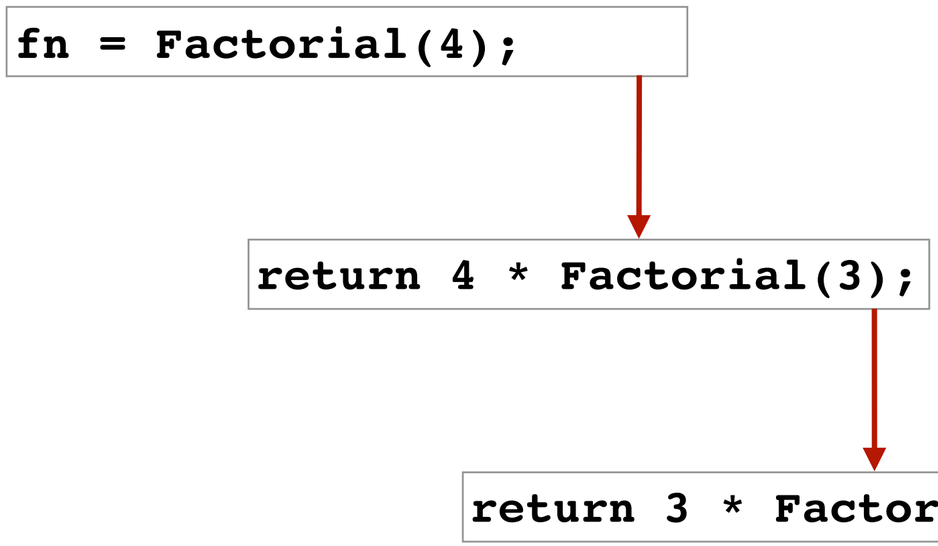
```
fn = Factorial(4);
```



```
return 4 * Factorial(3);
```

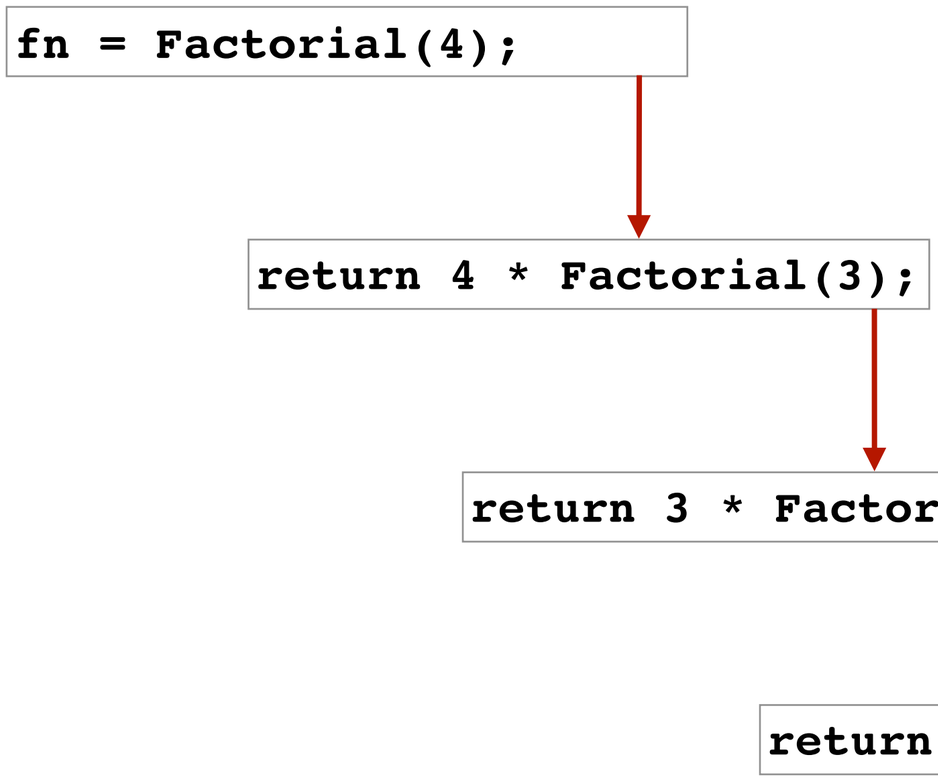
```
int Factorial(int n){
     if (n == 1)
          return 1;
     else
          return n * Factorial(n - 1);
}
```

Example: Factorial



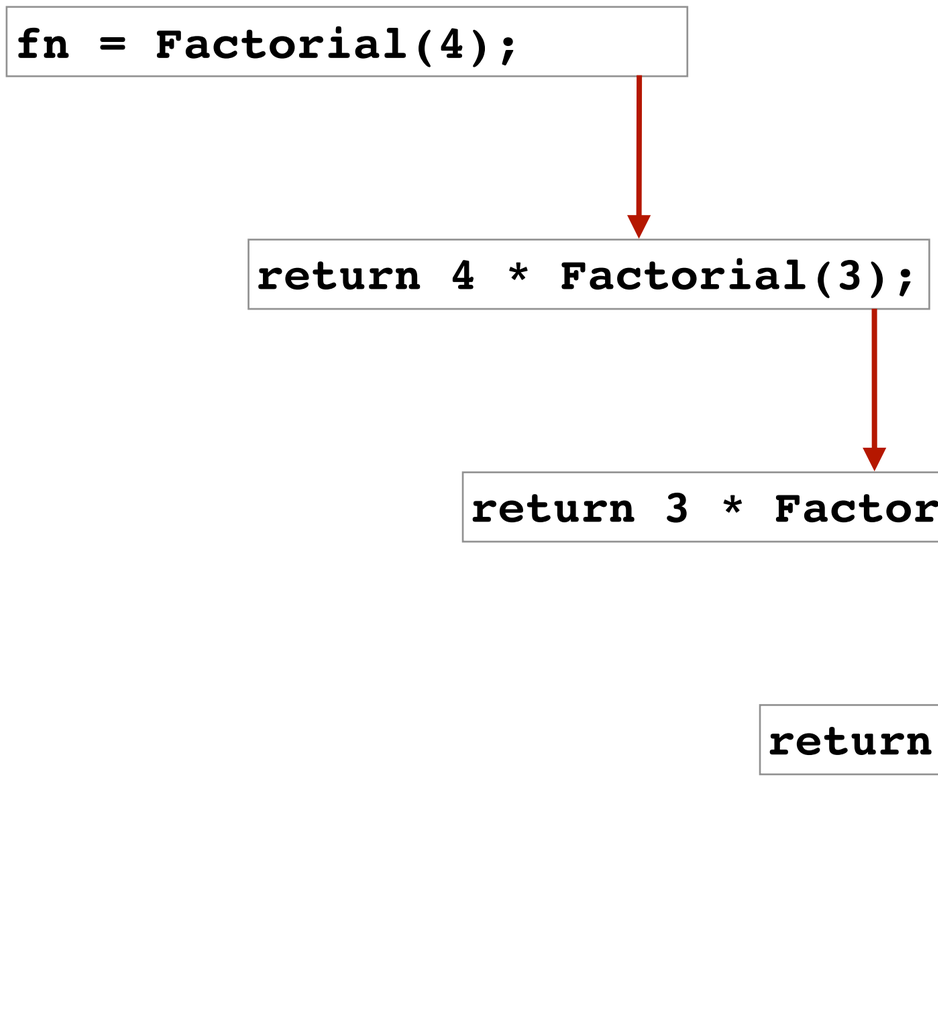

```
int Factorial(int n){
     if (n == 1)
          return 1;
     else
          return n * Factorial(n - 1);
}
```

Example: Factorial



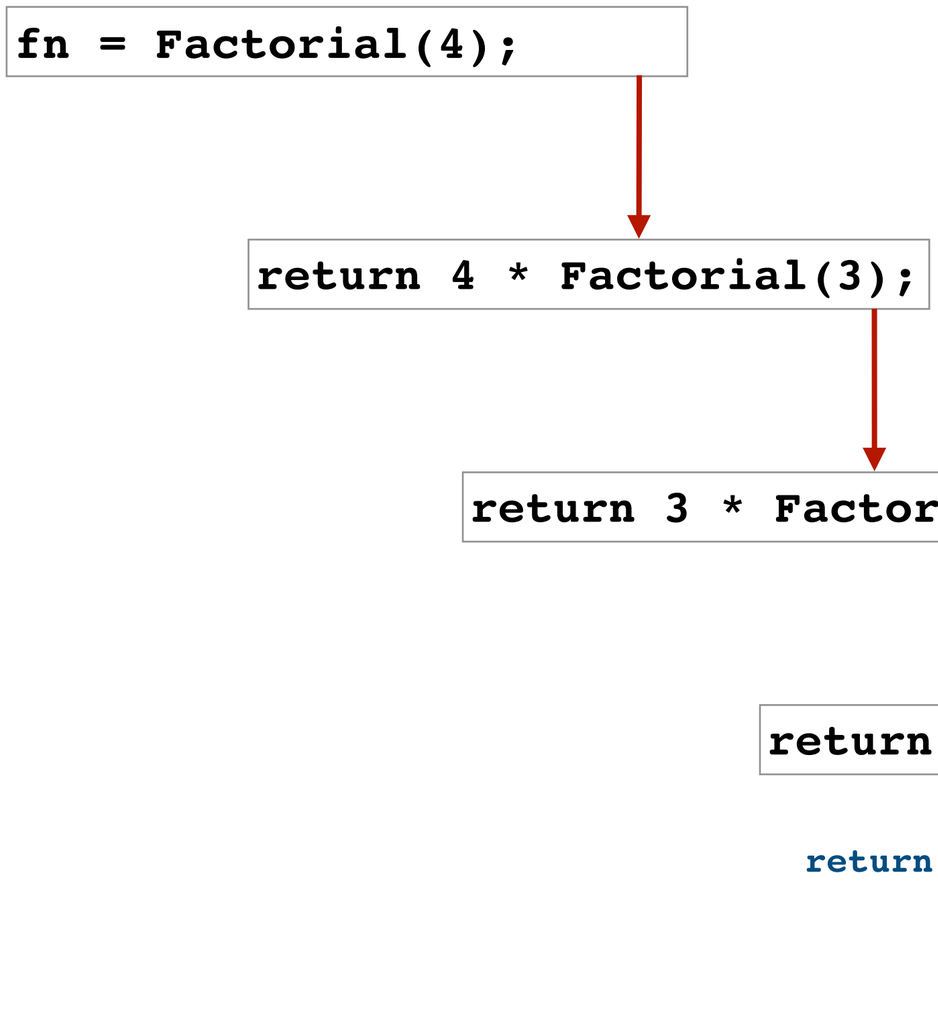

```
int Factorial(int n){
     if (n == 1)
          return 1;
     else
          return n * Factorial(n - 1);
}
```

Example: Factorial



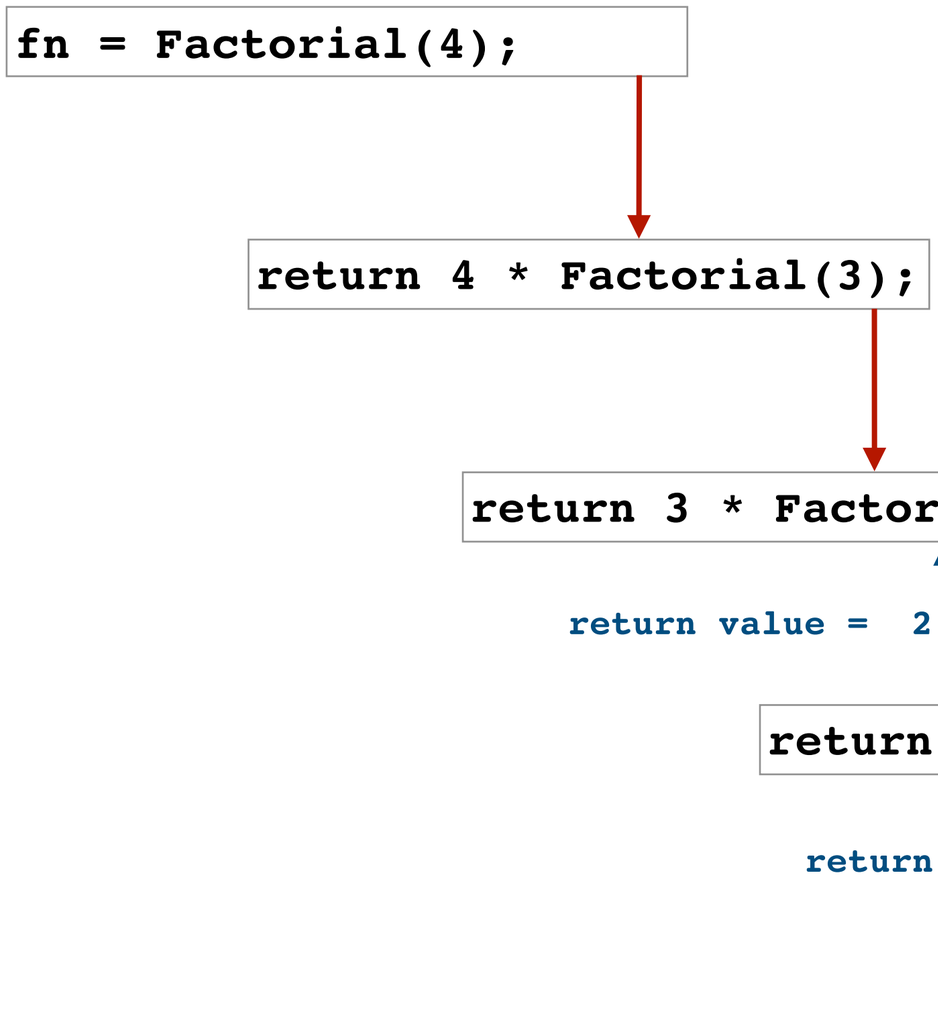

```
int Factorial(int n){
     if (n == 1)
          return 1;
     else
          return n * Factorial(n - 1);
}
```

Example: Factorial



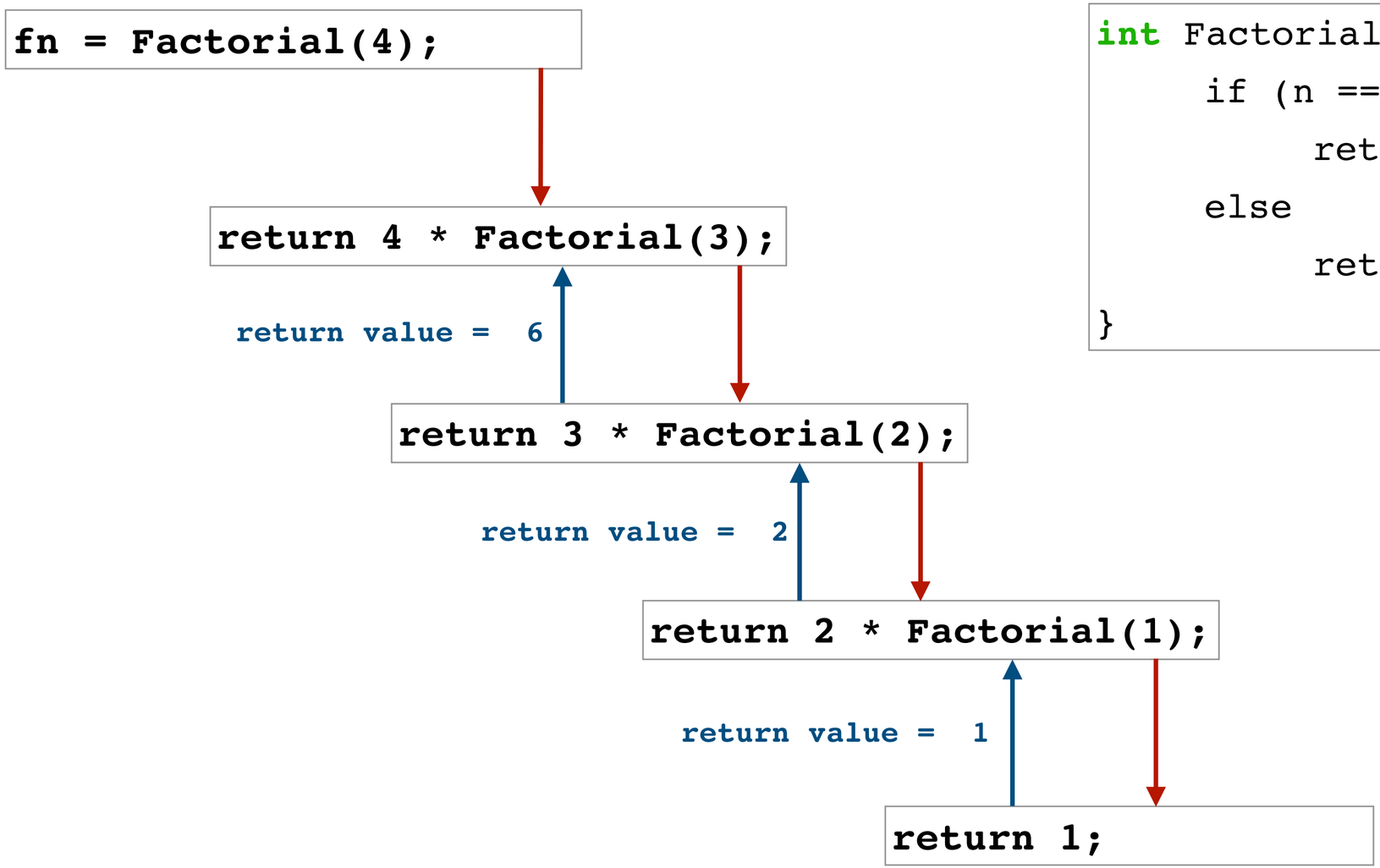

```
int Factorial(int n){
     if (n == 1)
          return 1;
     else
          return n * Factorial(n - 1);
}
```

Example: Factorial



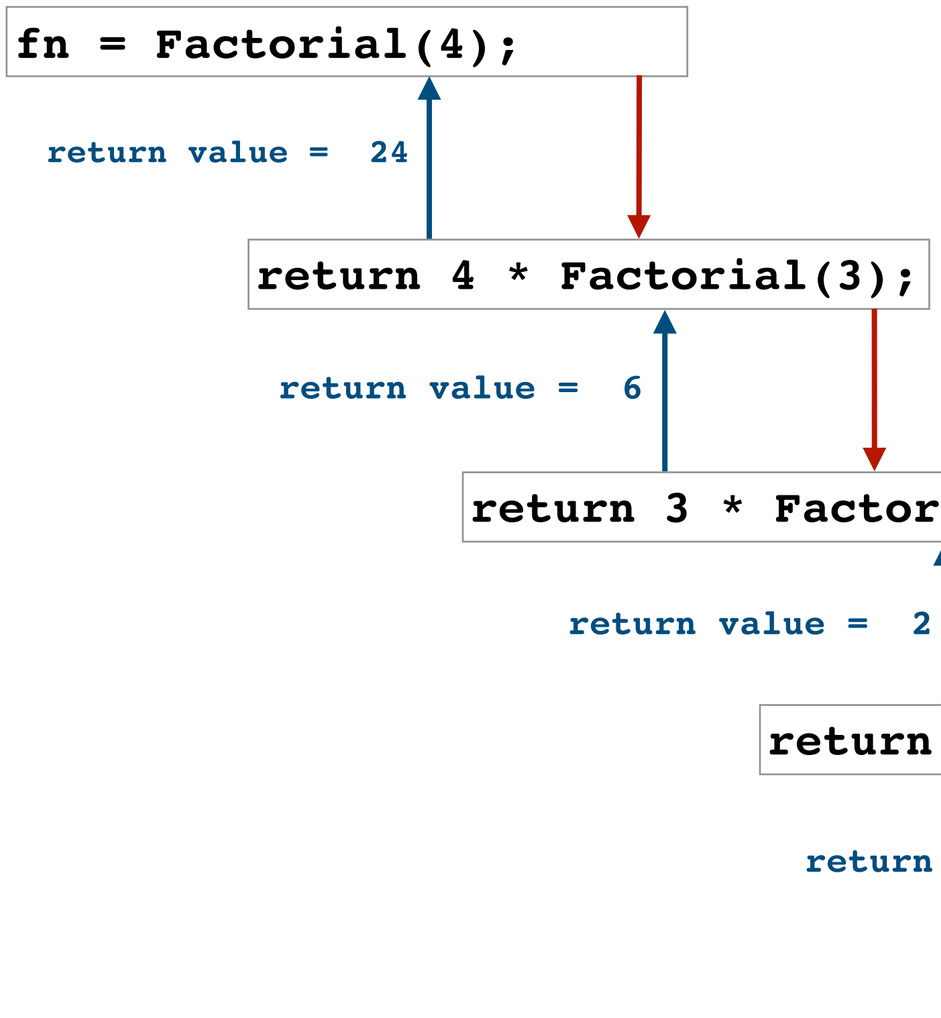

```
int Factorial(int n){
     if (n == 1)
          return 1;
     else
          return n * Factorial(n - 1);
}
```

Example: Factorial




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

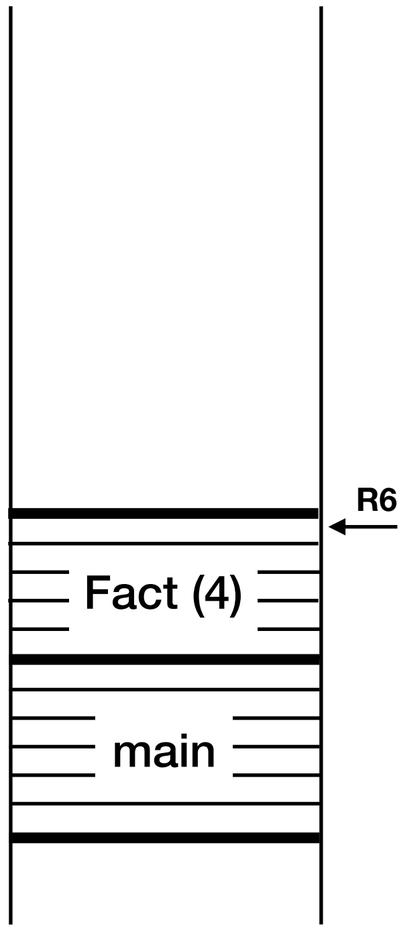
Example: Factorial



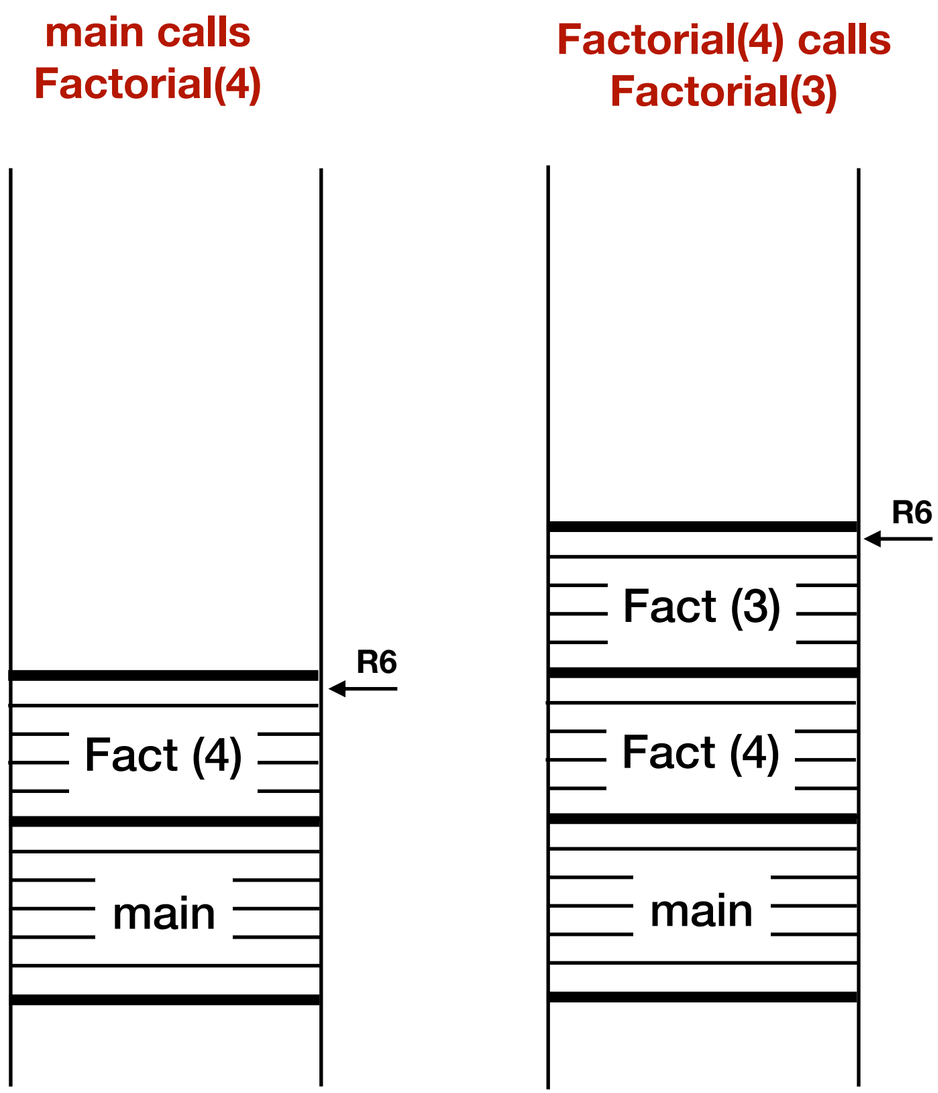

```
int Factorial(int n){
     if (n == 1)
          return 1;
     else
          return n * Factorial(n - 1);
}
```

RTS During Execution of Factorial

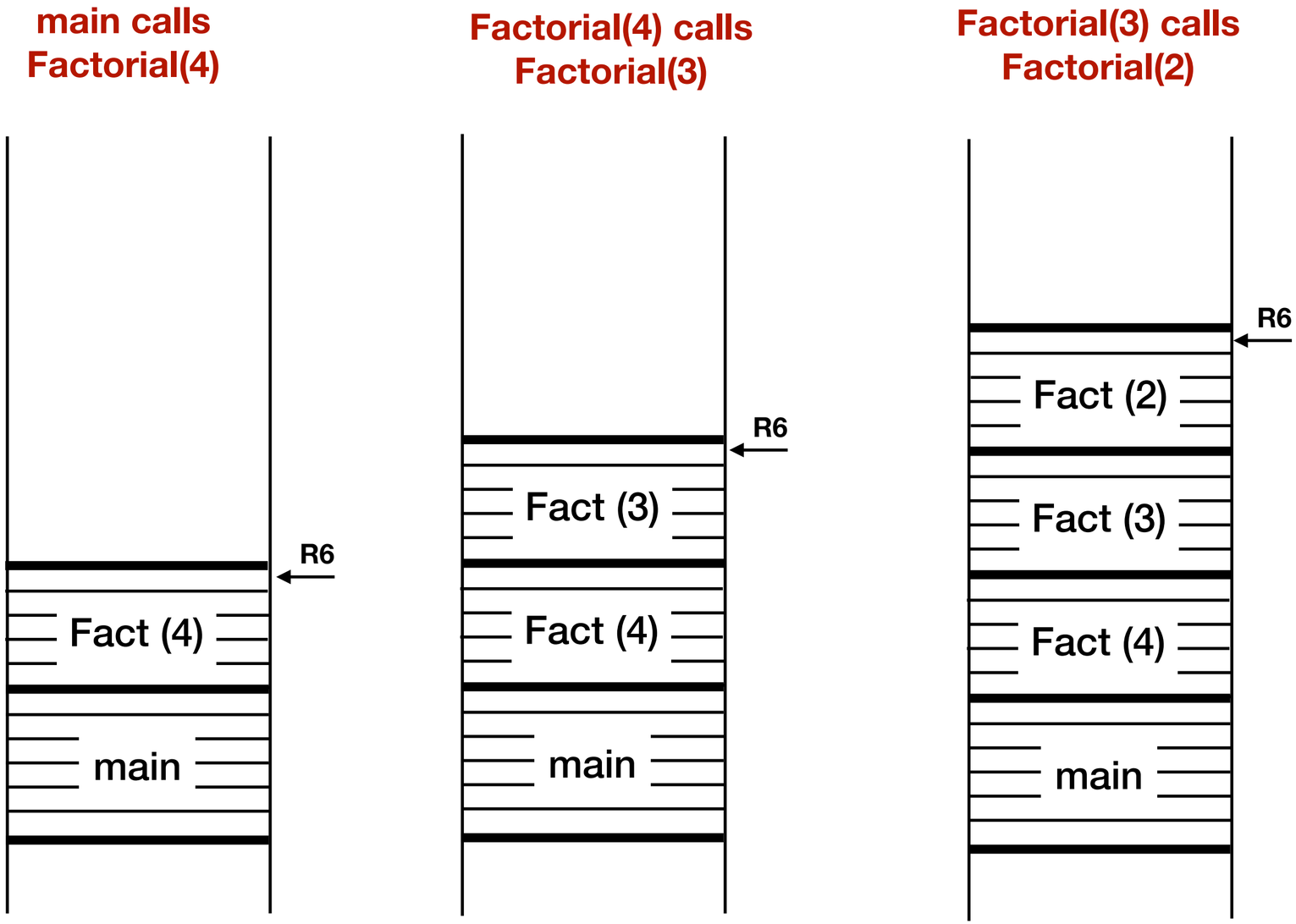
main calls Factorial(4)



RTS During Execution of Factorial

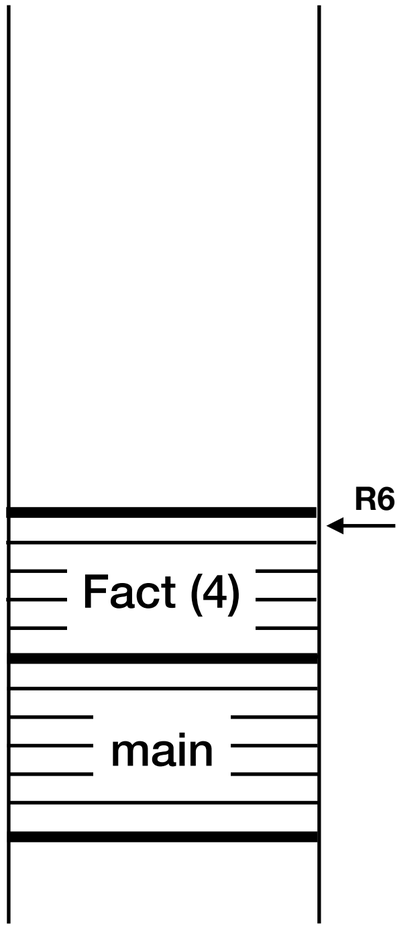


RTS During Execution of Factorial

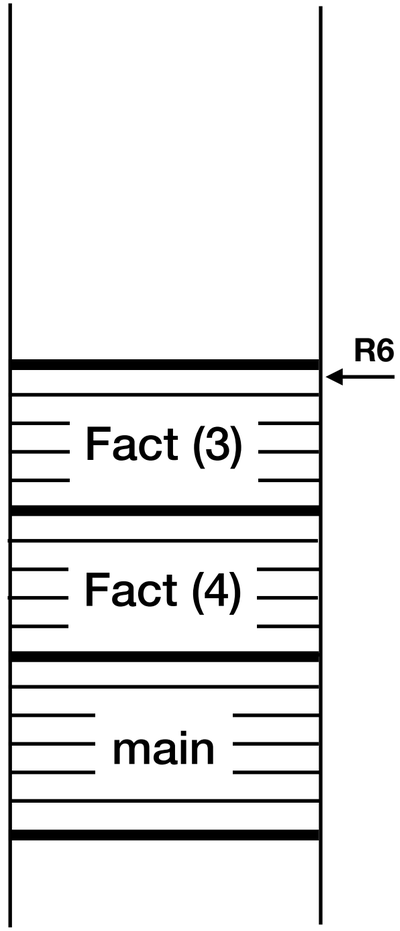


RTS During Execution of Factorial

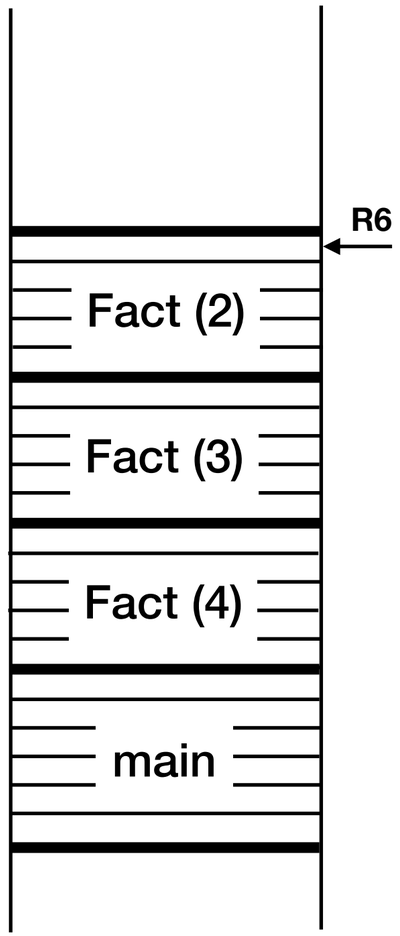
main calls Factorial(4)



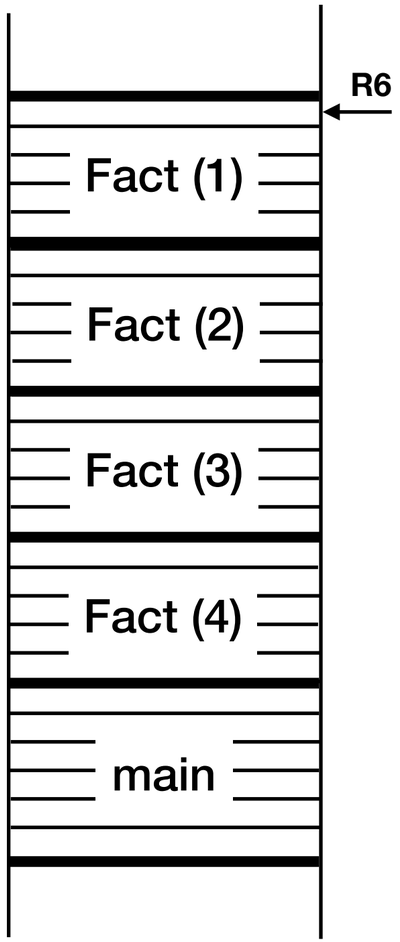
Factorial(4) calls Factorial(3)



Factorial(3) calls Factorial(2)

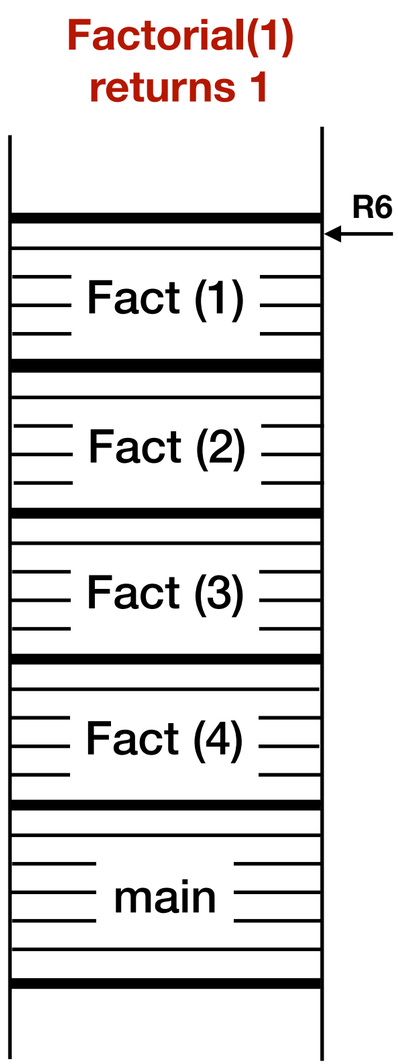


Factorial(2) calls Factorial(1)

