C++ Programming

Instructor: Rita Kuo
Office: CS 520E
Phone: Ext. 4405
E-mail: rita.kuo@uvu.edu

Mapping zyBooks Chapters

Topic	zyBooks Chapter
Recursion	
Merge Sort	
Recursion using C/C++	7.1, 7.2, 7.5
Recursion Types	7.2, 7.3, 7.6, 7.7, 7.9
Stack Overflow	7.8

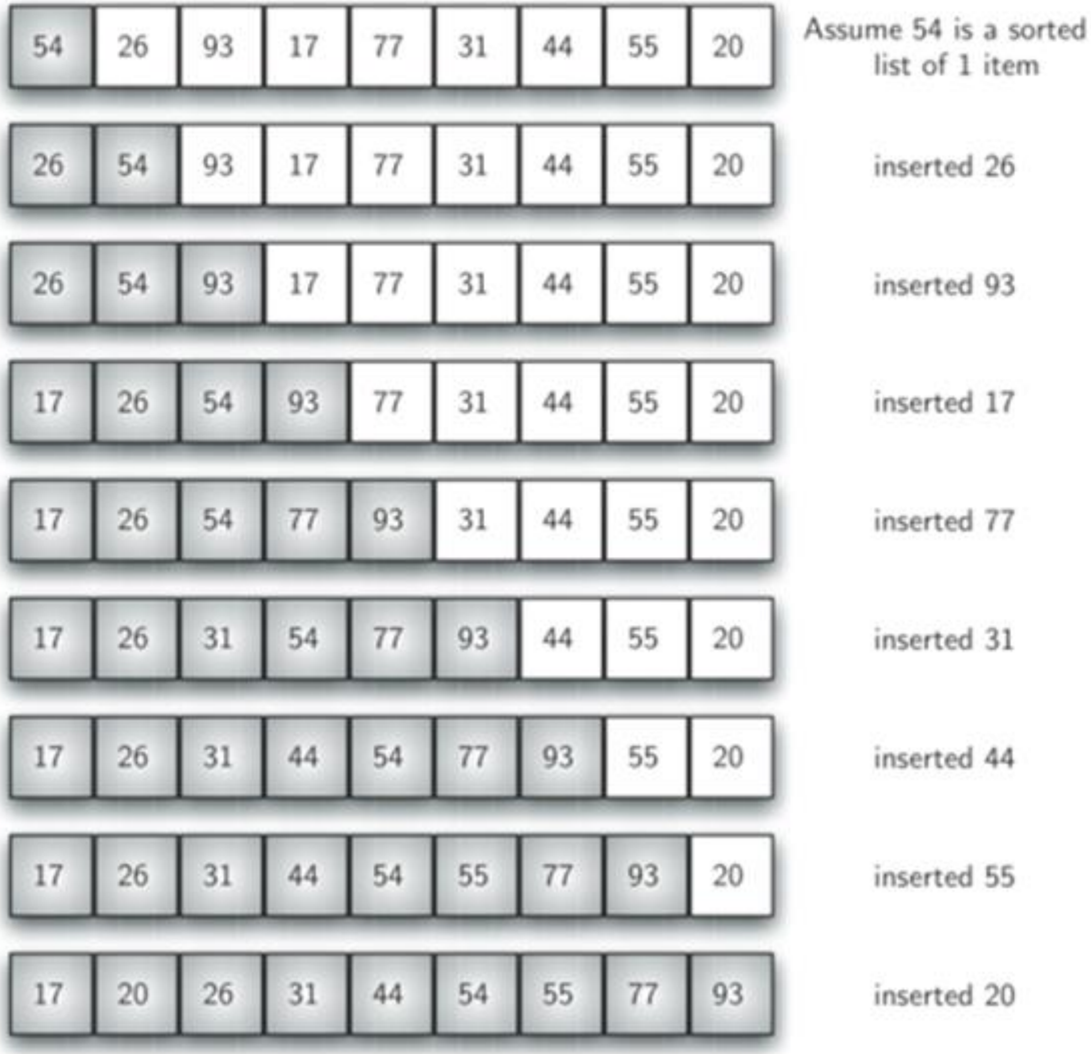
Self-study Chapters: 7.4

Recursion

Algorithm

- Definition
  - □ Any well-defined computational procedure
  - □ Takes some value, or set of values, as input
  - □ Produces some value, or set of values as output.
- Sorting Problem
  - □ Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
  - □ Output: A reordered $<a', a'_2, \ldots, a'_n>$ of the input sequence such that
  - □ $a'_1 <= a'_2 <= \ldots <= a'_n$
- Pseudocode
  - □ Informal high-level description of the operating principle of a computer program or other algorithm

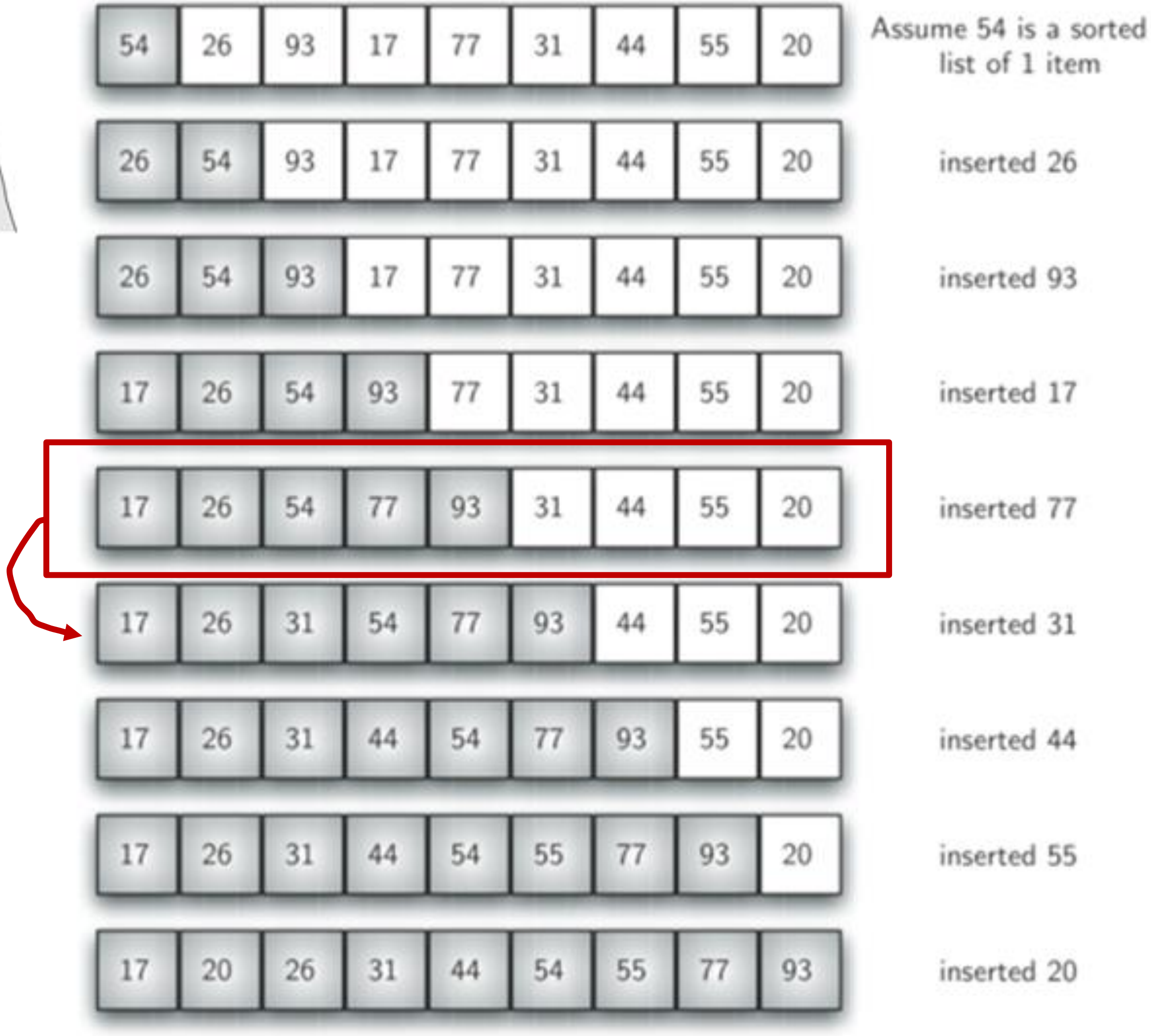
Insertion Sort



54	26	93	17	77	31	44	55	20	Assume 54 is a sorted list of 1 item
26	54	93	17	77	31	44	55	20	inserted 26
26	54	93	17	77	31	44	55	20	inserted 93
17	26	54	93	77	31	44	55	20	inserted 17
17	26	54	77	93	31	44	55	20	inserted 77
17	26	31	54	77	93	44	55	20	inserted 31
17	26	31	44	54	77	93	55	20	inserted 44
17	26	31	44	54	55	77	93	20	inserted 55
17	20	26	31	44	54	55	77	93	inserted 20

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html

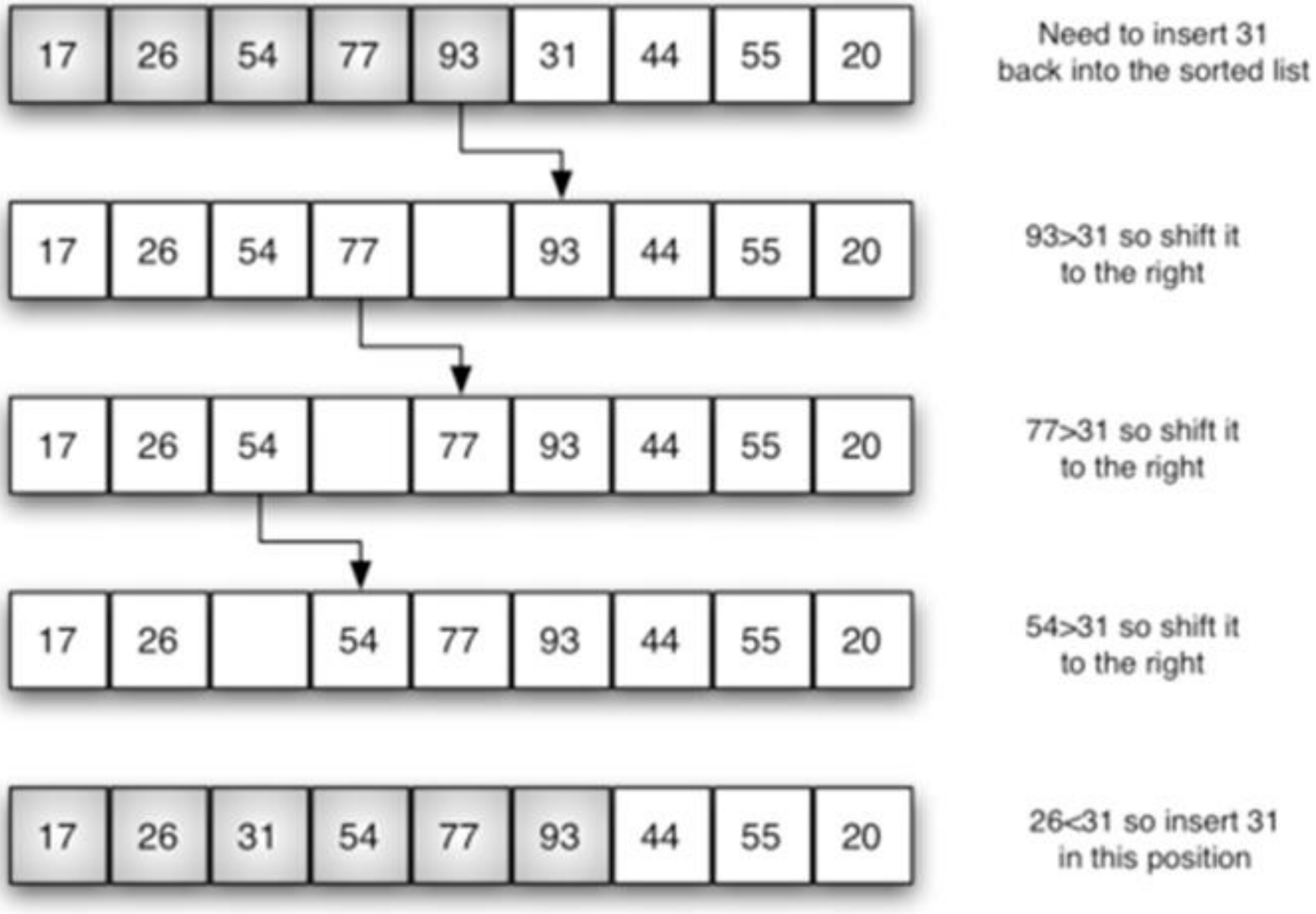
Insertion Sort



54	26	93	17	77	31	44	55	20	Assume 54 is a sorted list of 1 item
26	54	93	17	77	31	44	55	20	inserted 26
26	54	93	17	77	31	44	55	20	inserted 93
17	26	54	93	77	31	44	55	20	inserted 17
17	26	54	77	93	31	44	55	20	inserted 77
17	26	31	54	77	93	44	55	20	inserted 31
17	26	31	44	54	77	93	55	20	inserted 44
17	26	31	44	54	55	77	93	20	inserted 55
17	20	26	31	44	54	55	77	93	inserted 20

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html

Insertion Sort

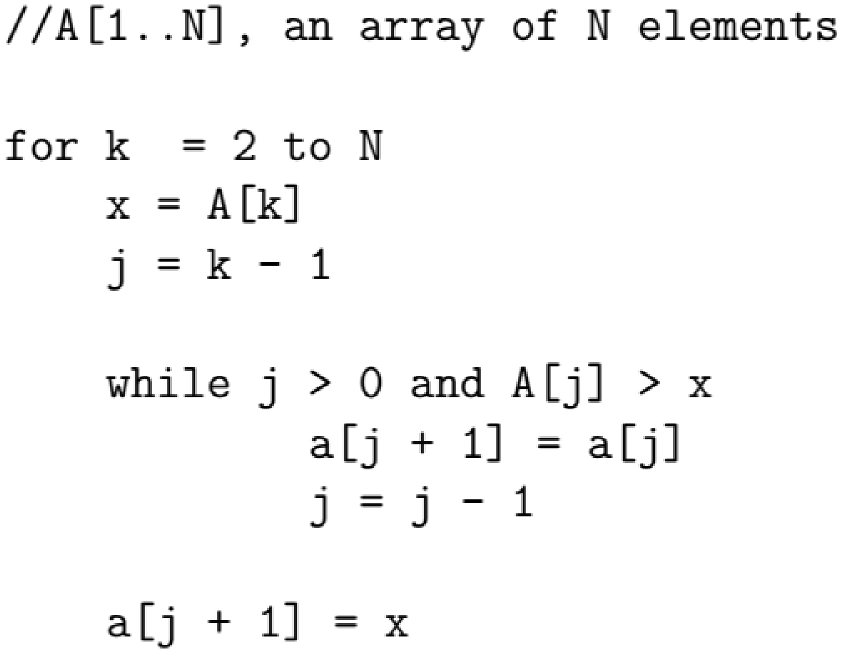


17	26	54	77	93	31	44	55	20	Need to insert 31 back into the sorted list
17	26	54	77		93	44	55	20	93>31 so shift it to the right
17	26	54		77	93	44	55	20	77>31 so shift it to the right
17	26		54	77	93	44	55	20	54>31 so shift it to the right
17	26	31	54	77	93	44	55	20	26<31 so insert 31 in this position

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html

Insertion Sort

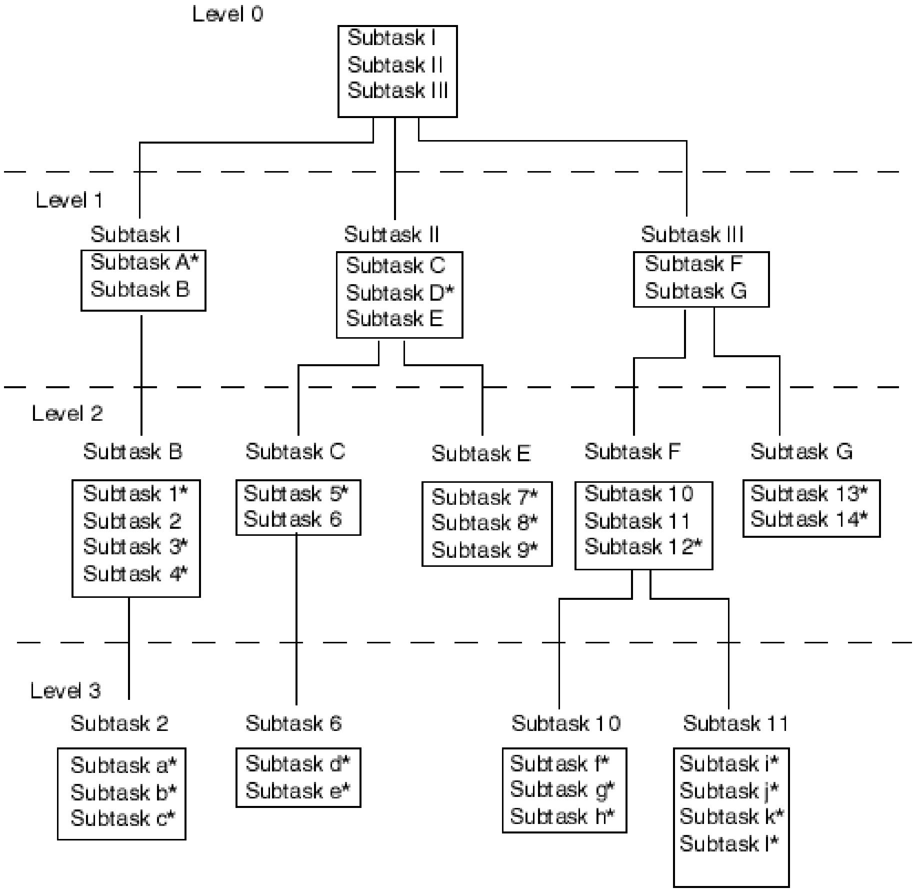
- Algorithm



```
//A[1..N], an array of N elements

for k  = 2 to N
    x = A[k]
    j = k - 1

    while j > 0 and A[j] > x
            a[j + 1] = a[j]
            j = j - 1

    a[j + 1] = x
```

Top-Down Design

- Think about the program in abstract (major steps or subtasks)
- Refine each steps by thinking about the details in each subtasks



Level 0

Subtask I
Subtask II
Subtask III

Level 1

Subtask I
Subtask A*
Subtask B

Subtask II
Subtask C
Subtask D*
Subtask E

Subtask III
Subtask F
Subtask G

Level 2

Subtask B
Subtask 1*
Subtask 2
Subtask 3*
Subtask 4*

Subtask C
Subtask 5*
Subtask 6

Subtask E
Subtask 7*
Subtask 8*
Subtask 9*

Subtask F
Subtask 10
Subtask 11
Subtask 12*

Subtask G
Subtask 13*
Subtask 14*

Level 3

Subtask 2
Subtask a*
Subtask b*
Subtask c*

Subtask 6
Subtask d*
Subtask e*

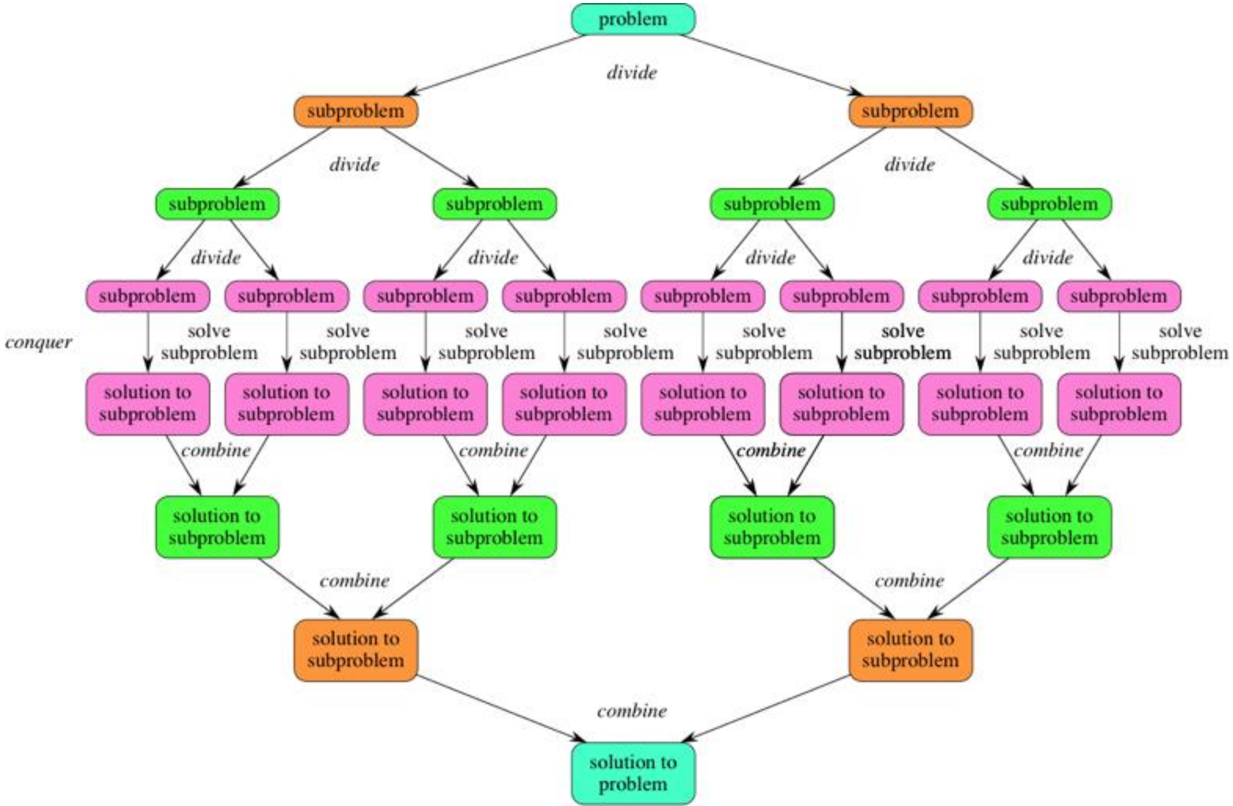
Subtask 10
Subtask f*
Subtask g*
Subtask h*

Subtask 11
Subtask i*
Subtask j*
Subtask k*
Subtask l*

http://www.eecs.wsu.edu/~cs150/tdd.htm

Divide and Conquer

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem
- **Conquer** the subproblems by solving them **recursively**
- **Combine** the solutions to the subproblems into the solution for the original problem



https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

Merge Sort

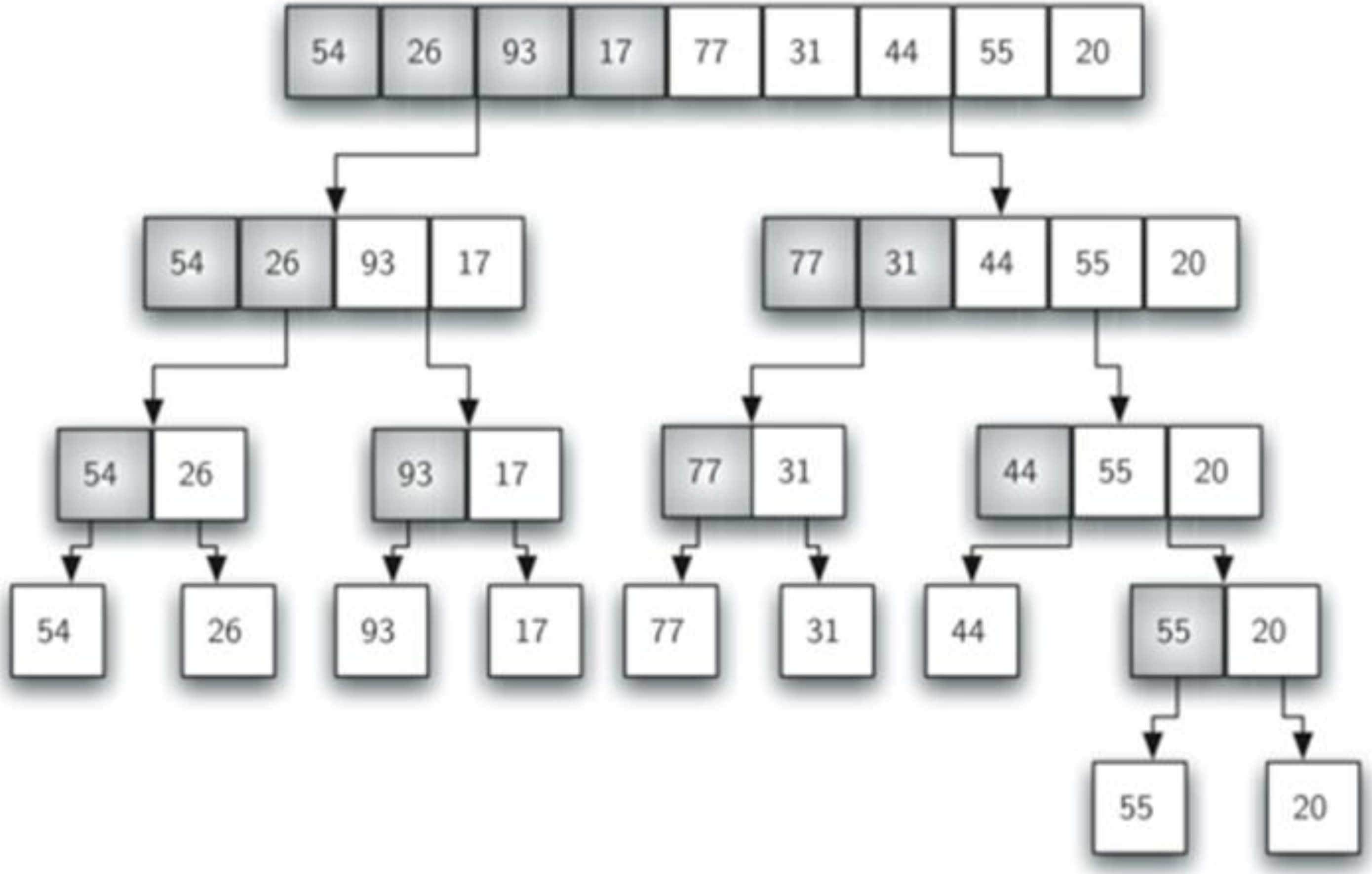
Divide and Conquer

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem
- **Conquer** the subproblems by solving them **recursively**
- **Combine** the solutions to the subproblems into the solution for the original problem

- Merge Sort
  - **Divide**: Divide the n-element sequence to be sorted into **two subsequence** of n/2 elements each
  - **Conquer**: Sort the two subsequences recursively using **merge sort**
  - **Combine**: **Merge** the two sorted subsequences to produce the sorted answer

Merge Sort

- Splitting the list

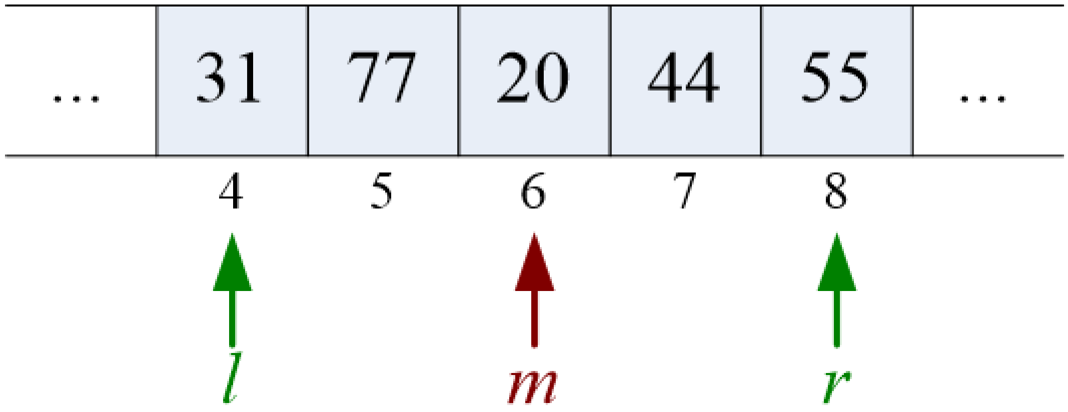


http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

- Algorithm - Merge Sort

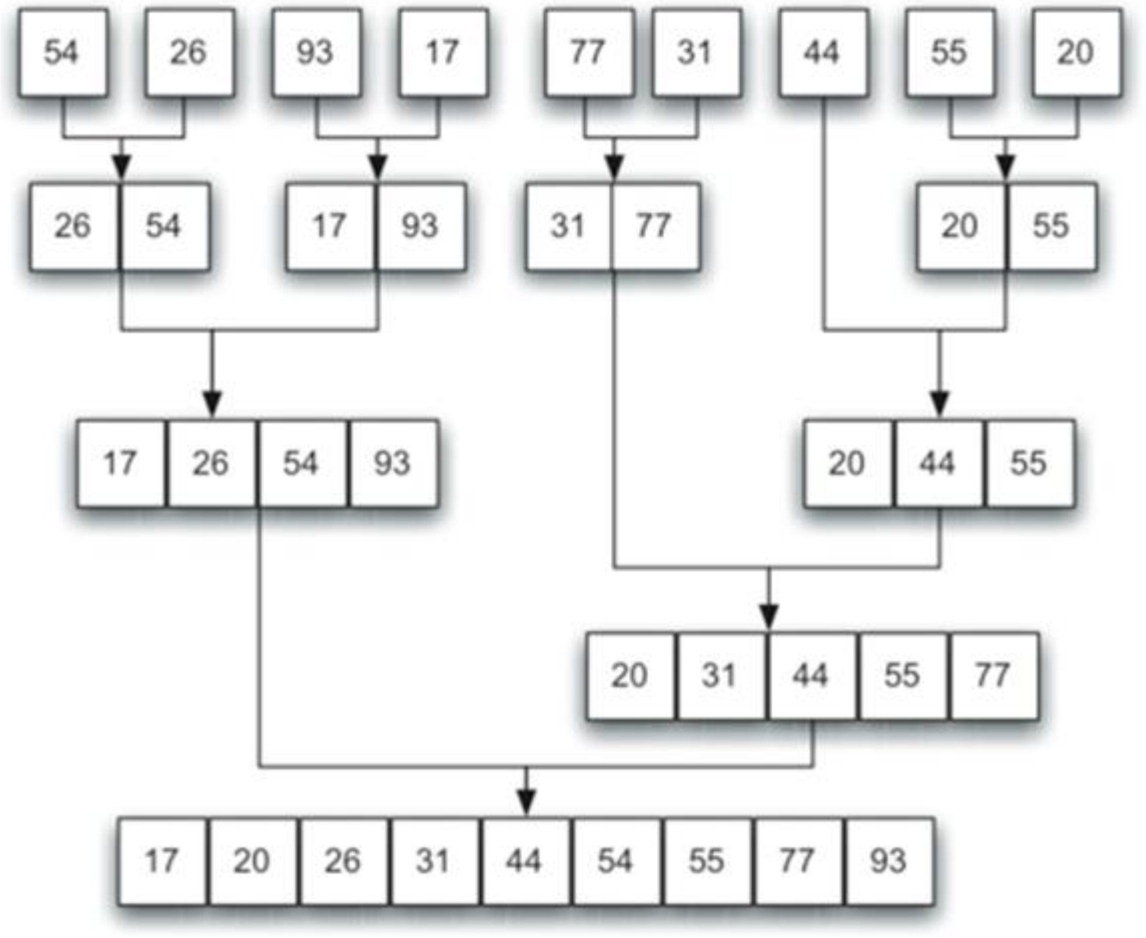
```
MergeSort ($A$, $l$, $r$)
if $r > l$
    $m = (l + r) / 2$          //find the middle index of the list
    MergeSort ($A$, $l$, $m$)
    MergeSort ($A$, $m + 1$, $r$)
    Merge ($A$, $l$, $m$, $r$)
```