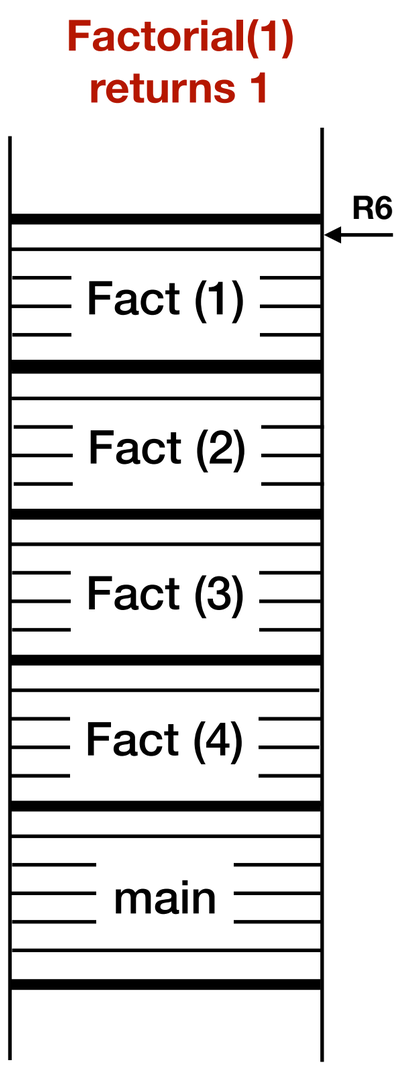


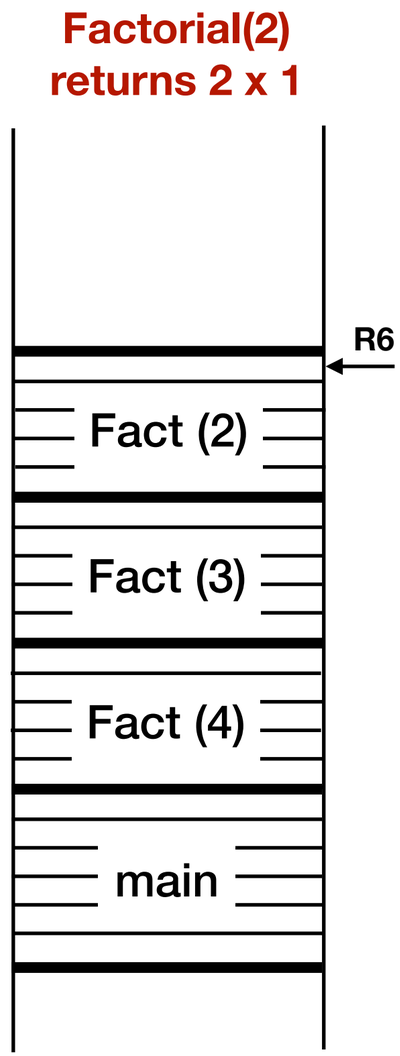
RTS During Execution of Factorial

RTS During Execution of Factorial



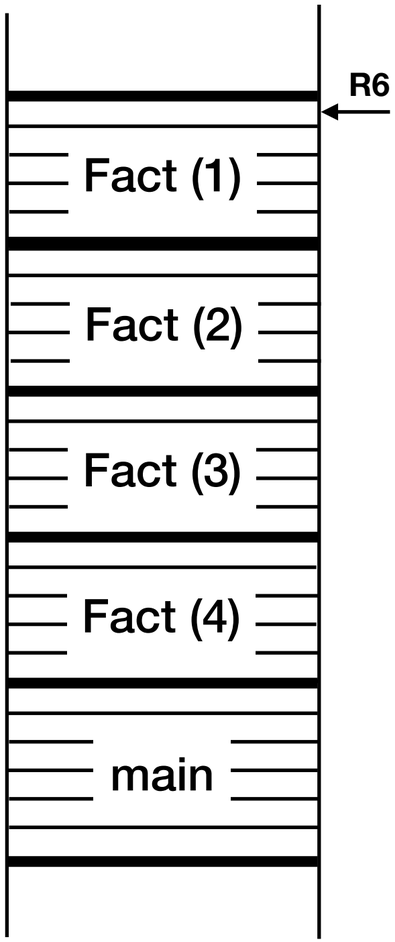
RTS During Execution of Factorial



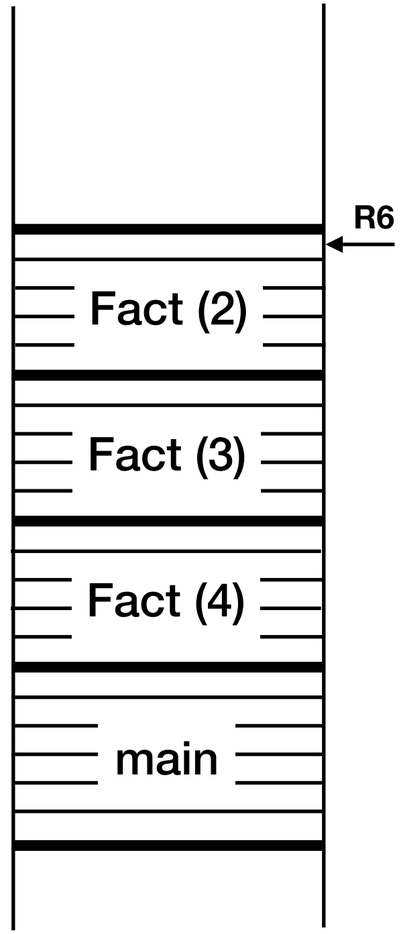


RTS During Execution of Factorial

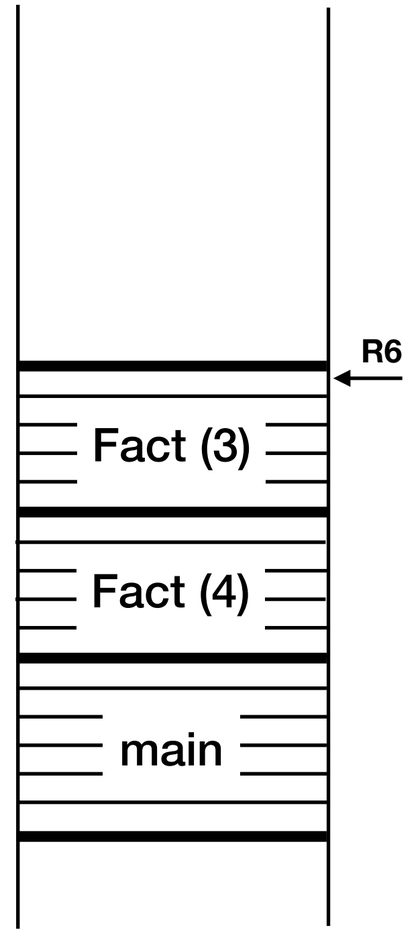
Factorial(1) returns 1



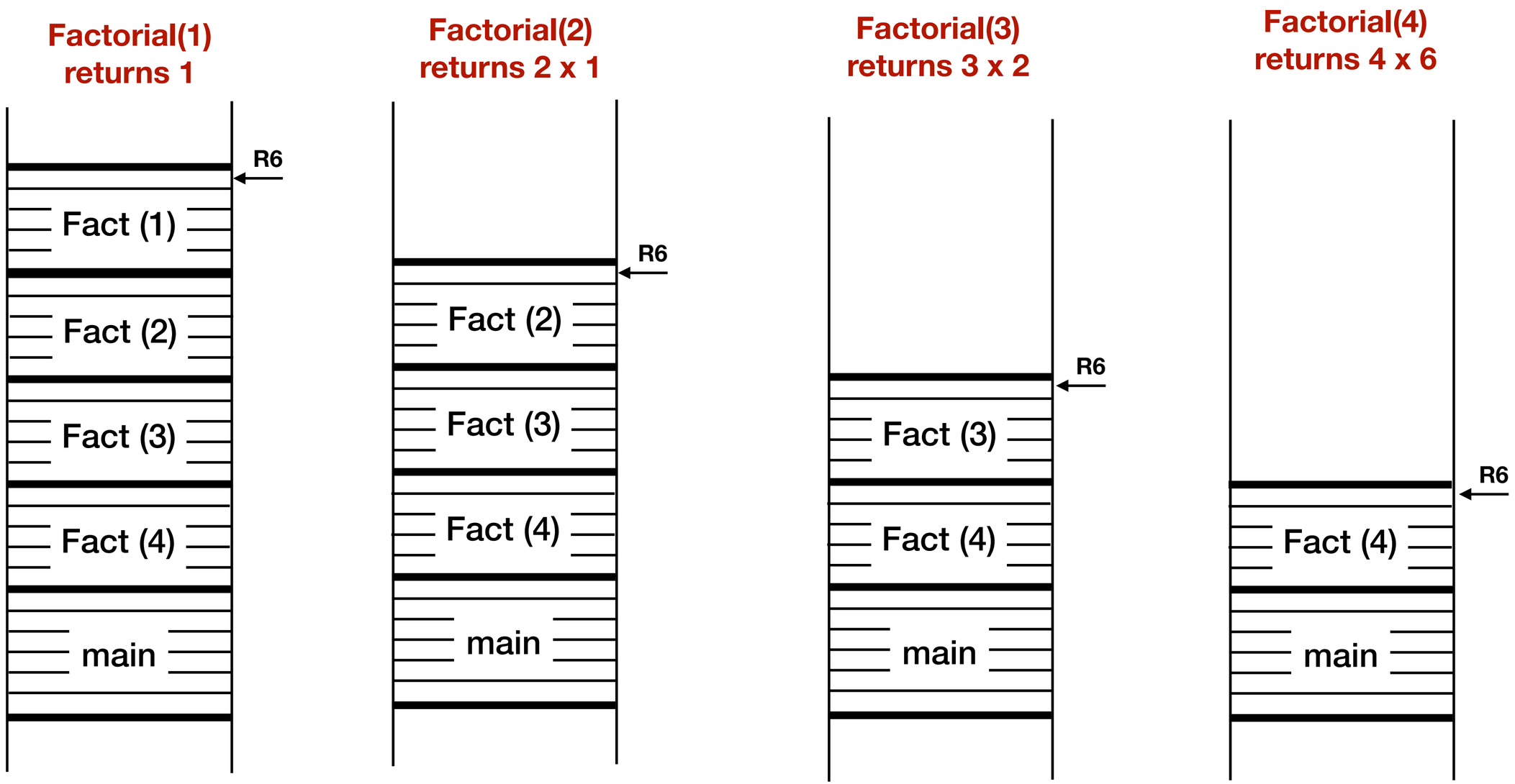
Factorial(2) returns 2 x 1



Factorial(3) returns 3 x 2



RTS During Execution of Factorial



Example: Fibonacci series

Example: Fibonacci series

Mathematical definition:

Example: Fibonacci series

Mathematical definition:

Fibonacci Series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

Example: Fibonacci series

Mathematical definition:

Fibonacci Series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

$$f(n) = f(n-1) + f(n-2)$$

Example: Fibonacci series

Mathematical definition:

Fibonacci Series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = 1$$

Example: Fibonacci series

Mathematical definition:

Fibonacci Series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = 1$$

$$f(0) = 1$$

Example: Fibonacci series

Mathematical definition:

Fibonacci Series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = 1$$

$$f(0) = 1$$

Fibonacci (3) = Fibonacci(2) + Fibonacci(1)

Example: Fibonacci series

Mathematical definition:

Fibonacci Series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = 1$$

$$f(0) = 1$$

Fibonacci (3) = Fibonacci(2) + Fibonacci(1)

= (Fibonacci(1) + Fibonacci(0)) + Fibonacci(1)

Example: Fibonacci series

Mathematical definition:

Fibonacci Series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = 1$$

$$f(0) = 1$$

Fibonacci (3) = Fibonacci(2) + Fibonacci(1)

= (Fibonacci(1) + Fibonacci(0)) + Fibonacci(1)

= 1 + 1 + 1 = 3

Example: Fibonacci series

Example: Fibonacci series

```
int Fibonacci (int n) {
    int sum;
    if (n == 0 || n == 1)
        return 1;
    else {
        sum = (Fibonacci(n-1) + Fibonacci(n-2));
        return sum;
        }
}
```

Example: Binary search

```c
int binary(int arr[], int n, int key){
  int start = 0;       // Left pointer
  int end = n - 1; // Right pointer

  while (end >= start){
    int mid = (start + end) / 2; // Pick middle element

    // Logic to focus search on left or right of mid
    if (key == arr[mid])
      return mid;
    else if (key < arr[mid])
      end = mid - 1;
    else
      start = mid + 1;
  }
  return -1; // Loop exited, element not present.
}
```

Can we implement binary search in *recursive* way?

Example: Binary search

```c
int binary(int arr[], int n, int key){
  int start = 0;       // Left pointer
  int end = n - 1; // Right pointer

  while (end >= start){
    int mid = (start + end) / 2; // Pick middle element

    // Logic to focus search on left or right of mid
    if (key == arr[mid])
      return mid;
    else if (key < arr[mid])
      end = mid - 1;
    else
      start = mid + 1;
  }
  return -1; // Loop exited, element not present.
}
```

Can we implement binary search in *recursive* way?

- **Need**: A mechanism to keep track of the `start` and `end` indices across recursive calls; local variables won't do (why?).

Example: Binary search

Example: Binary search

```
int binary_search(int item, int list[], int start, int end){
    int middle = (end + start)/2;
    if (end < start)
      return -1;                          // Did not find key
    else if (_________________________) // Found item!
        return middle;
    else if (_________________________)  // Search left half
        return binary_search(item, list, start, middle-1);
    else                                  // Search right half
        return binary_search(_____________________________);
}
```

Example: Binary search

Mechanism

```
int binary_search(int item, int list[], int start, int end){
    int middle = (end + start)/2;
    if (end < start)
      return -1;                          // Did not find key
    else if (_________________________) // Found item!
        return middle;
    else if (_________________________)  // Search left half
        return binary_search(item, list, start, middle-1);
    else                                  // Search right half
        return binary_search(_____________________________);
}
```

Example: Binary search

```
int binary_search(int item, int list[], int start, int end){
    int middle = (end + start)/2;
    if (end < start)
      return -1;                          // Did not find key
    else if (_____________________________) // Found item!
        return middle;
    else if (_____________________________)  // Search left half
        return binary_search(item, list, start, middle-1);
    else                                    // Search right half
        return binary_search(_____________________________);
}
```

Example: Binary search

```
int binary_search(int item, int list[], int start, int end){
    int middle = (end + start)/2;
    if (end < start)
      return -1;                          // Did not find key
    else if (list[middle] == item) // Found item!
        return middle;
    else if (_____________________)  // Search left half
        return binary_search(item, list, start, middle-1);
    else                                  // Search right half
        return binary_search(_____________________________);
}
```

Example: Binary search

```
int binary_search(int item, int list[], int start, int end){
    int middle = (end + start)/2;
    if (end < start)
      return -1;                          // Did not find key
    else if (list[middle] == item) // Found item!
        return middle;
    else if (item < list[middle])  // Search left half
        return binary_search(item, list, start, middle-1);
    else                                  // Search right half
        return binary_search(_____________________________);
}
```

Example: Binary search

```
int binary_search(int item, int list[], int start, int end){
    int middle = (end + start)/2;
    if (end < start)
      return -1;                          // Did not find key
    else if (list[middle] == item) // Found item!
        return middle;
    else if (item < list[middle])  // Search left half
        return binary_search(item, list, start, middle-1);
    else                                  // Search right half
        return binary_search(item, list, middle+1, end);
}
```

Concept check

Concept check

- What are the base/terminating cases in binary search?

Concept check

- What are the base/terminating cases in binary search?

- What are the recursive cases in binary search?

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.

- Recall the steps ….

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.

- Recall the steps ….

1. Choose first element of given array as pivot.

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.

- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.

- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.
3. Increment left pointer while the element it points to is less than pivot

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.

- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.
3. Increment left pointer while the element it points to is less than pivot
4. Decrement right pointer while the element it points to is greater than pivot.

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.
- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.
3. Increment left pointer while the element it points to is less than pivot
4. Decrement right pointer while the element it points to is greater than pivot.
    1. If pointers cross/overlap, **split array**, then **repeat** on each subarray.

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.
- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.
3. Increment left pointer while the element it points to is less than pivot
4. Decrement right pointer while the element it points to is greater than pivot.
    1. If pointers cross/overlap, **split array**, then **repeat** on each subarray.
5. If neither pointers can move swap elements.

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.
- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.
3. Increment left pointer while the element it points to is less than pivot
4. Decrement right pointer while the element it points to is greater than pivot.
    1. If pointers cross/overlap, **split array**, then **repeat** on each subarray.
5. If neither pointers can move swap elements.
6. Repeat 3-5 while left pointer < right pointer.

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.
- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.
3. Increment left pointer while the element it points to is less than pivot
4. Decrement right pointer while the element it points to is greater than pivot.
    1. If pointers cross/overlap, **split array**, then **repeat** on each subarray.
5. If neither pointers can move swap elements.
6. Repeat 3-5 while left pointer < right pointer.

We wrote one function to split array

Example: Quicksort

- We already saw Quicksort last time and remarked it was recursive … but provided an iterative solution.
- Recall the steps ….

1. Choose first element of given array as pivot.
2. Maintain pointers from the left and right.
3. Increment left pointer while the element it points to is less than pivot
4. Decrement right pointer while the element it points to is greater than pivot.
    1. If pointers cross/overlap, **split array**, then **repeat** on each subarray.
5. If neither pointers can move swap elements.
6. Repeat 3-5 while left pointer < right pointer.

And a loop to repeat on subarrays after split

We wrote one function to split array

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, _________);

    // And right half
    QuickSort(arr, ___________, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = ___________;
  int left = _________; // Initialize left
  int right = _______;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (_____________________);

    do right--; // Decrement right till …
    while (_____________________);

    if (___________) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, ____________, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = ___________;
  int left = _________; // Initialize left
  int right = _______;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (_____________________);

    do right--; // Decrement right till …
    while (_____________________);

    if (____________) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, pivotVal + 1, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = ___________;
  int left = _________; // Initialize left
  int right = _______;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (_____________________);

    do right--; // Decrement right till …
    while (_____________________);

    if (___________) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, pivotVal + 1, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = arr[start];
  int left = _________; // Initialize left
  int right = _______;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (_____________________);

    do right--; // Decrement right till …
    while (_____________________);

    if (___________) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, pivotVal + 1, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = arr[start];
  int left = start - 1; // Initialize left
  int right = _______;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (_____________________);

    do right--; // Decrement right till …
    while (_____________________);

    if (___________) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, pivotVal + 1, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = arr[start];
  int left = start - 1; // Initialize left
  int right = end + 1;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (_____________________);

    do right--; // Decrement right till …
    while (_____________________);

    if (_____________) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, pivotVal + 1, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = arr[start];
  int left = start - 1; // Initialize left
  int right = end + 1;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (arr[left] < pivotVal);

    do right--; // Decrement right till …
    while (_____________________);

    if (___________) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, pivotVal + 1, end);
  }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = arr[start];
  int left = start - 1; // Initialize left
  int right = end + 1;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (arr[left] < pivotVal);

    do right--; // Decrement right till …
    while (arr[right] > pivotVal);

    if (_____) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Recursive implementation

```c
void Swap(int* one, int* two){
  int temp = *one;
  *one = *two;
  *two = temp;
}
```

```c
void QuickSort(int arr[], int start, int end){
  if (start < end){
    int pivotVal = partition(arr, start, end);

    // Now sort left half
    QuickSort(arr, start, pivotVal);

    // And right half
    QuickSort(arr, pivotVal + 1, end);
 }
}
```

```c
int partition(int arr[], int start, int end){

  int pivotVal = arr[start];
  int left = start - 1; // Initialize left
  int right = end + 1;   // Initialize right

  while(1){
    do left++; // Increment left till …
    while (arr[left] < pivotVal);

    do right--; // Decrement right till …
    while (arr[right] > pivotVal);

    if (left >= right) // Split if overlap
      return right;

    Swap(&arr[left], &arr[right]);
  }
}
```

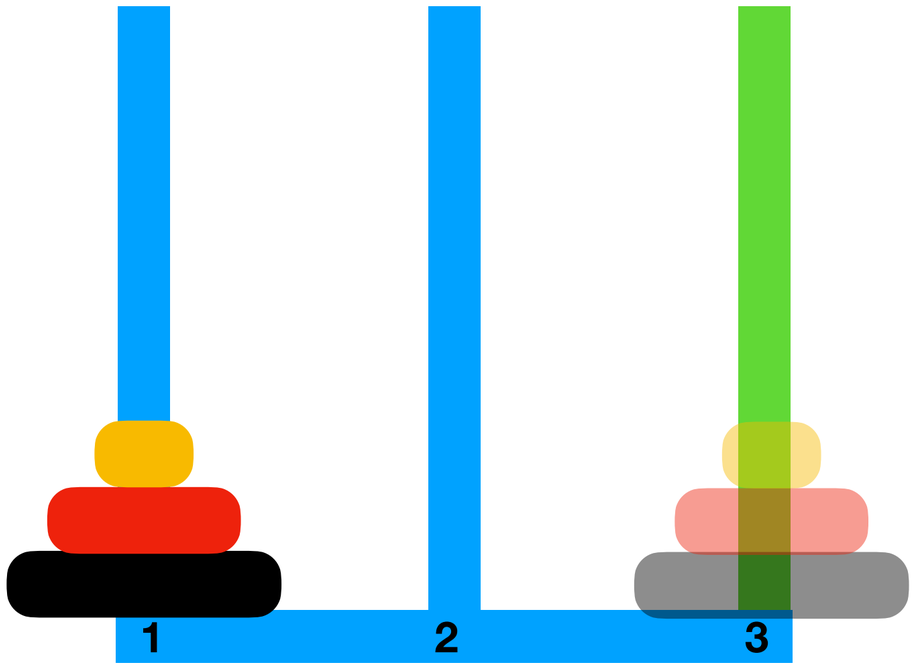
Check Gitlab for reference material: https://github.com/iabraham/ece220-fa25/blob/main/lec12_1007/quicksort.c

Concept check

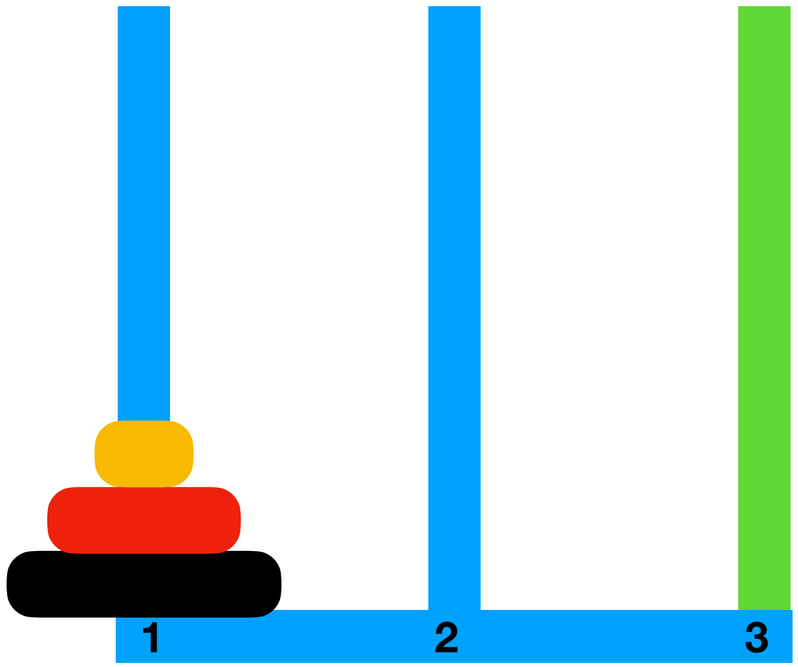
Concept check

- Why did iterative version of Quicksort need a stack to work but iterative binary search did not?

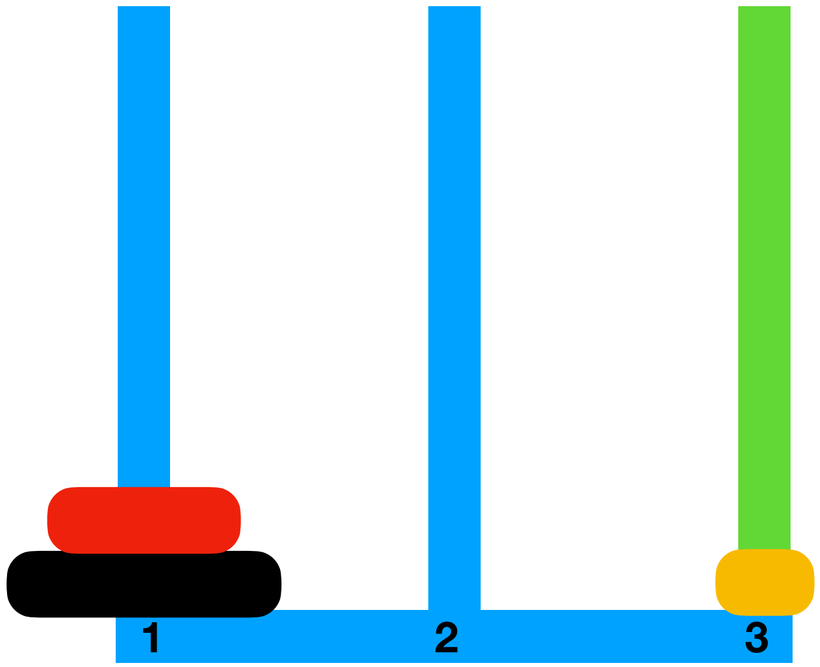
Example: Towers of Hanoi



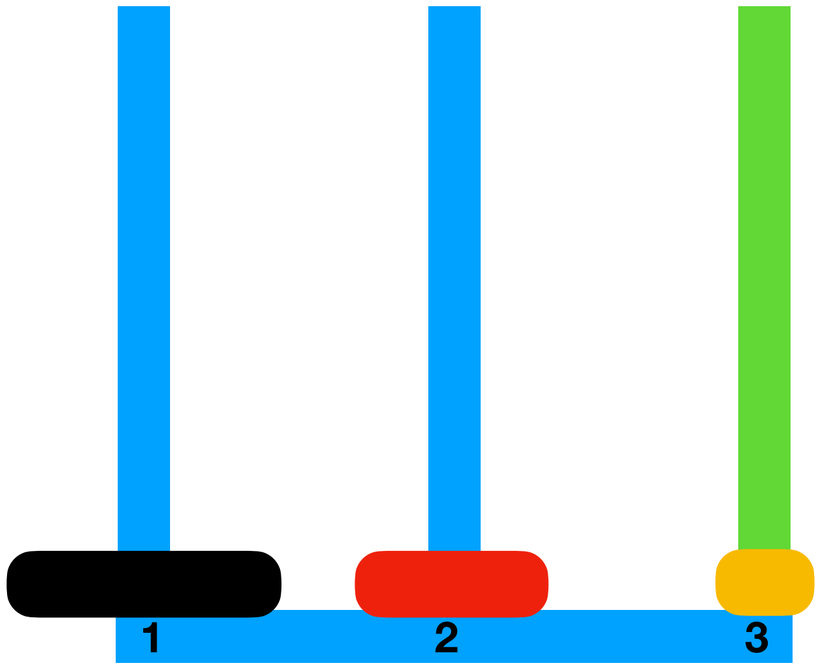
Example: Towers of Hanoi



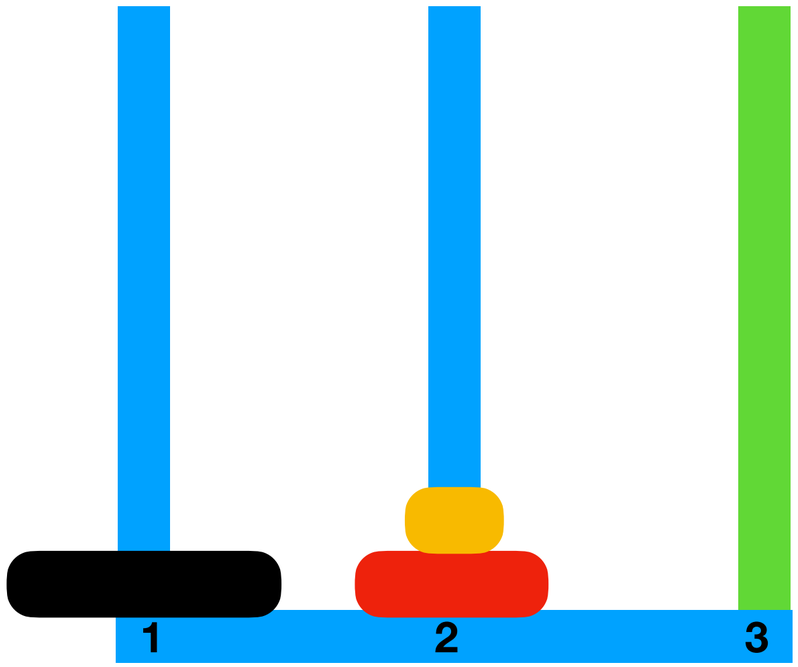
Example: Towers of Hanoi



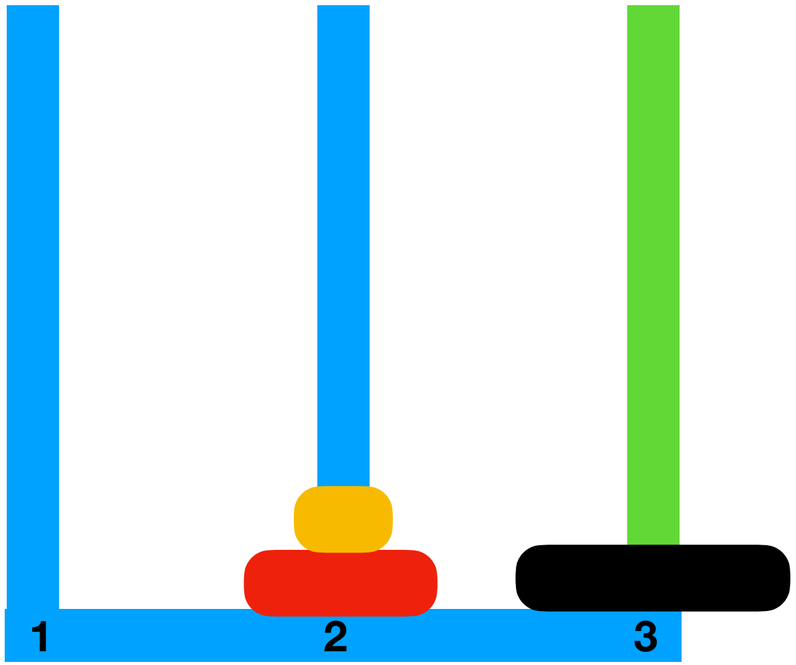
Example: Towers of Hanoi



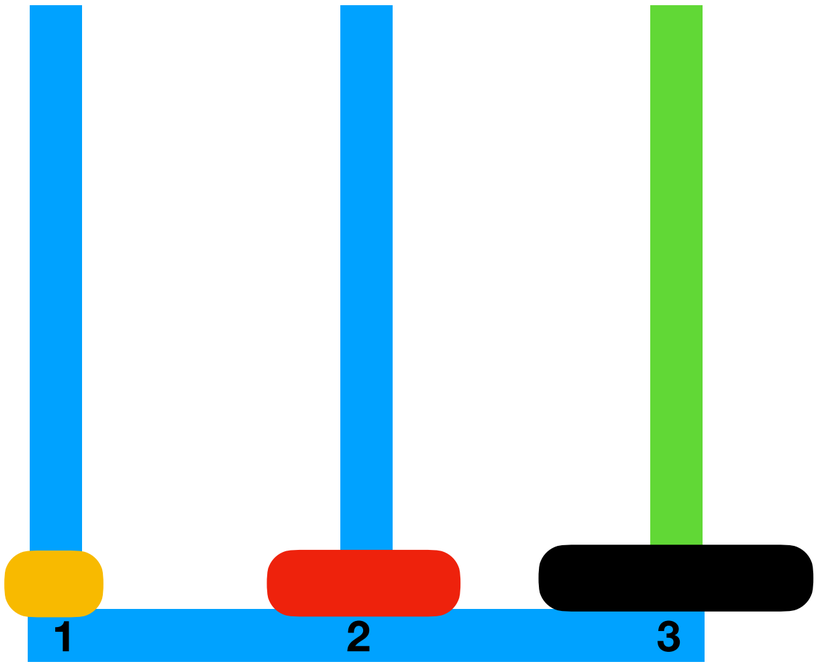
Example: Towers of Hanoi



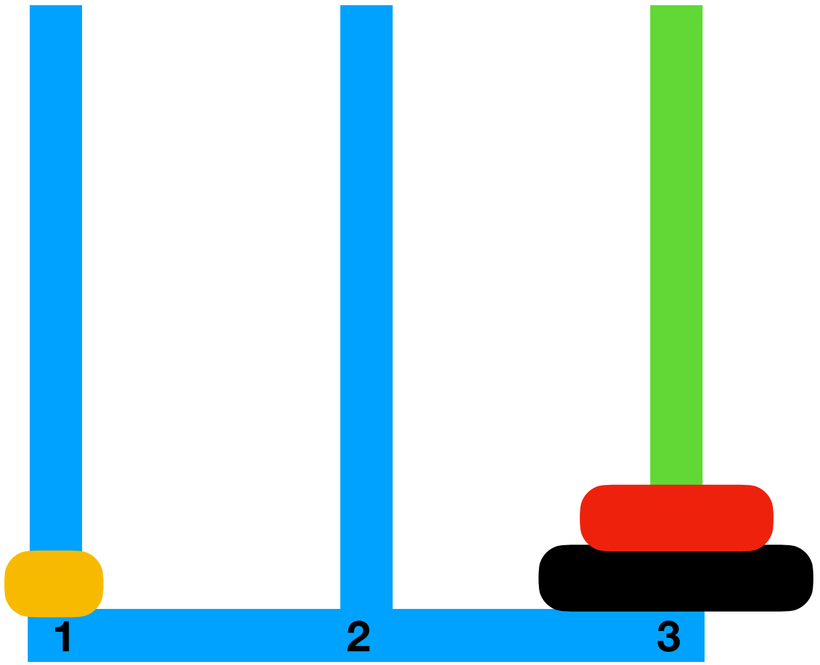
Example: Towers of Hanoi



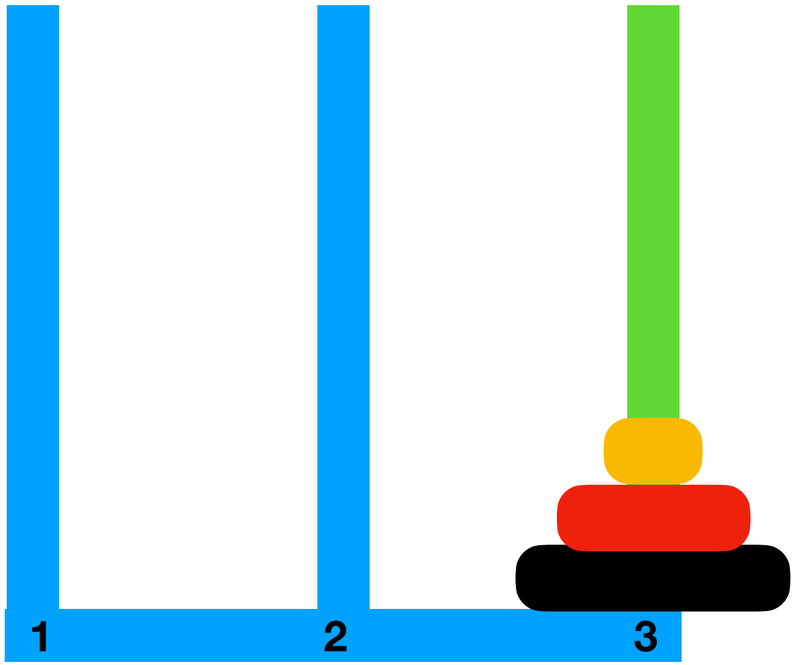
Example: Towers of Hanoi



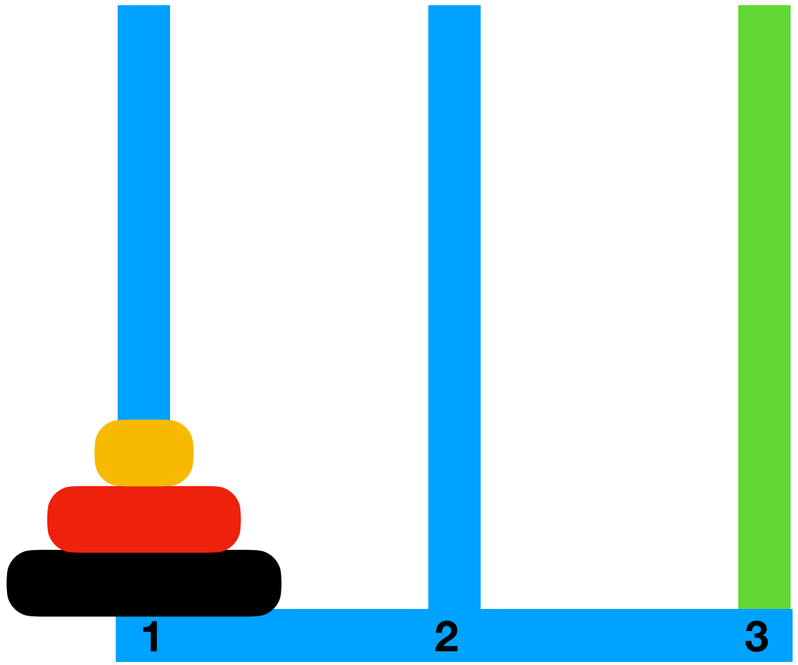
Example: Towers of Hanoi



Example: Towers of Hanoi



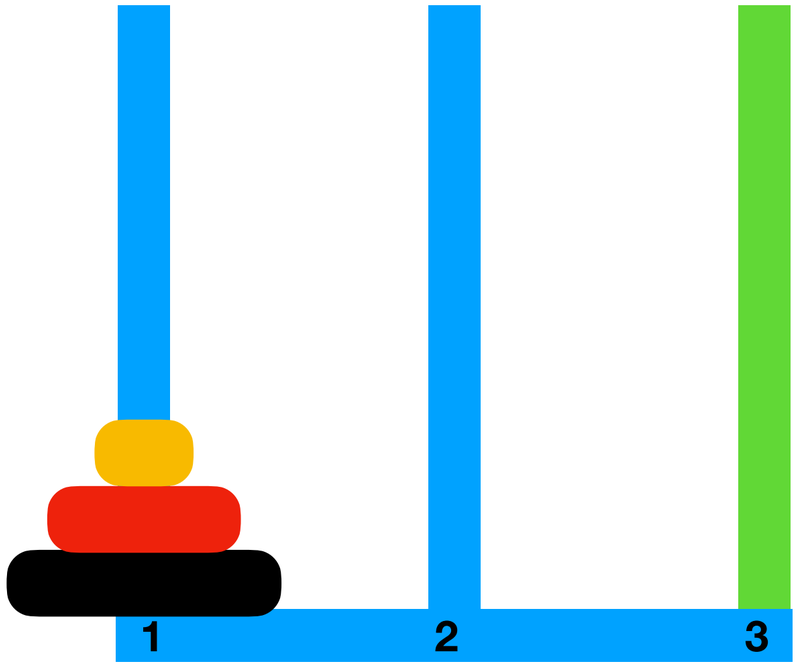
Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

Example: Towers of Hanoi

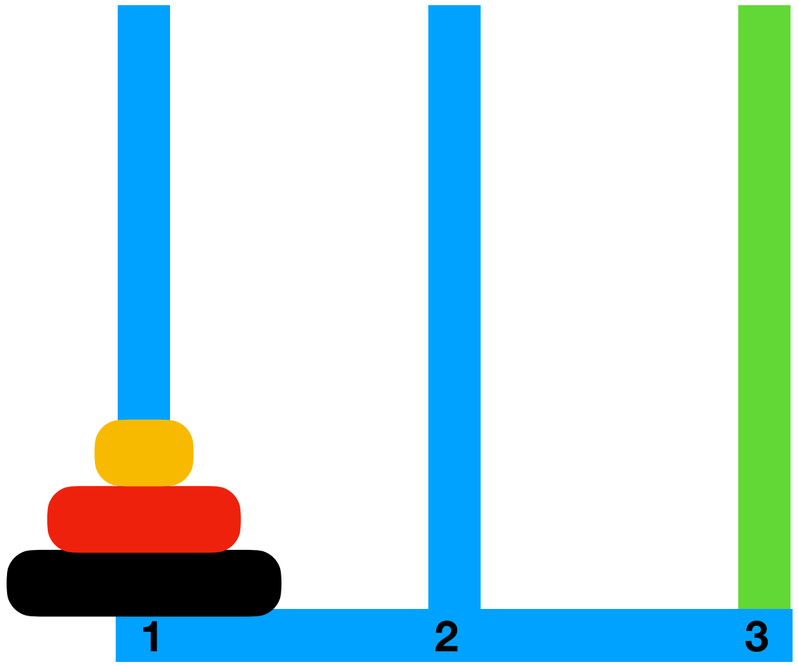


move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)

Example: Towers of Hanoi



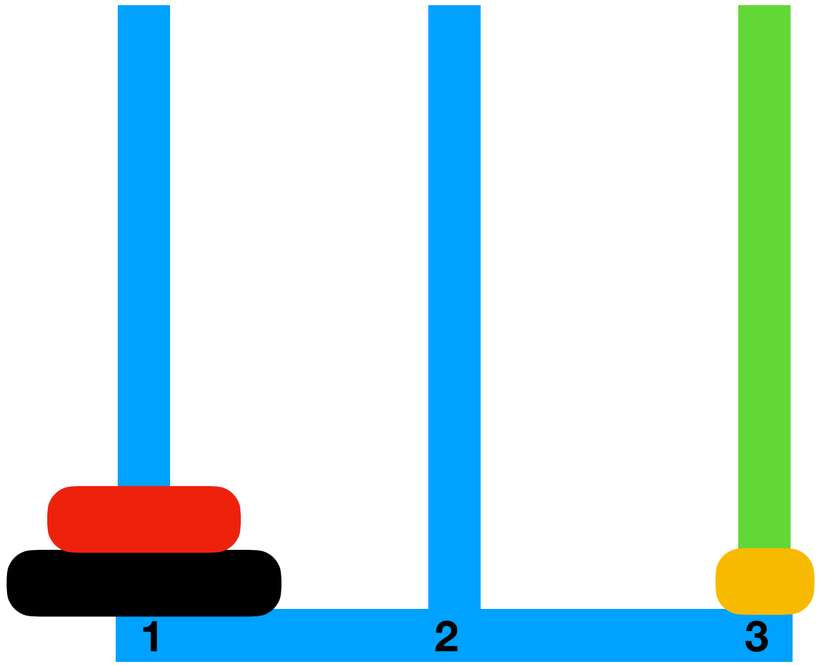
move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

dmove(disk_1, r1, r3)

move(disks2, r1, r2, r3)