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

- Merging the sublists



54 26 93 17 77 31 44 55 20

26 54 17 93 31 77 20 55

17 26 54 93 20 44 55

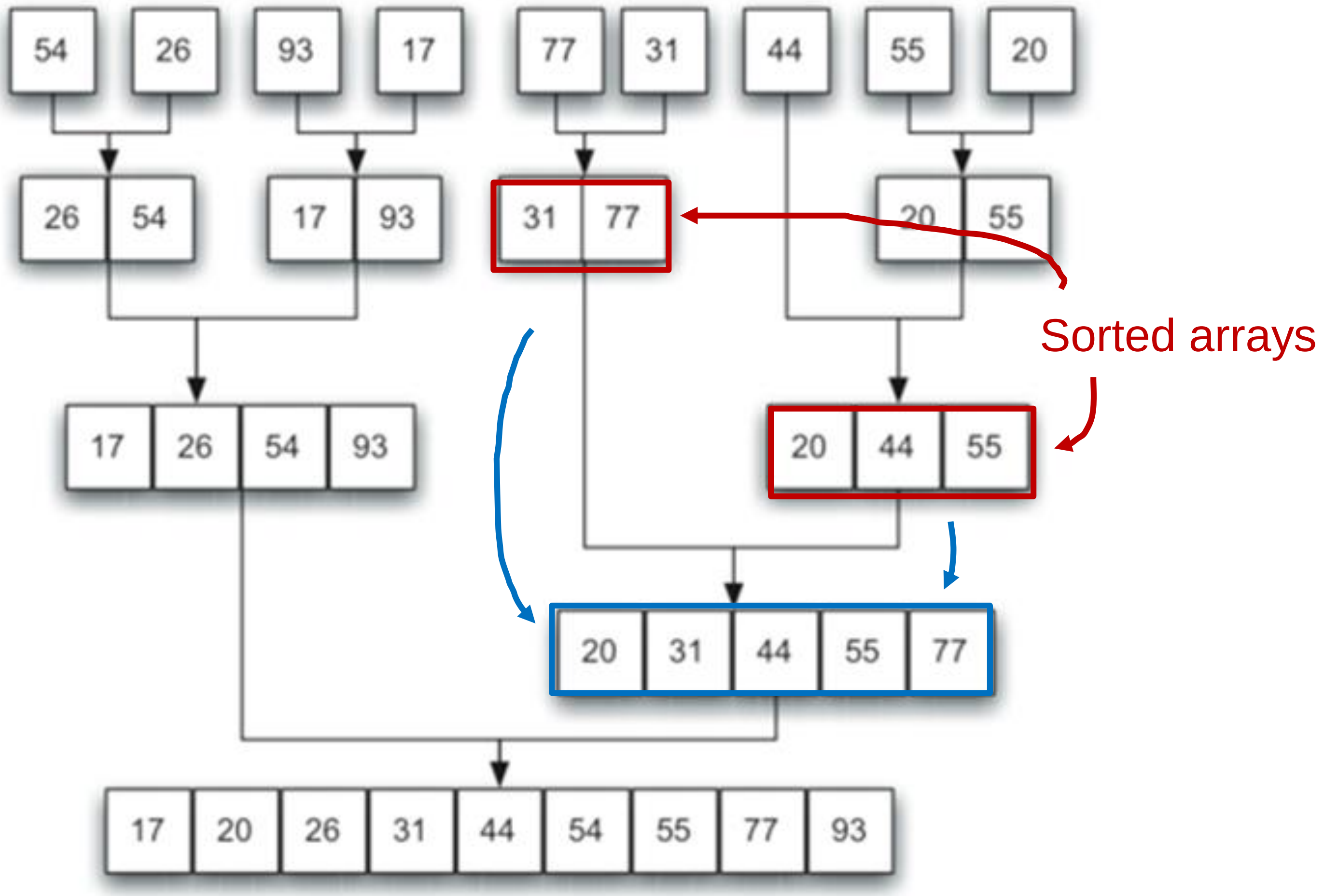
20 31 44 55 77

17 20 26 31 44 54 55 77 93

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

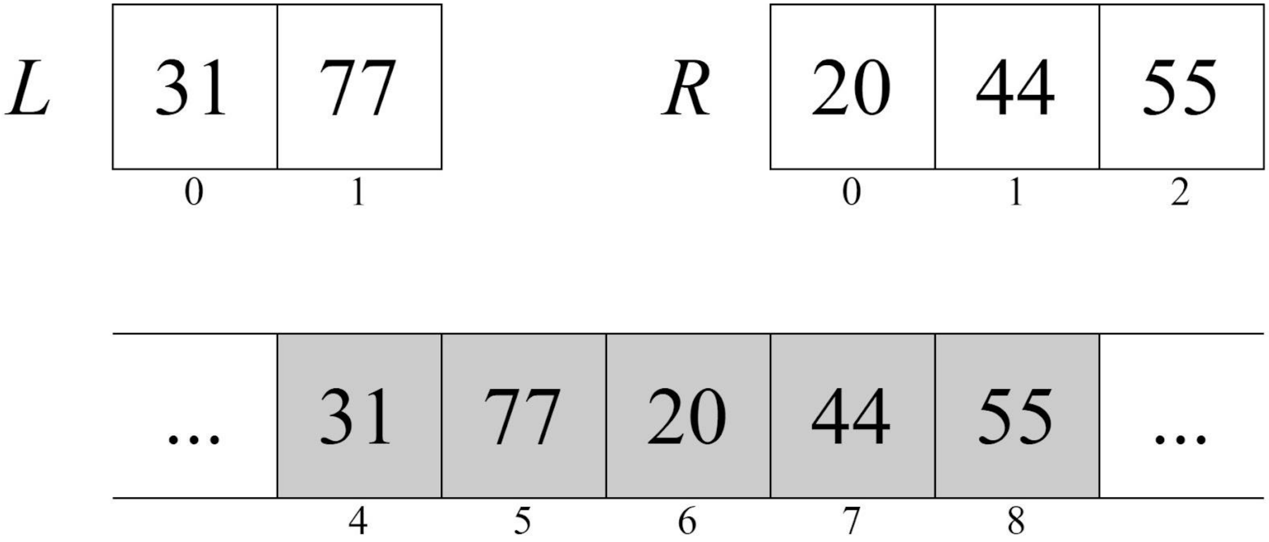
- Merging the sublists



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

- Merging the sublists



L	31	77
	0	1

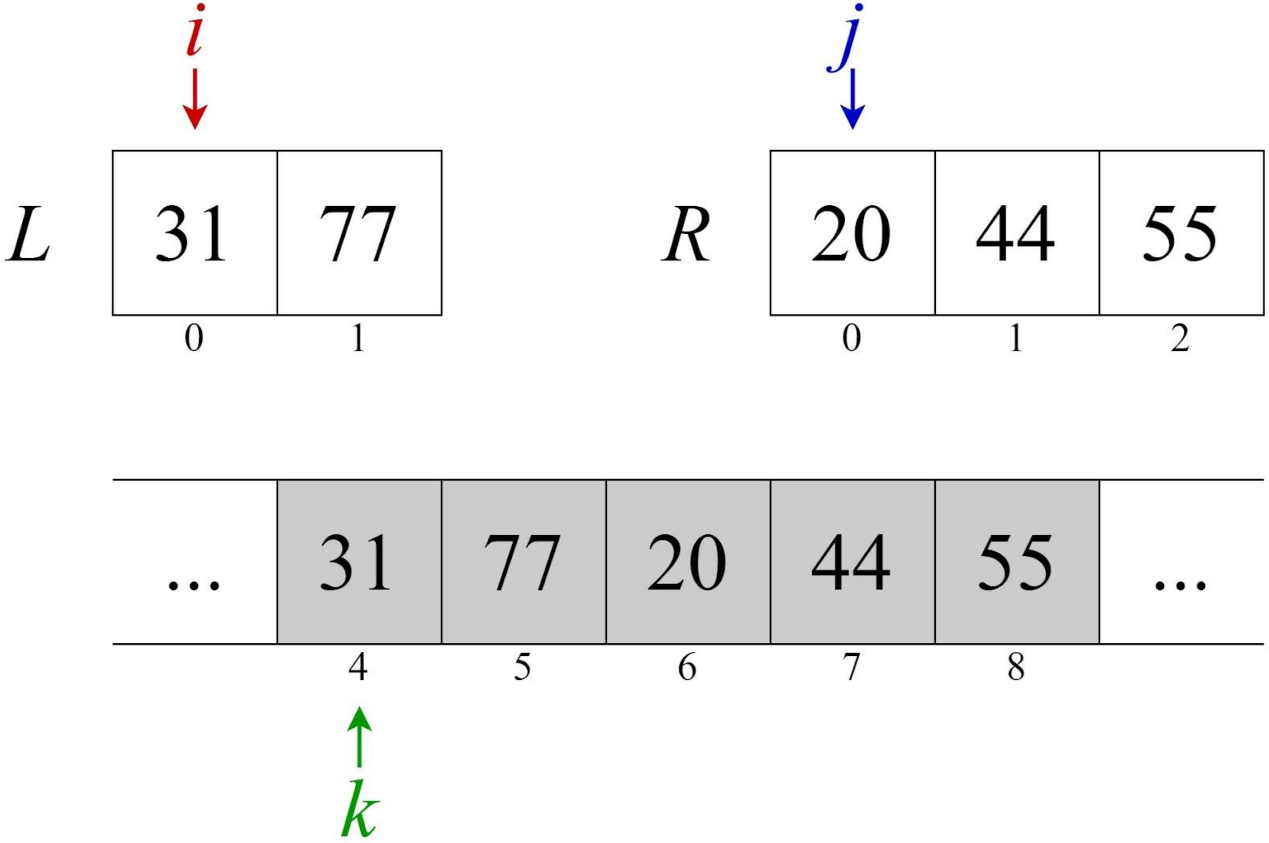
R	20	44	55
	0	1	2

...	31	77	20	44	55	...
	4	5	6	7	8	

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

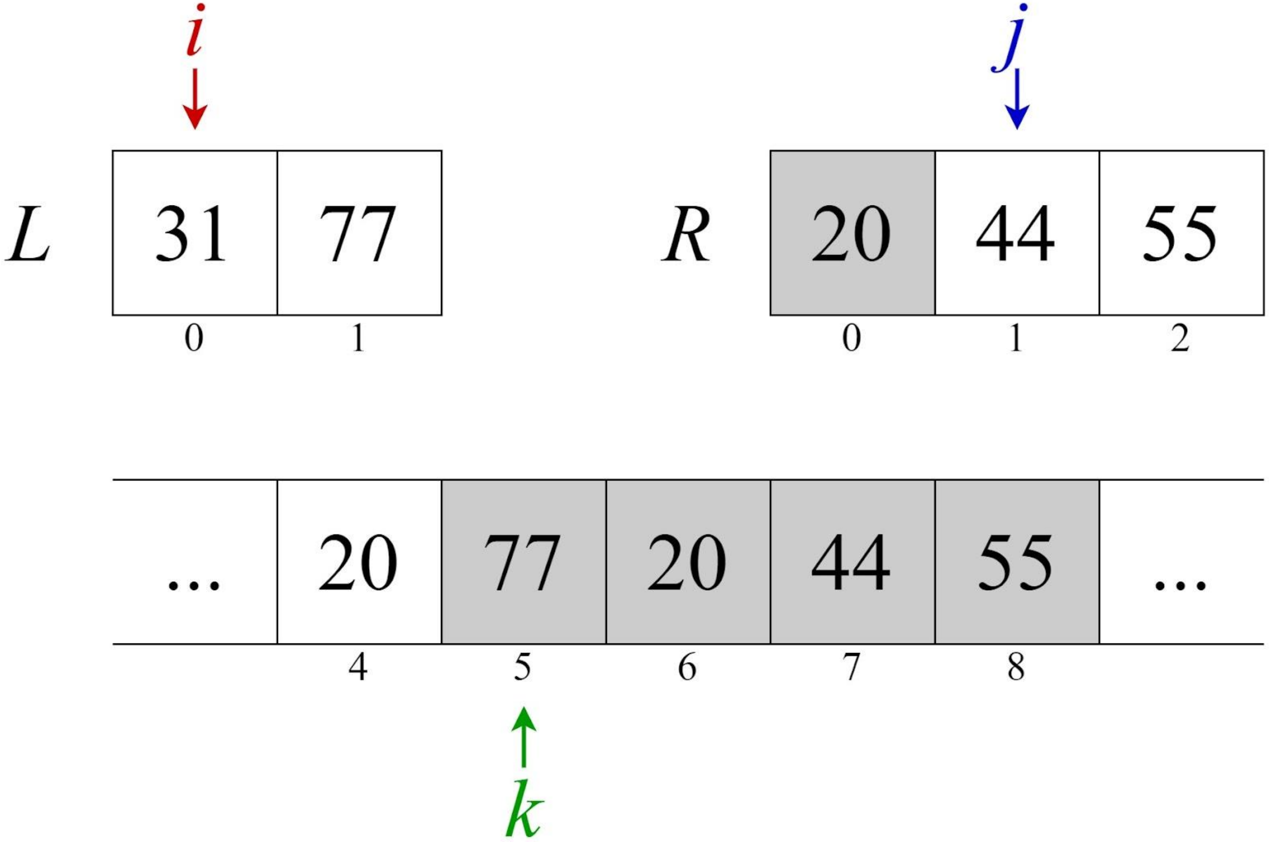
- Merging the sublists



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

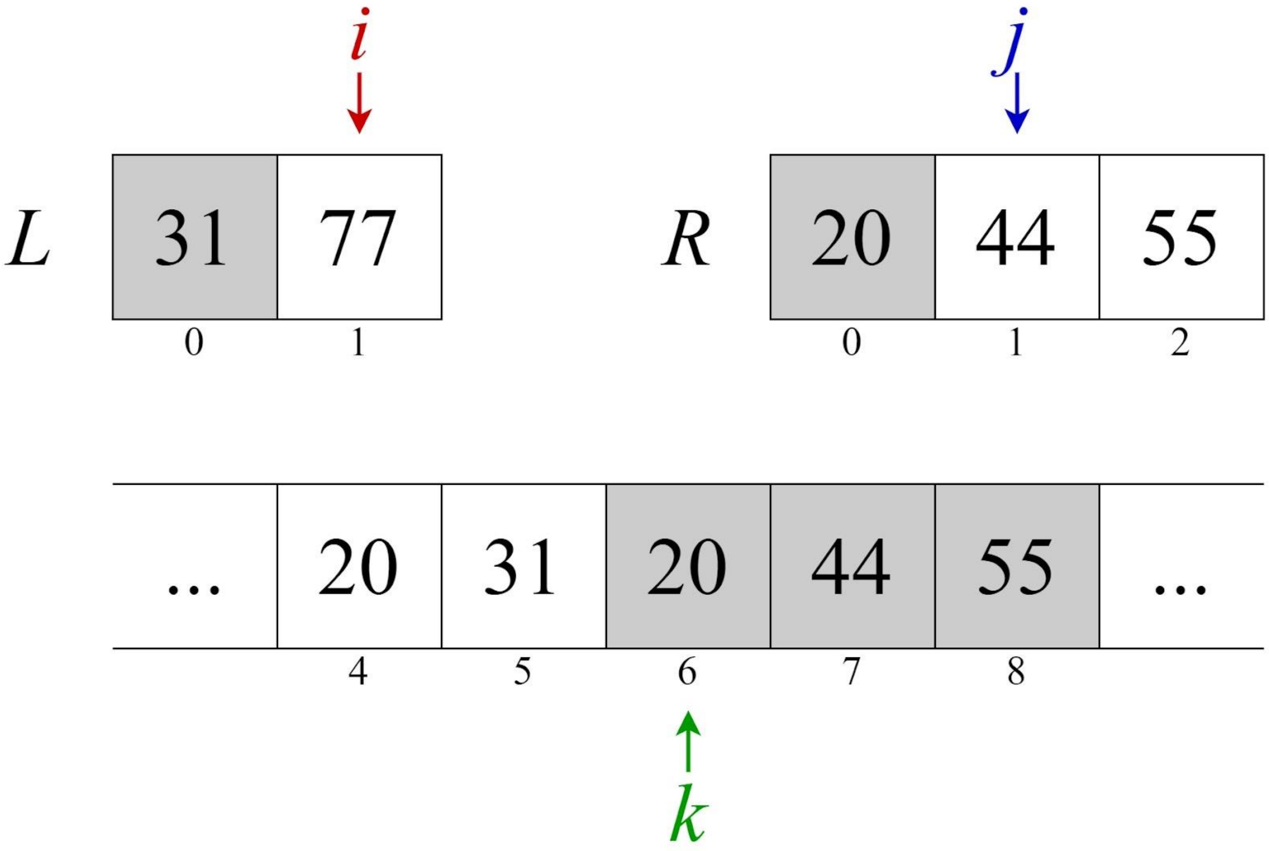
- Merging the sublists



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

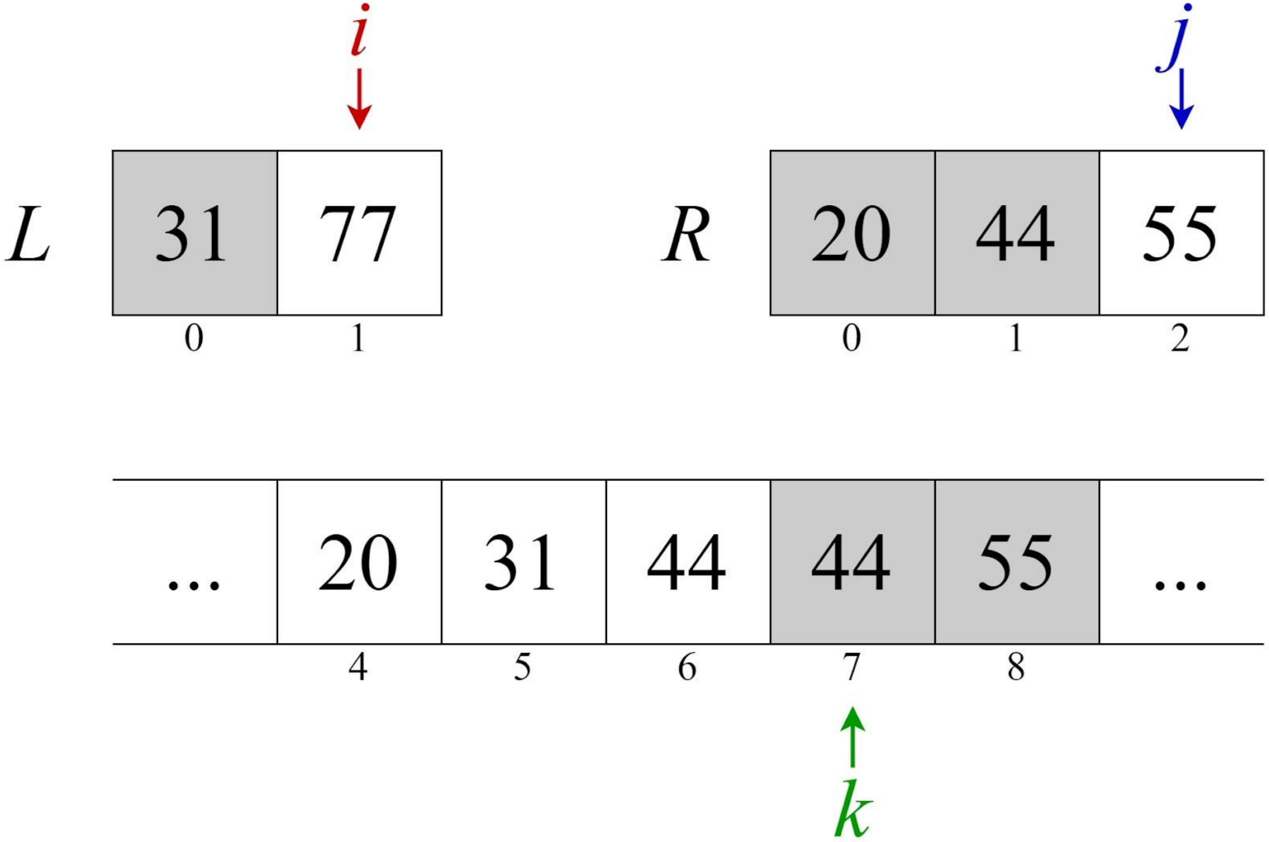
- Merging the sublists