Example: Towers of Hanoi



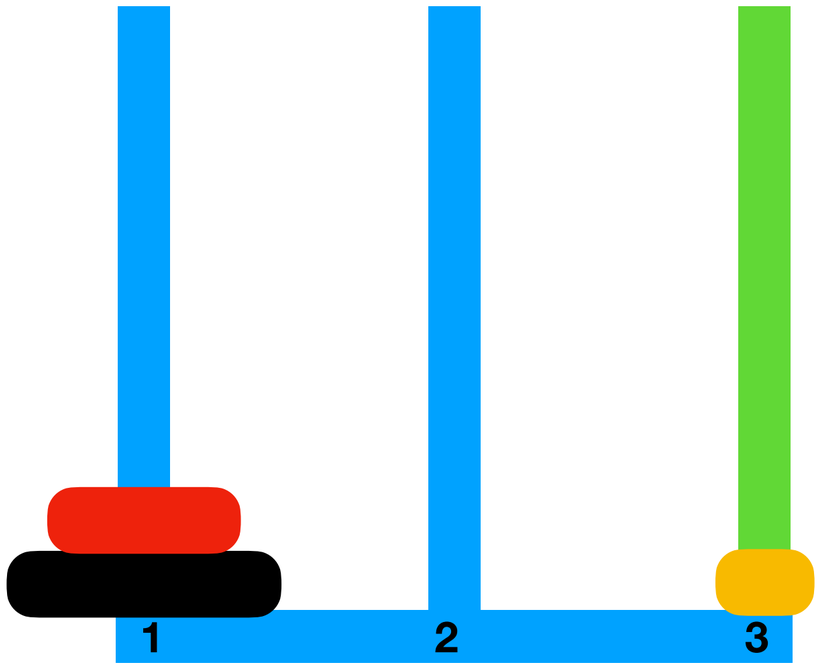
move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

dmove(disk_1, r1, r3)

move(disks2, r1, r2, r3)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

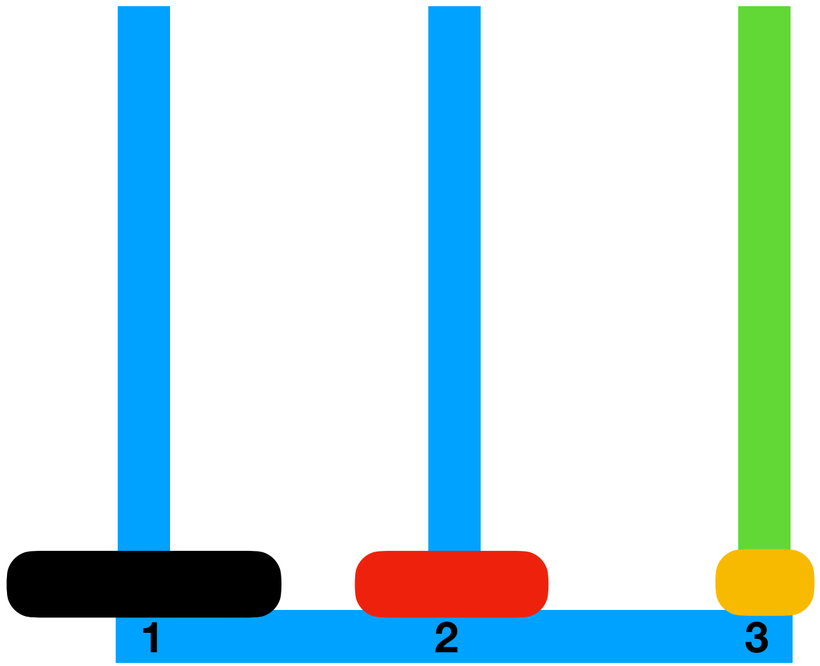
move(disks3, r1, r3, r2)

dmove(disk_1, r1, r3)

dmove(disk_2, r1, r2)

move(disks2, r1, r2, r3)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

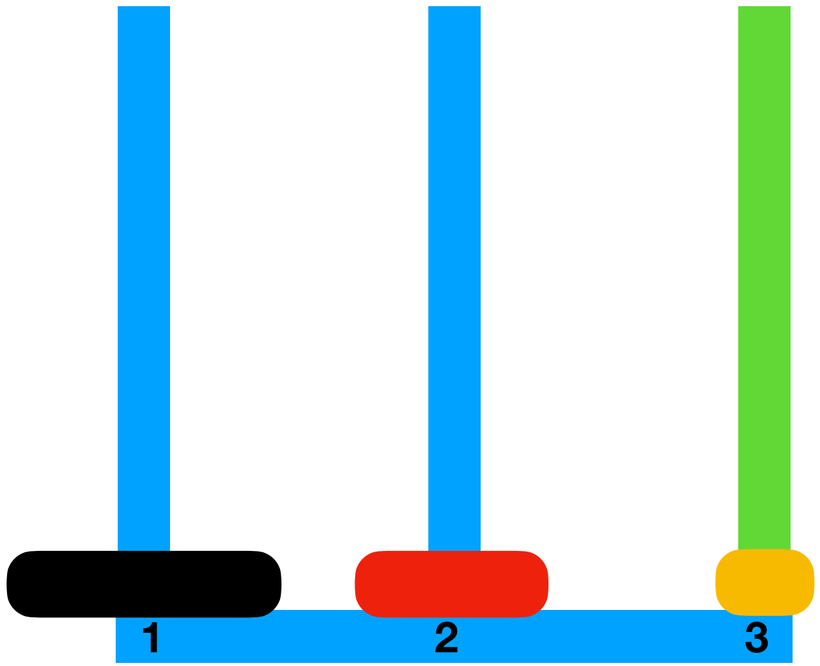
move(disks3, r1, r3, r2)

dmove(disk_1, r1, r3)

dmove(disk_2, r1, r2)

move(disks2, r1, r2, r3)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

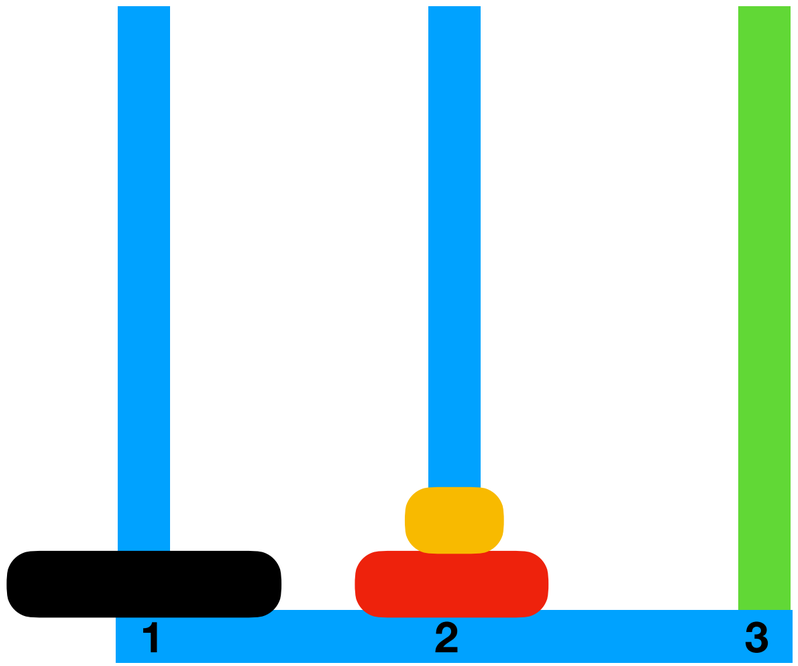
move(disks2, r1, r2, r3)

dmove(disk_1, r1, r3)

dmove(disk_2, r1, r2)

dmove(disk_1, r3, r2)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

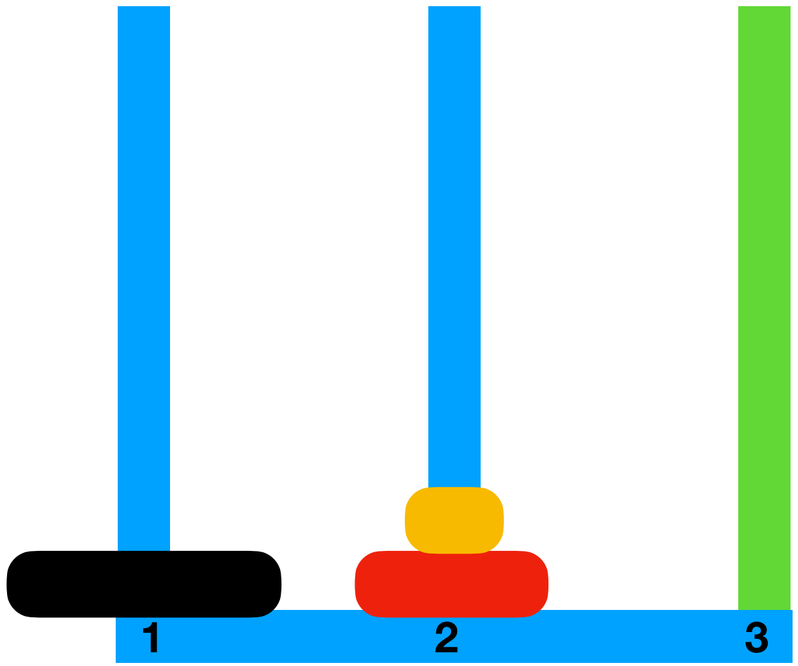
move(disks2, r1, r2, r3)

dmove(disk_1, r1, r3)

dmove(disk_2, r1, r2)

dmove(disk_1, r3, r2)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)

+

dmove(disk_3, r1, r3)

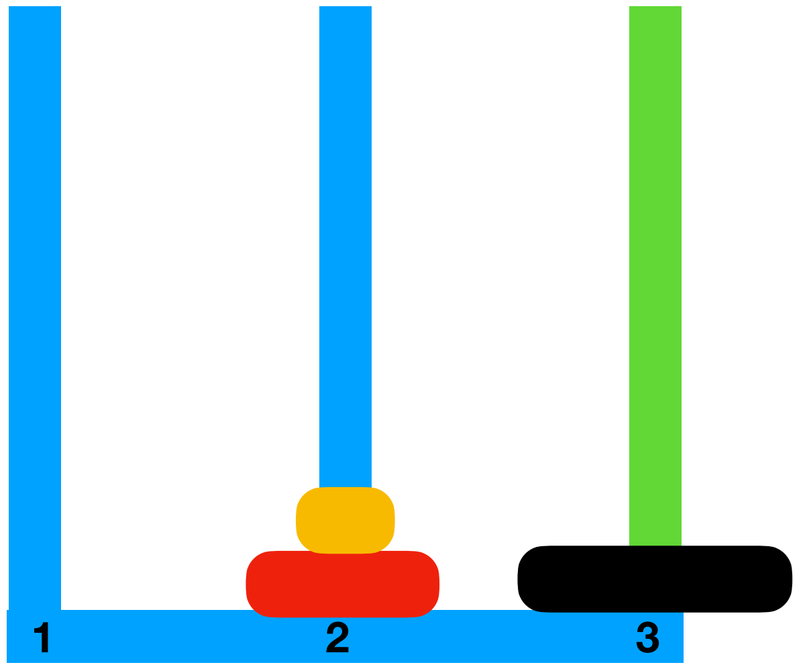
dmove(disk_1, r1, r3)

dmove(disk_2, r1, r2)

dmove(disk_1, r3, r2)

Example: Towers of Hanoi

move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*



move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)

+

dmove(disk_3, r1, r3)

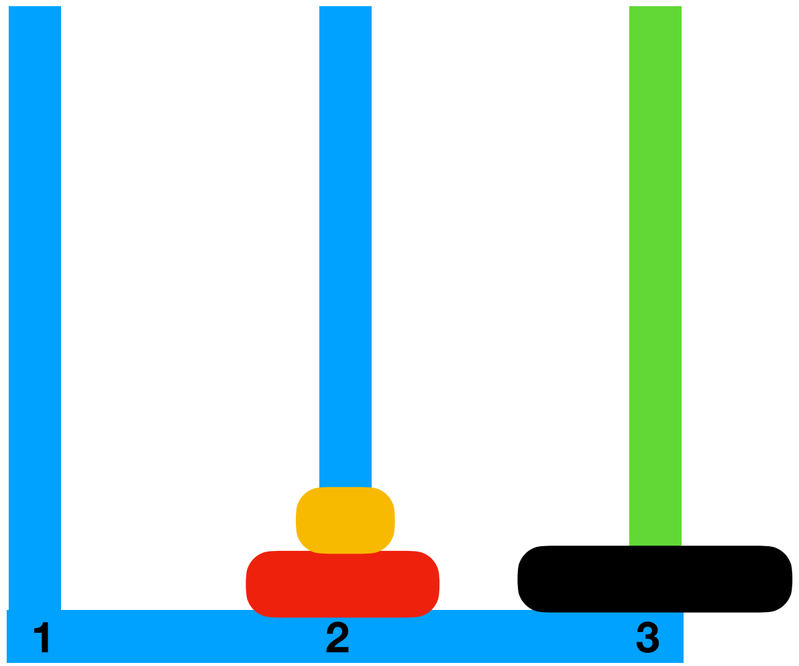
dmove(disk_1, r1, r3)

dmove(disk_2, r1, r2)

dmove(disk_1, r3, r2)

Example: Towers of Hanoi

move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

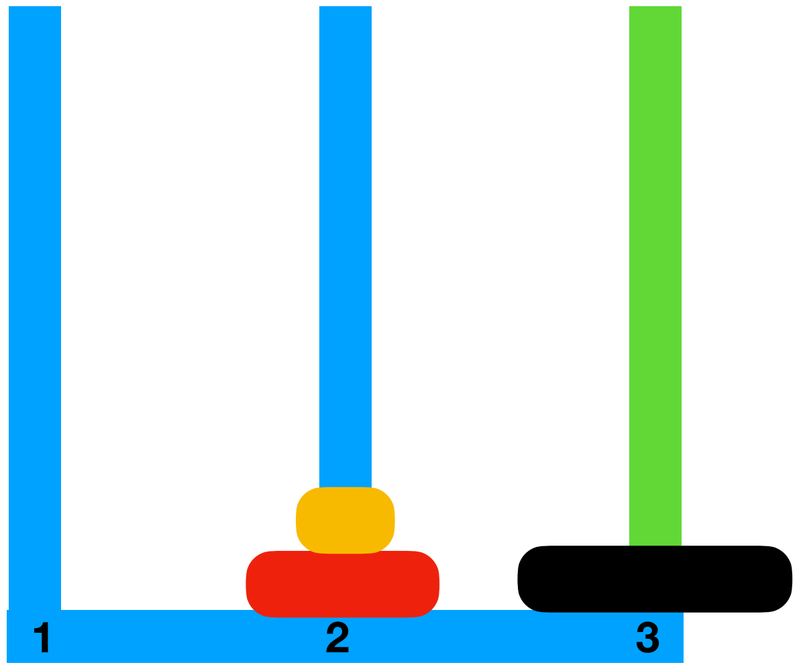


move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)
+
dmove(disk_3, r1, r3)
+
move(disks2, r2, r3, r1)

dmove(disk_1, r1, r3)
dmove(disk_2, r1, r2)
dmove(disk_1, r3, r2)

Example: Towers of Hanoi



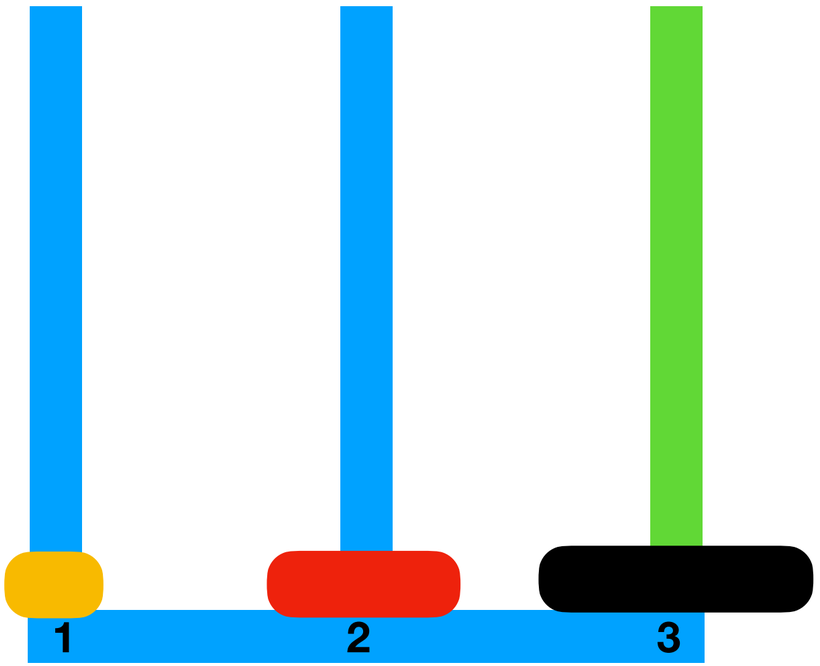
move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)
+
dmove(disk_3, r1, r3)
+
move(disks2, r2, r3, r1)

dmove(disk_1, r1, r3)
dmove(disk_2, r1, r2)
dmove(disk_1, r3, r2)
dmove(disk_1, r2, r1)

Example: Towers of Hanoi



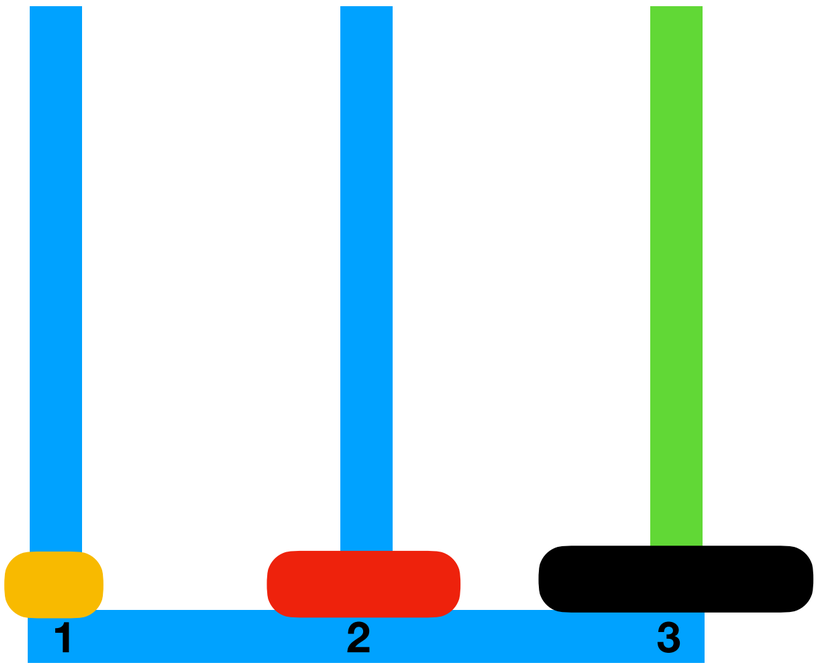
move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)
+
dmove(disk_3, r1, r3)
+
move(disks2, r2, r3, r1)

dmove(disk_1, r1, r3)
dmove(disk_2, r1, r2)
dmove(disk_1, r3, r2)
dmove(disk_1, r2, r1)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

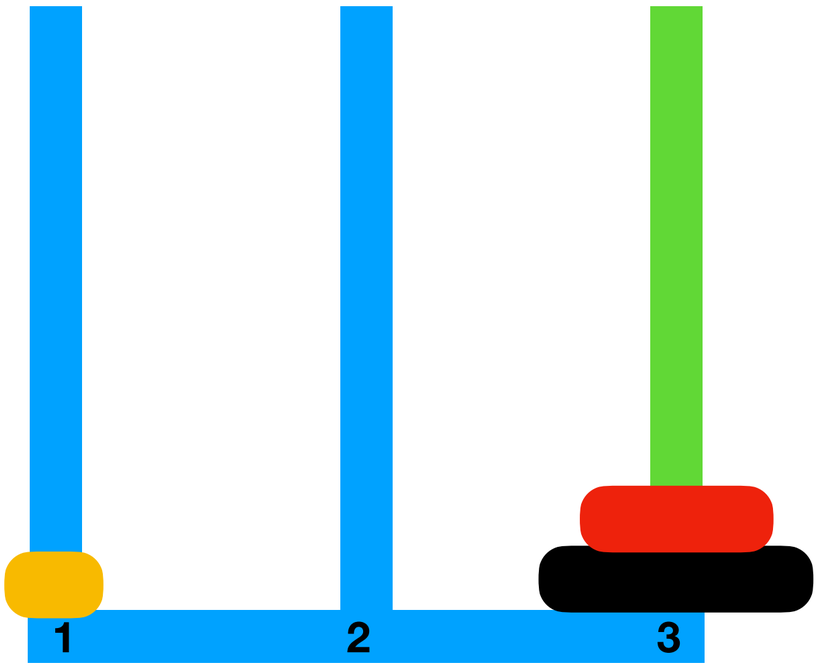
move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)
+
dmove(disk_3, r1, r3)
+
move(disks2, r2, r3, r1)

dmove(disk_1, r1, r3)
dmove(disk_2, r1, r2)
dmove(disk_1, r3, r2)

dmove(disk_1, r2, r1)
dmove(disk_2, r2, r3)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

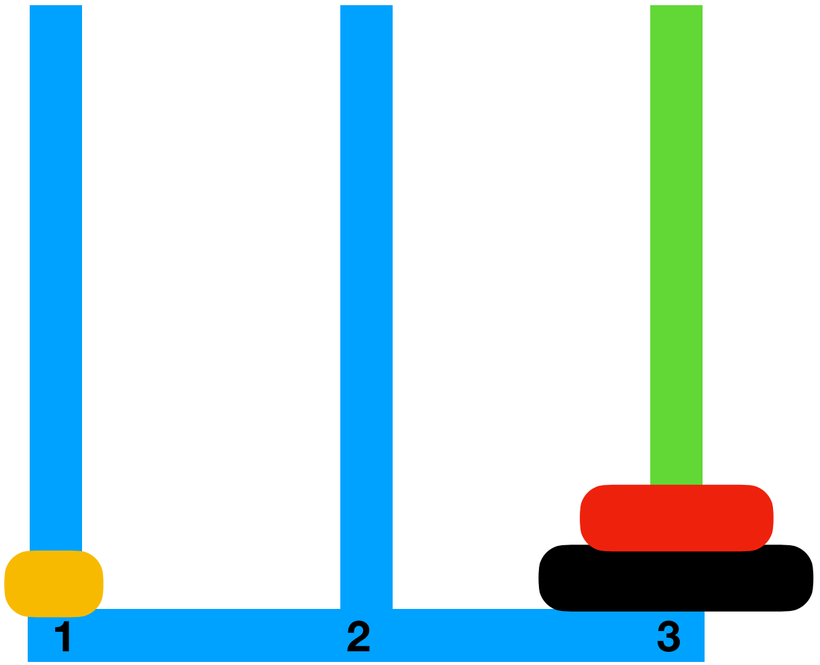
move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)
+
dmove(disk_3, r1, r3)
+
move(disks2, r2, r3, r1)

dmove(disk_1, r1, r3)
dmove(disk_2, r1, r2)
dmove(disk_1, r3, r2)

dmove(disk_1, r2, r1)
dmove(disk_2, r2, r3)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

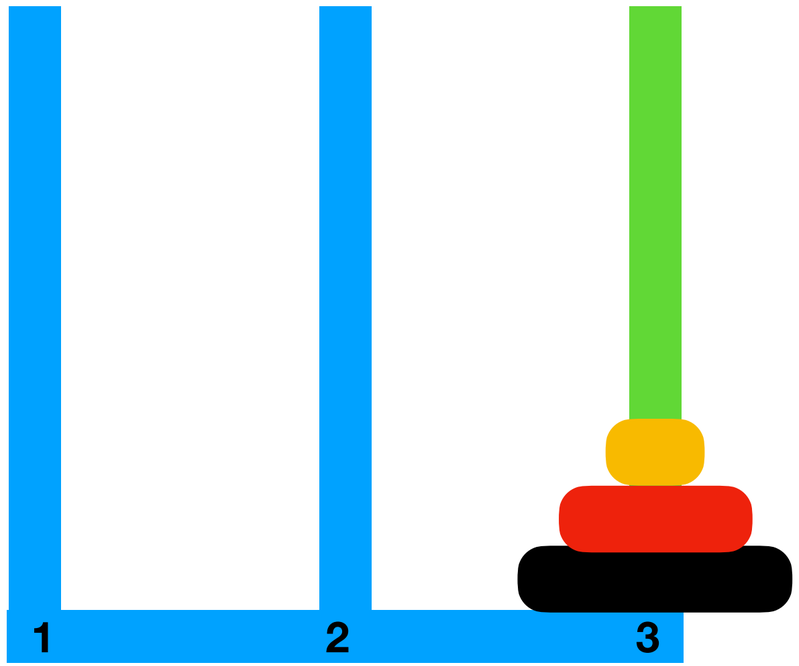
move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)
+
dmove(disk_3, r1, r3)
+
move(disks2, r2, r3, r1)

dmove(disk_1, r1, r3)
dmove(disk_2, r1, r2)
dmove(disk_1, r3, r2)

dmove(disk_1, r2, r1)
dmove(disk_2, r2, r3)
dmove(disk_1, r1, r3)

Example: Towers of Hanoi



move(diskStacks, src, dest, temp) *// Move 2+ disks*
dmove(disk_num, src, dest) *// Move single disk*

move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)
+
dmove(disk_3, r1, r3)
+
move(disks2, r2, r3, r1)

dmove(disk_1, r1, r3)
dmove(disk_2, r1, r2)
dmove(disk_1, r3, r2)

dmove(disk_1, r2, r1)
dmove(disk_2, r2, r3)
dmove(disk_1, r1, r3)

Recursion in Towers of Hanoi

Recursion in Towers of Hanoi

move(disks3, r1, r3, r2)

move(disks4, r1, r3, r2)

Recursion in Towers of Hanoi

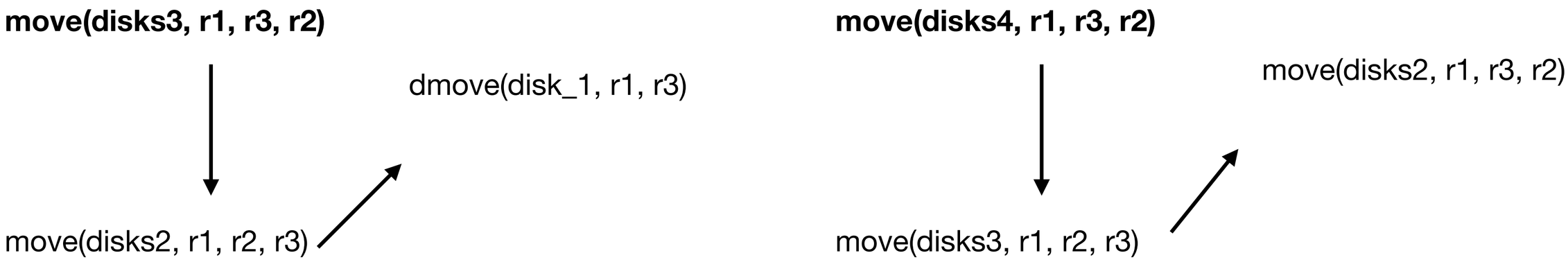
move(disks3, r1, r3, r2)

move(disks2, r1, r2, r3)

move(disks4, r1, r3, r2)

move(disks3, r1, r2, r3)