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

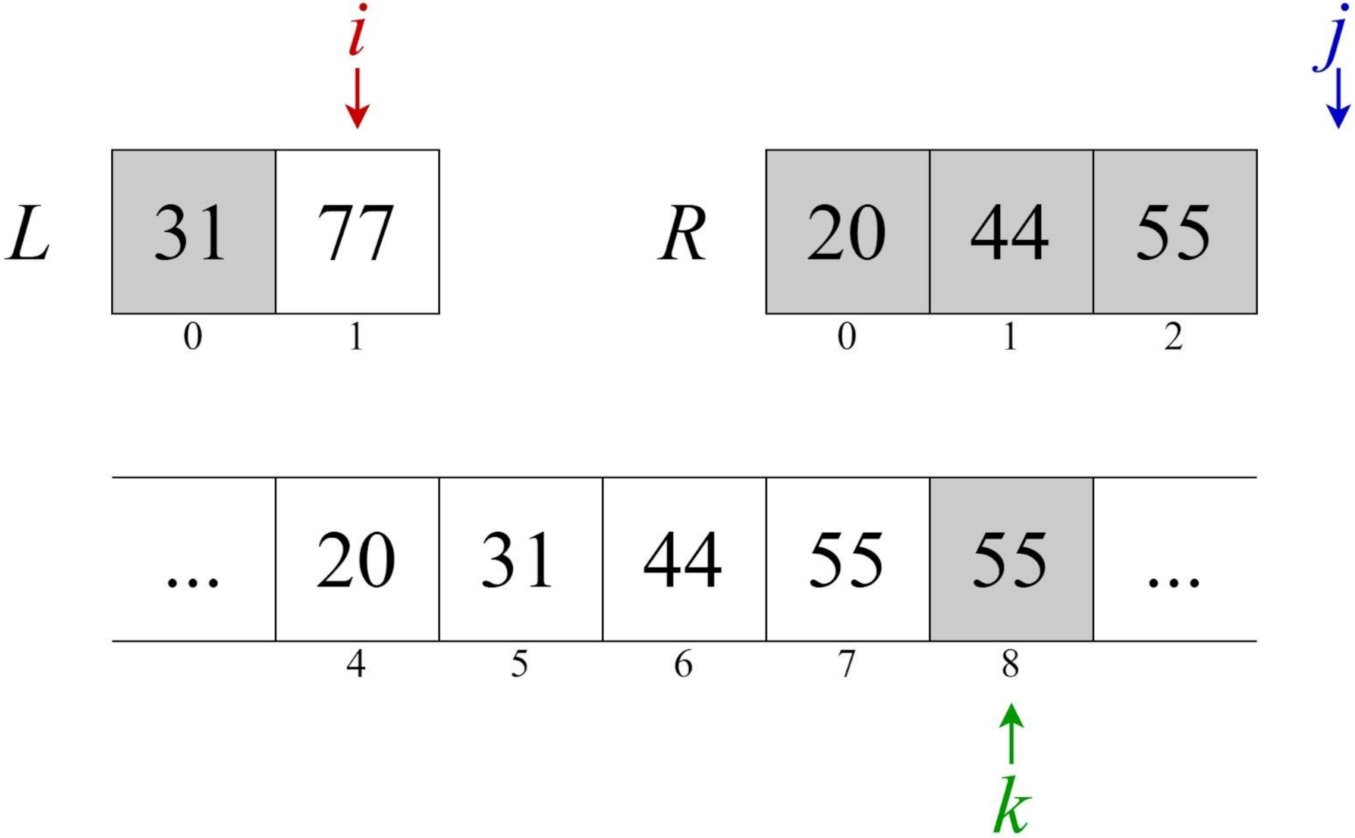
- Merging the sublists



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

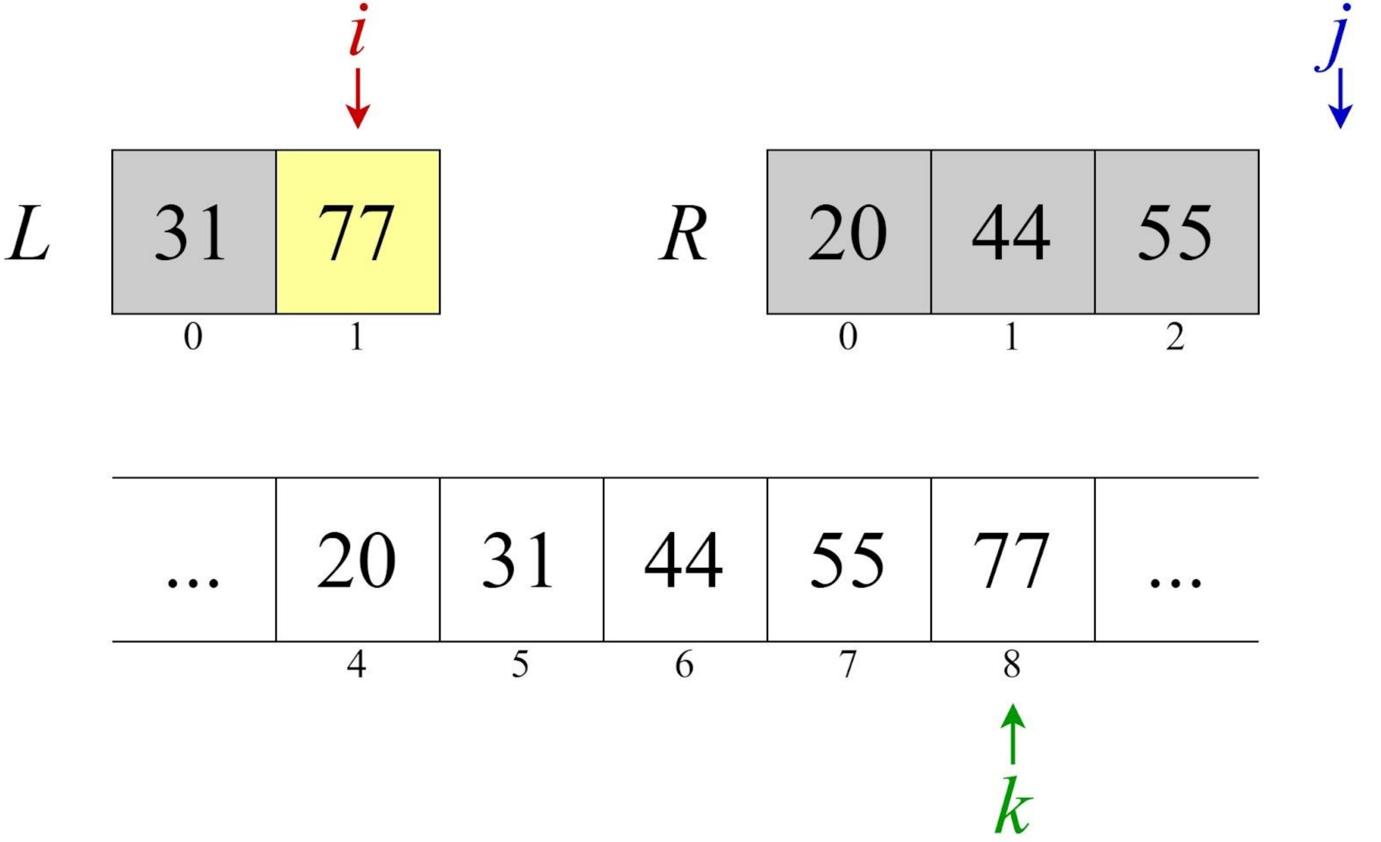
- Merging the sublists



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

- Merging the sublists



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Merge Sort

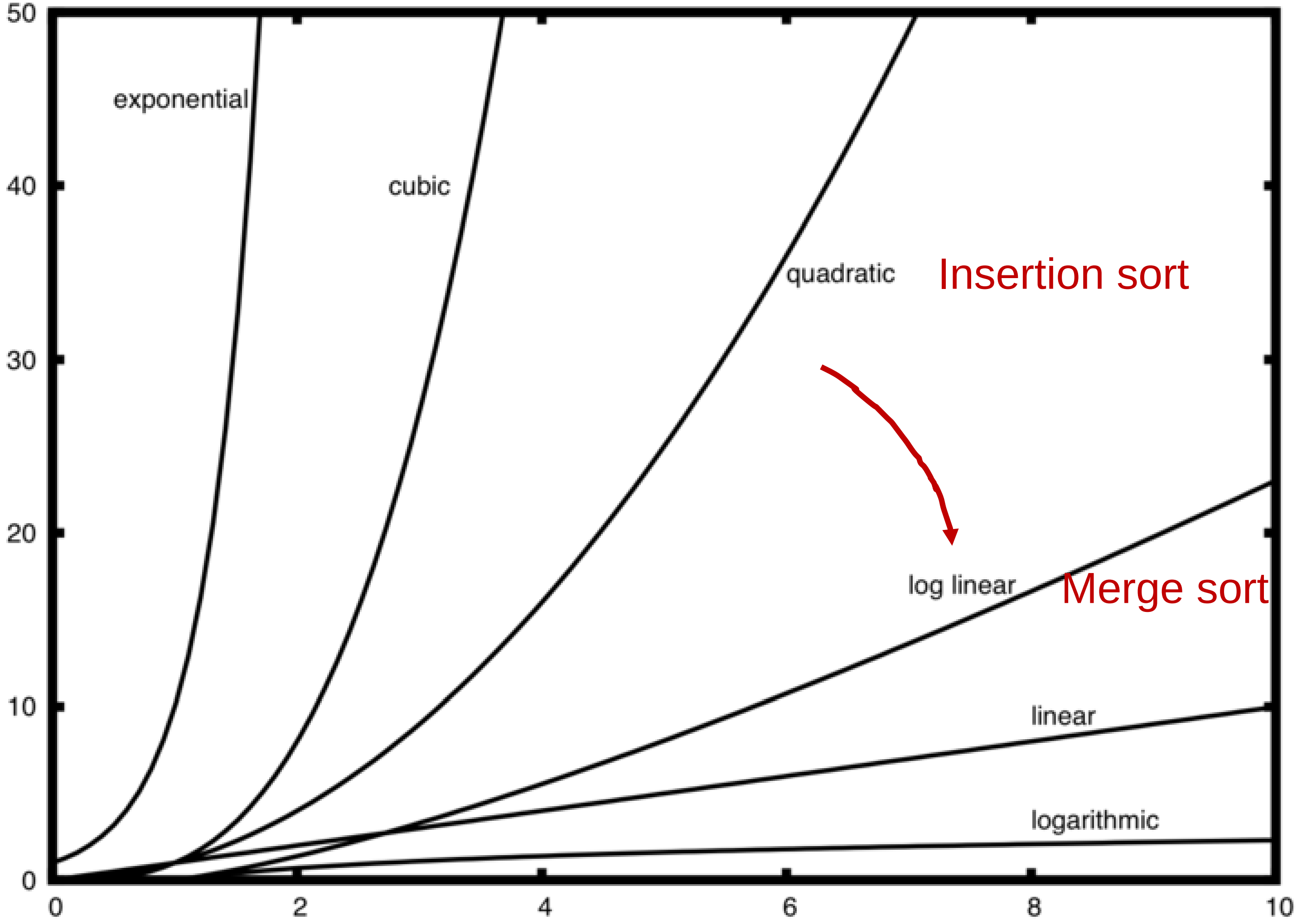
- Algorithm - Merge

Merge (*A*, *p*, *q*, *r*)

```
// copy the left part of the array to L
//               right part of the array to R
L = A[p:q]
R = A[q+1:r]
// i and j are index pointing to L and R
i = 0
j = 0
k = p
// copy the smallest item to the original array
while (i < len(L) and j < len(R))
    if L[i] <= R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
    k = k + 1
// copy the remaining elements to the original array
if (i < len(L))
    A[k:r] = L[i:len(L) - 1]
if (j < len(R))
    A[k:r] = R[j:len(R) - 1]
```

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

Analyzing Algorithm



Insertion sort

Merge sort

Comparing Merge Sort and Insertion Sort

- Time Complexity:
  - Merge Sort: **O(n lg n)**
  - Insertion Sort: **O(n2)**
- Space Complexity
  - Merge Sort: **O(n)**, cannot be preferred over the place where memory is a problem.
  - Insertion Sort: **O(1)**
- Efficiency
  - Merge Sort: efficient in terms of time
  - Insertion Sort: efficient in terms of space
- More information
  - https://www.geeksforgeeks.org/merge-sort-vs-insertion-sort/

Recursion using C/C++

Review - Functions

- What is a function?
  - A small program, with its own declarations and statements
- Benefits
  - Divide a program into small pieces that are easier for people to understand and modify
  - Avoid duplicating code that’s used more than once
- The form of a function in C++

```
return-val name-of-function (list of formal parameters)
{
    body of function
}
```

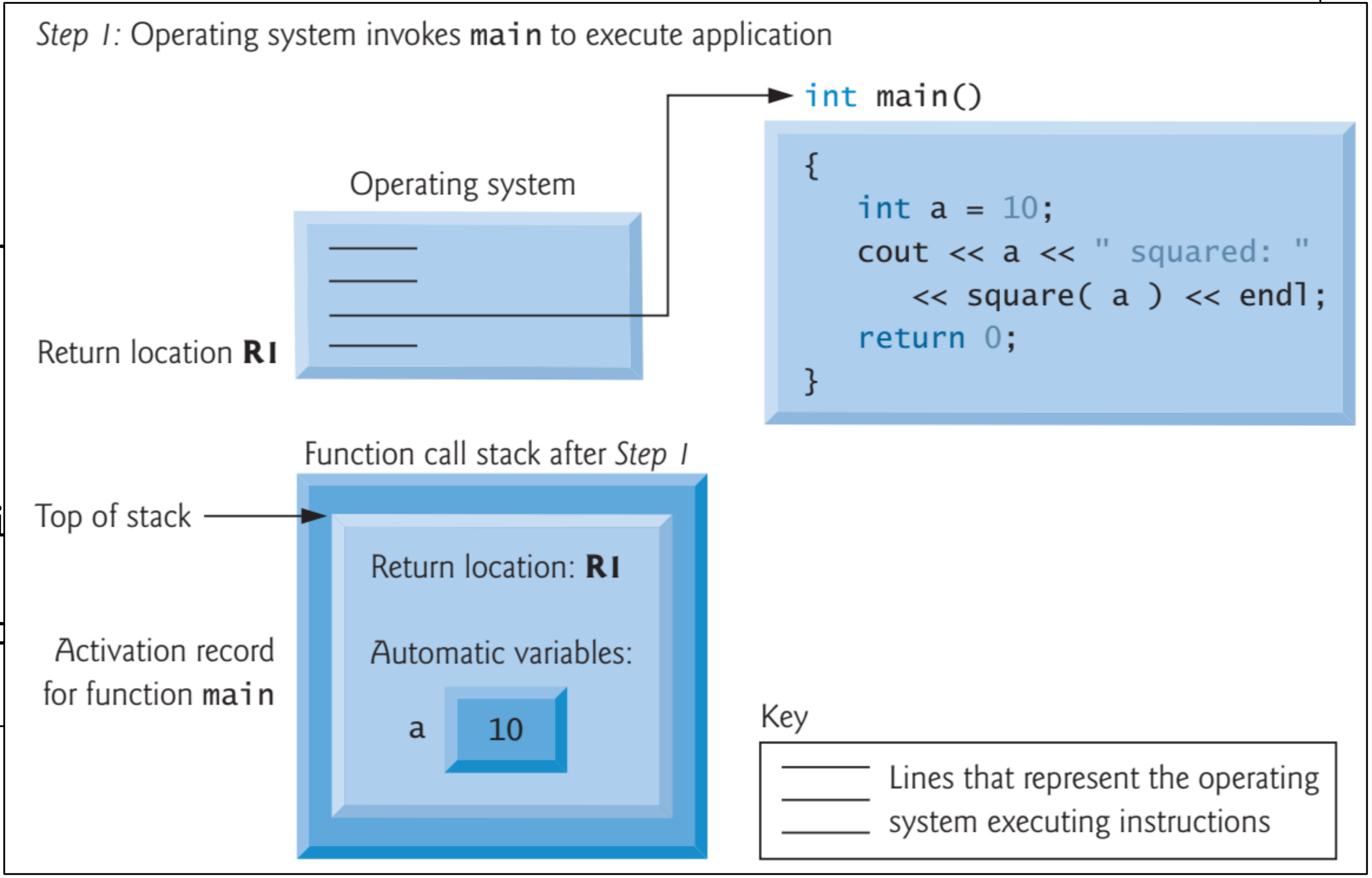
- Example

```
int main()
{
  cout << “Hello World!“<< endl;
  return 0;
}
```

Review - Function Call Stack and Stack Frames

- Example

```
#include <iostream>
using na

int squa
{
        r
}

int main
{
        i


        c
}
```



Step 1: Operating system invokes `main` to execute application

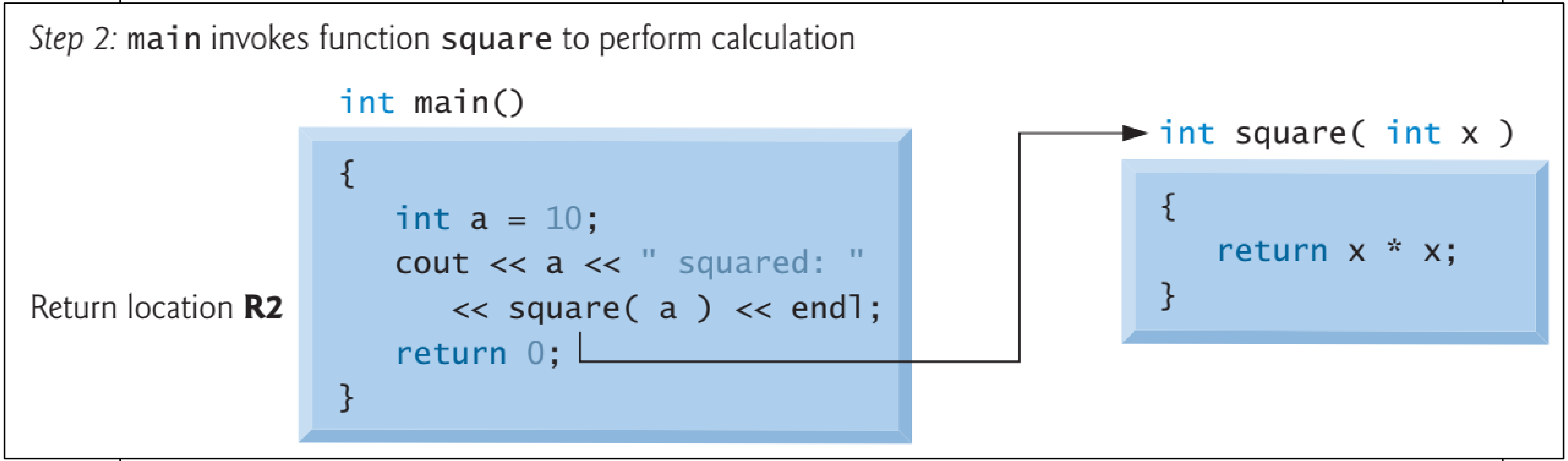
```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

Operating system
Return location **R1**
Function call stack after *Step 1*
Top of stack
Activation record for function main
Return location: **R1**
Automatic variables:
a
10
Key
Lines that represent the operating system executing instructions

Review - Function Call Stack and Stack Frames

- Example

```
#include <iostream>
using namespace std;
```



Step 2: **main** invokes function **square** to perform calculation

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

Return location **R2**

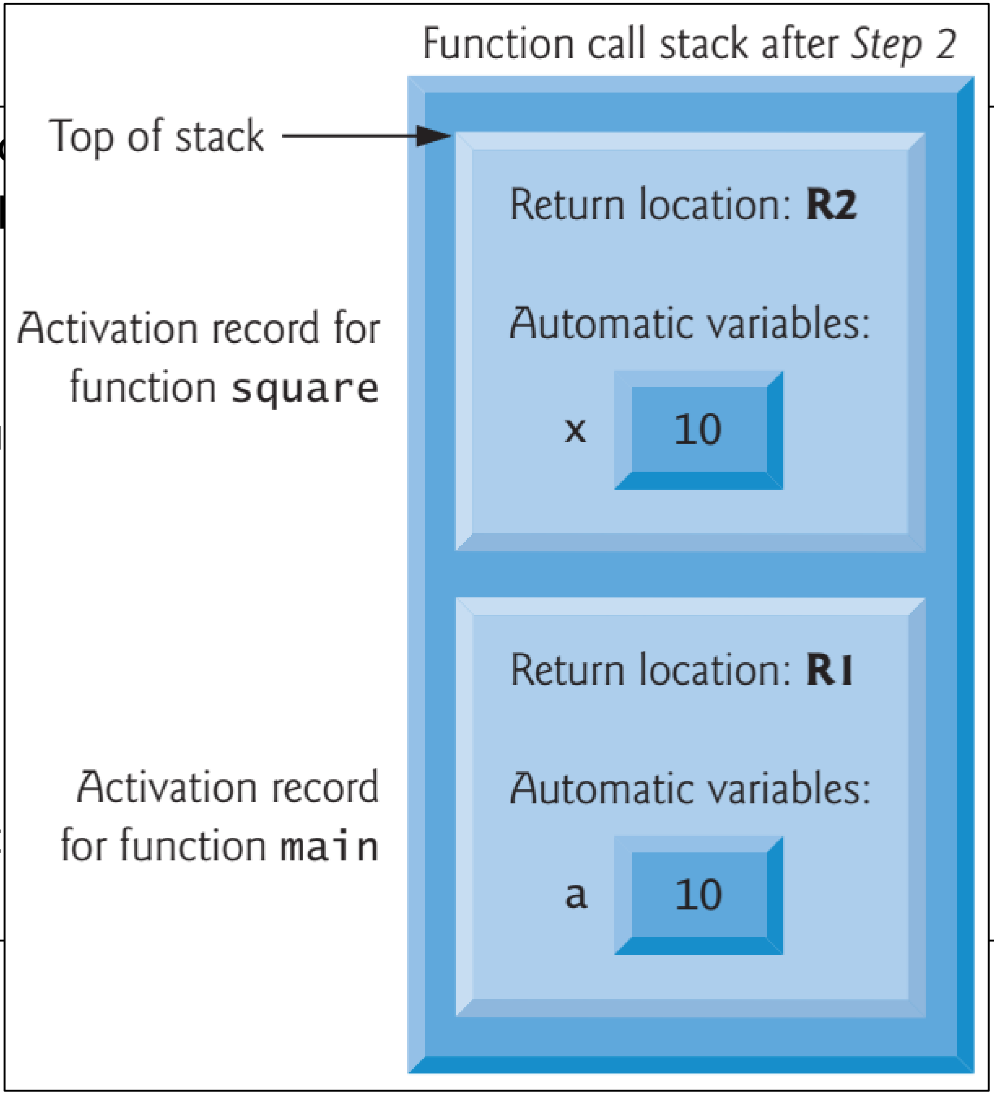
```
int square( int x )
{
   return x * x;
}
```

```
        cout << a << " squared: " << << endl;
}
```

Review - Function Call Stack and Stack Frames

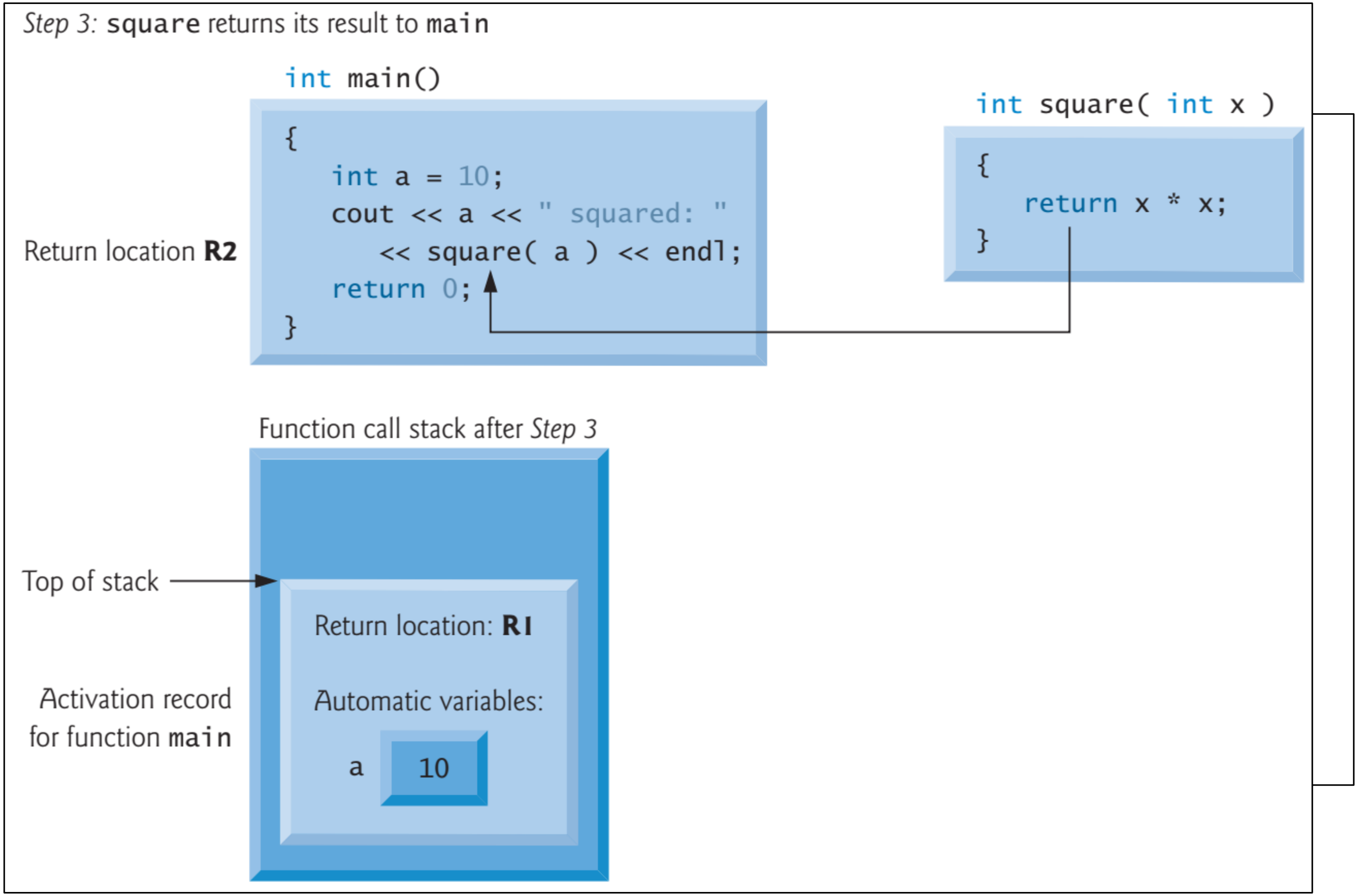
- Example