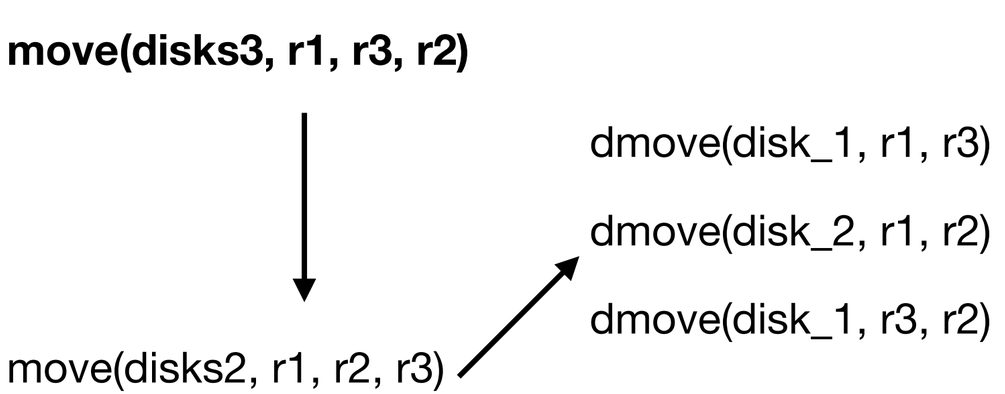
Recursion in Towers of Hanoi

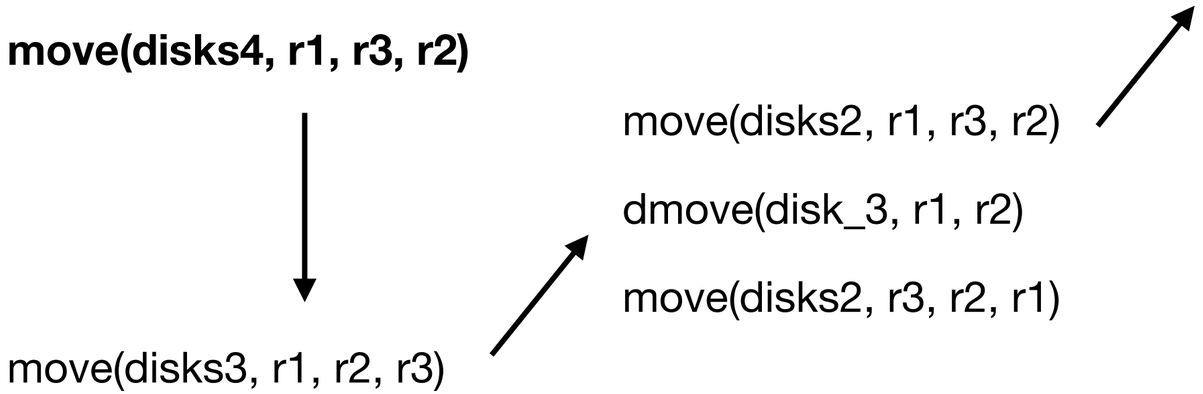


Recursion in Towers of Hanoi

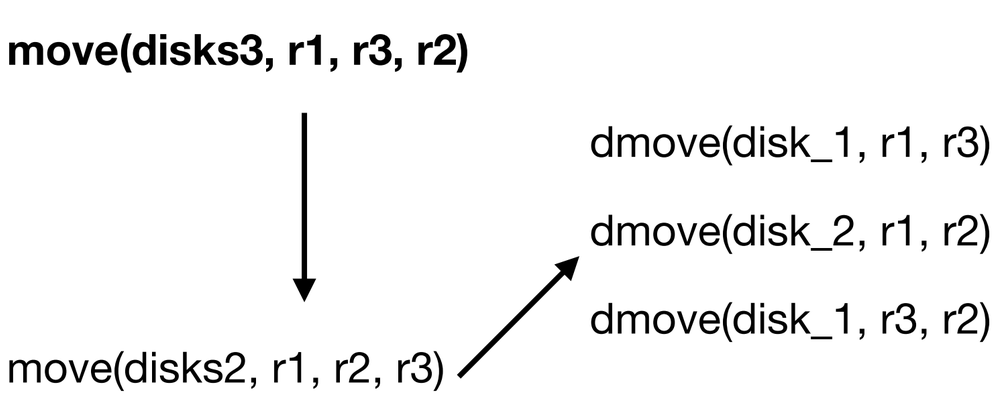
Recursion in Towers of Hanoi

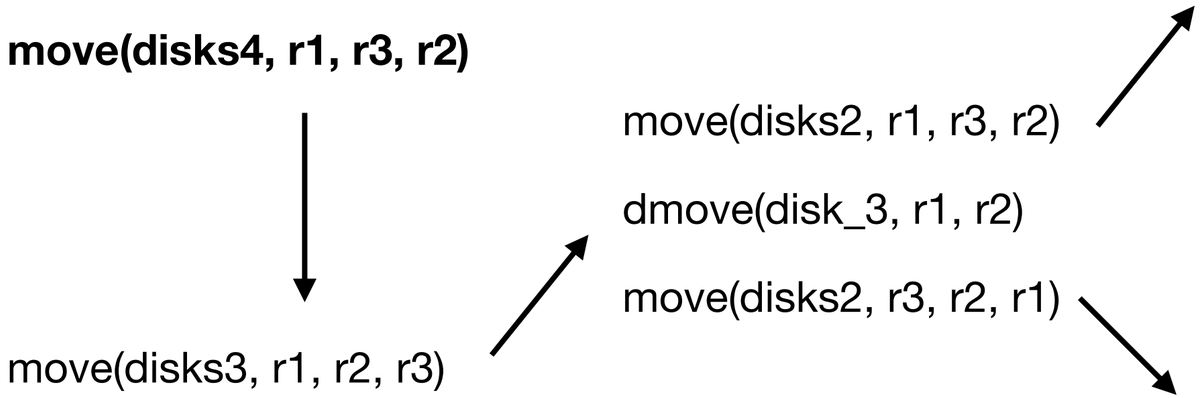
Recursion in Towers of Hanoi



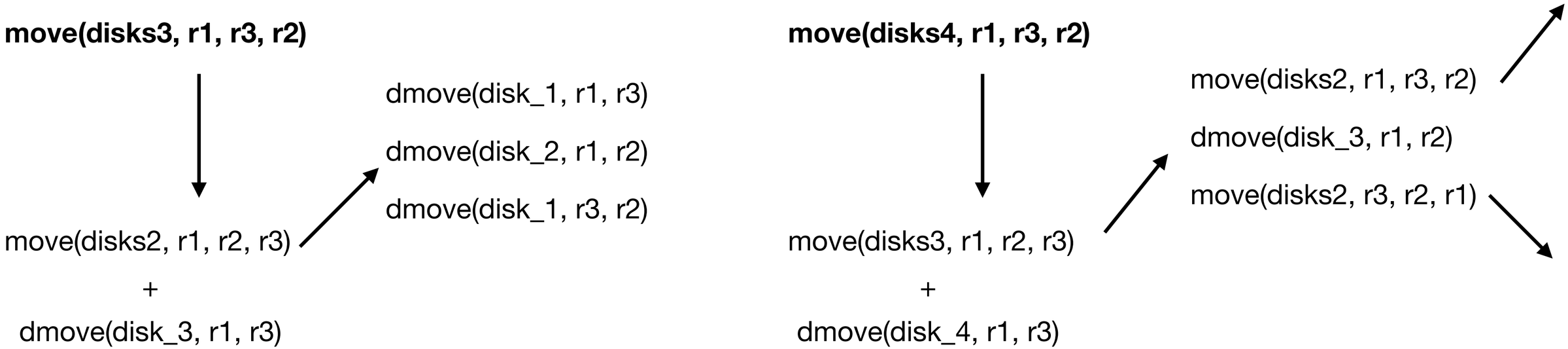


Recursion in Towers of Hanoi

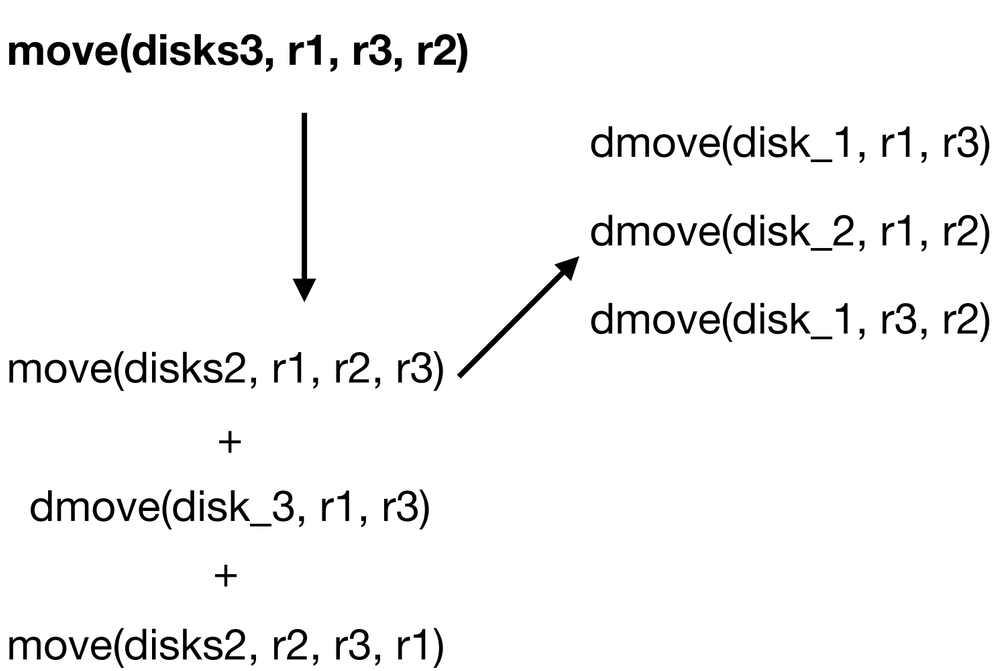


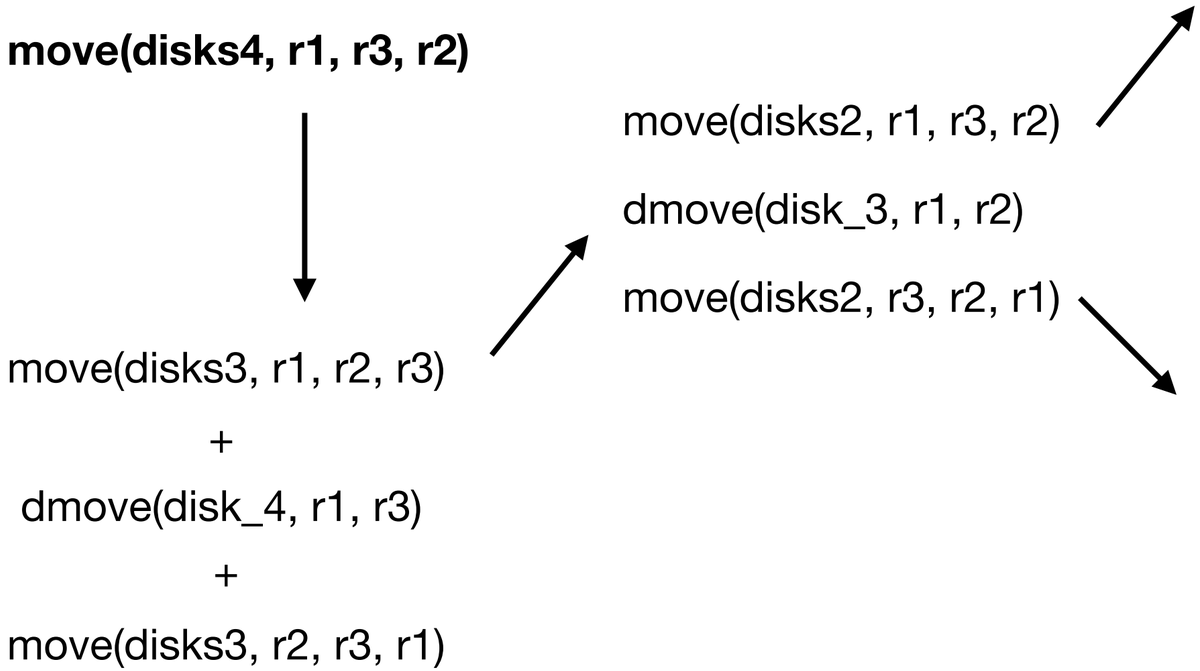


Recursion in Towers of Hanoi

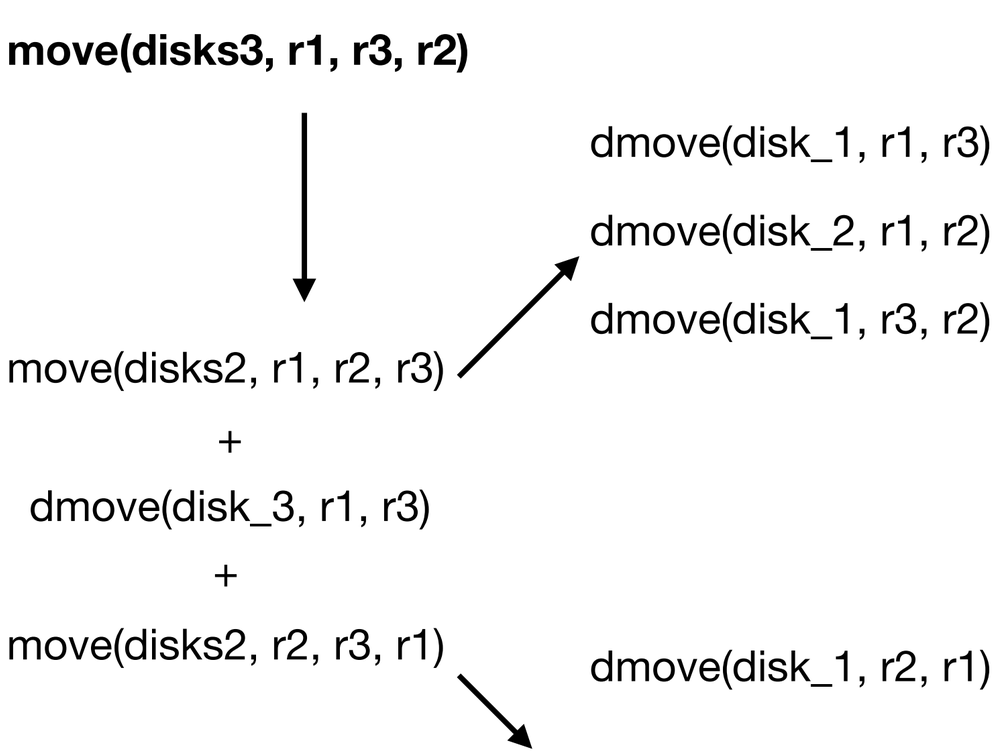


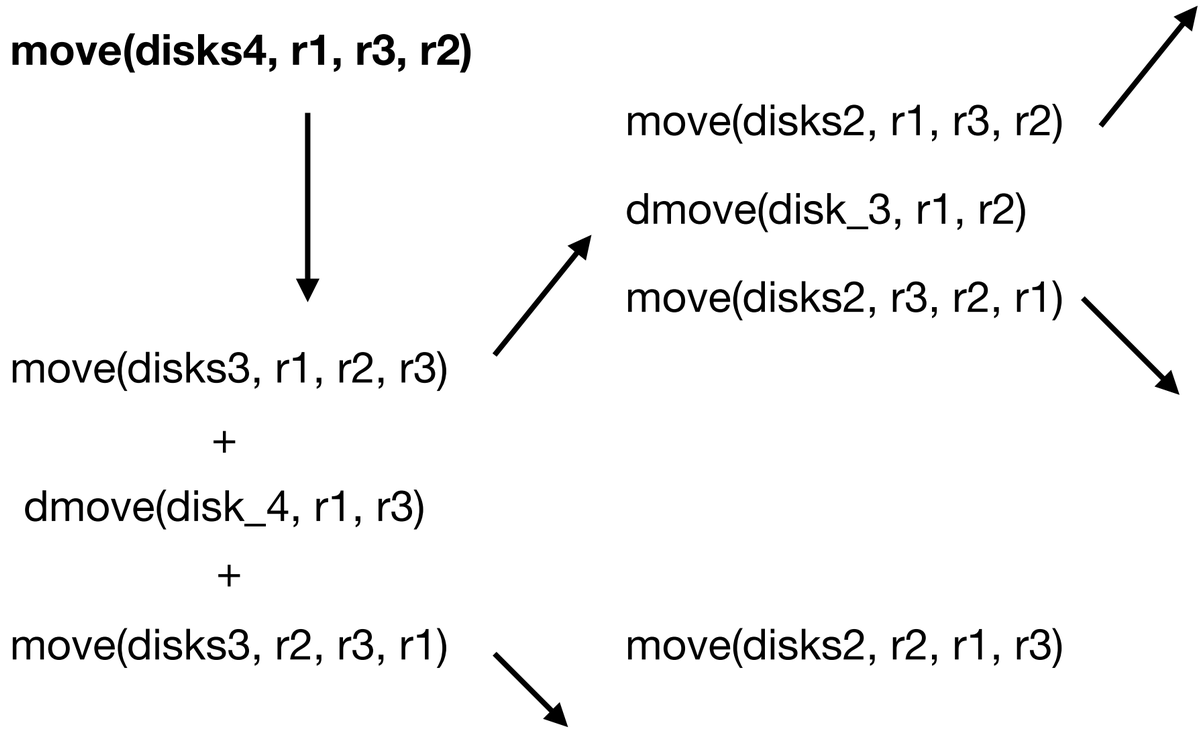
Recursion in Towers of Hanoi



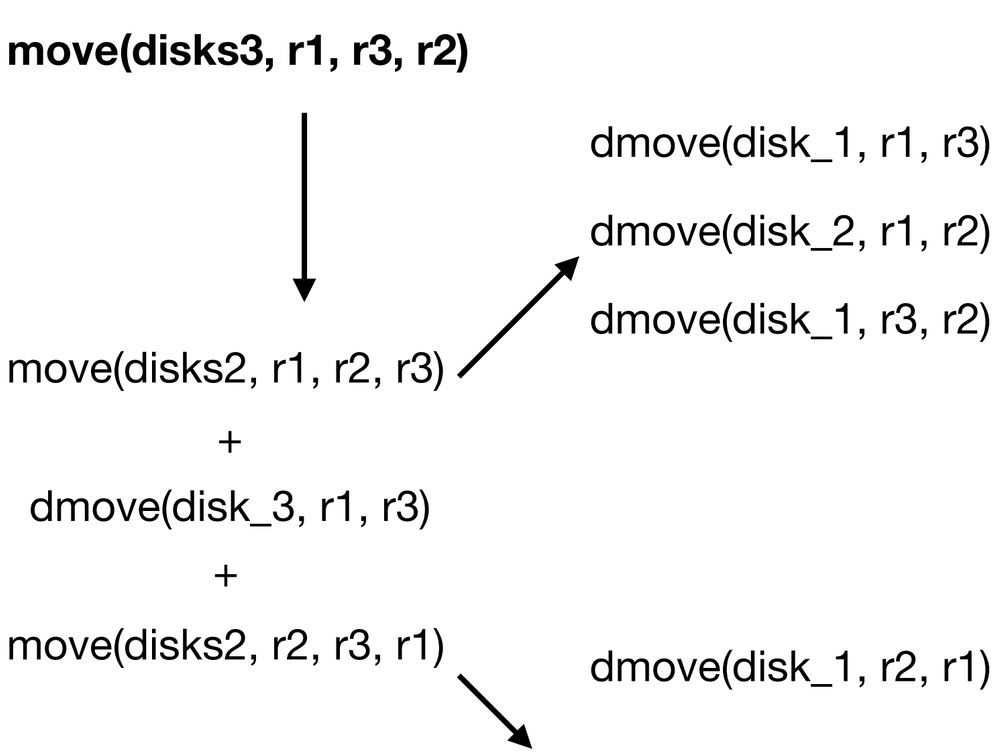


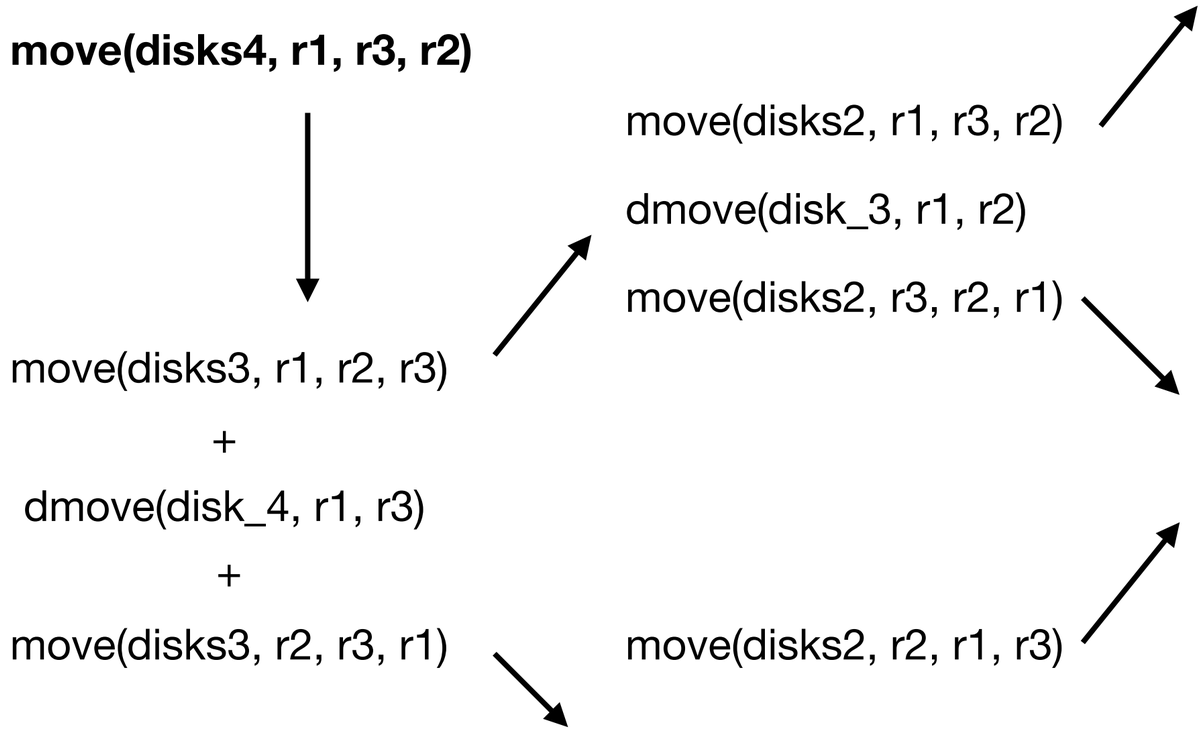
Recursion in Towers of Hanoi



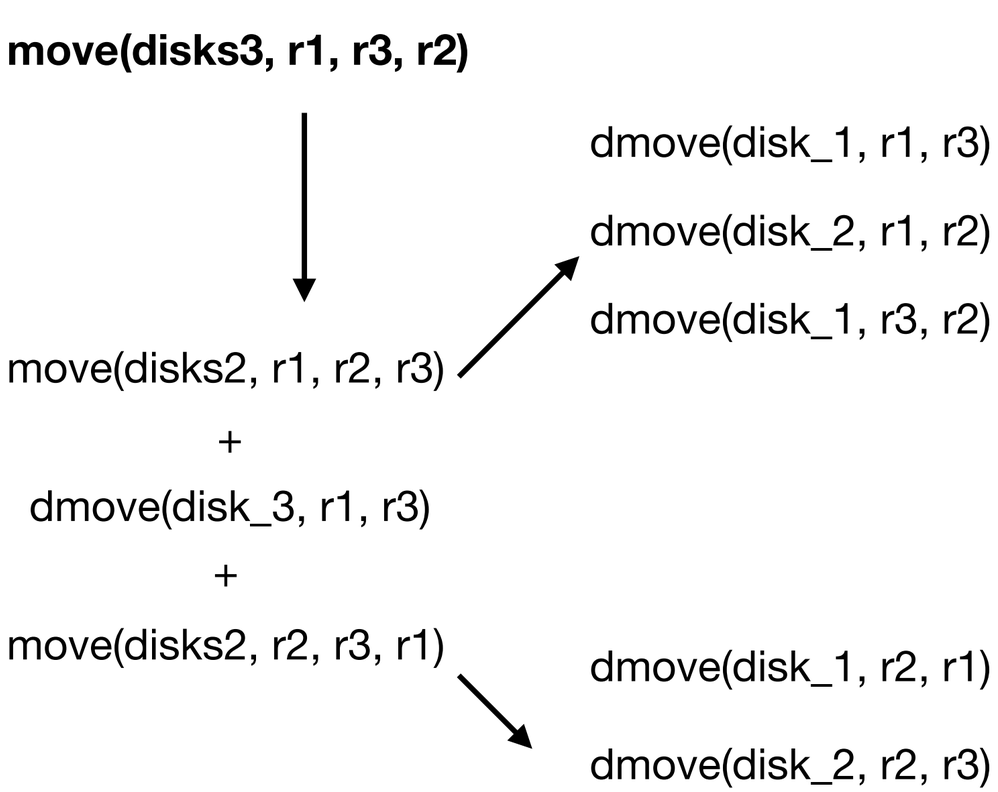


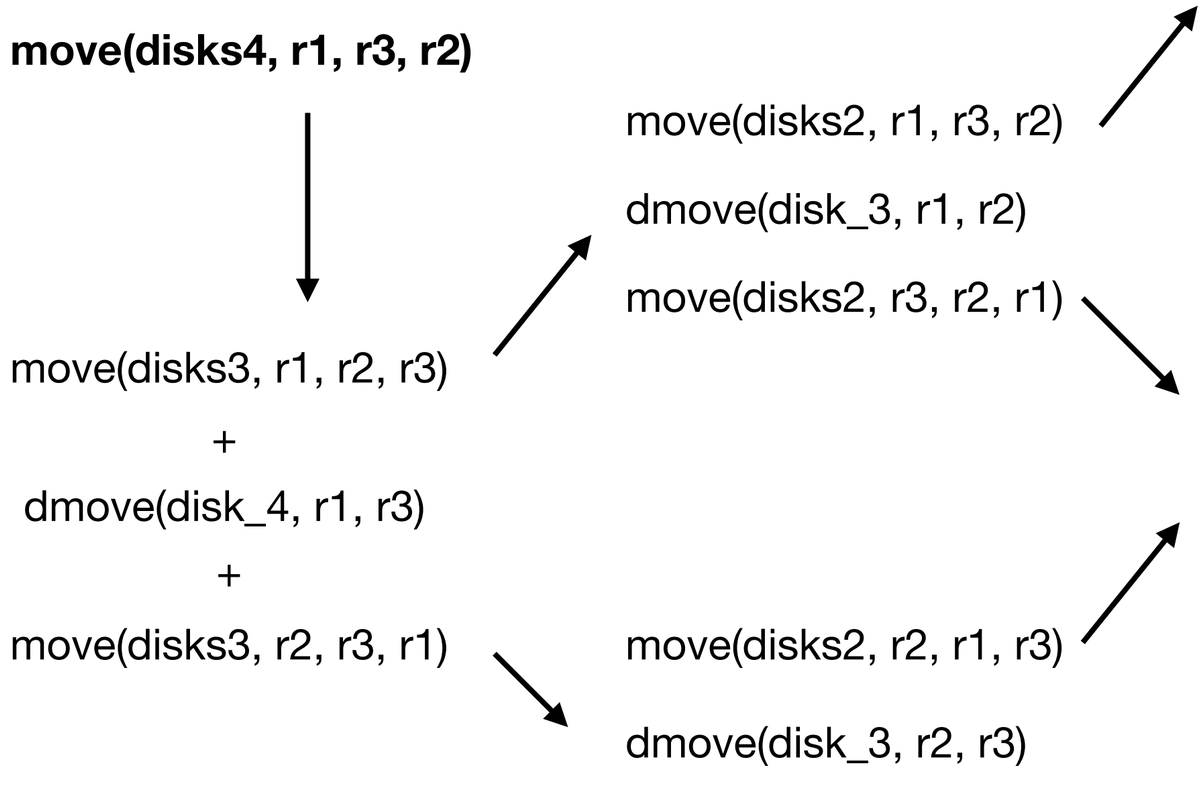
Recursion in Towers of Hanoi



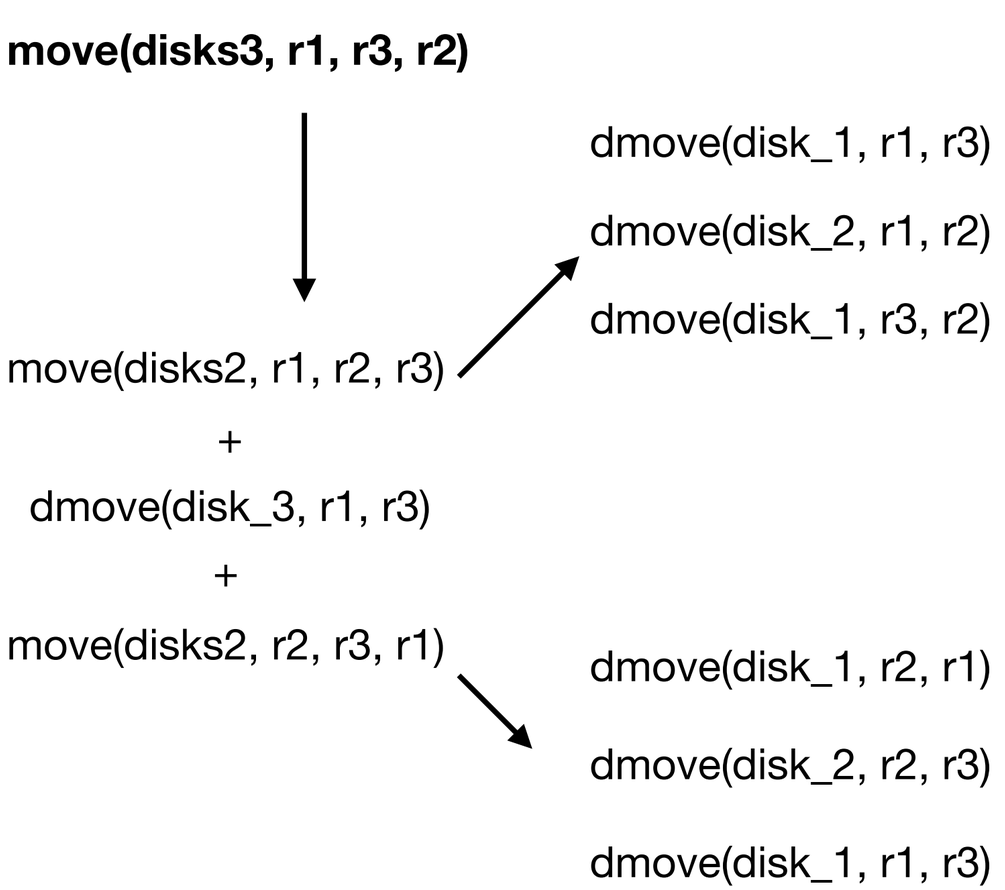


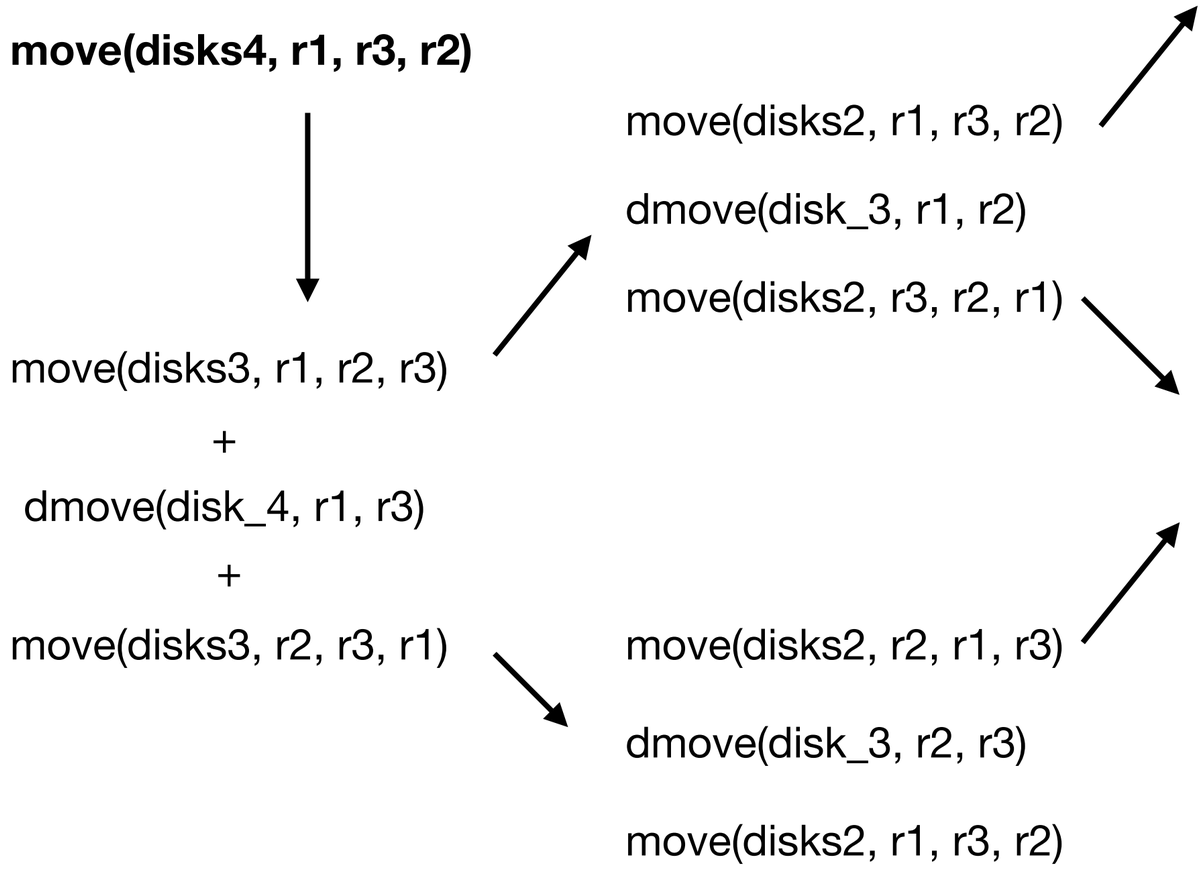
Recursion in Towers of Hanoi



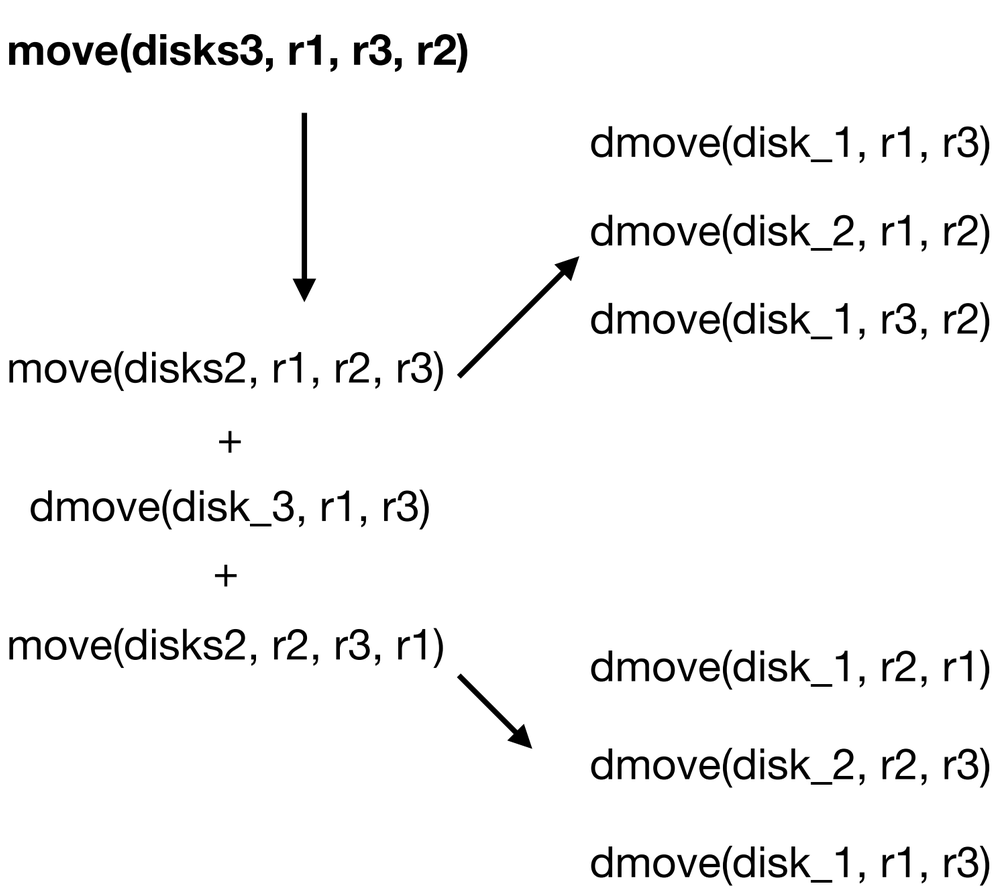


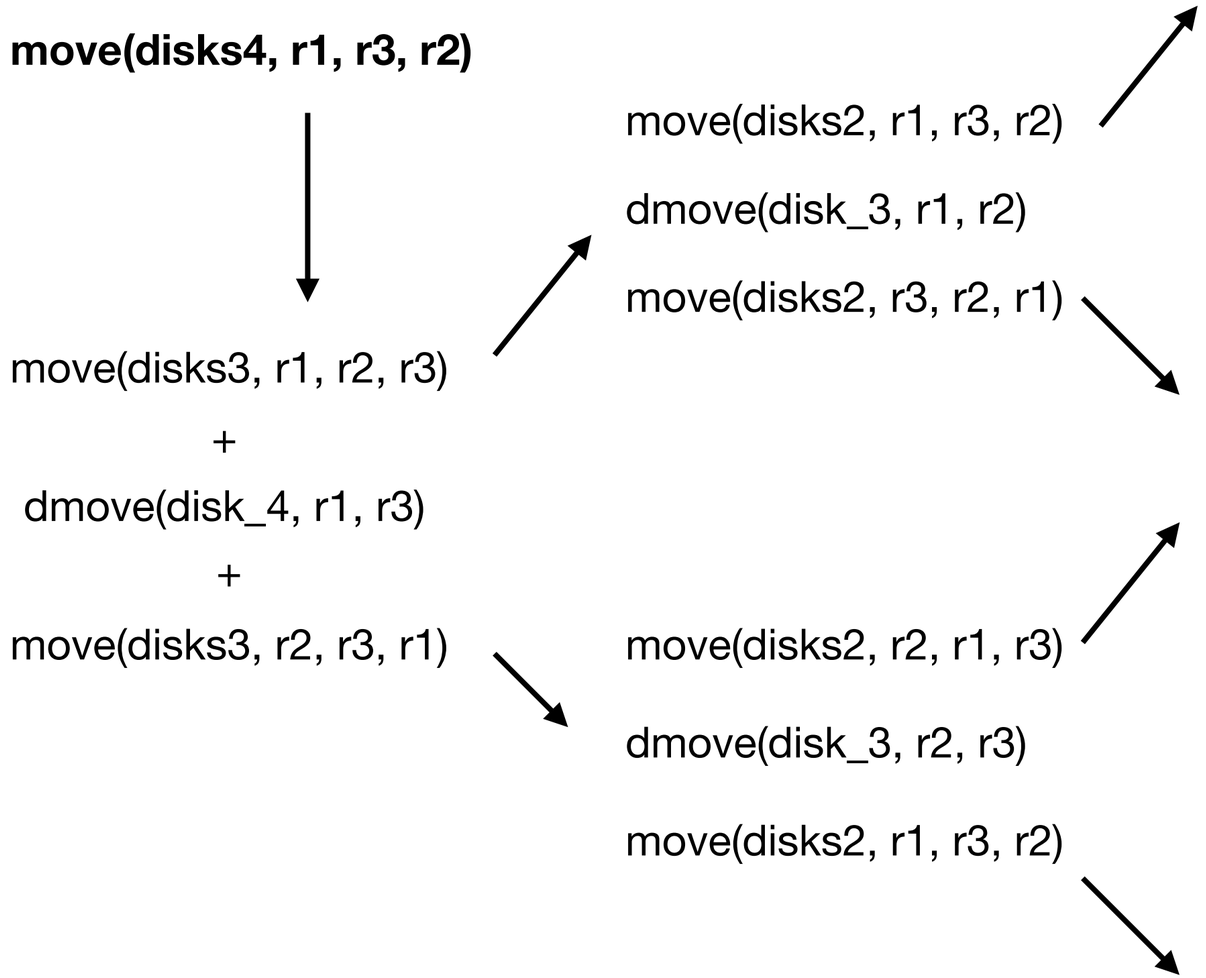
Recursion in Towers of Hanoi





Recursion in Towers of Hanoi





Towers of Hanoi: General formula?

Towers of Hanoi: General formula?

```
void move(disknum, src, dest, temp){
  if (disknum > 1){
    move(disknum - 1, src, temp, dest);

    printf("Moved disk %d from rod %d to %d\n", disknum, src, dest);

    move(disknum - 1, temp, dest, src);
  }
  else
    printf("Moved disk 1 from rod %d to %d\n", src, dest);
}
```

Towers of Hanoi: General formula?

```
void move(disknum, src, dest, temp){
  if (disknum > 1){
    move(disknum - 1, src, temp, dest);
    printf("Moved disk %d from rod %d to %d\n", disknum, src, dest);

    move(disknum - 1, temp, dest, src);
  }
  else
    printf("Moved disk 1 from rod %d to %d\n", src, dest);
}
```

Move a sub-stack to from src to intermediate rod

Towers of Hanoi: General formula?

Move a sub-stack to from src to intermediate rod

Direct move of single disk.

```
void move(disknum, src, dest, temp){
  if (disknum > 1){
    move(disknum - 1, src, temp, dest);
    printf("Moved disk %d from rod %d to %d\n", disknum, src, dest);
    move(disknum - 1, temp, dest, src);
  }
  else
    printf("Moved disk 1 from rod %d to %d\n", src, dest);
}
```

Towers of Hanoi: General formula?

```
void move(disknum, src, dest, temp){
  if (disknum > 1){
    move(disknum - 1, src, temp, dest);
    printf("Moved disk %d from rod %d to %d\n", disknum, src, dest);
    move(disknum - 1, temp, dest, src);
  }
  else
    printf("Moved disk 1 from rod %d to %d\n", src, dest);
}
```

Move a sub-stack to from src to intermediate rod

Direct move of single disk.

Move the sub-stack from intermediate to dest rod

Towers of Hanoi: General formula?

```
void move(disknum, src, dest, temp){
  if (disknum > 1){
    move(disknum - 1, src, temp, dest);
    printf("Moved disk %d from rod %d to %d\n", disknum, src, dest);
    move(disknum - 1, temp, dest, src);
  }
  else
    printf("Moved disk 1 from rod %d to %d\n", src, dest);
}
```

Move a sub-stack to from src to intermediate rod

Direct move of single disk.

Move the sub-stack from intermediate to dest rod

Base case; all moves involving sub-stacks end-up here.

Recursion in LC3

Recursion in LC3

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 5;
    running_sum(5);
}
```

Recursion in LC3

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 5;
    running_sum(5);
}
```

How can we write equivalent LC3 code?

Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 5;
    running_sum(5);
}
```

How can we write equivalent LC3 code?

Recall function calls are implemented using the run time stack.

Recursion in LC3

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 5;
    running_sum(5);
}
```

How can we write equivalent LC3 code?

Recall function calls are implemented using the run time stack.

Recursive calls need not be treated any different from normal function calls!

Generating an activation record

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)
4. Execute function

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)
4. Execute function
5. *Callee* tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)
4. Execute function
5. *Callee* tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)
6. Return to caller (`RET`)

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)
4. Execute function
5. *Callee* tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)
6. Return to caller (`RET`)

Callee

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)
4. Execute function
5. *Callee* tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)
6. Return to caller (`RET`)

Callee

7. *Caller* tear-down (pop callee's return value and arguments from stack)

Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)
4. Execute function
5. *Callee* tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)
6. Return to caller (`RET`)

Callee

7. *Caller* tear-down (pop callee's return value and arguments from stack)

Caller

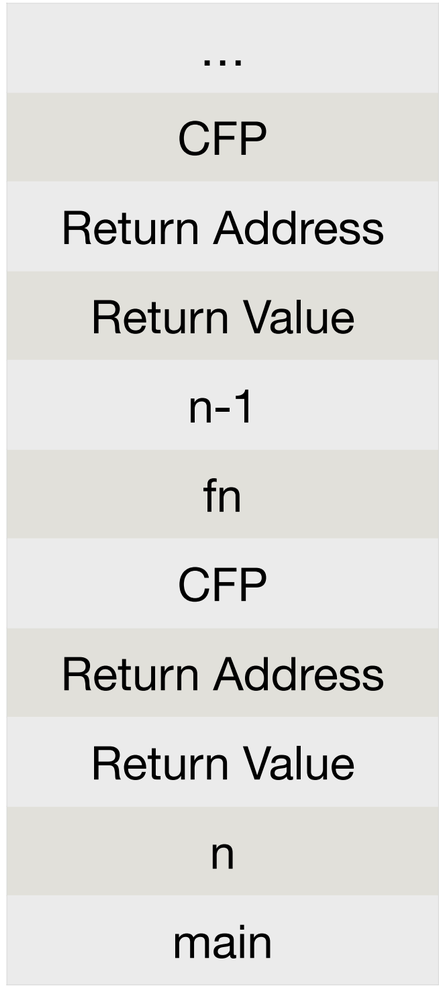
Check Github for reference material: https://github.com/iabraham/ece220-sp25/blob/main/lec12/README.md

Recursion in LC3

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

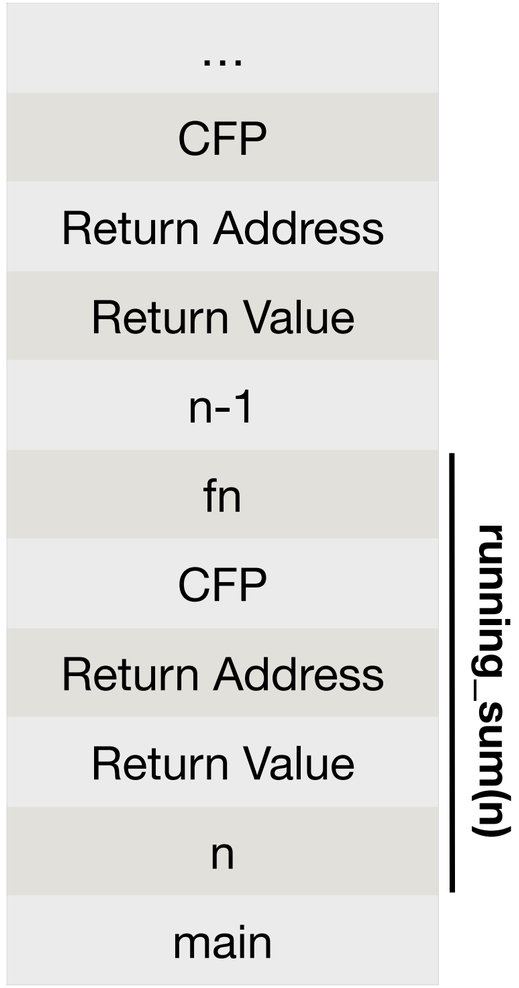
Recursion in LC3

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```



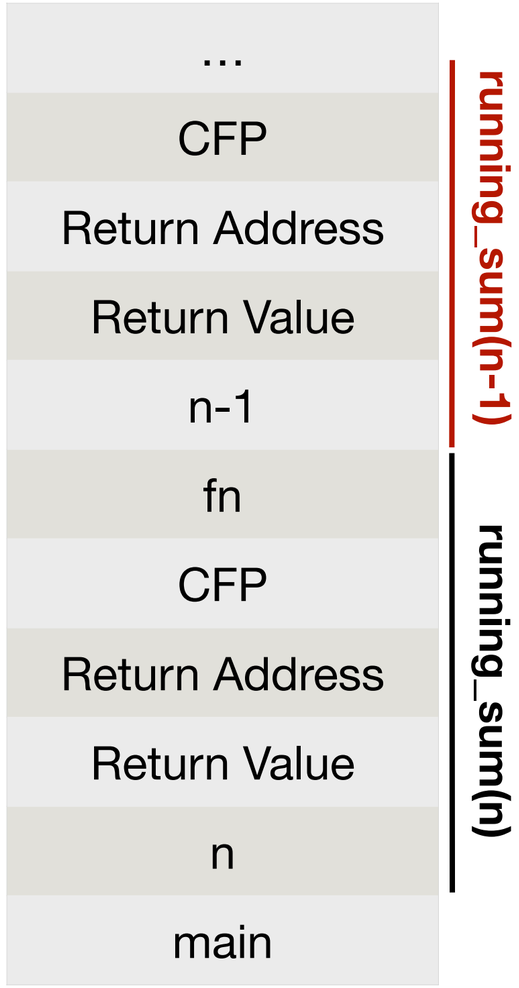
Recursion in LC3

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```



Recursion in LC3

```
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}

int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

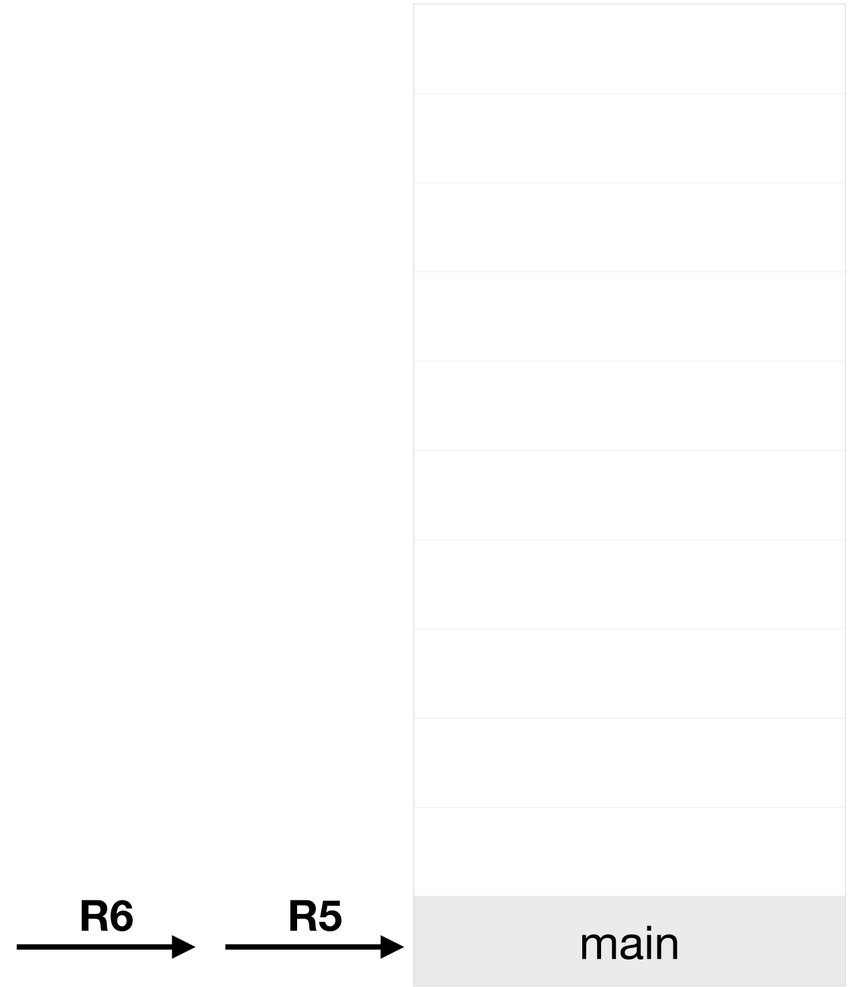


Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
```

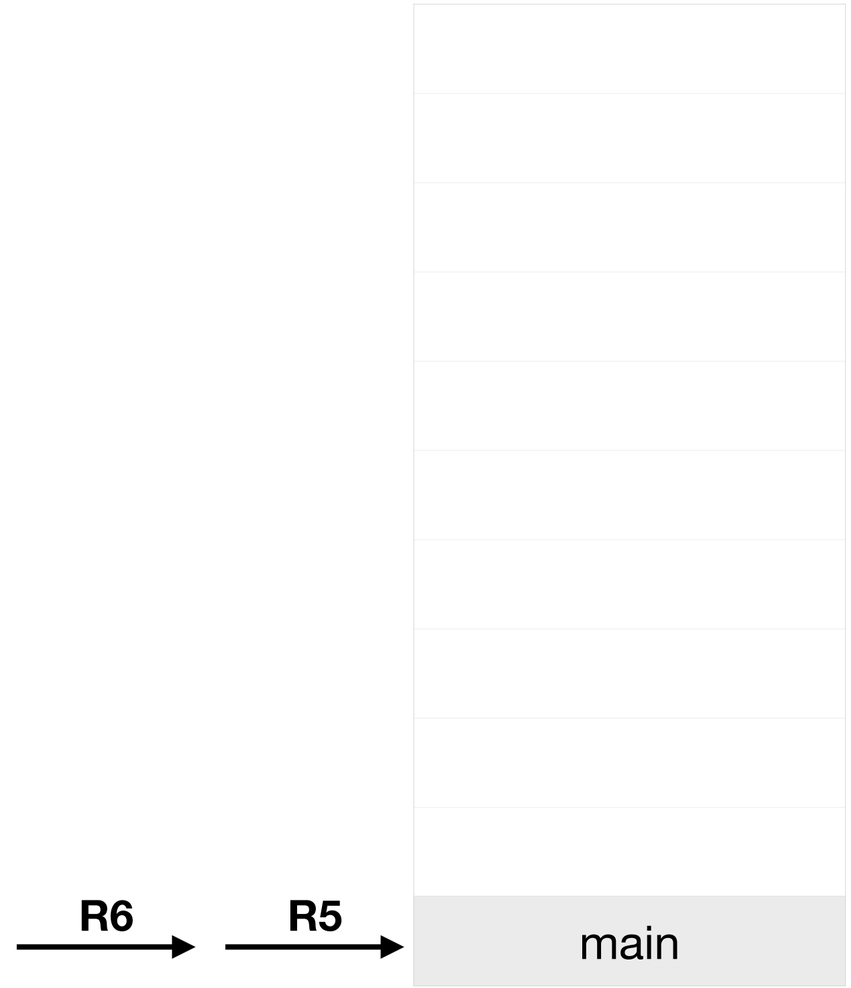


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
```

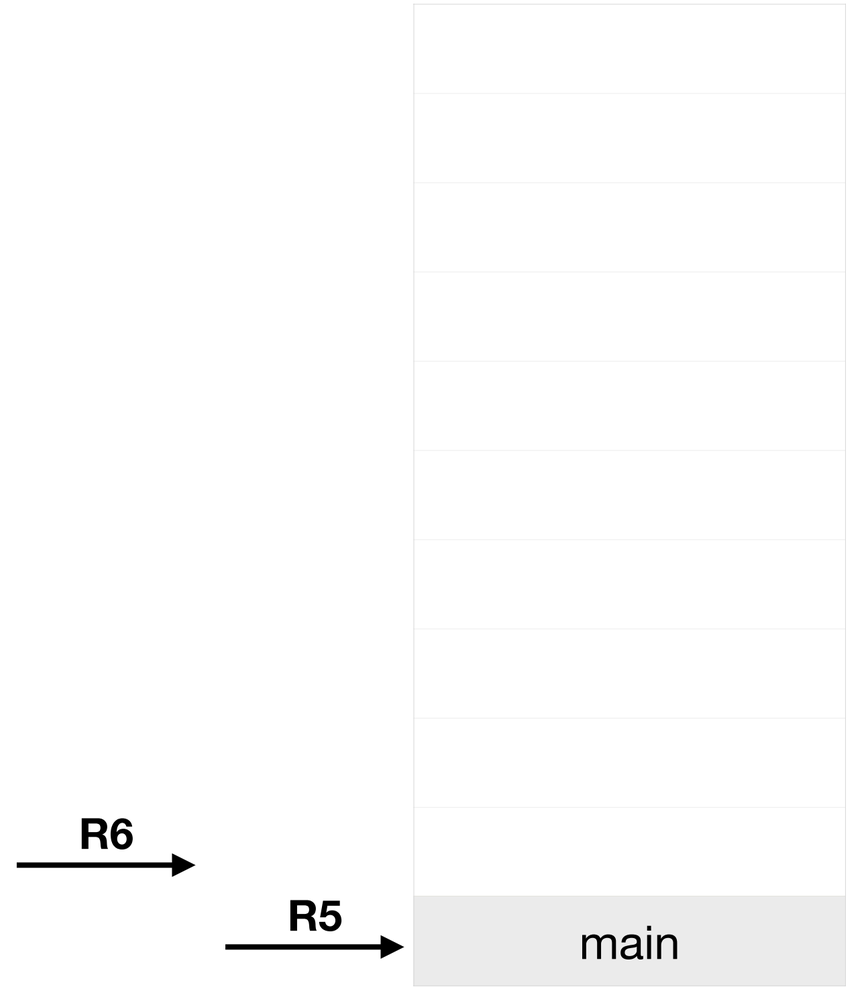


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
```

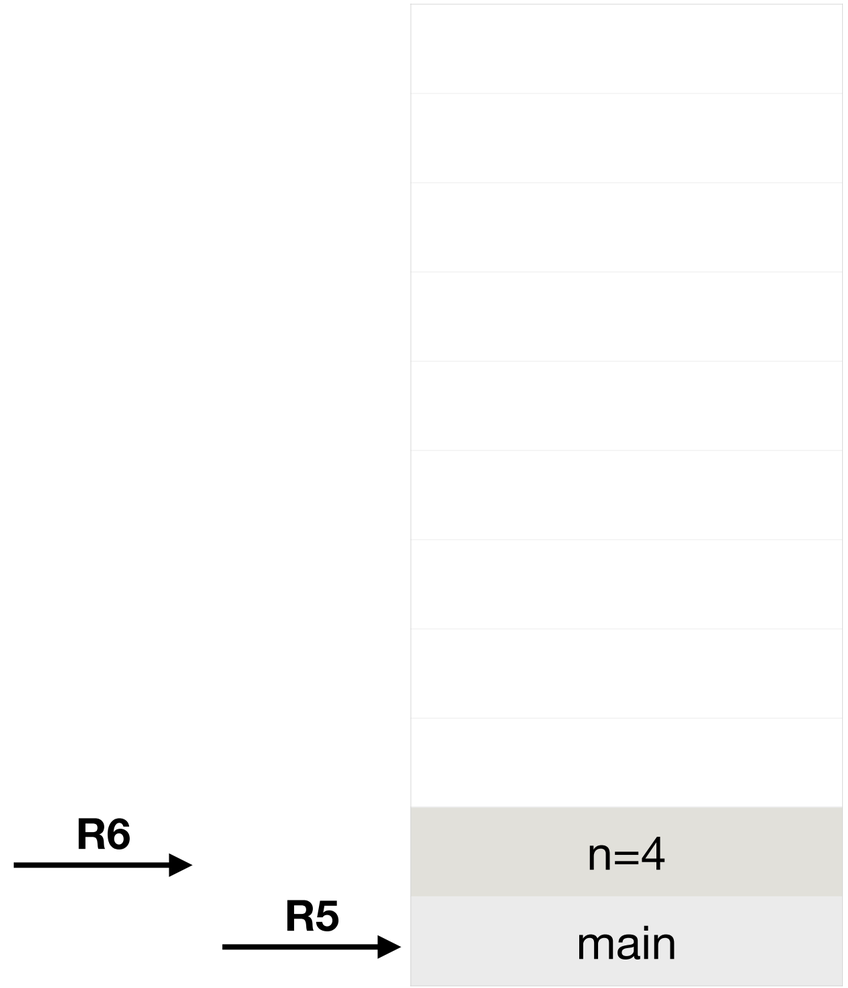


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
```

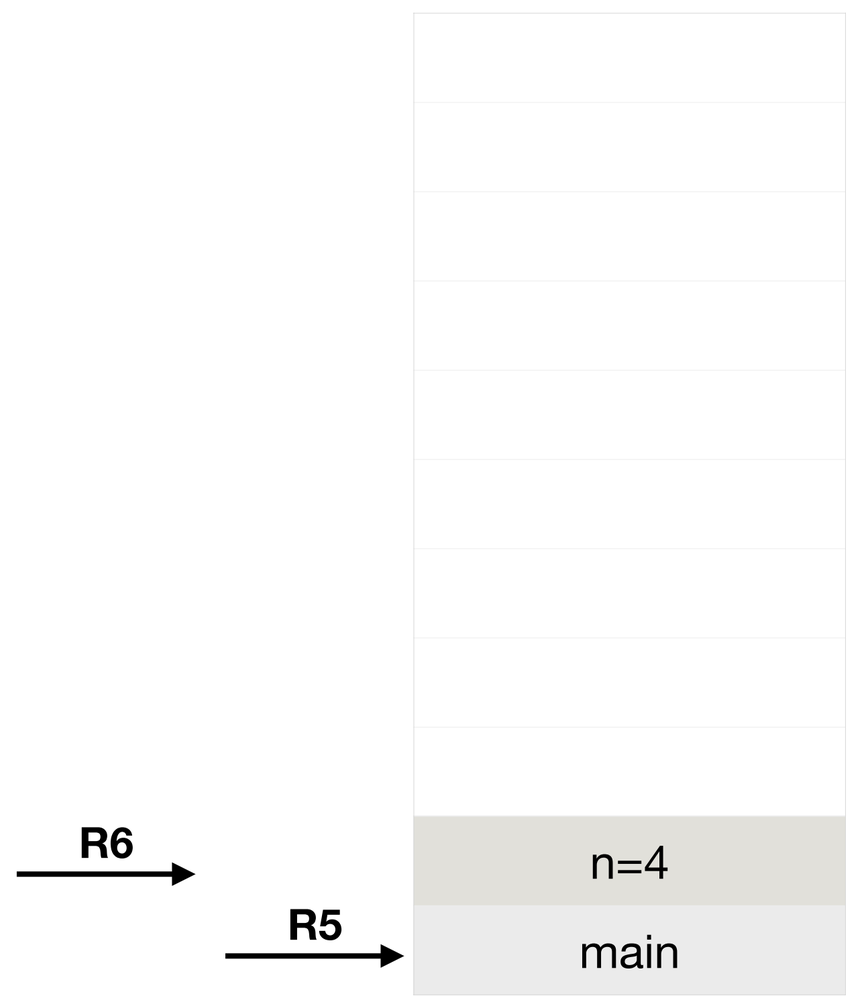


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
```



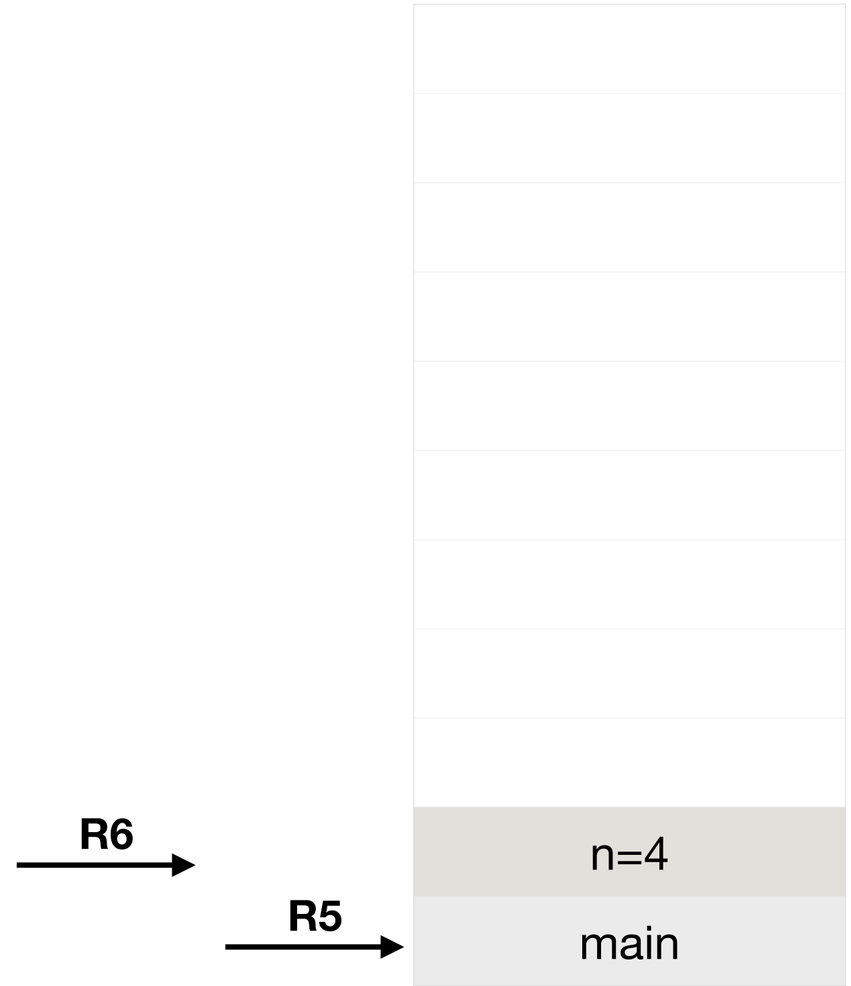
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
```

R0 | 4



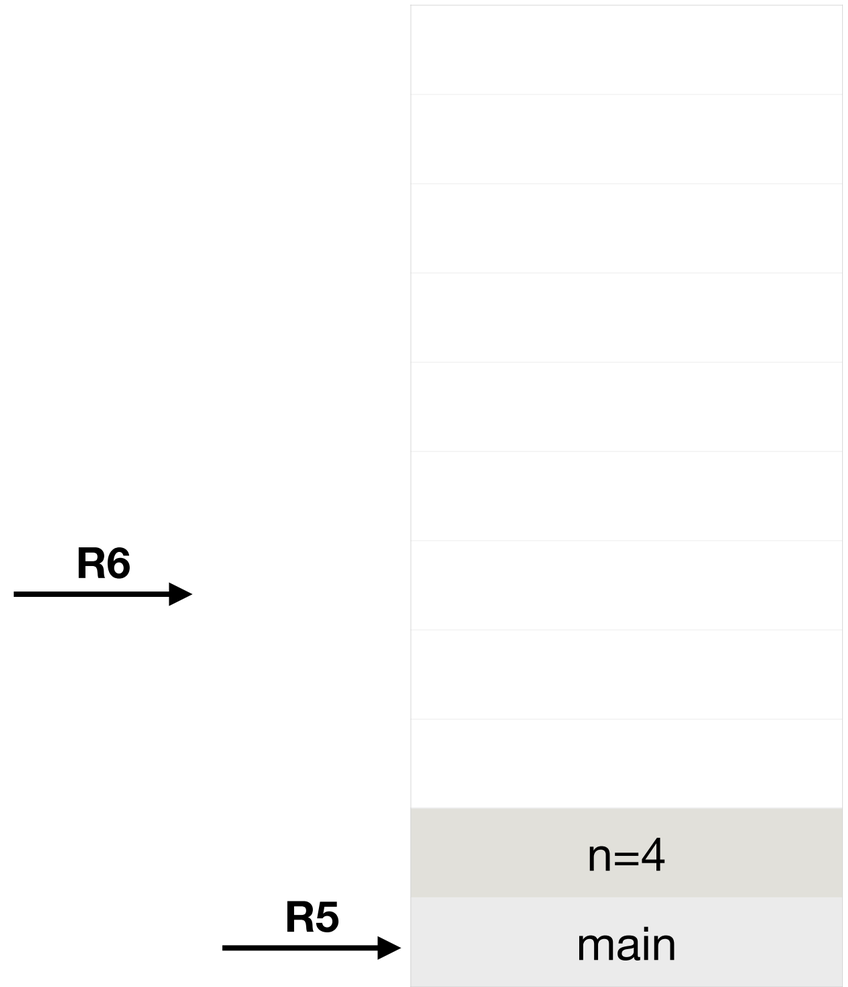
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

Recursion in LC3

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
```

R0	4



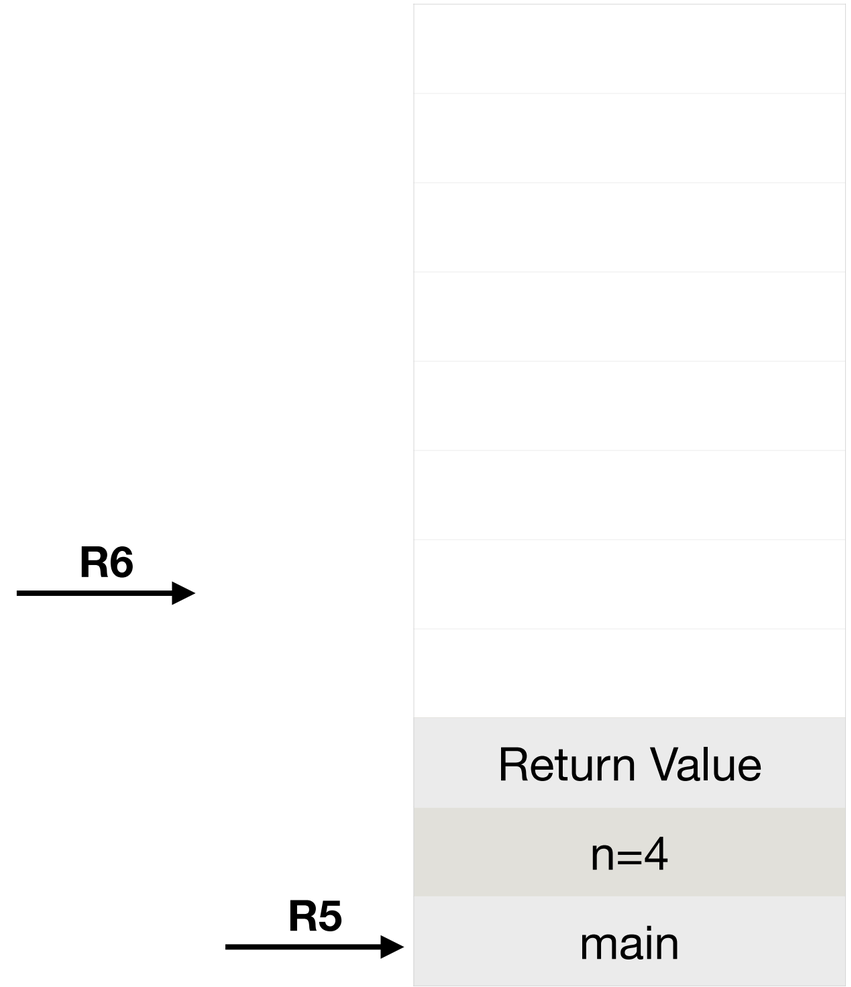
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

Recursion in LC3

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
```

R0	4



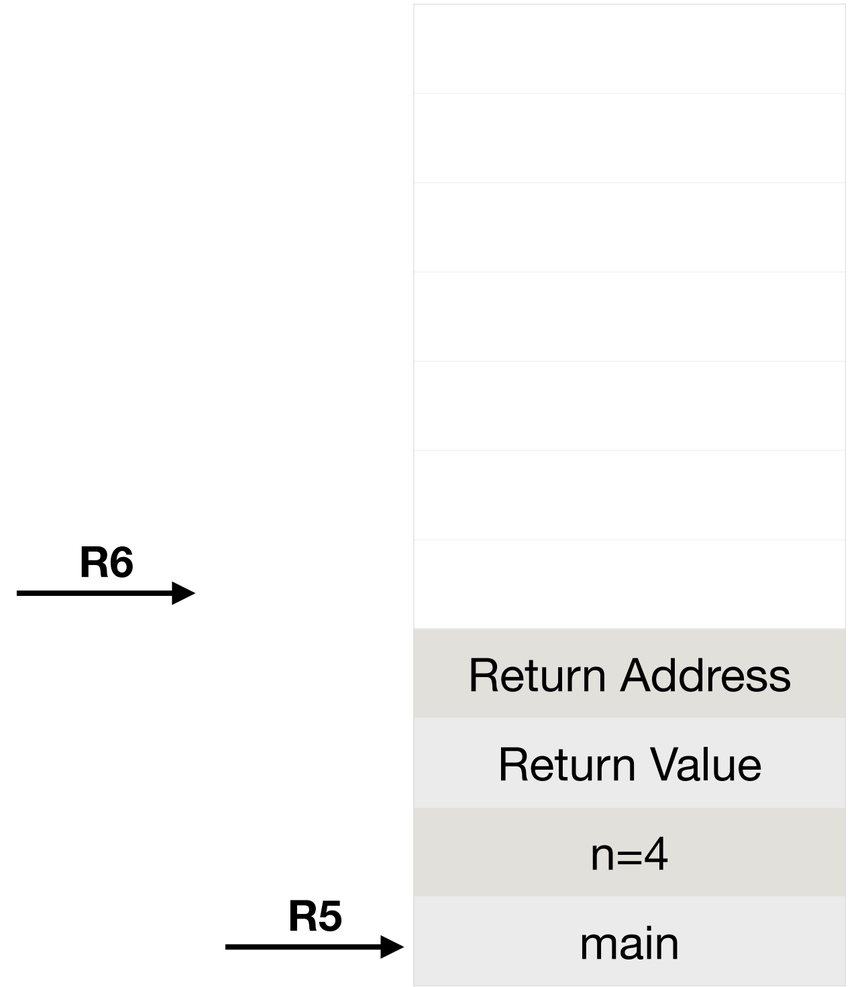
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
```

R0 | 4



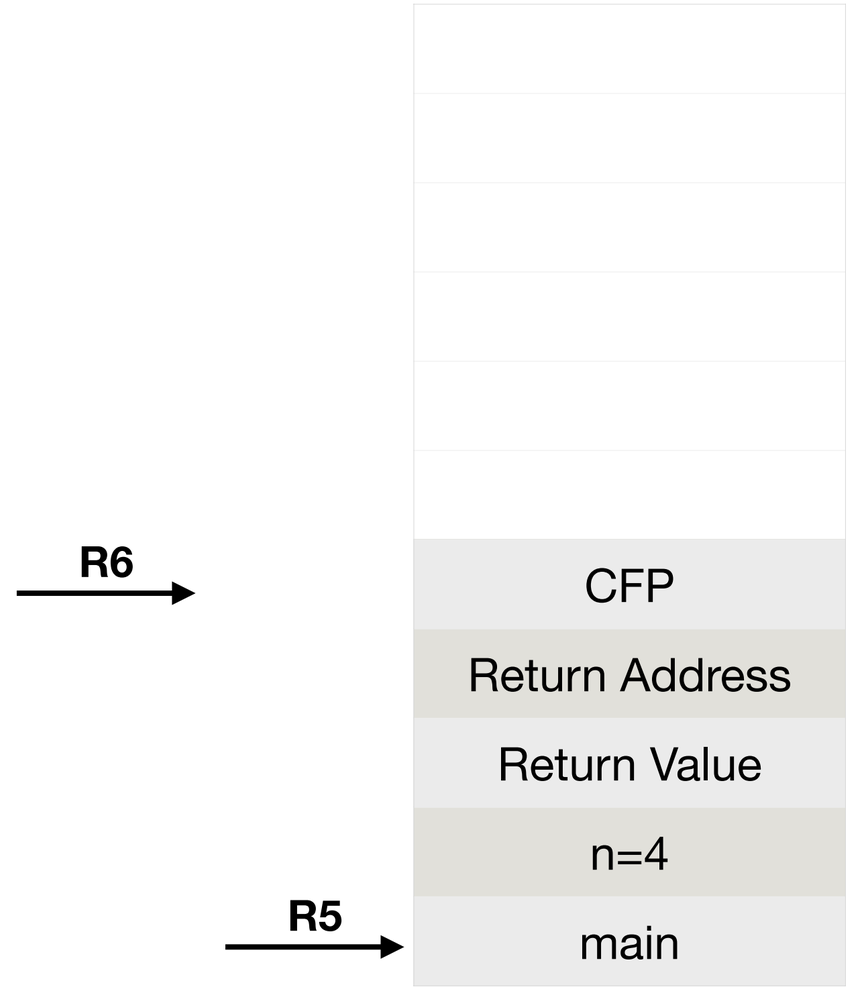
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0
```

R0	4

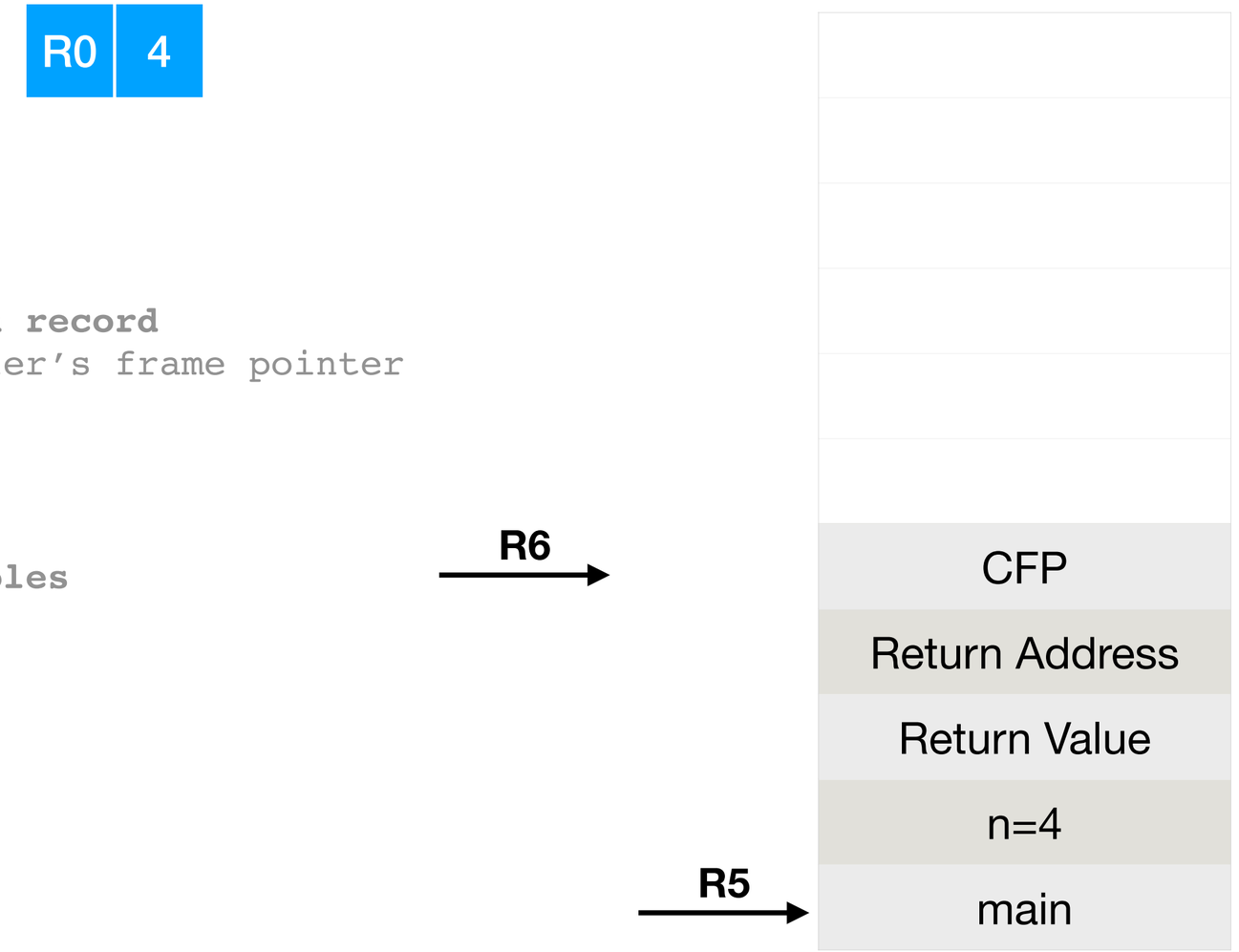


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
```

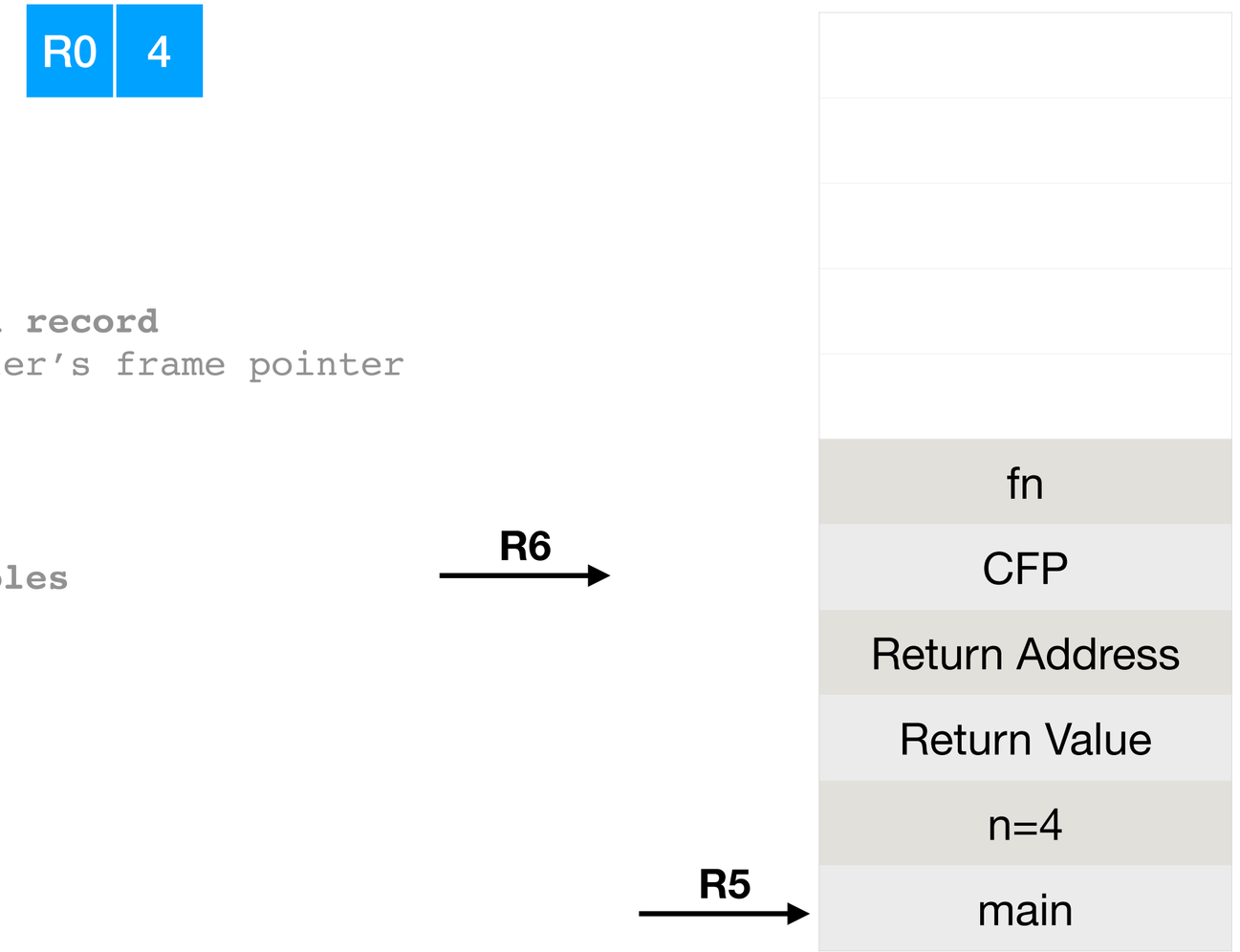


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
```



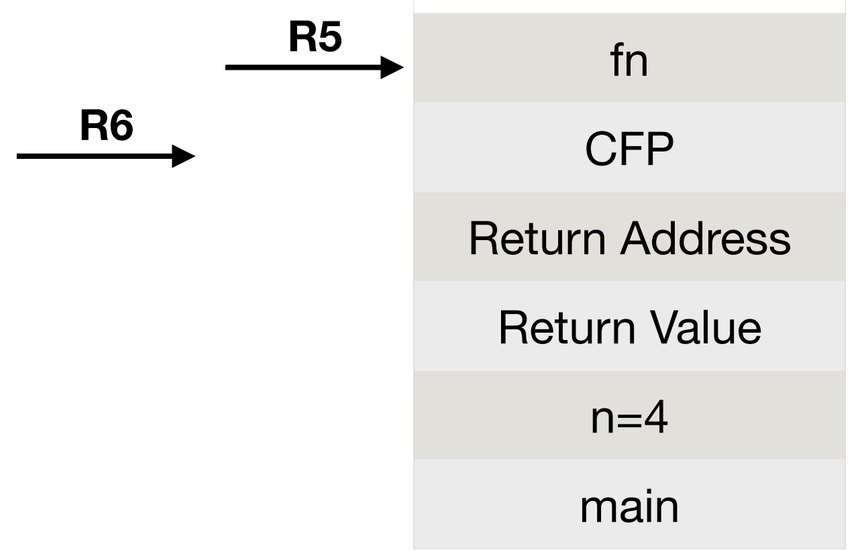
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
ADD R5, R6, #-1
```

R0	4



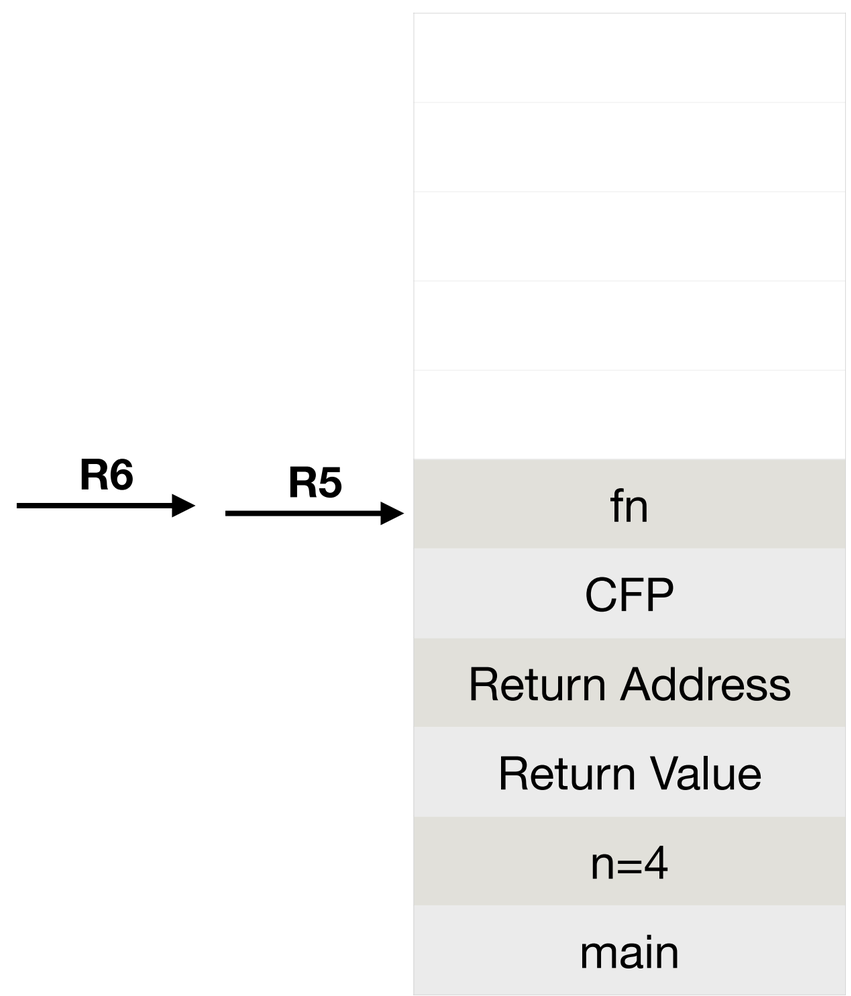
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
ADD R5, R6, #-1
ADD R6, R6, #-1
```

R0	4



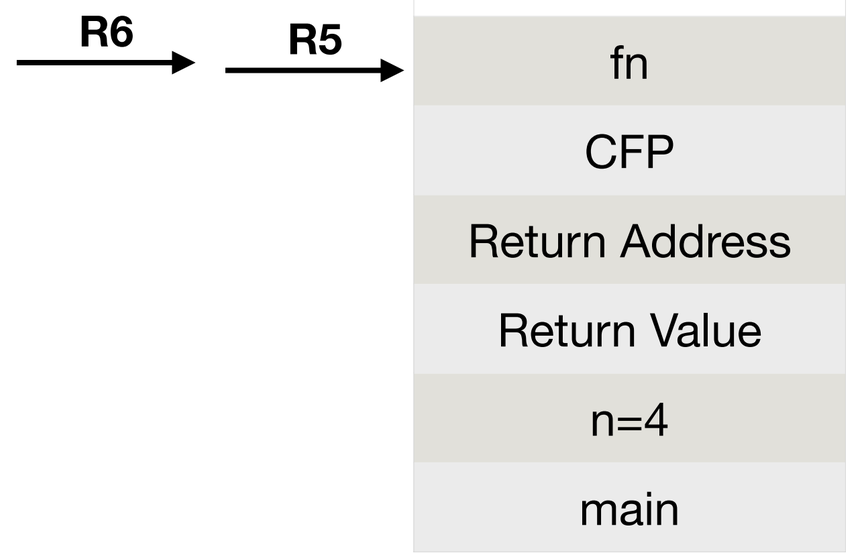
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
ADD R5, R6, #-1
ADD R6, R6, #-1

;function logic
```

R0	4



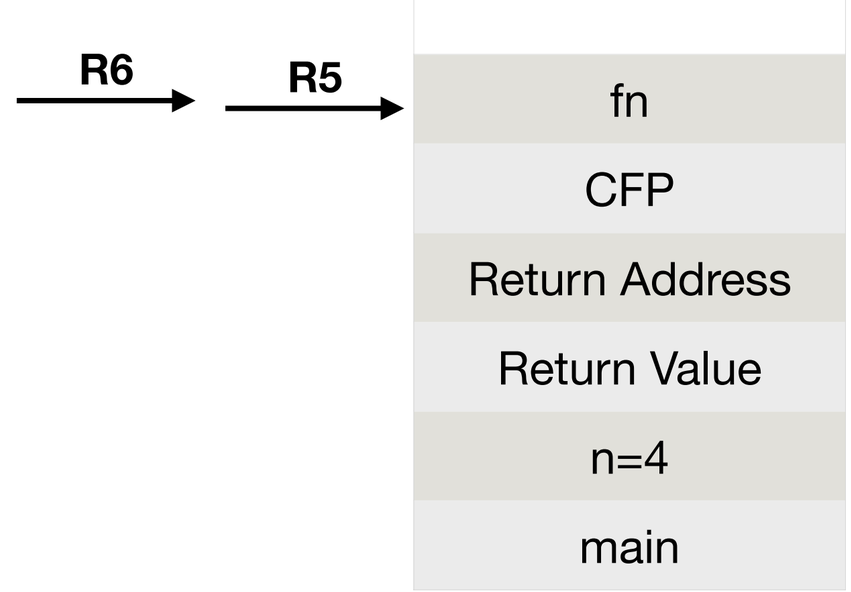
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
ADD R5, R6, #-1
ADD R6, R6, #-1