```
#include <i
using names

int square(
{
        retu
}

int main()
{
        int

        cout
}
```


Function call stack after *Step 2*
Top of stack
Activation record for function square
Return location: **R2**
Automatic variables:
x
10
Activation record for function main
Return location: **R1**
Automatic variables:
a
10

Participation Activity 6.8.2

Review - Function Call Stack and Stack Frames



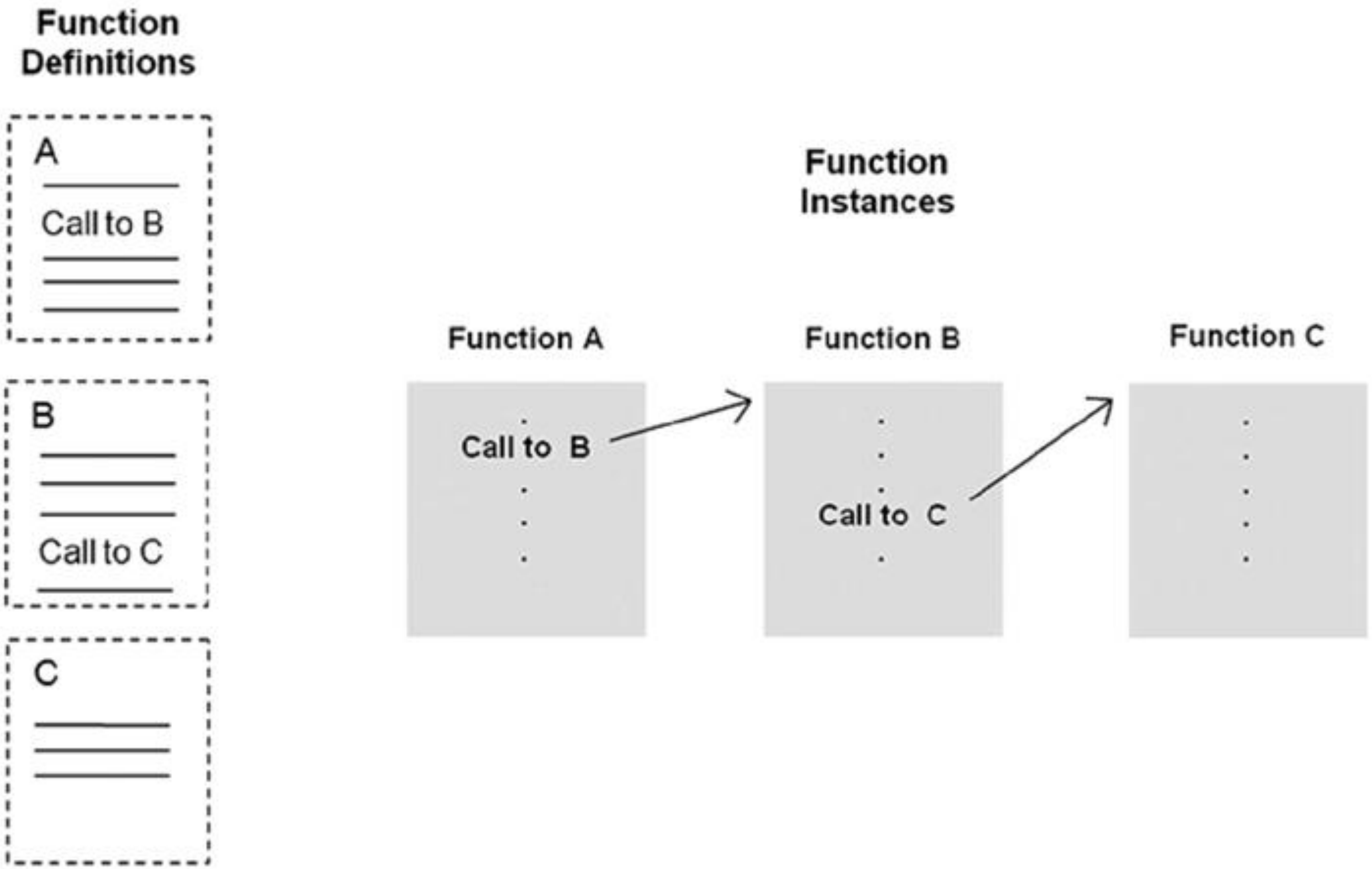
Step 3: **square** returns its result to **main**

```
int main()
{
    int a = 10;
    cout << a << " squared: "
       << square( a ) << endl;
    return 0;
}
```

```
int square( int x )
{
   return x * x;
}
```

Introduction

- General Function Call



Introduction

- Recursive Function Call
  - A function that calls itself

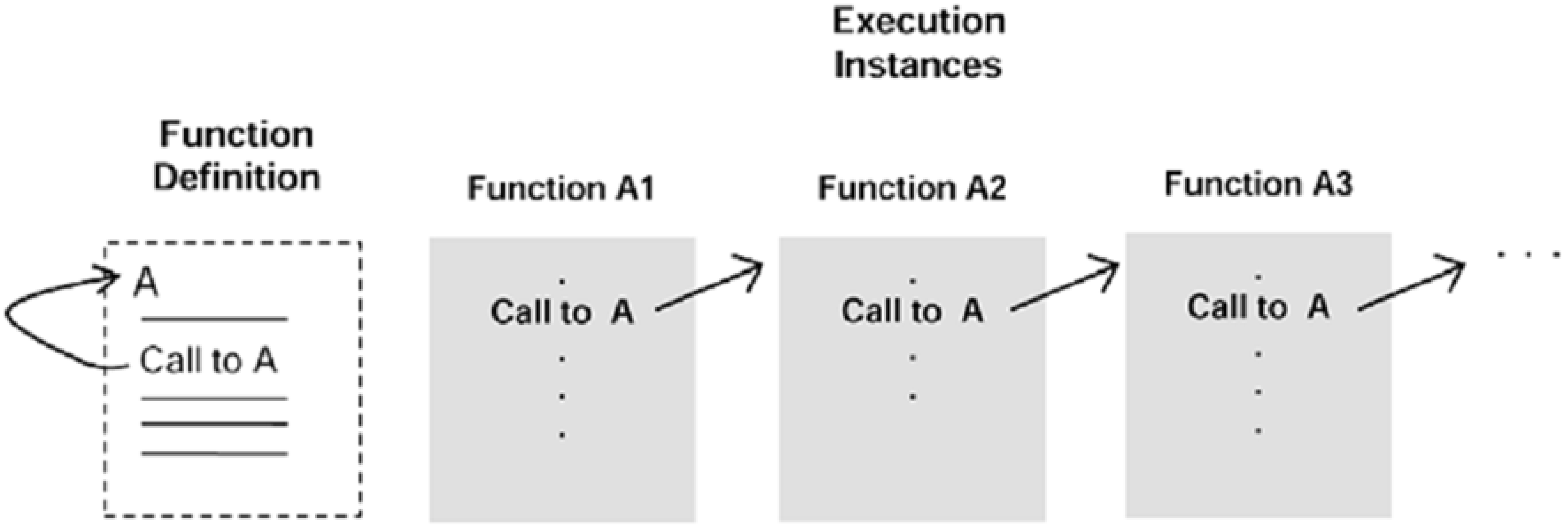


FIGURE 11-3 Recursive Function Execution Instances

Factorial Function

- Factorial
  - □ `n! = n × (n-1) × (n-2) × … × 2 × 1`
  - □ E.g., `n = 4`
`n! = 4! = 4 × 3 × 2 × 1`
- Factorial in for loop

```
unsigned long f(int n)
{
         int factorial = 1;
         int i;

         for (i = n; i >= 1; i--)
                  factorial *= i;

         return factorial;
}
```

Factorial Function

- Factorial
  - `n! = n × (n-1) × (n-2) × … × 2 × 1`
  - E.g.,
    ```
    n = 4
    n! = 4! = 4 × 3 × 2 × 1
    ```
- Define a function `f`
  - `f(n)      = n × (n-1) × (n-2) × … × 2 × 1`

Factorial Function

- Factorial
  - □ n! = n × (n-1) × (n-2) × … × 2 × 1
  - □ E.g.,
    n = 4
    n! = 4! = 4 × 3 × 2 × 1
- Define a function f
  - □ f(n) = n × (n-1) × (n-2) × … × 2 × 1
  - □ n = n-1
    f(n-1) = (n-1) × ((n-1)-1) × ((n-1)-2) × … × 2 × 1
    = (n-1) × (n-2) × (n-3) × … × 2 × 1

Factorial Function

- Factorial
  - $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$
  - E.g., $n = 4$

    $n! = 4! = 4 \times 3 \times 2 \times 1$
- Define a function f
  - $f(n) = n \times \underbrace{(n-1) \times (n-2) \times \ldots \times 2 \times 1}$
  - $n = n-1$

    $$\begin{aligned} f(n-1) &= (n-1) \times ((n-1)-1) \times ((n-1)-2) \times \ldots \times 2 \times 1 \\ &= (n-1) \times (n-2) \times (n-3) \times \ldots \times 2 \times 1 \end{aligned}$$

Factorial Function

- Factorial
  - $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$
  - E.g., $n = 4$

    $n! = 4! = 4 \times 3 \times 2 \times 1$
- Define a function f
  - $f(n) = n \times \underbrace{(n-1) \times (n-2) \times \ldots \times 2 \times 1}$
  - $n = n-1$

    $f(n-1) = (n-1) \times ((n-1)-1) \times ((n-1)-2) \times \ldots \times 2 \times 1$

    $= (n-1) \times (n-2) \times (n-3) \times \ldots \times 2 \times 1$
  - $f(n) = n \times f(n-1)$

Factorial Function

- Factorial
  - □ n! = n × (n-1) × (n-2) × … × 2 × 1
  - □ E.g., n = 4

    n! = 4! = 4 × 3 × 2 × 1
- Define a function f
  - □ $f(n) = n \times \underbrace{(n-1) \times (n-2) \times \dots \times 2 \times 1}_{}$
  - □ n = n-1

    $f(n-1) = (n-1) \times ((n-1)-1) \times ((n-1)-2) \times \dots \times 2 \times 1$

    $= (n-1) \times (n-2) \times (n-3) \times \dots \times 2 \times 1$
  - □ $f(n) = n \times f(n-1)$
- Function f in C/C++

```
unsigned long f(int n)
{
          return n * f(n-1);
}
```

Factorial Function

- Factorial
  - □ n! = n × (n-1) × (n-2) × … × 2 × 1
  - □ E.g.,　　n = 4
    　　　　n! = 4! = 4 × 3 × 2 × 1
- Define a function f
  - □ f(n)　　= n × (n-1) × (n-2) × … × 2 × 1
  - □ n = n-1
    f(n-1)　　= (n-1) × ((n-1)-1) × ((n-1)-2) × … × 2 × 1
    　　　　　= (n-1) × (n-2) × (n-3) × … × 2 × 1
  - □ f(n)　　= n × f(n-1)
- Function f in C/C++

```
unsigned long f(int n)
{
         return n * f(n-1);
}
```

→ n × (n-1) × (n-2) × … × 2 × 1 × 0 × -1 × -2 × …

Factorial Function

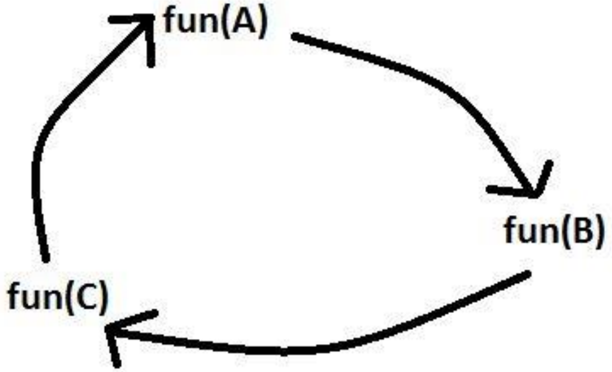
- Factorial
  - n! = n × (n-1) × (n-2) × … × 2 × 1
  - E.g., n = 4
    n! = 4! = 4 × 3 × 2 × 1
- Define a function f
  - $f(n) = n \times \underbrace{(n-1) \times (n-2) \times \ldots \times 2 \times 1}$
  - n = n-1
    $f(n-1) = (n-1) \times ((n-1)-1) \times ((n-1)-2) \times \ldots \times 2 \times 1$
    $= (n-1) \times (n-2) \times (n-3) \times \ldots \times 2 \times 1$
  - f(n) = n × f(n-1) if n > 1
    = 1 if n is 0
- Function f in C/C++

```
unsigned long f(int n)
{
        if (n == 0)
                return 1;
        else
                return n * f(n-1);
}
```

Recursion Types

Types of Recursions

- Direct Recursion
  - □ A function calls itself from within itself
  - □ Types
    - Tail recursion
    - Head recursion
    - Tree recursion
    - Nested recursion
- Indirect Recursion
  - □ A function call itself from more than one function call one another mutually

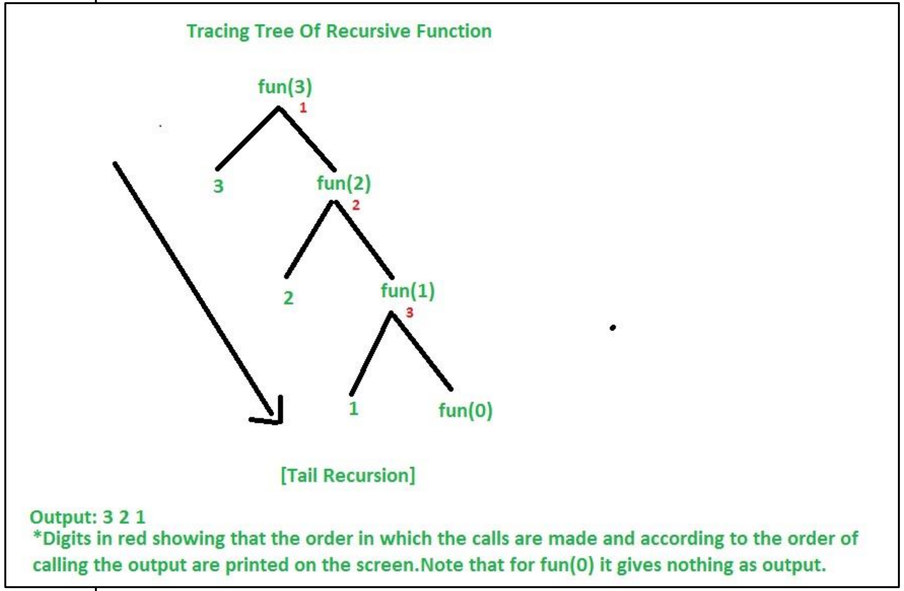


https://www.geeksforgeeks.org/types-of-recursions/

Tail Recursion

- A recursive function calling itself and that recursive call is the last statement in the function
- After that call the recursive function performs nothing
- Example: factorial number
- Example: count down

```
#include <iostream>
using namespace std;

void fun(int n)
{
    if (n > 0) {
        cout << n << " ";