;function logic
;base case (n==1)
```

R0	4



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

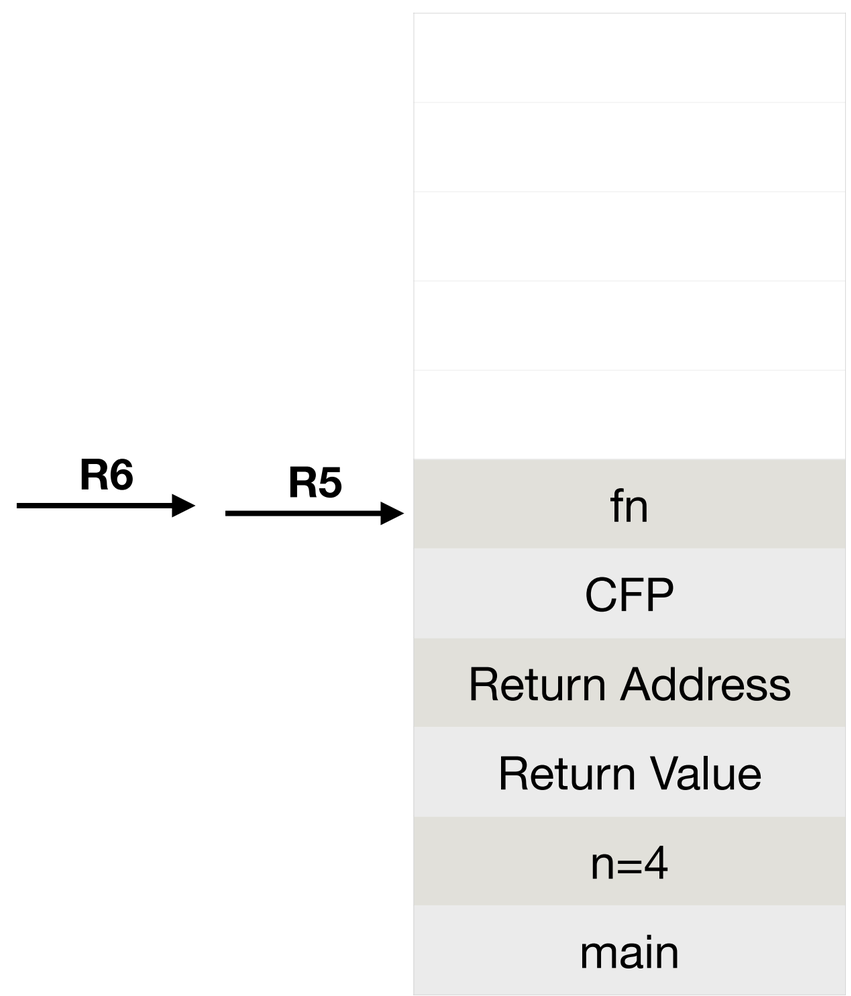
Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
ADD R5, R6, #-1
ADD R6, R6, #-1

;function logic
;base case (n==1)
LDR R1, R5, #4
```

R0	4

R1	4



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

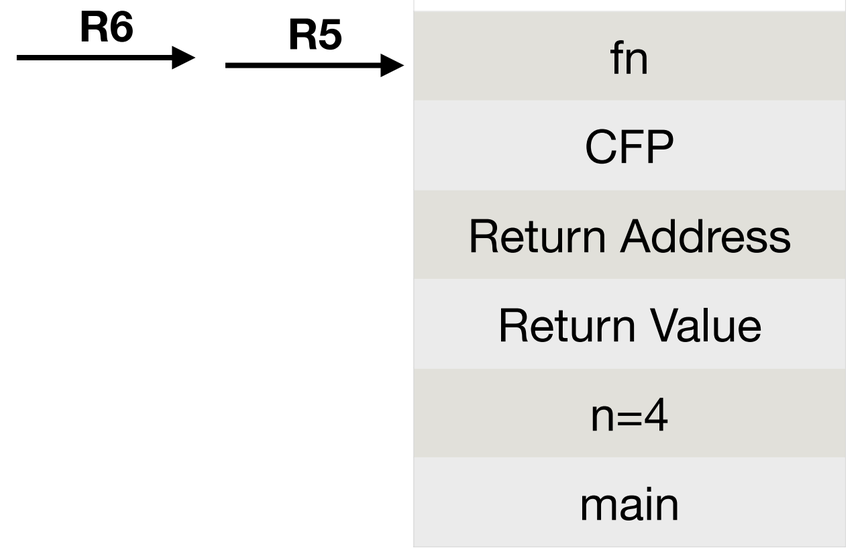
```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
ADD R5, R6, #-1
ADD R6, R6, #-1

;function logic
;base case (n==1)
LDR R1, R5, #4
ADD R2, R1, #-1
```

R0	4

R1	4

R2	3



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

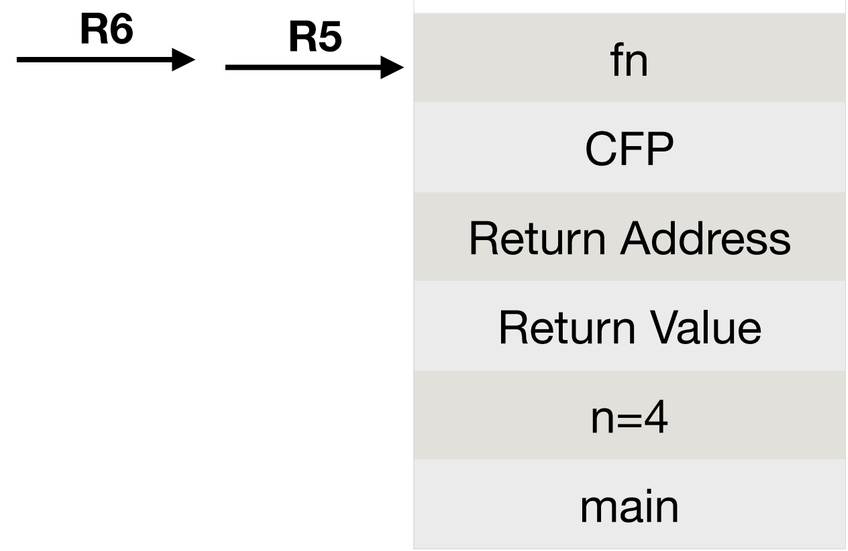
```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
;Caller set-up
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
JSR RUNNING

RUNNING
;callee set-up of Running(n)'s activation record
;push return value, return address & caller's frame pointer
ADD R6, R6, #-3
STR R7, R6, #1
STR R5, R6, #0

;update frame pointer & push local variables
ADD R5, R6, #-1
ADD R6, R6, #-1

;function logic
;base case (n==1)
LDR R1, R5, #4
ADD R2, R1, #-1
BRz BASE_CASE
```

R0	4

R1	4

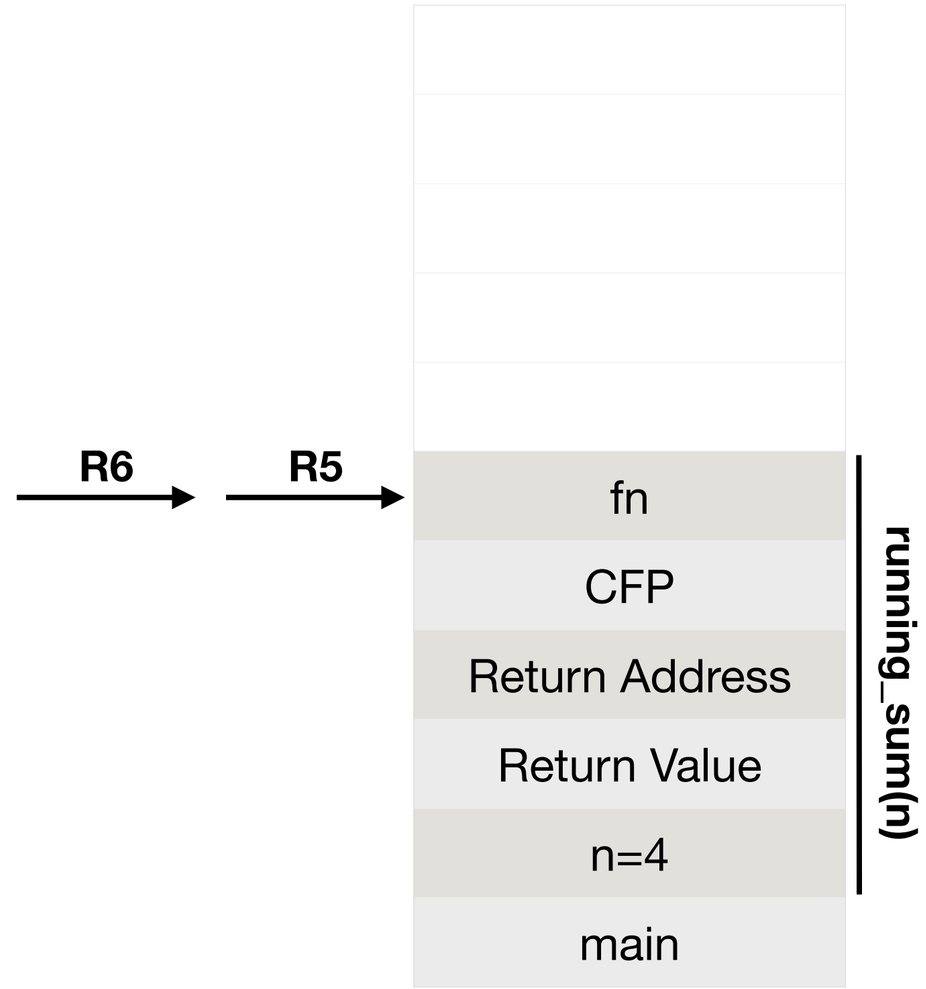
R2	3



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```



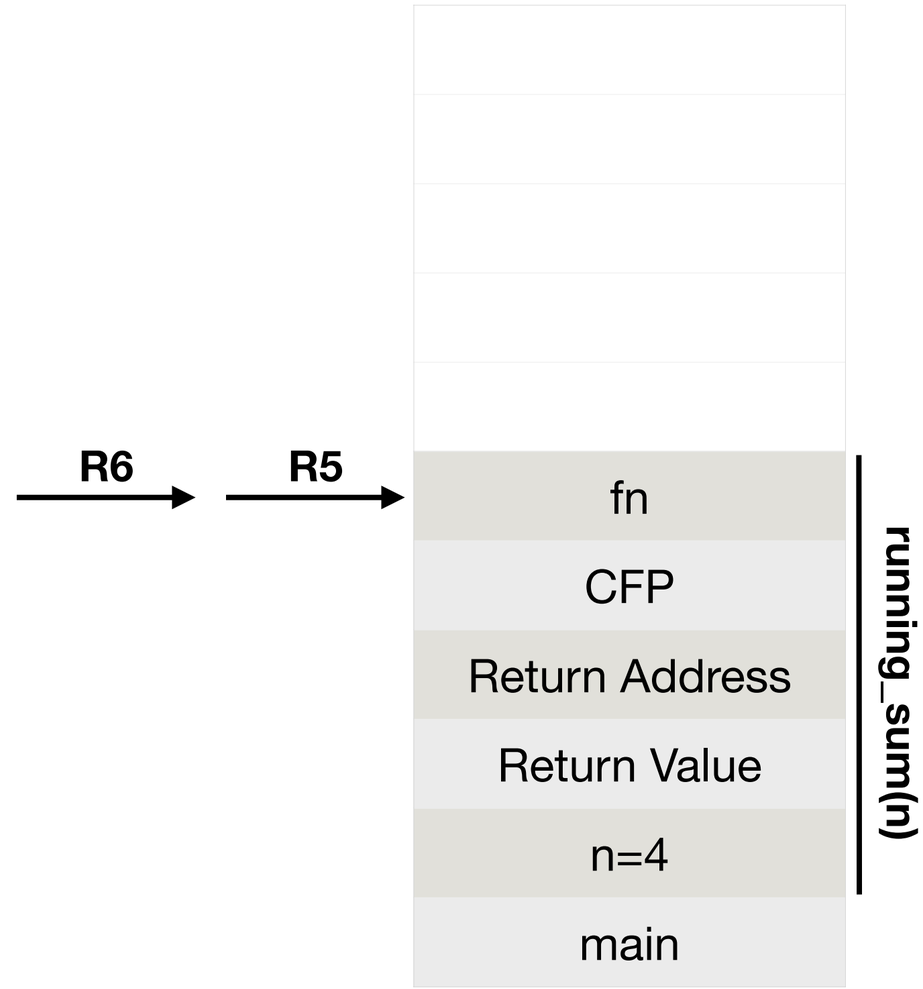
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
```



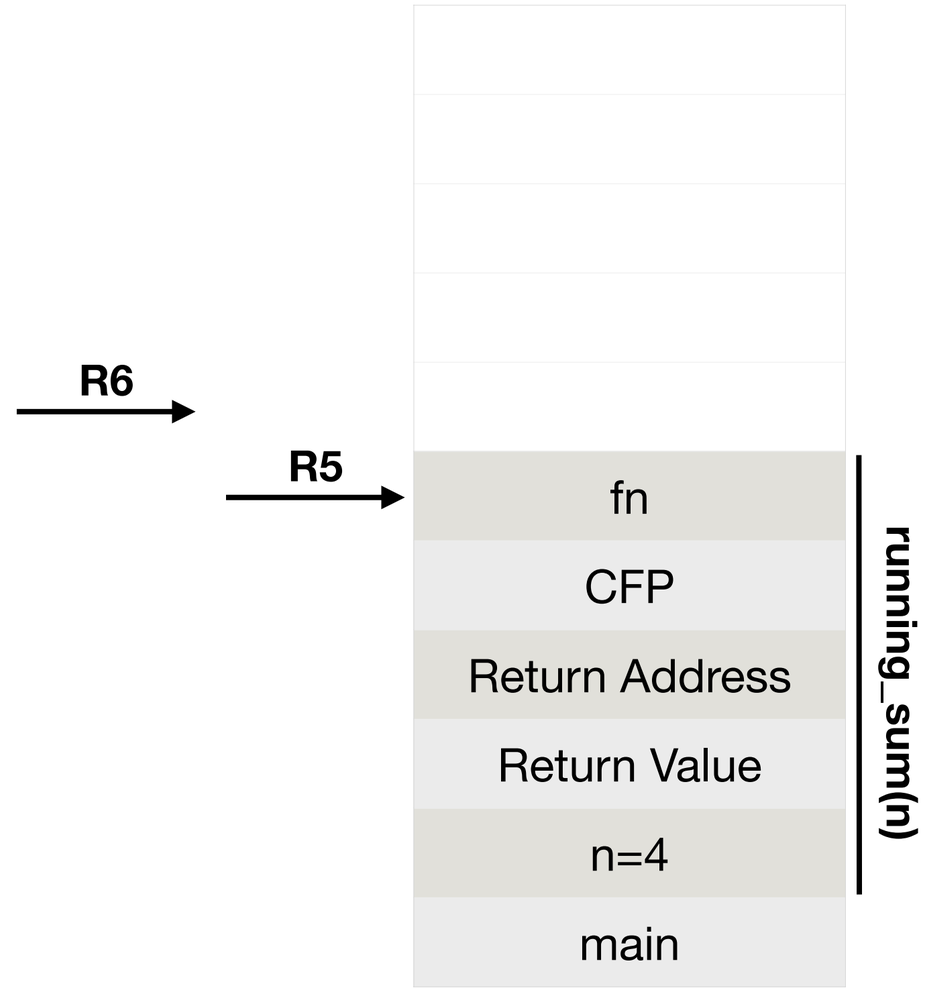
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
```



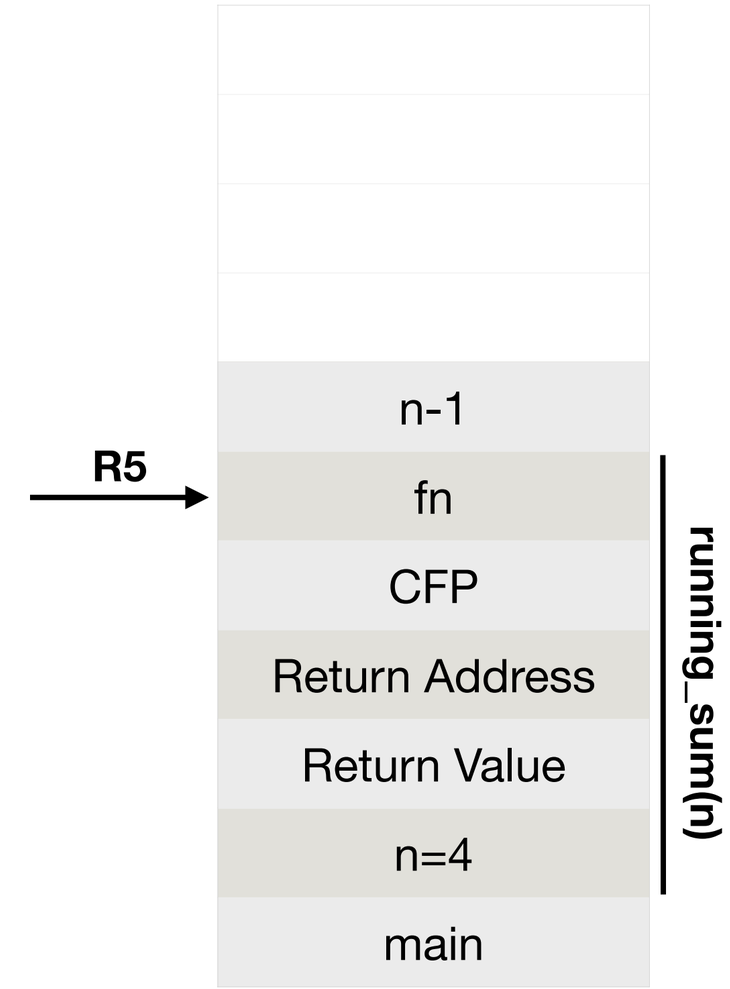
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
```



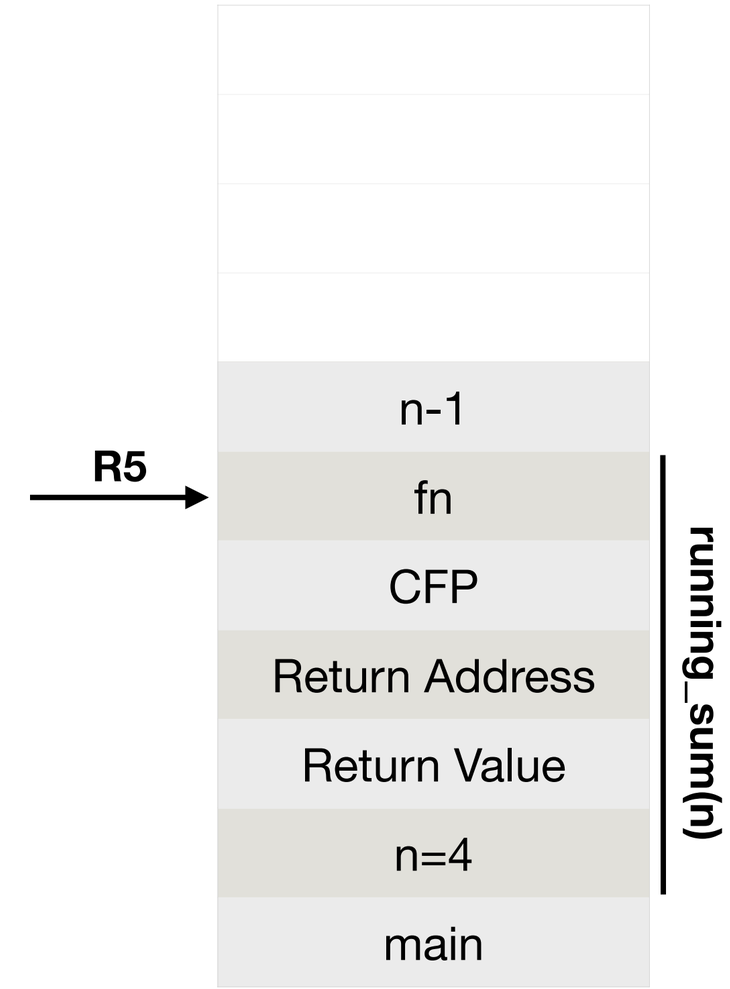
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```



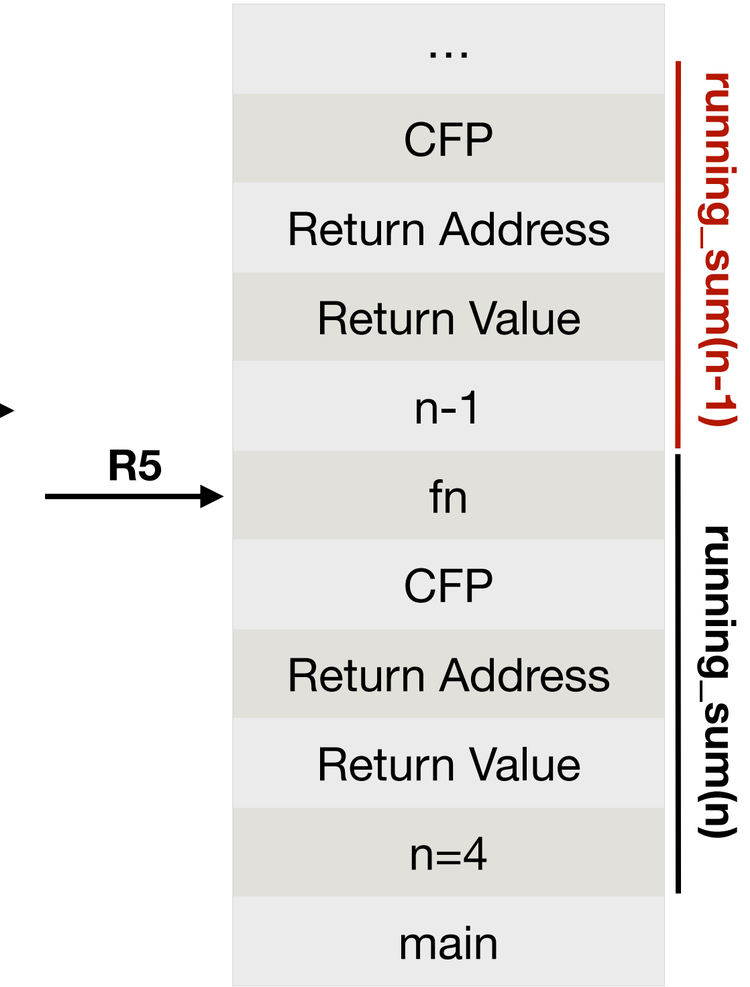
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```



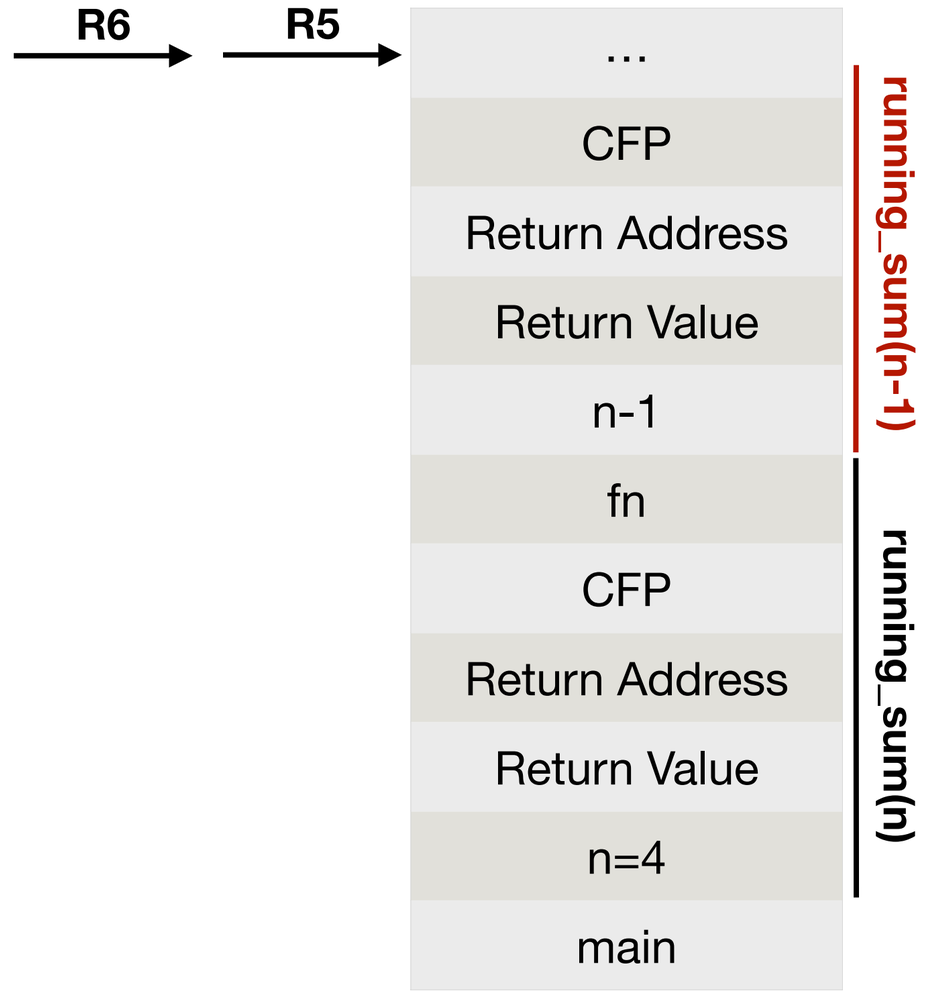
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```



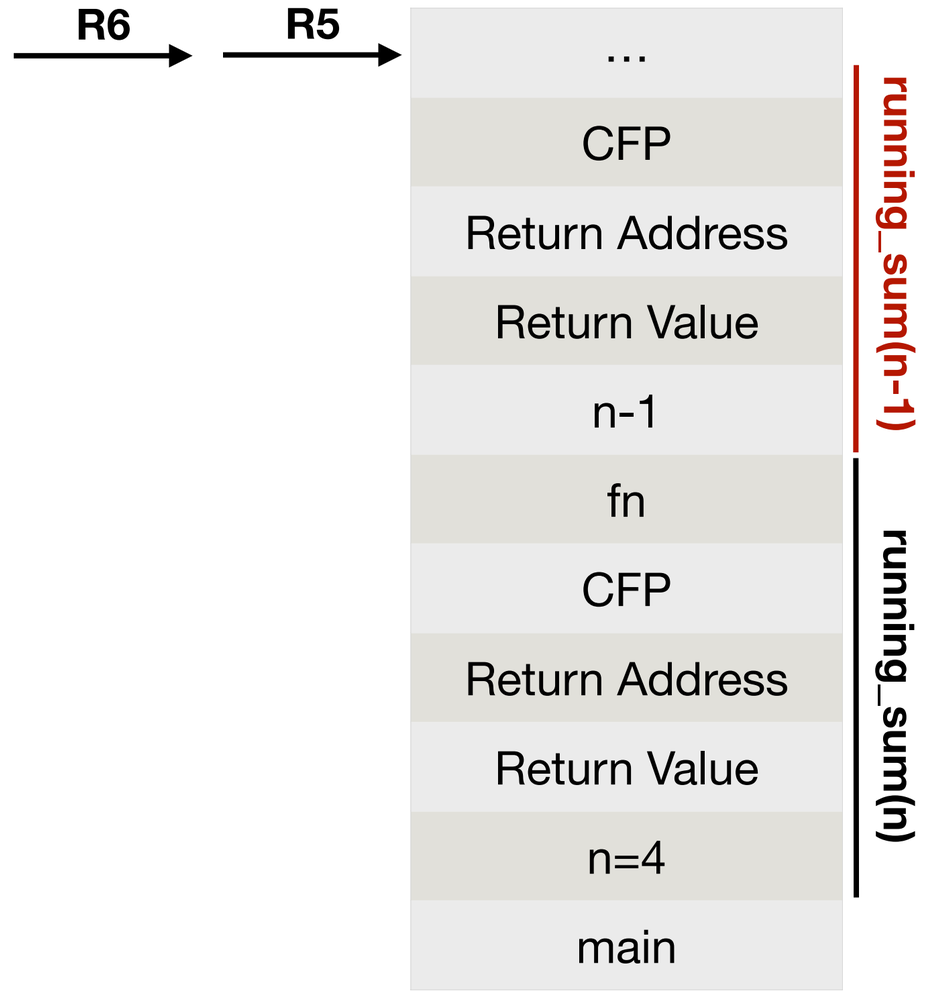
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```



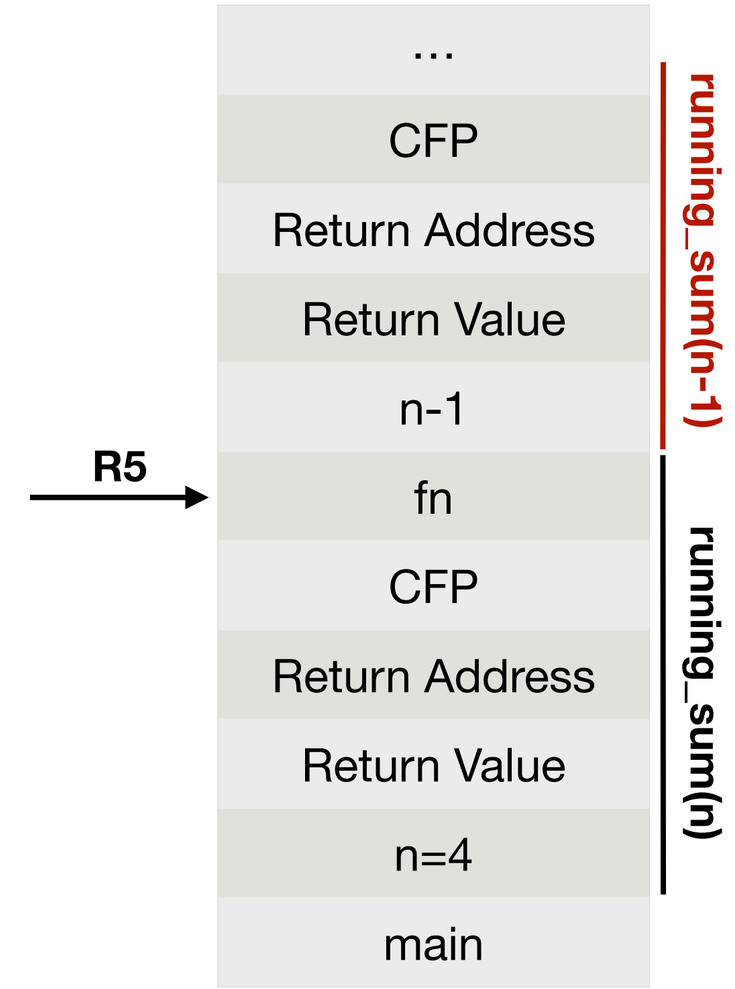
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```



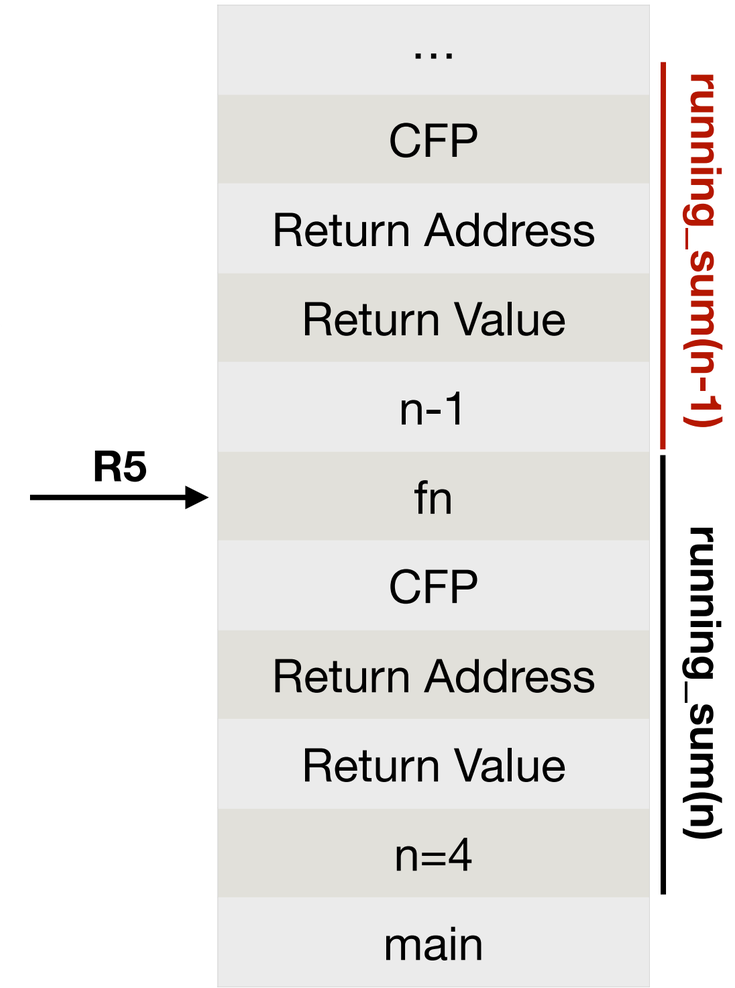
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)



;Caller tear-down for Running(n-1)
```



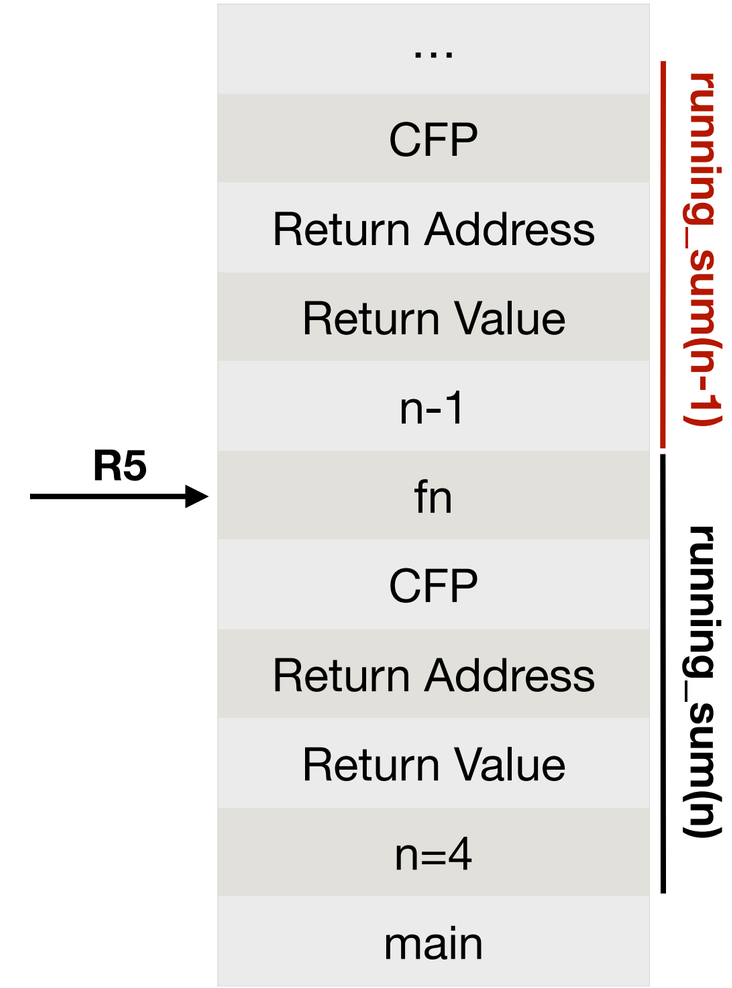
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)



;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
```



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

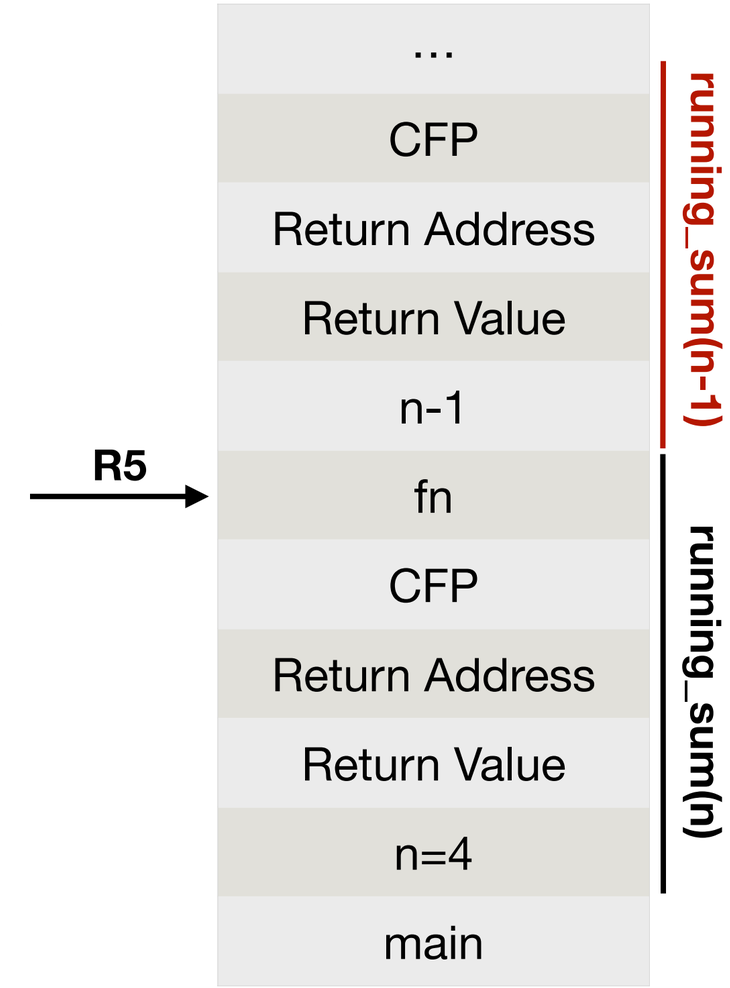
Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)



;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
```

R0 | 6



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

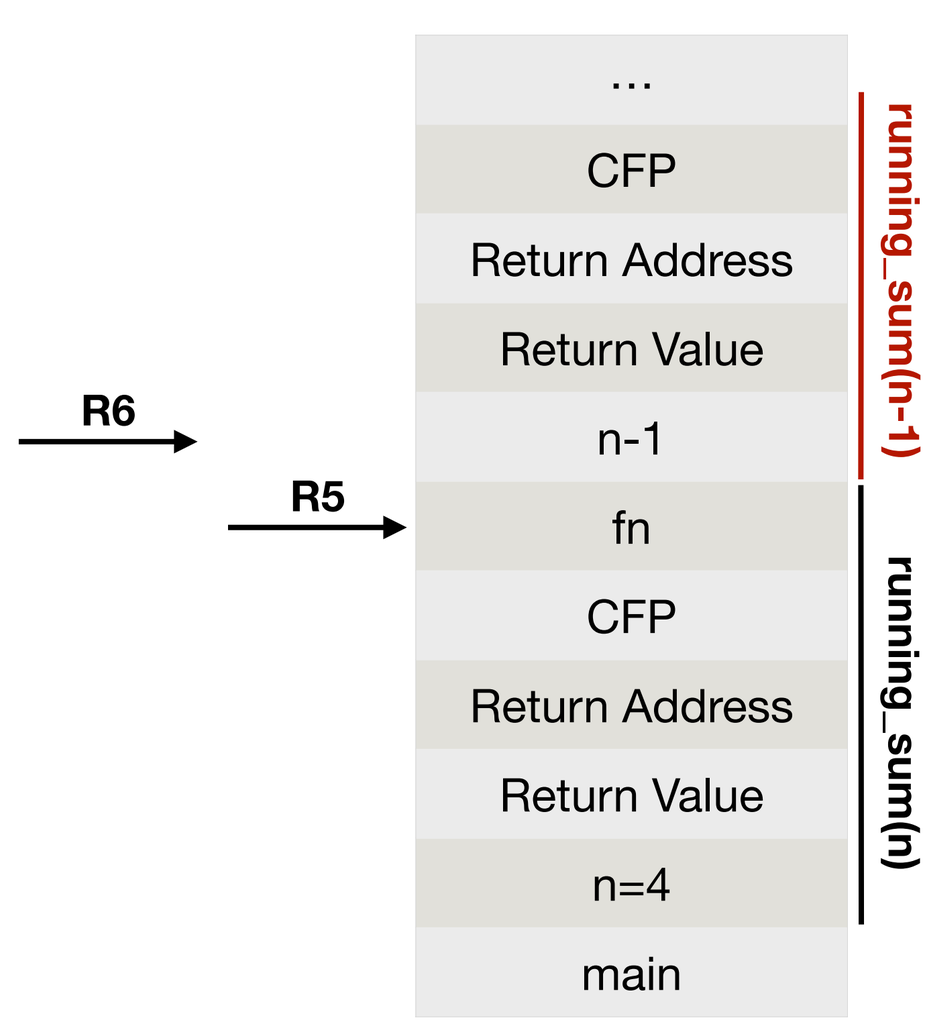
```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

R0 | 6

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```

```
;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1
```



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

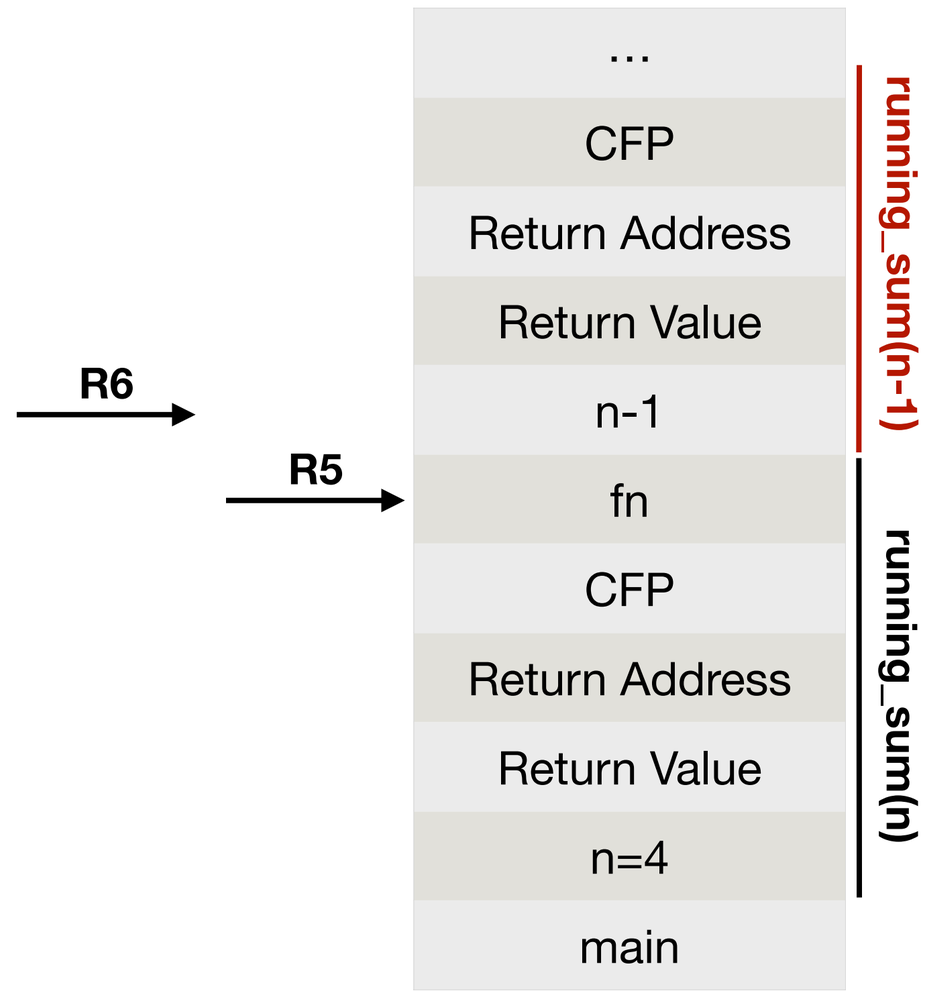
Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

R0 | 6

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)


;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
```



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

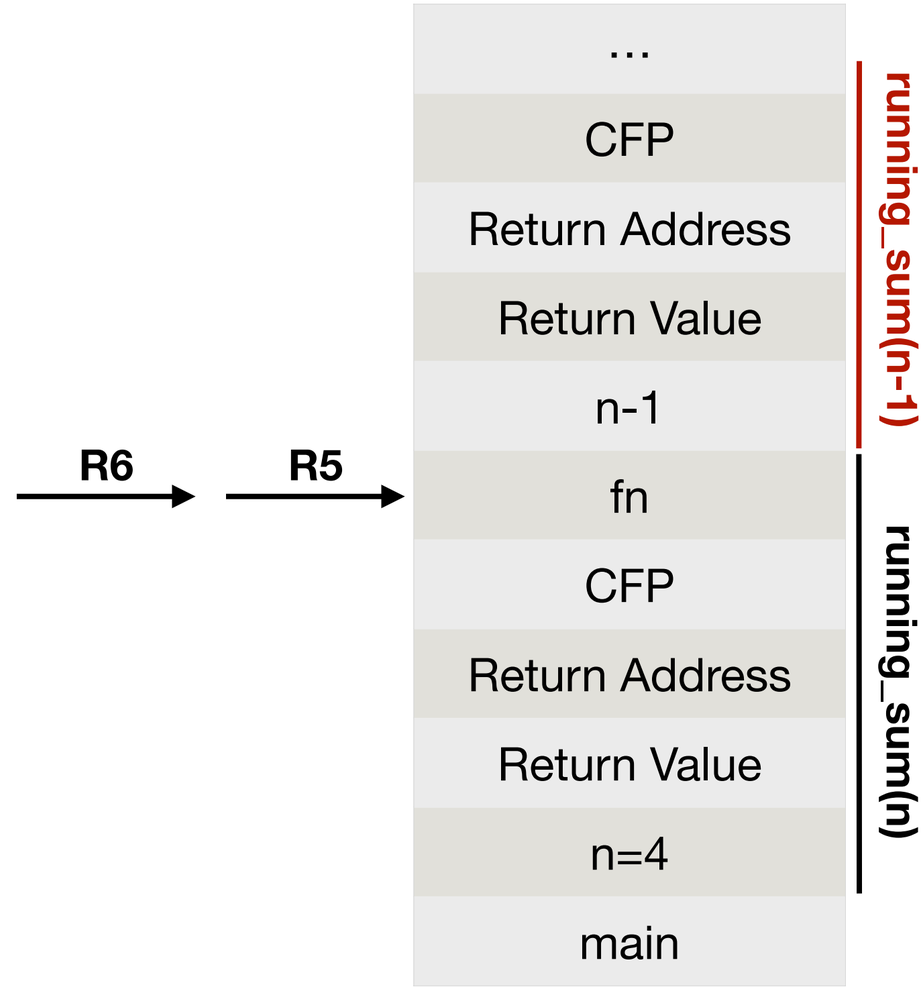
Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

R0 | 6

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)



;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1
```



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

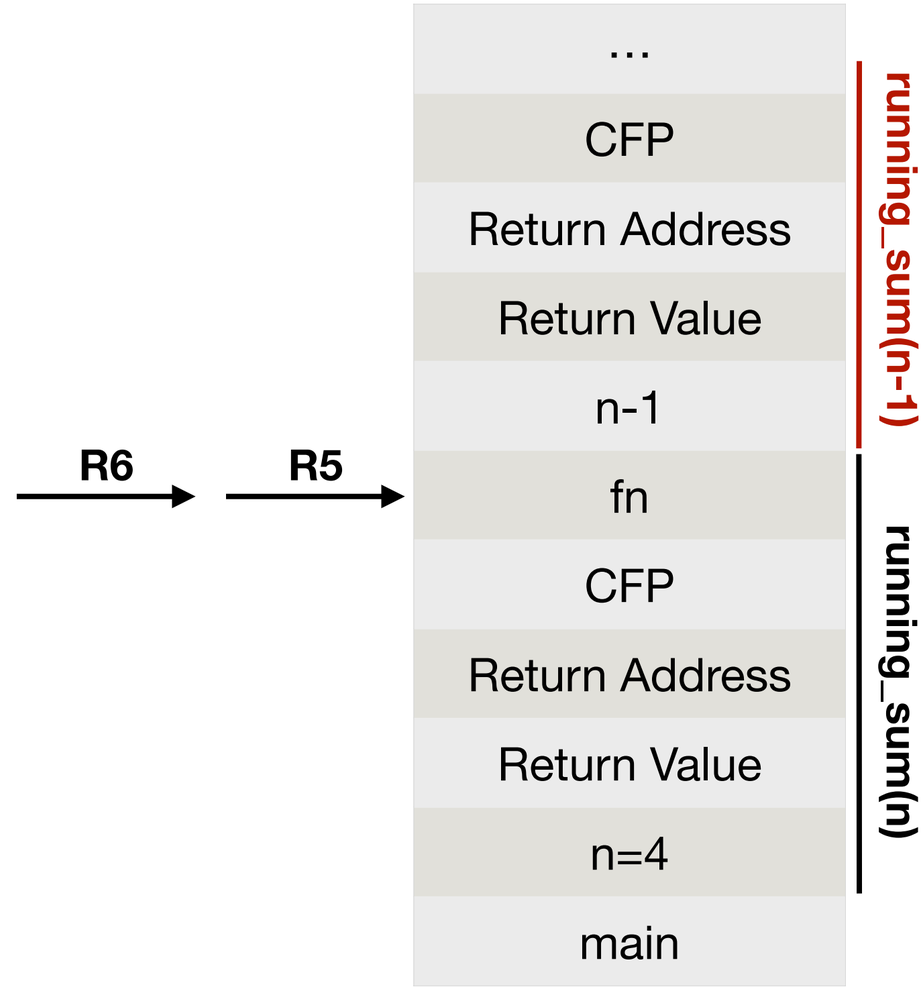
Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

R0 | 6

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)



;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1

;calculate n + Running(n-1)
```



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

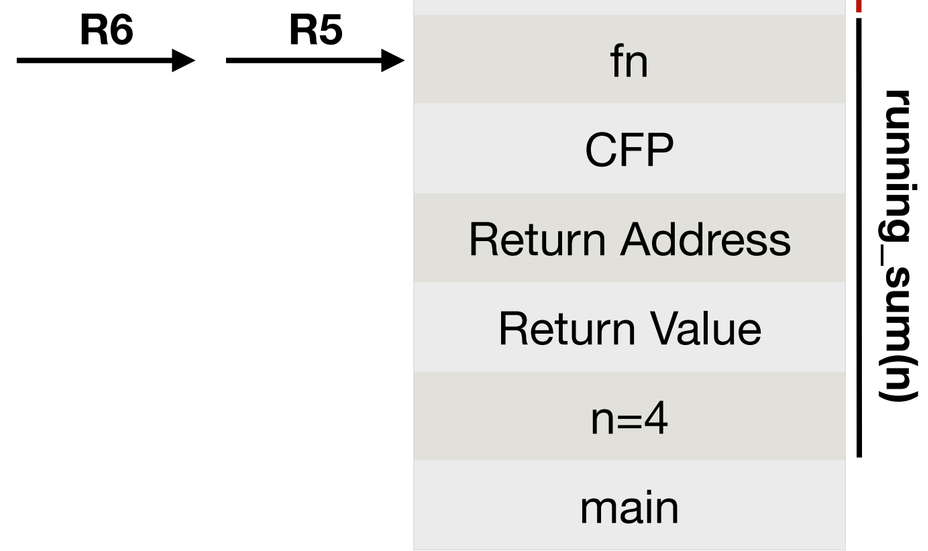
R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```

R0 | 6

R1 | 4

```
;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1

;calculate n + Running(n-1)
LDR R1, R5, #4
```



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

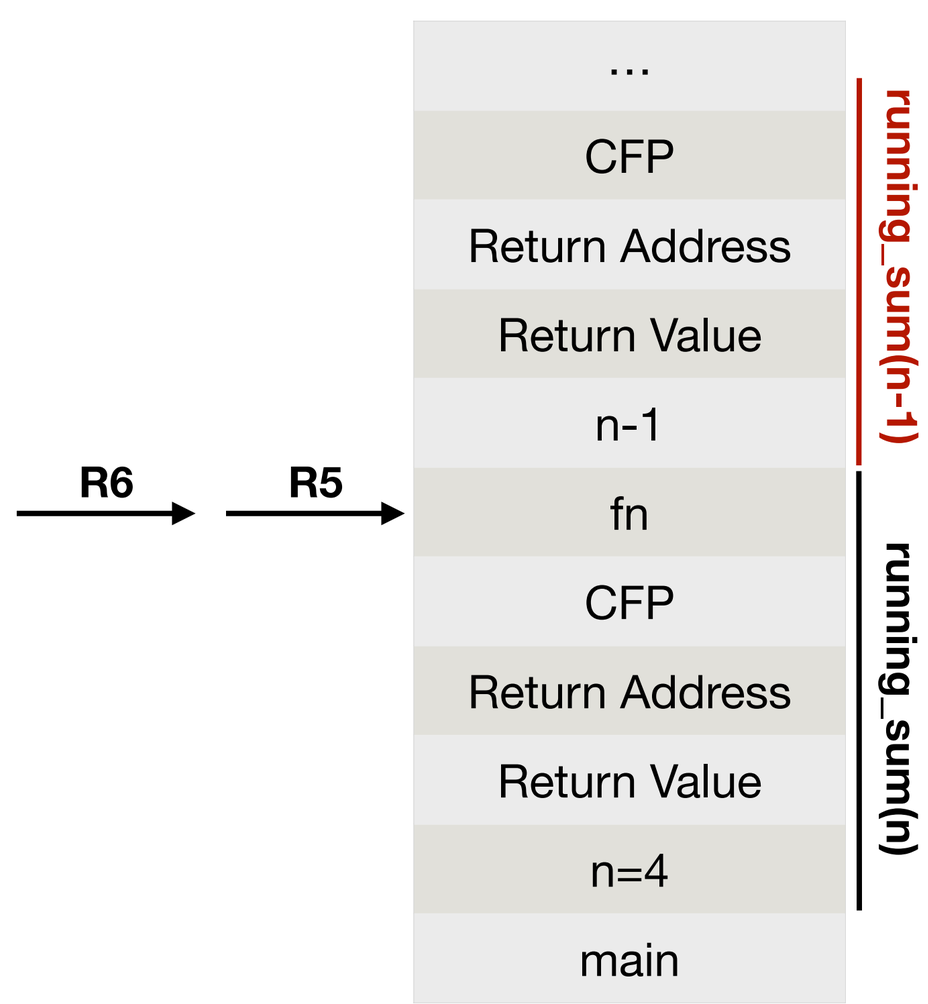
R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```

R0 | 10

R1 | 4

```
;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1

;calculate n + Running(n-1)
LDR R1, R5, #4
ADD R0, R1, R0
```



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

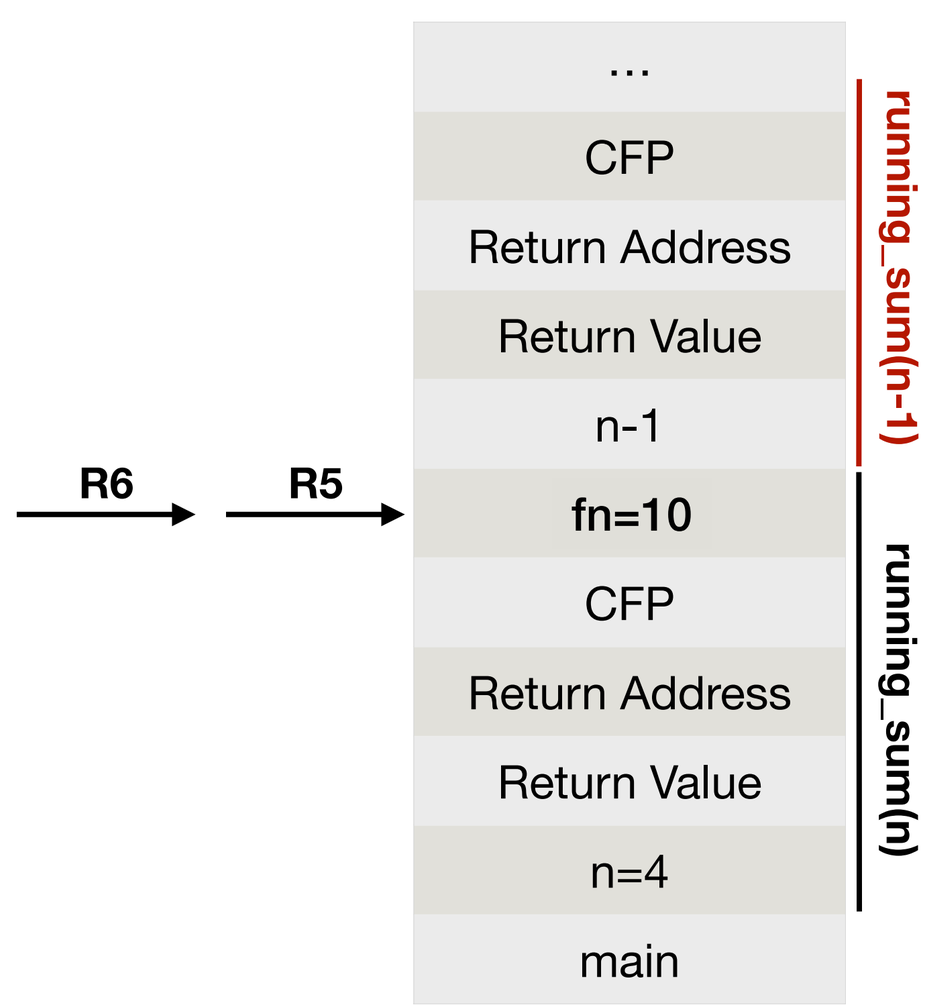
R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```

R0 | 10

R1 | 4

```
;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1

;calculate n + Running(n-1)
LDR R1, R5, #4
ADD R0, R1, R0
STR R0, R5, #0 ;store result in fn
```



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

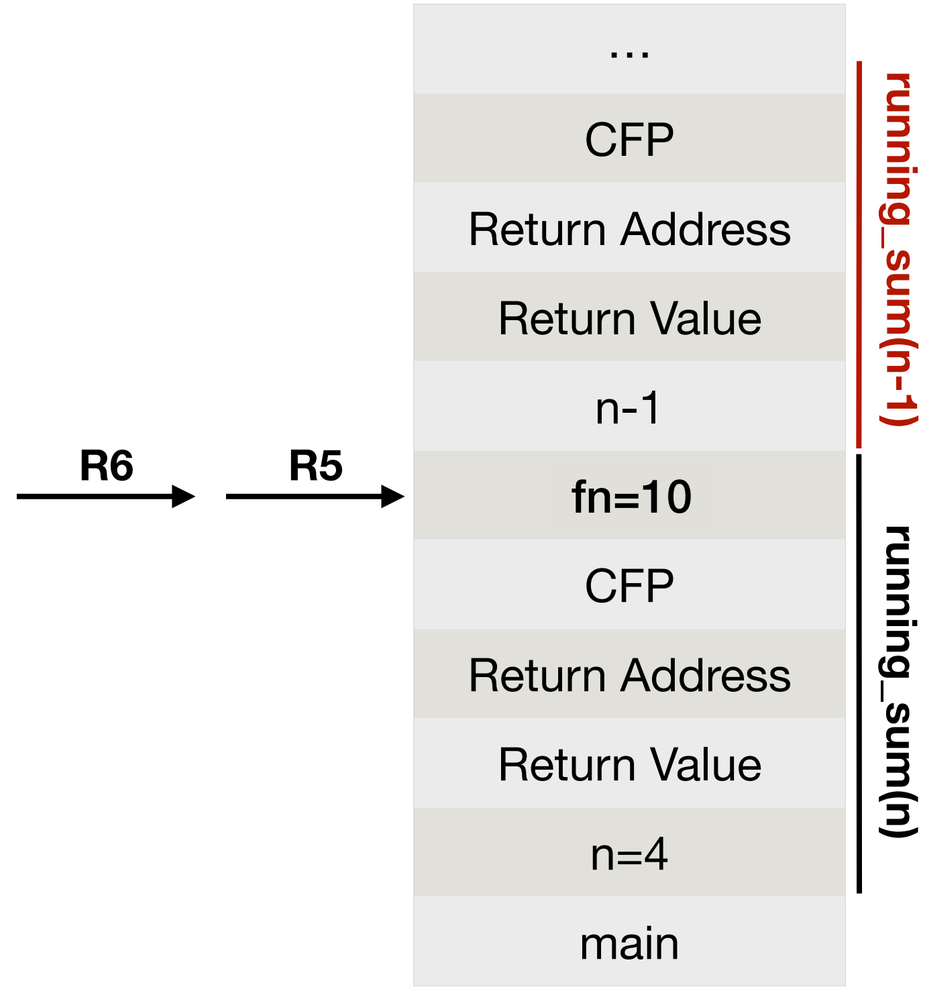
R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)
```

R0 | 10

R1 | 4

```
;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1

;calculate n + Running(n-1)
LDR R1, R5, #4
ADD R0, R1, R0
STR R0, R5, #0 ;store result in fn

;ready to return
```



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

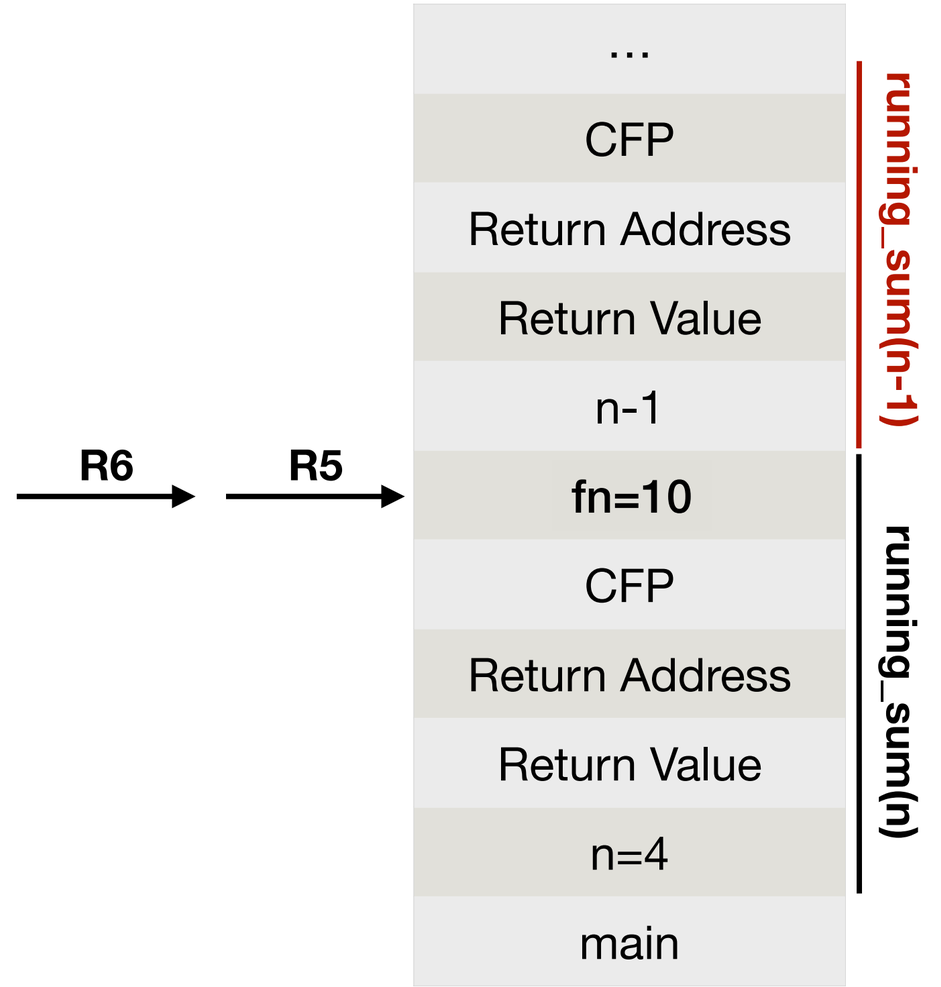
```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

R2 | 3

```
;Recursive case
;Caller setup: push argument n-1 onto RTS
ADD R6, R6, #-1
STR R2, R6, #0 ; R2 = n - 1
JSR RUNNING ; call Running(n-1)



;Caller tear-down for Running(n-1)
;pop Running(n-1)'s return value to R0
LDR R0, R6, #0
ADD R6, R6, #1

;pop Running(n-1)'s argument
ADD R6, R6, #1

;calculate n + Running(n-1)
LDR R1, R5, #4
ADD R0, R1, R0
STR R0, R5, #0 ;store result in fn

;ready to return
BRnzp RETURN
```

R0 | 10

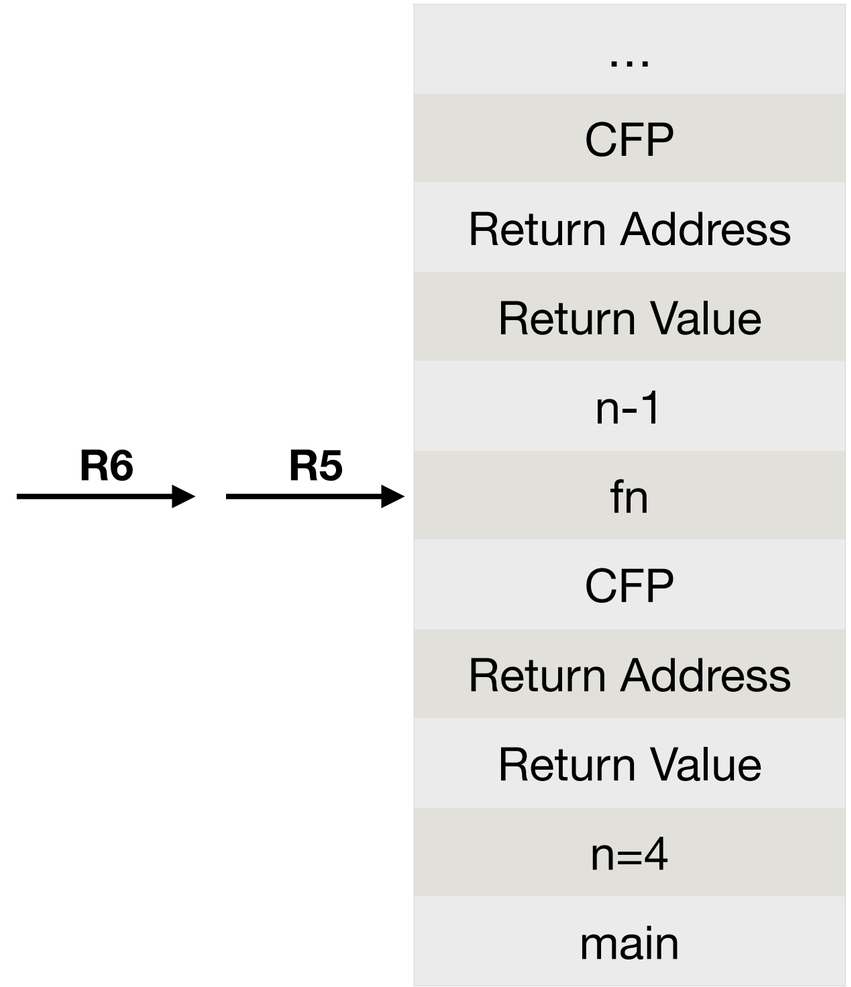
R1 | 4



Recursion in LC3

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

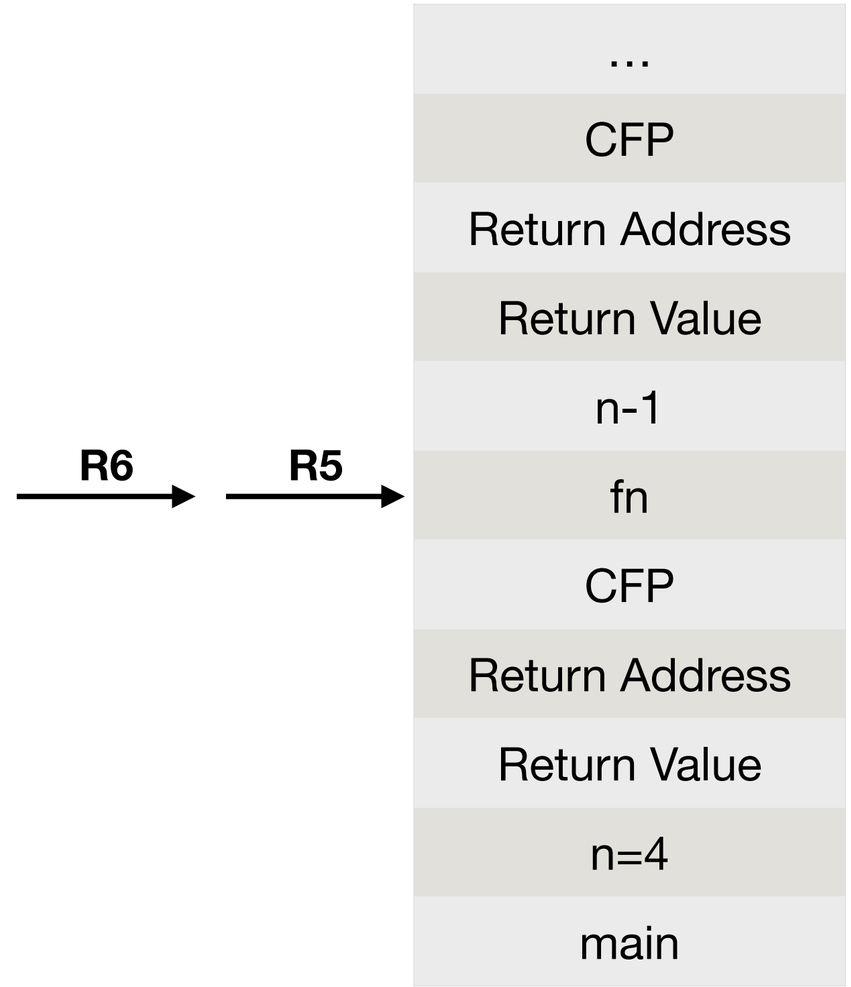


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
```

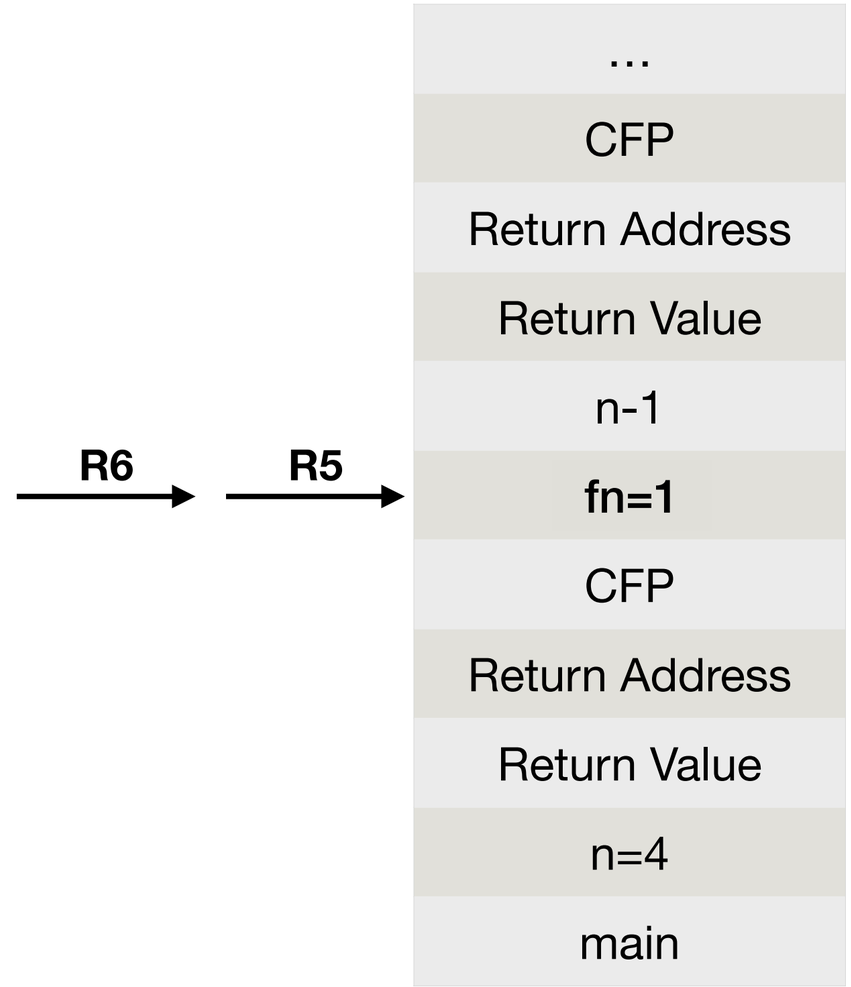


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1
```

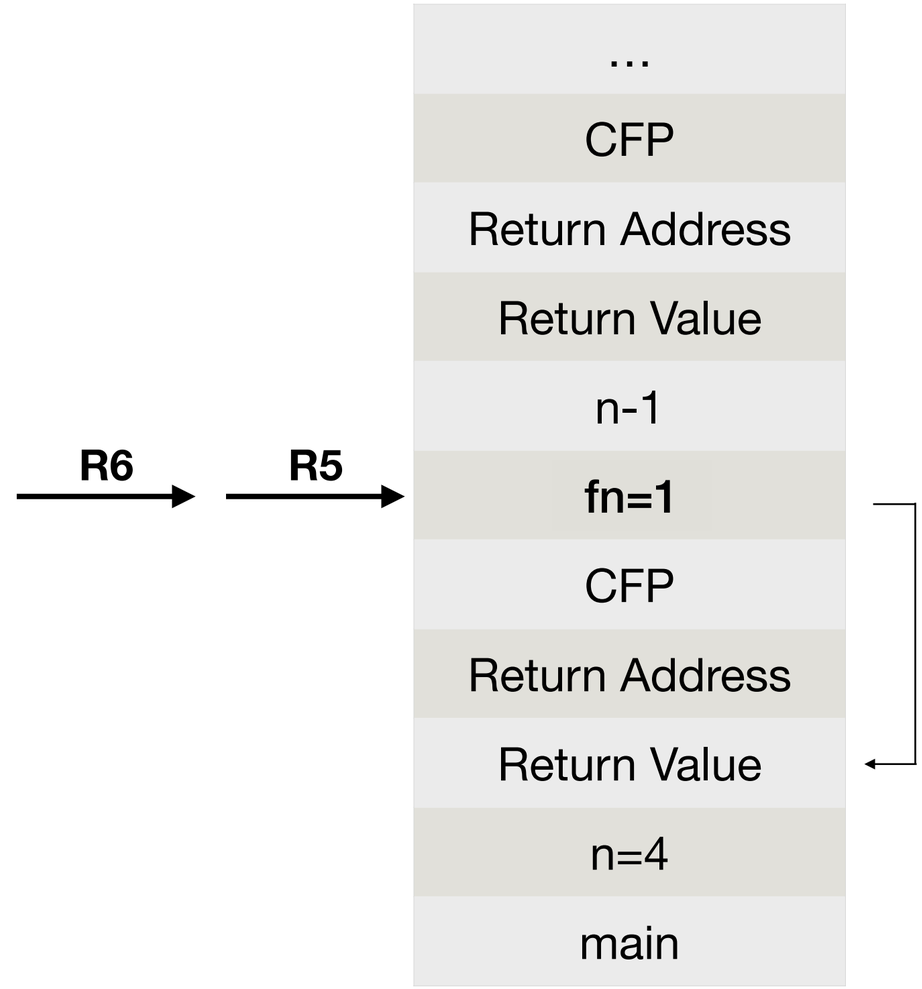


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3
```

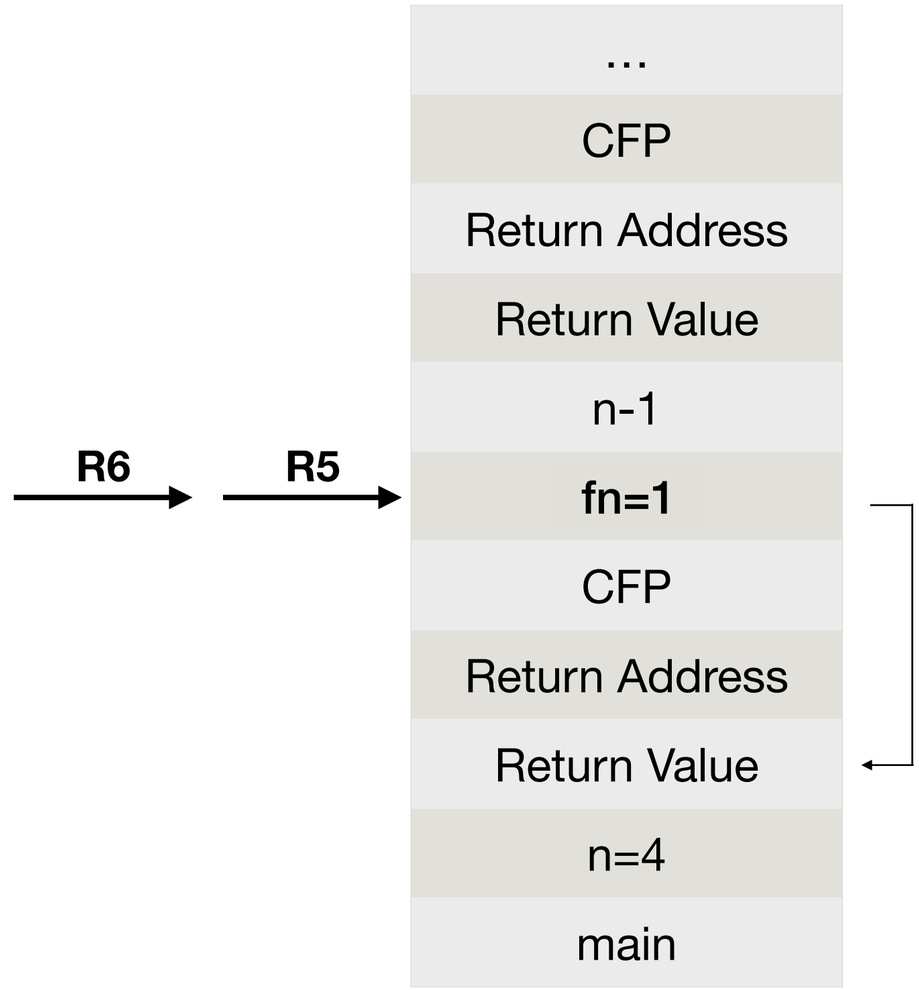


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

Recursion in LC3

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
```

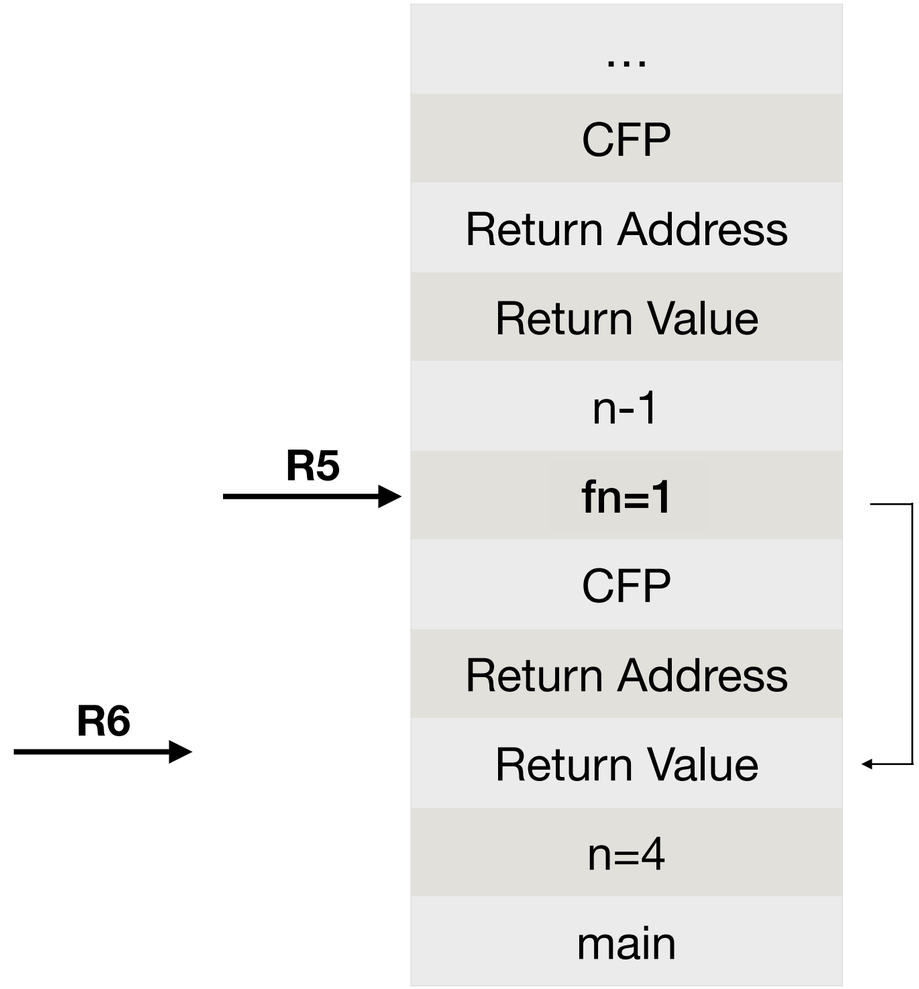


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

Recursion in LC3

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
ADD R6, R5, #3 ;pop local variables
```

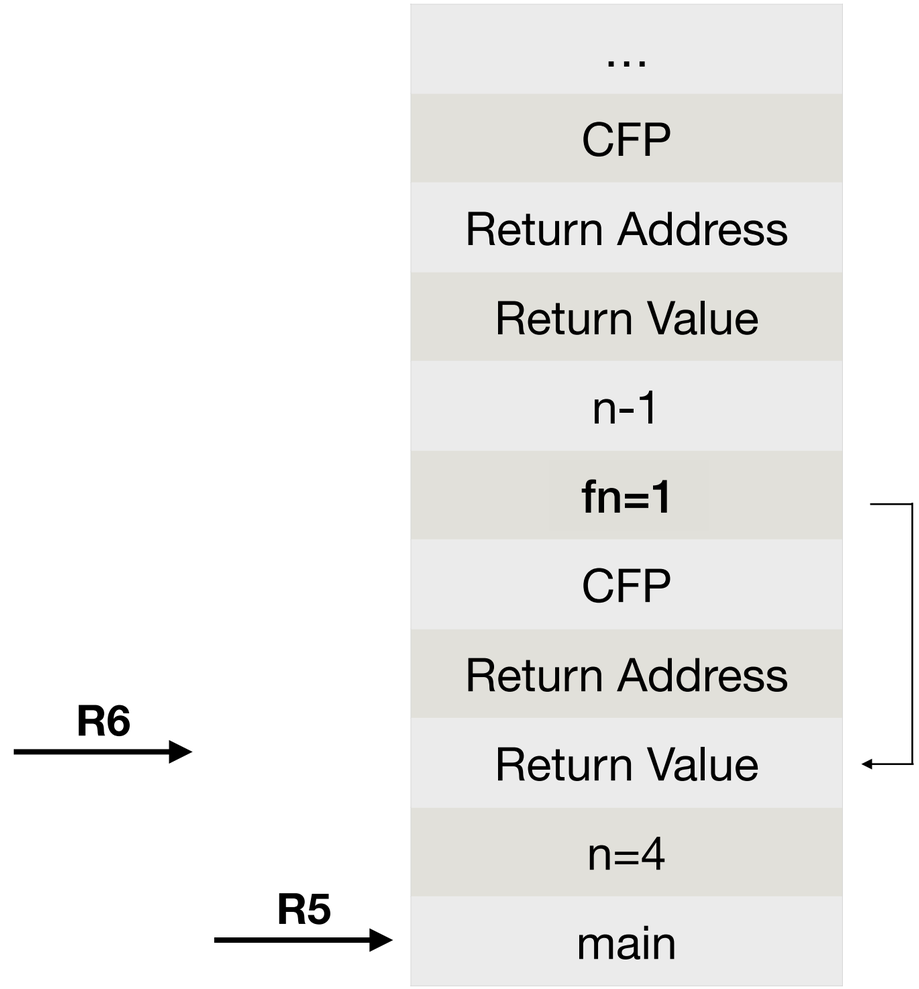


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
ADD R6, R5, #3 ;pop local variables

;restore caller's frame pointer and return address
LDR R5, R6, #-2
```

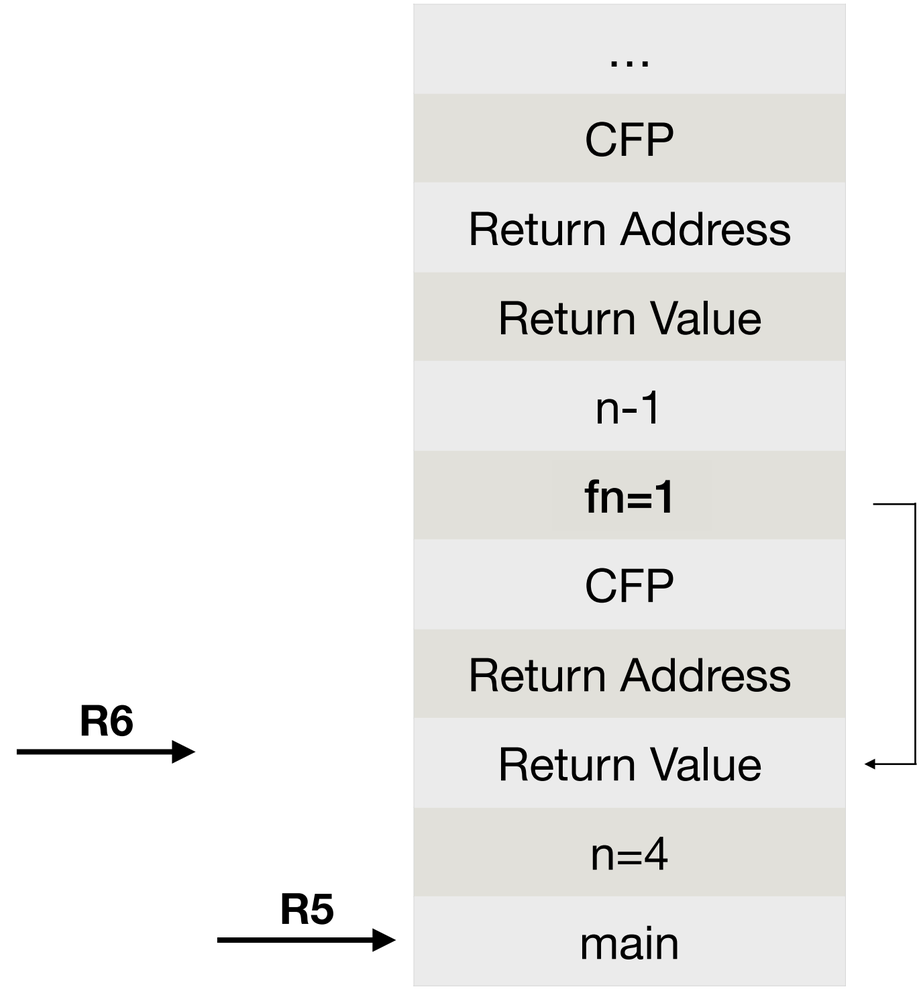


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

Recursion in LC3

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
ADD R6, R5, #3 ;pop local variables

;restore caller's frame pointer and return address
LDR R5, R6, #-2
LDR R7, R6, #-1
```

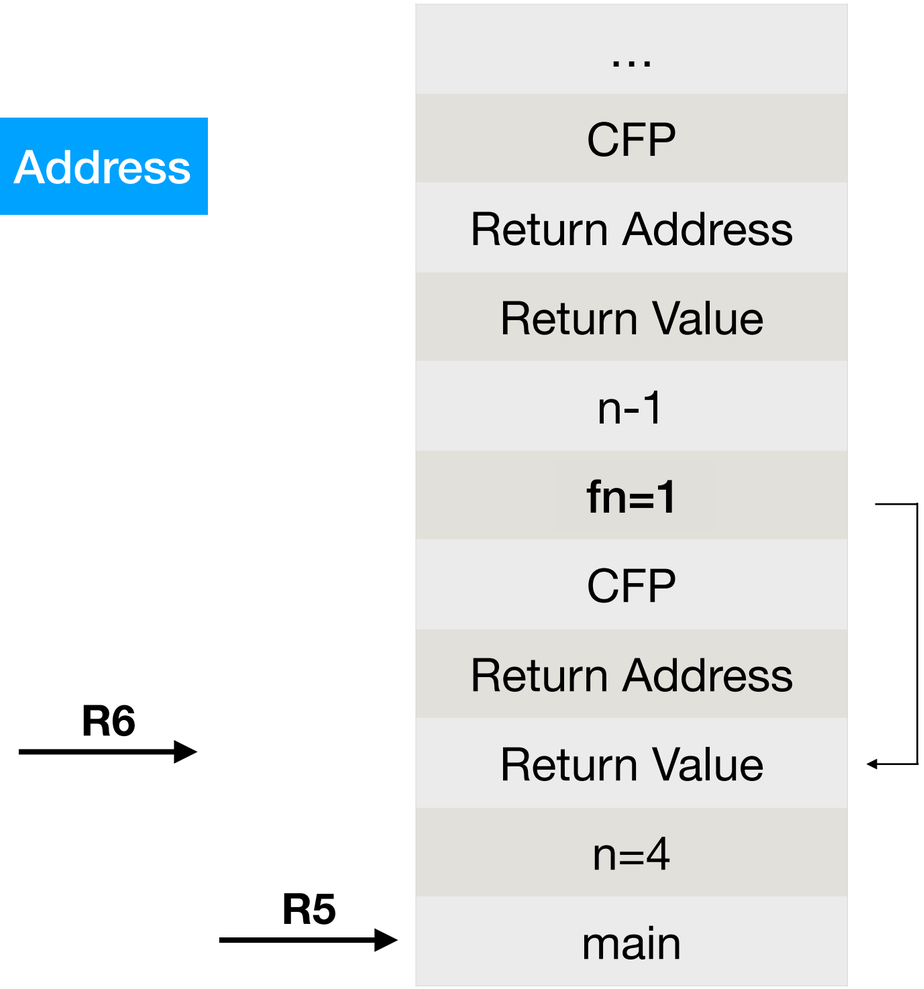


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
ADD R6, R5, #3 ;pop local variables

;restore caller's frame pointer and return address
LDR R5, R6, #-2
LDR R7, R6, #-1
```

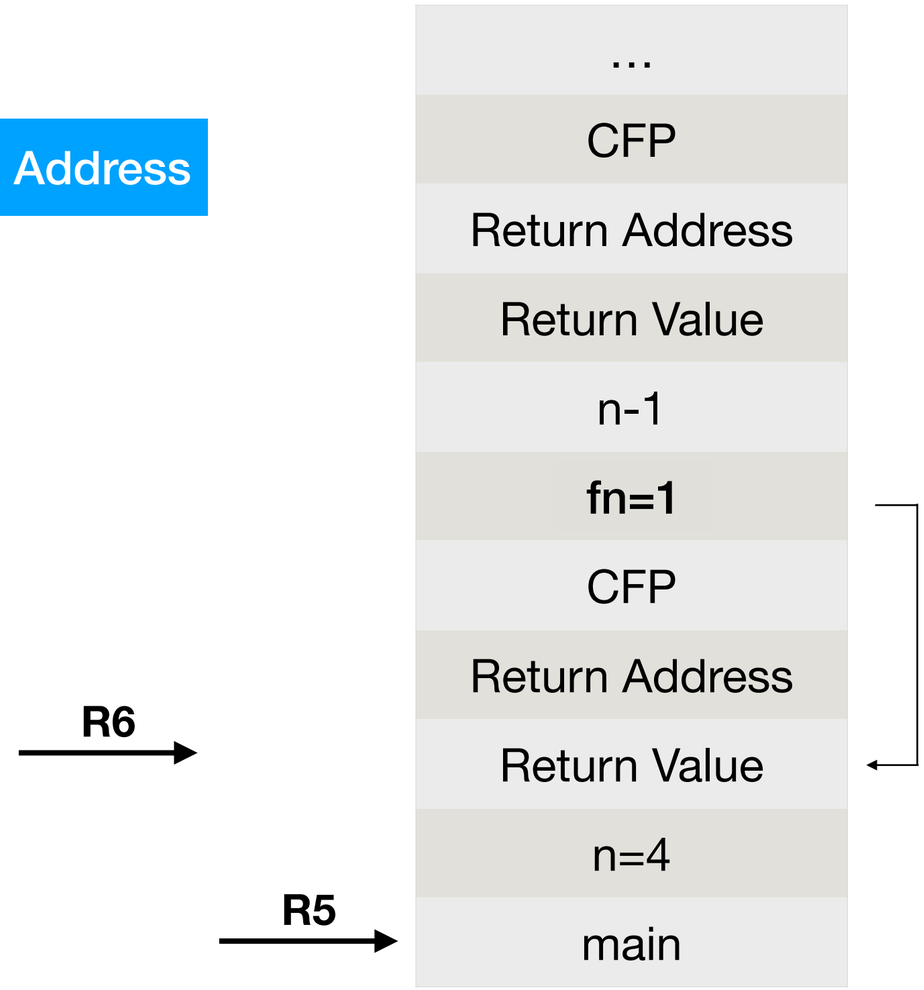


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
ADD R6, R5, #3 ;pop local variables

;restore caller's frame pointer and return address
LDR R5, R6, #-2
LDR R7, R6, #-1


;return to caller
```

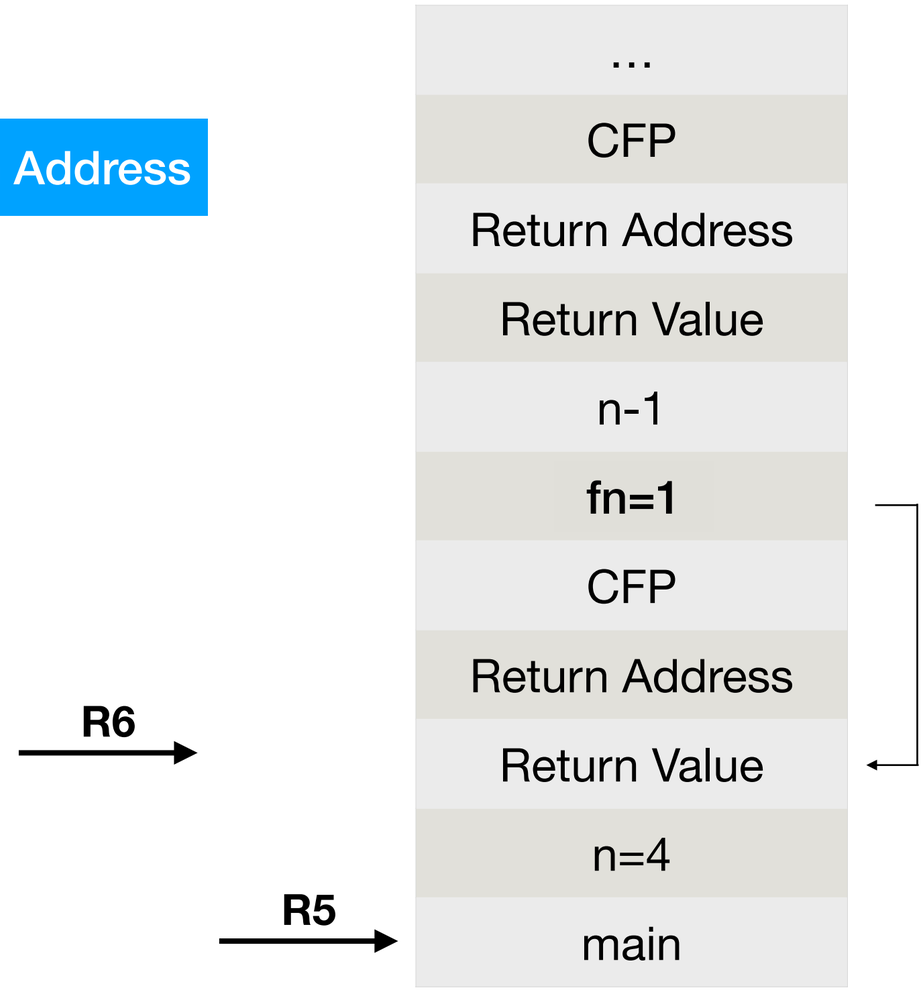


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(n);
}
```

```
BASE_CASE
AND R2, R2, #0
ADD R2, R2, #1
STR R2, R5, #0 ;set fn = 1

RETURN
;set return value
LDR R0, R5, #0
STR R0, R5, #3

;callee tear-down of Running(n)'s activation record
ADD R6, R5, #3 ;pop local variables

;restore caller's frame pointer and return address
LDR R5, R6, #-2
LDR R7, R6, #-1


;return to caller
RET
```

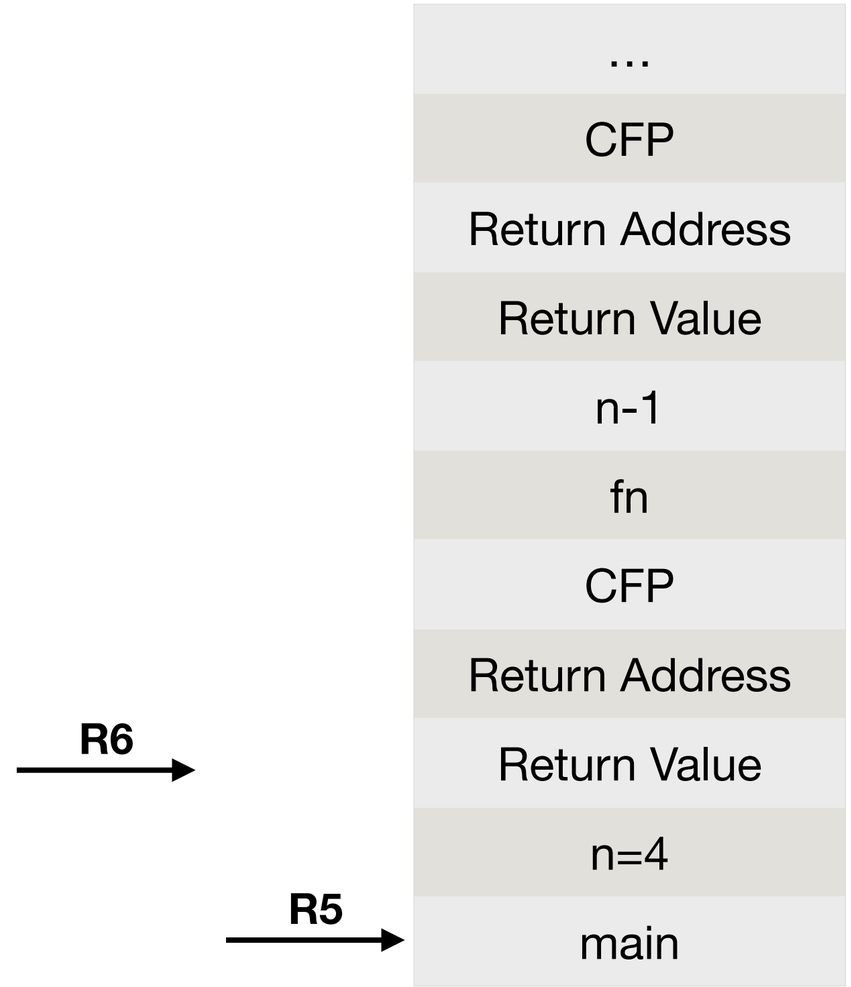


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

```
;Caller stack Tear-down for Running(n)
```

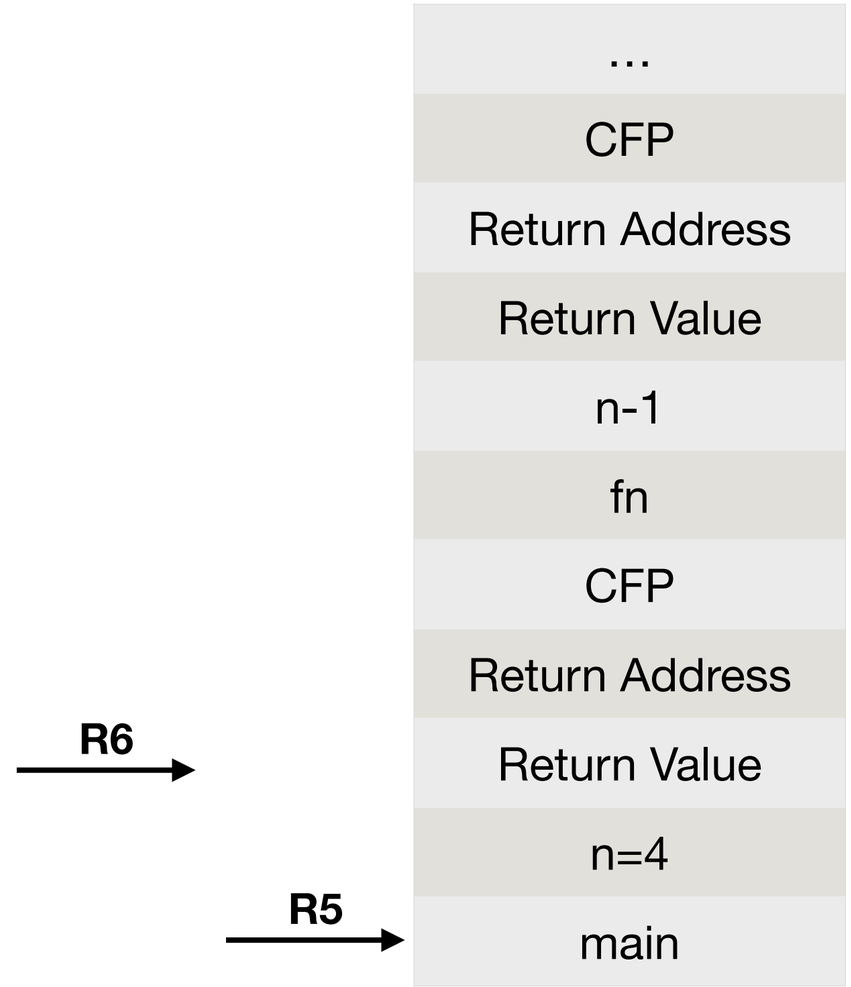


```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
```



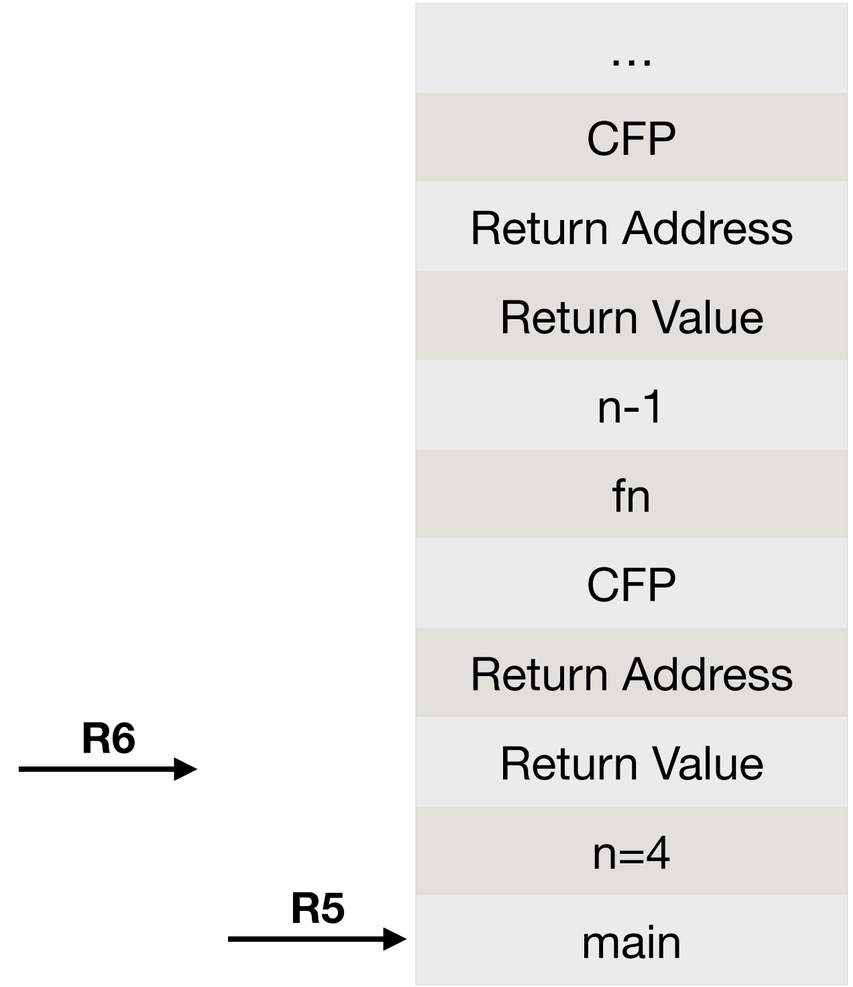
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
STR R0, R5, #-1 ;save return value to answer
```

Inside main's activation frame, answer is the second local variable



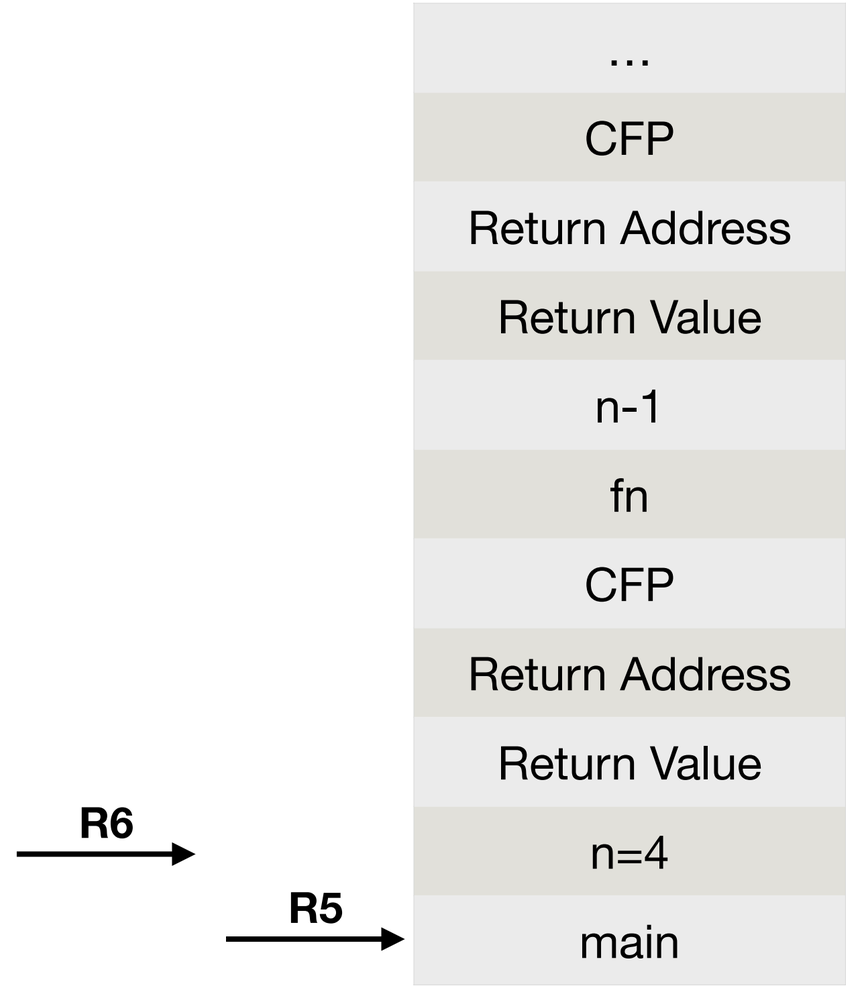
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

Recursion in LC3

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
STR R0, R5, #-1 ;save return value to answer
ADD R6, R6, #1  ;pop return value from stack
```

Inside main's activation frame, answer is the second local variable



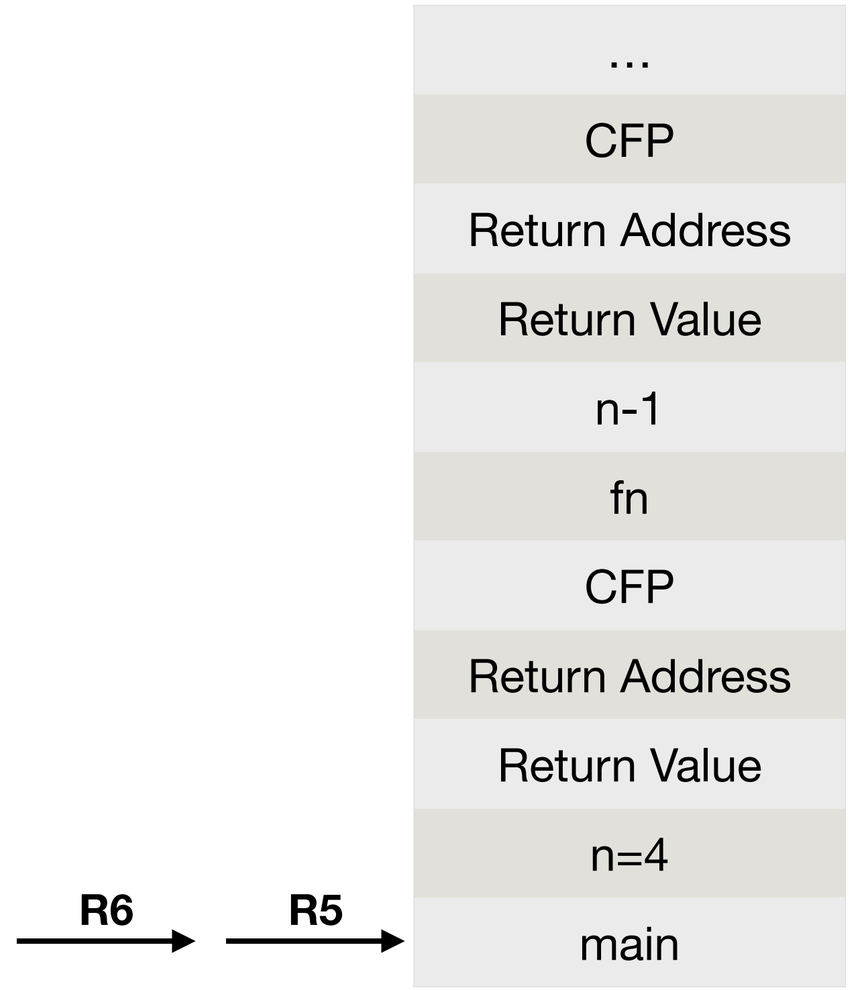
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

Recursion in LC3

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
STR R0, R5, #-1 ;save return value to answer
ADD R6, R6, #1  ;pop return value from stack
ADD R6, R6, #1  ;pop argument from stack
```

Inside main's activation frame, answer is the second local variable



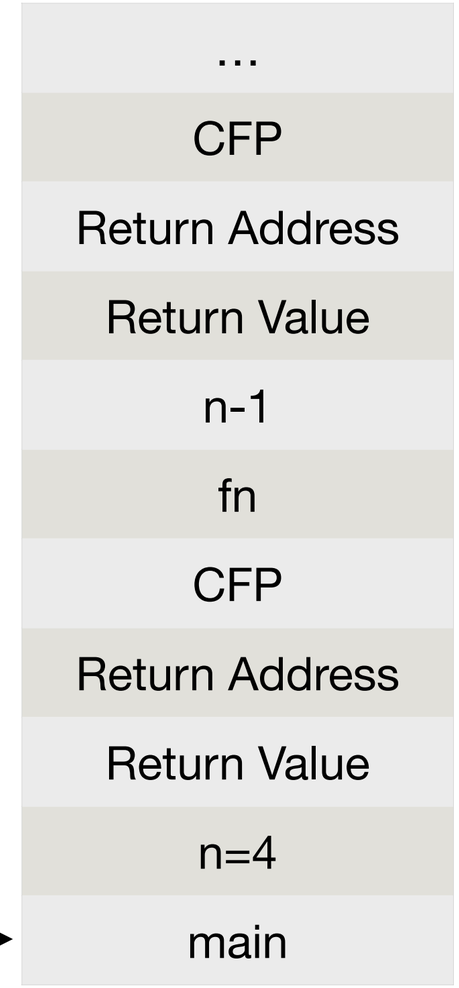
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

Recursion in LC3

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
STR R0, R5, #-1 ;save return value to answer
ADD R6, R6, #1  ;pop return value from stack
ADD R6, R6, #1  ;pop argument from stack
```

Inside main's activation frame, answer is the second local variable



Back to where we started!

```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

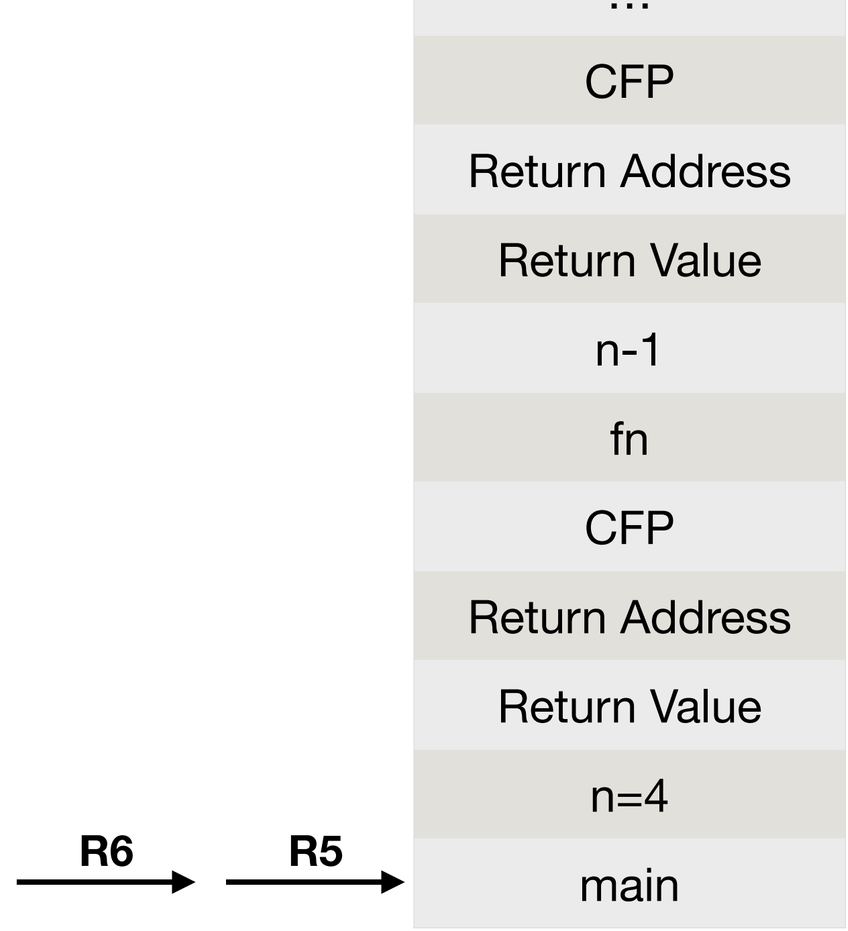
Recursion in LC3

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
STR R0, R5, #-1 ;save return value to answer
ADD R6, R6, #1  ;pop return value from stack
ADD R6, R6, #1  ;pop argument from stack
```

→ Inside main's activation frame, answer is the second local variable

Practice practice practice!

Back to where we started!



```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

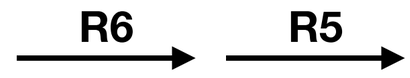
Recursion in LC3

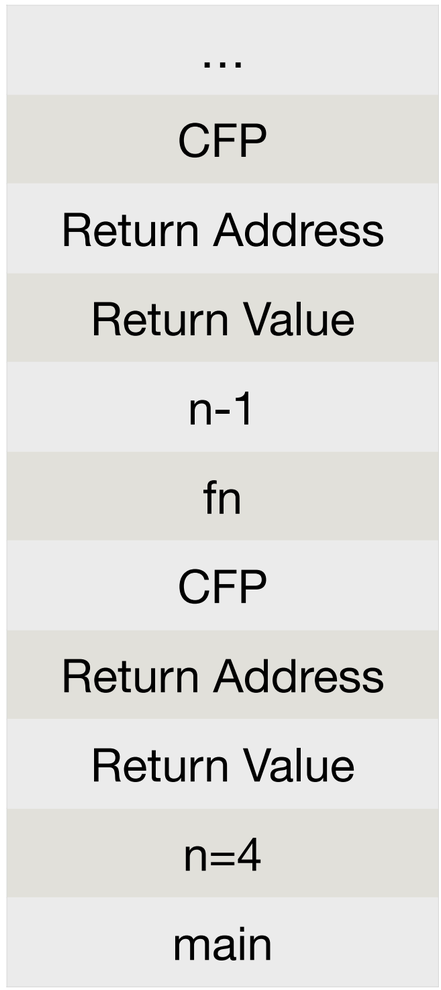
```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
STR R0, R5, #-1 ;save return value to answer
ADD R6, R6, #1  ;pop return value from stack
ADD R6, R6, #1  ;pop argument from stack
```

→ Inside main's activation frame, answer is the second local variable

Practice practice practice!

Back to where we started!





[Full LC3 Code]

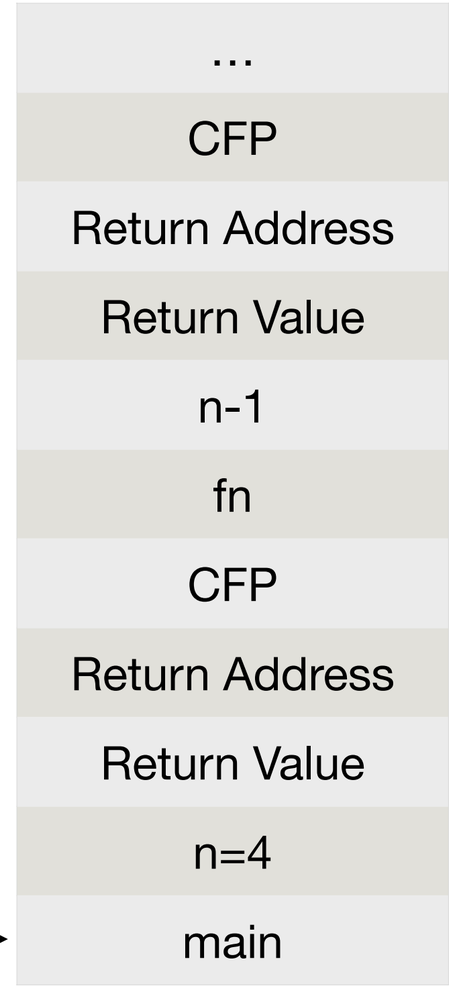
```c
int running_sum(int n){
    int fn;
    if (n==1)
        fn = 1;
    else
        fn = n + running_sum(n-1);
    return fn;
}
```

```c
int main(void){
    int n = 4;
    int answer;
    answer = running_sum(4);
}
```

Recursion in LC3

```
;Caller stack Tear-down for Running(n)
LDR R0, R6, #0  ;copy return value to R0
STR R0, R5, #-1 ;save return value to answer
ADD R6, R6, #1  ;pop return value from stack
ADD R6, R6, #1  ;pop argument from stack
```

Inside main's activation frame, answer is the second local variable



Practice practice practice!

Back to where we started!

[Github C to LC3 Steps]() [Full LC3 Code]()

Next time

- More problem solving with recursion.
  - A small chess problem
  - Solving a maze
- When is recursion good vs. bad?