        // Last statement in the function
        fun(n - 1);
    }
}

int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```



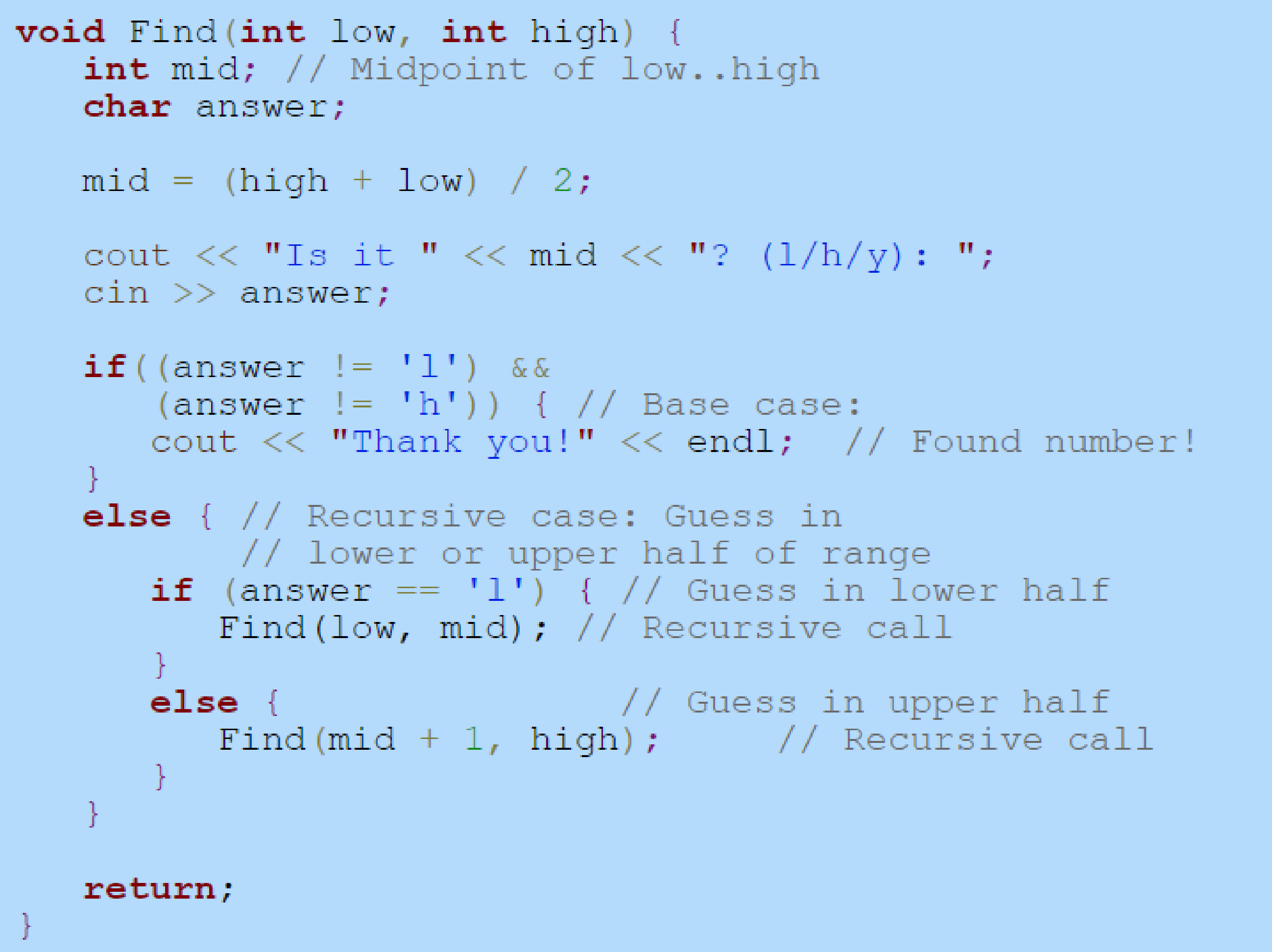
Output: 3 2 1
*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

https://www.geeksforgeeks.org/types-of-recursions/

Participation Activity 7.2.2

Tail Recursion

- Example: Binary Search

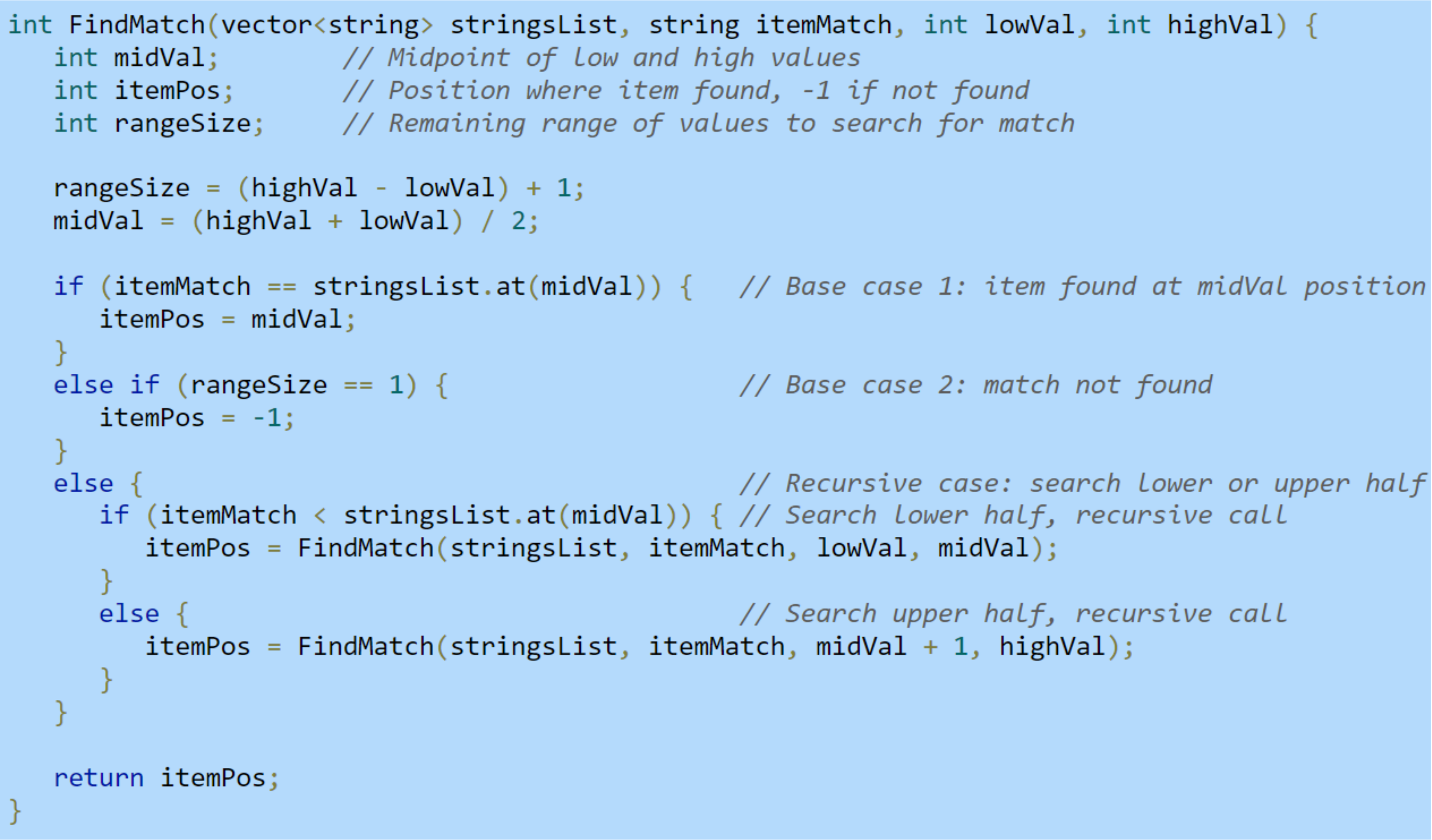


```
void Find(int low, int high) {
   int mid; // Midpoint of low..high
   char answer;

   mid = (high + low) / 2;

   cout << "Is it " << mid << "? (l/h/y): ";
   cin >> answer;

   if((answer != 'l') &&
      (answer != 'h')) { // Base case:
      cout << "Thank you!" << endl;  // Found number!
   }
   else { // Recursive case: Guess in
          // lower or upper half of range
      if (answer == 'l') { // Guess in lower half
         Find(low, mid); // Recursive call
      }
      else {               // Guess in upper half
         Find(mid + 1, high);     // Recursive call
      }
   }

   return;
}
```

Figure 7.3.2

Tail Recursion

- Example: Binary Search

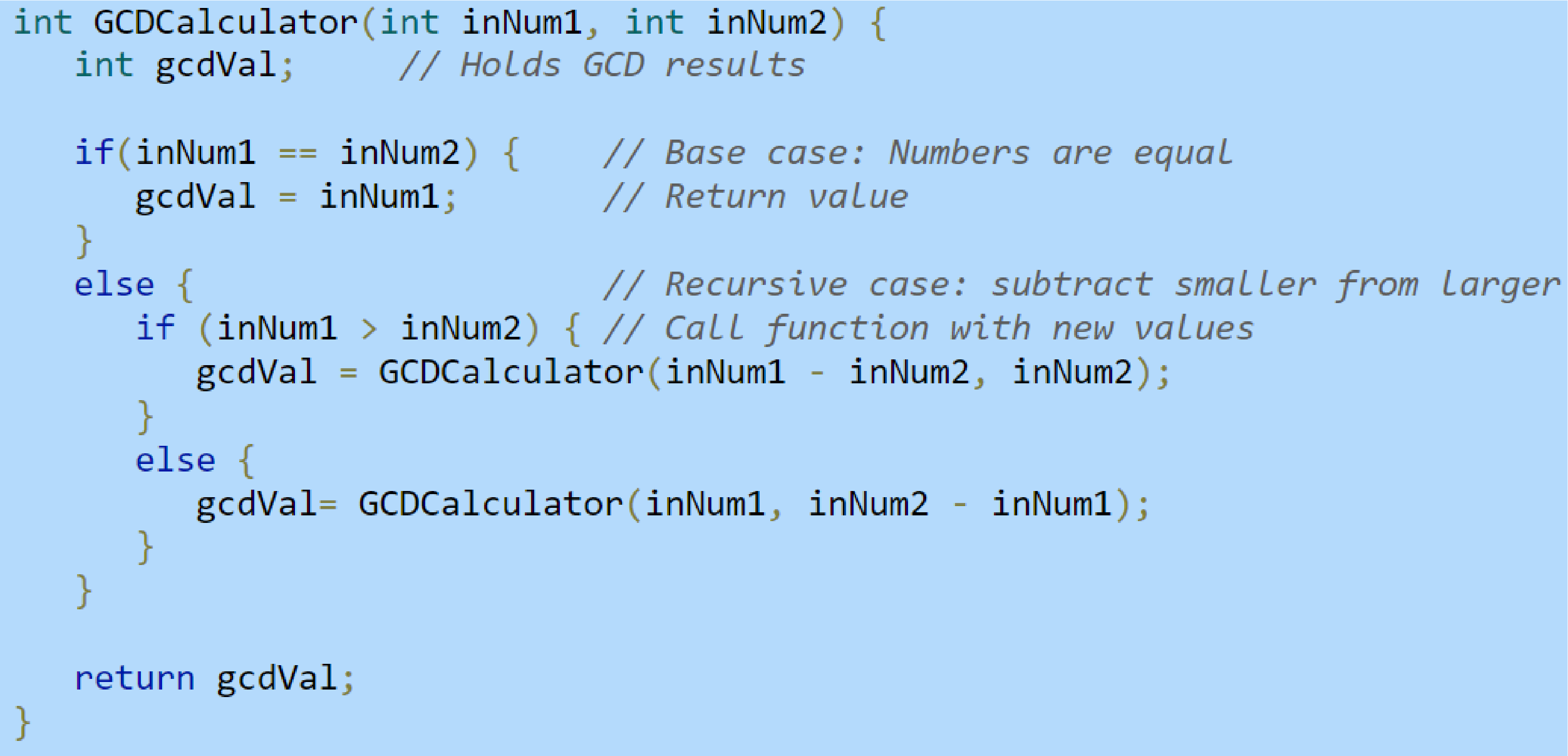


```
int FindMatch(vector<string> stringsList, string itemMatch, int lowVal, int highVal) {
   int midVal;        // Midpoint of low and high values
   int itemPos;       // Position where item found, -1 if not found
   int rangeSize;     // Remaining range of values to search for match

   rangeSize = (highVal - lowVal) + 1;
   midVal = (highVal + lowVal) / 2;

   if (itemMatch == stringsList.at(midVal)) {   // Base case 1: item found at midVal position
      itemPos = midVal;
   }
   else if (rangeSize == 1) {                   // Base case 2: match not found
      itemPos = -1;
   }
   else {                                       // Recursive case: search lower or upper half
      if (itemMatch < stringsList.at(midVal)) { // Search lower half, recursive call
         itemPos = FindMatch(stringsList, itemMatch, lowVal, midVal);
      }
      else {                                    // Search upper half, recursive call
         itemPos = FindMatch(stringsList, itemMatch, midVal + 1, highVal);
      }
   }

   return itemPos;
}
```

Figure 7.6.2

Tail Recursion

- Example: Calculating greatest common divisor

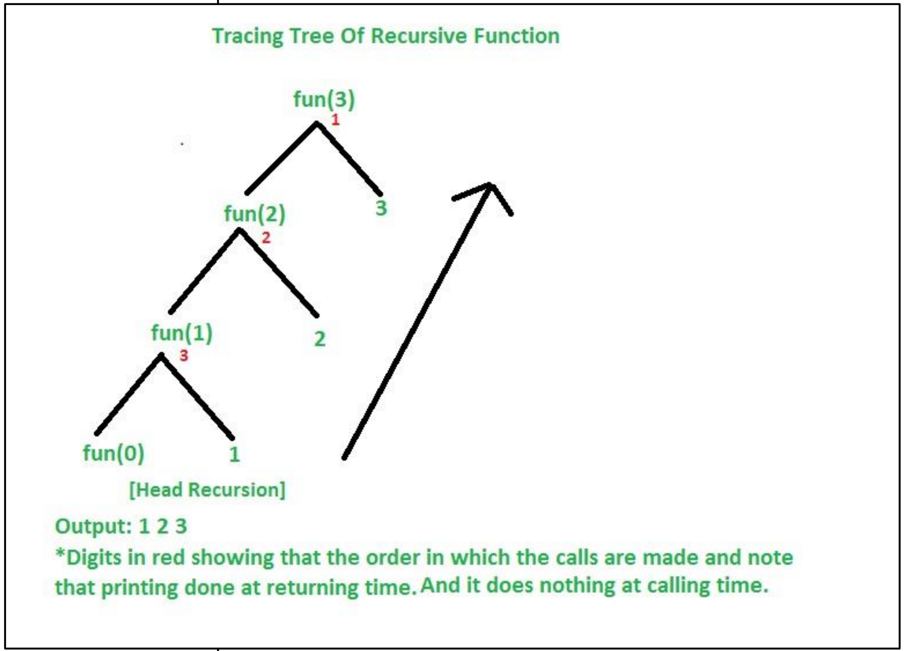


```
int GCDCalculator(int inNum1, int inNum2) {
   int gcdVal;     // Holds GCD results

   if(inNum1 == inNum2) {    // Base case: Numbers are equal
      gcdVal = inNum1;       // Return value
   }
   else {                    // Recursive case: subtract smaller from larger
      if (inNum1 > inNum2) { // Call function with new values
         gcdVal = GCDCalculator(inNum1 - inNum2, inNum2);
      }
      else {
         gcdVal= GCDCalculator(inNum1, inNum2 - inNum1);
      }
   }

   return gcdVal;
}
```

Head Recursion

- A recursive function calling itself and that recursive call is the first statement in the function
- There’s no statement, no operation before the call
- Example: Counting

```
#include <iostream>
using namespace std;

void fun(int n)
{
    if (n > 0) {

        // First statement in the function
        fun(n - 1);

        cout << " "<< n;
    }
}

int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```


Tracing Tree Of Recursive Function

[Head Recursion]

Output: 1 2 3

*Digits in red showing that the order in which the calls are made and note that printing done at returning time. And it does nothing at calling time.

https://www.geeksforgeeks.org/types-of-recursions/

Head Recursion

- Example: Print the data in the linked list reversely

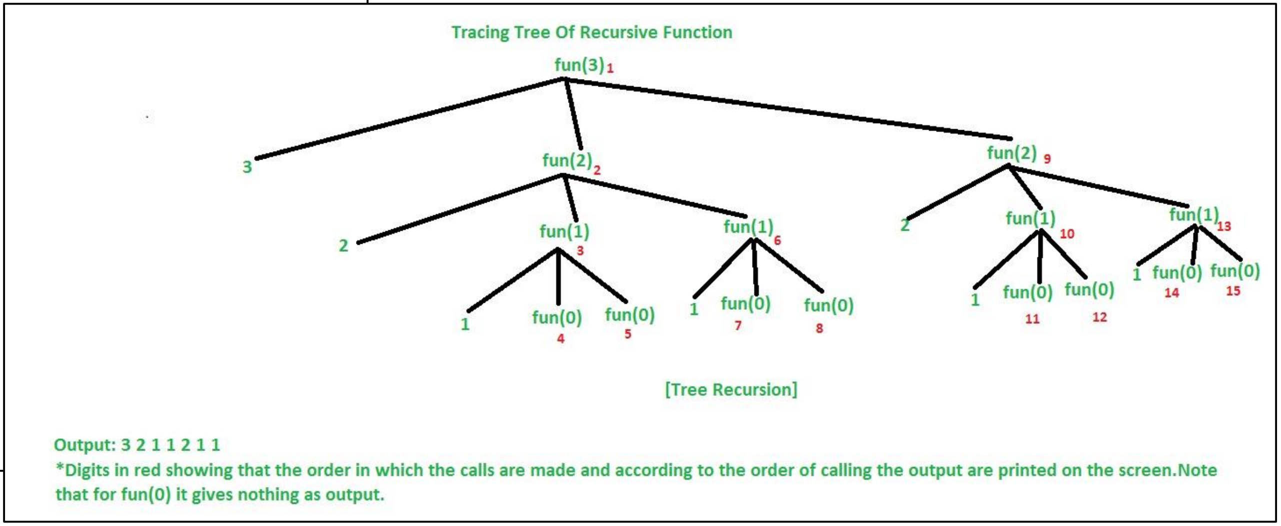
```
void reverse_print_list(struct node_t *head)
{
        if (!head)
                return;
        reverse_print_list(head->next);
        print_node(head);
}
```

Written in C language

Tree Recursion

- A recursive function calling itself for more than one time
- Example:

```
#include <iostream>
using namespace std;

void fun(int n)
{
    if (n > 0)
    {
        cout << " " << n;

        // Calling once
        fun(n - 1);

        // Calling twice
        fun(n - 1);
    }
}

int main()
{
    fun(3);
    return 0;
}
```



Tracing Tree Of Recursive Function

[Tree Recursion]

Output: 3 2 1 1 2 1 1

*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

https://www.geeksforgeeks.org/types-of-recursions/

Tree Recursion

- Example: Fibonacci Sequence

  -
    ```
    0  1    2      3      4      5      6      7      8
    0, 1,    1,     2,     3,     5,     8,     13,    21, …
    0, 1, (0+1), (1+1), (1+2), (2+3), (3+5), (5+8), (8+13), …
    ```

  -
    ```
    fib(n)   = fib(n-1) + fib(n-2)        otherwise
             = 1                          if n = 1
             = 0                          if n = 0
    ```
  - Function fib in C/C++

```
unsigned fib(int m)
{
        if (m == 0)
                return 0;
        else if (m == 1)
                return 1;
        else
                return fib(m - 1) + fib(m - 2);
}
```

Tree Recursion

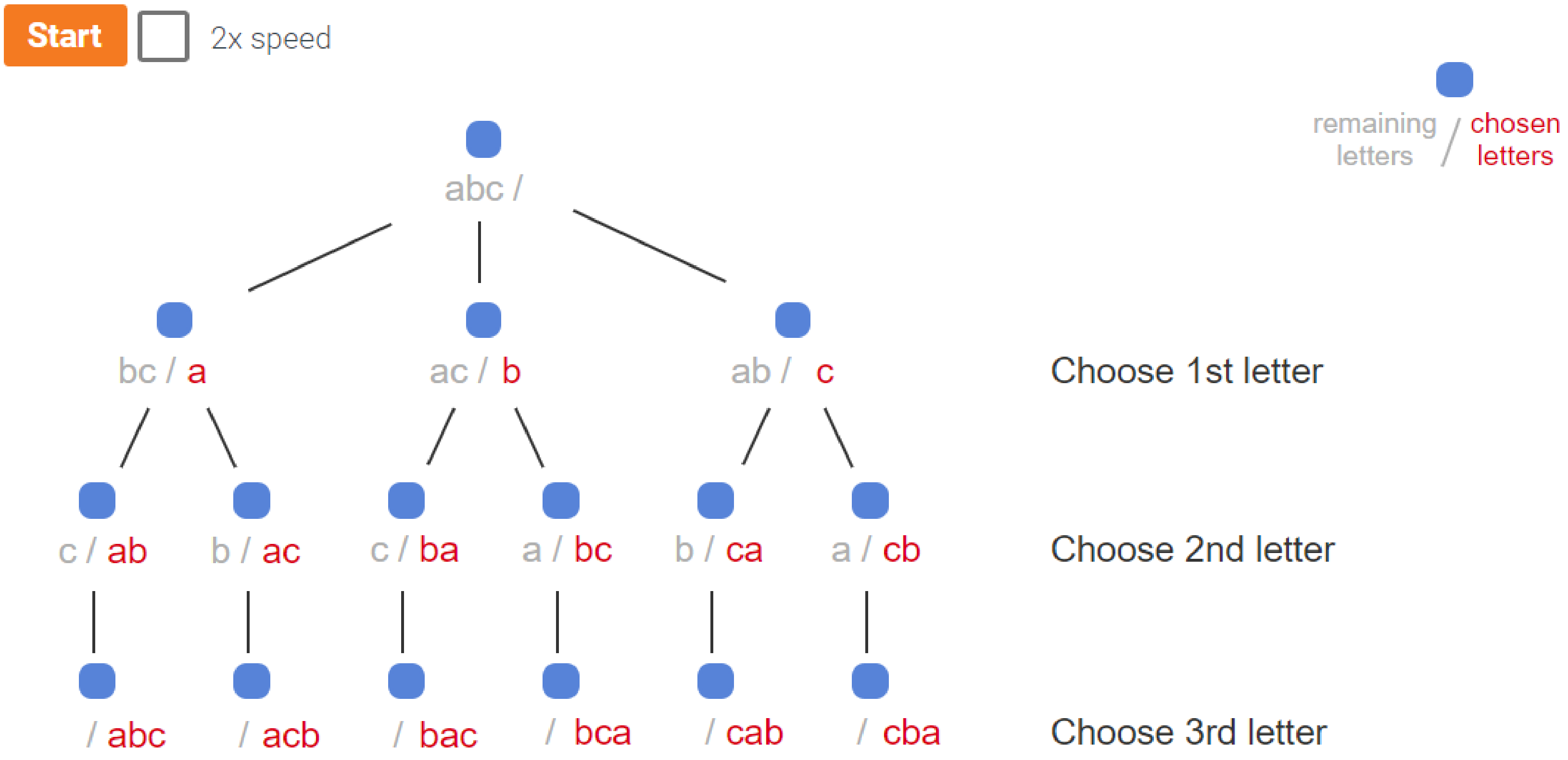
- Example: Merge Sort

```
MergeSort ($A, l, r$)
if $r > l$
    $m = (l + r) / 2$            //find the middle index of the list
    MergeSort ($A, l, m$)
    MergeSort ($A, m + 1, r$)
    Merge ($A, l, m, r$)
```

Participation Activity 7.7.1

Tree Recursion

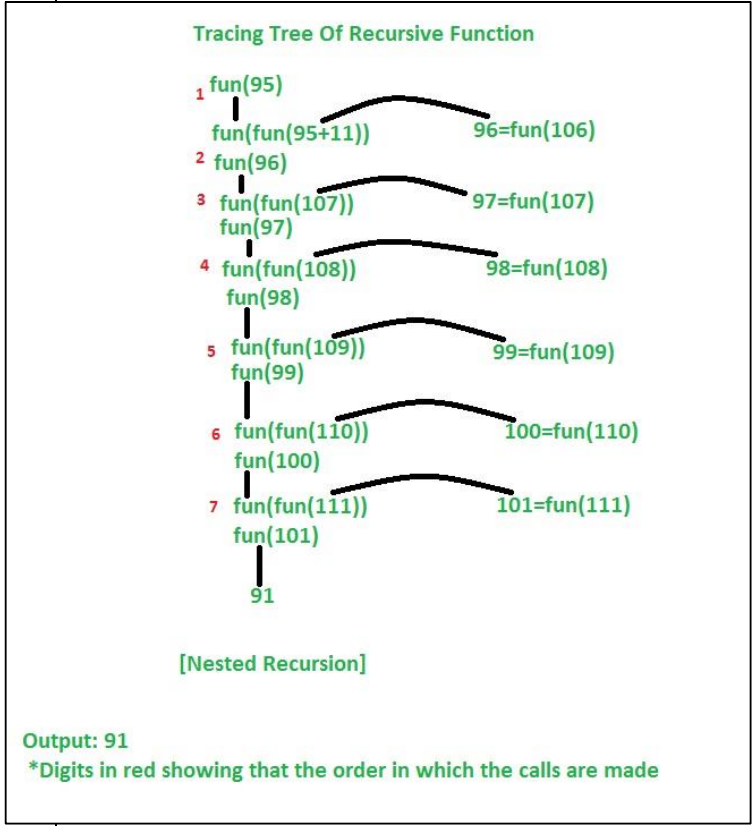
- Example: Permutation



Nested Recursion

- A recursive function will pass the parameter as a recursive call → recursion inside recursion
- Example:

```
#include <iostream>
using namespace std;

int fun(int n)
{
    if (n > 100)
        return n - 10;

    // A recursive function passing parameter
    // as a recursive call or recursion inside
    // the recursion
    return fun(fun(n + 11));
}

int main()
{
    int r;
    r = fun(95);

    cout << " " << r;

    return 0;
}
```



[Nested Recursion]

Output: 91

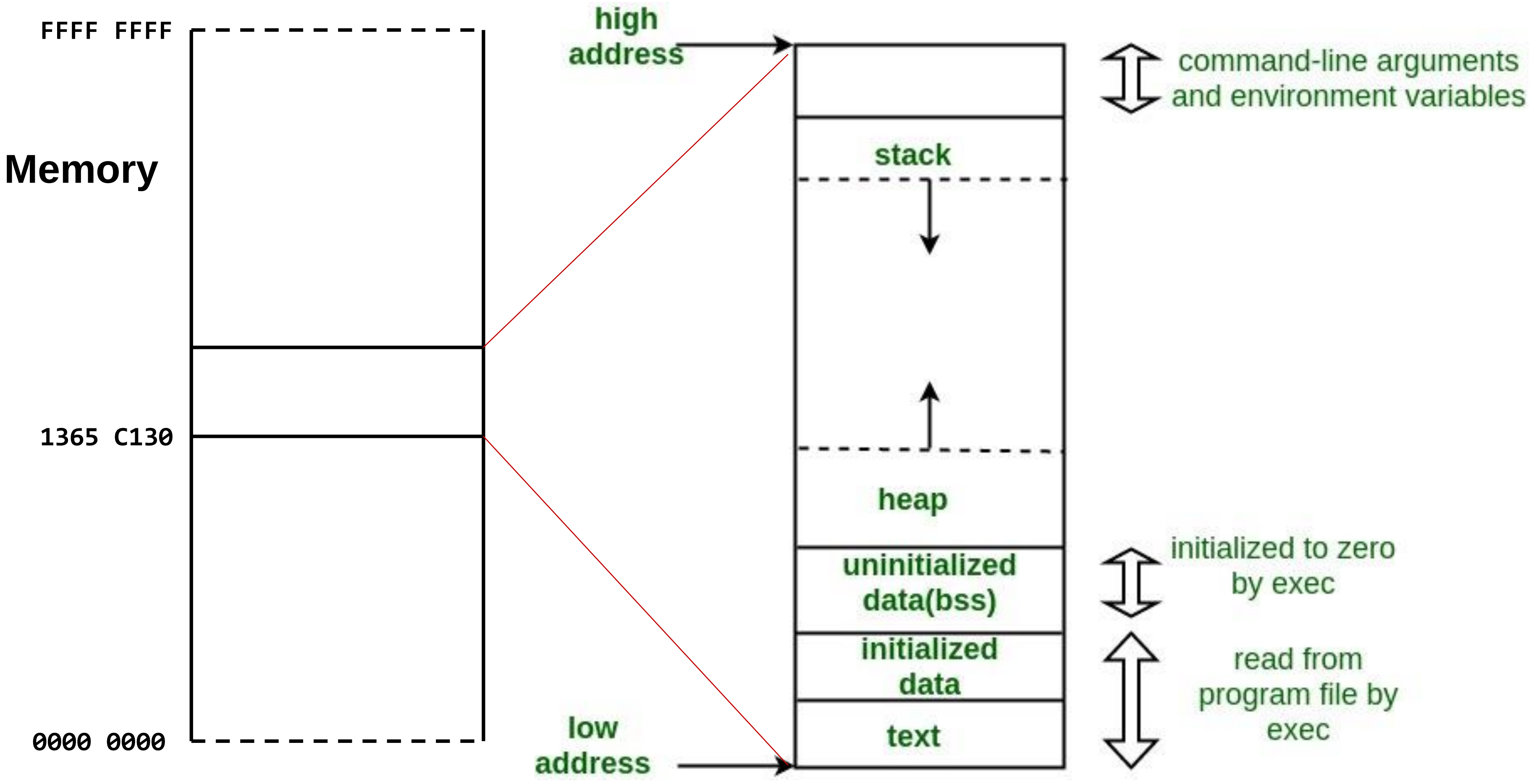
*Digits in red showing that the order in which the calls are made

https://www.geeksforgeeks.org/types-of-recursions/

Stack Overflow

Memory Layout

- Memory
  - Variables correspond to locations in the computer's memory



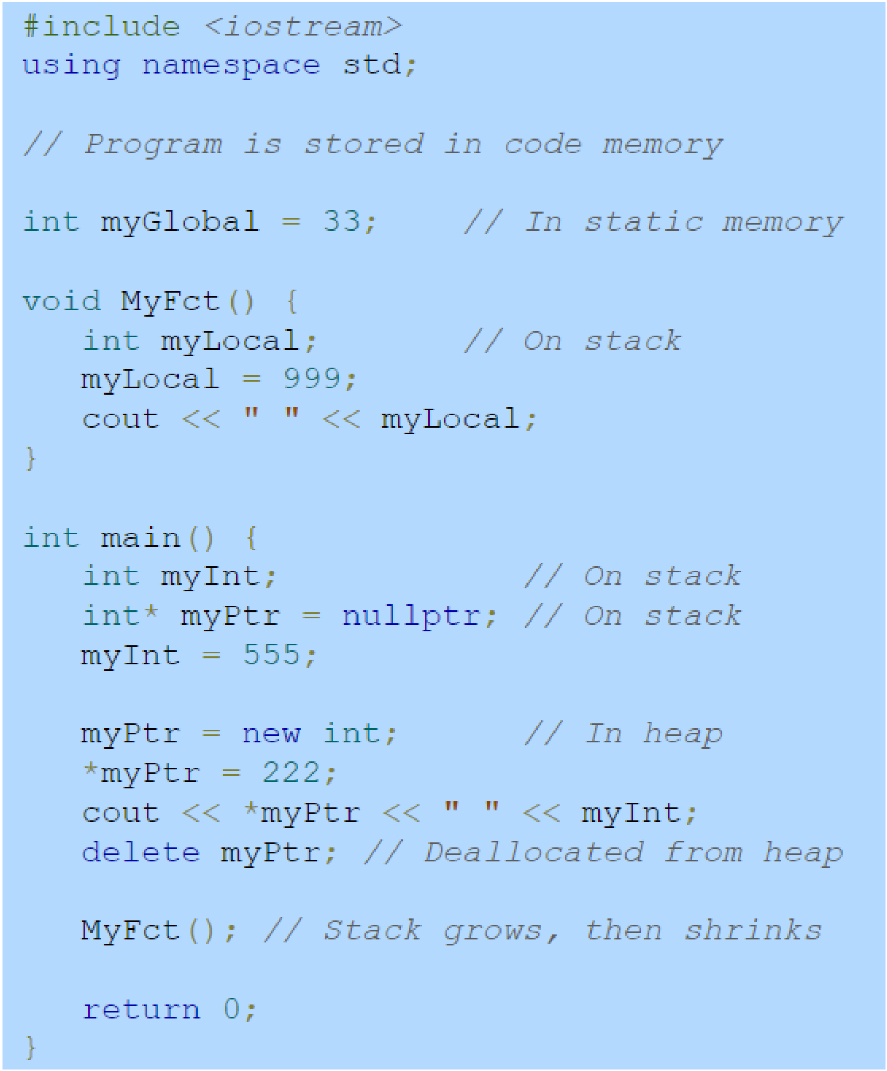
https://www.geeksforgeeks.org/memory-layout-of-c-program/

Participation Activity 10.7.1

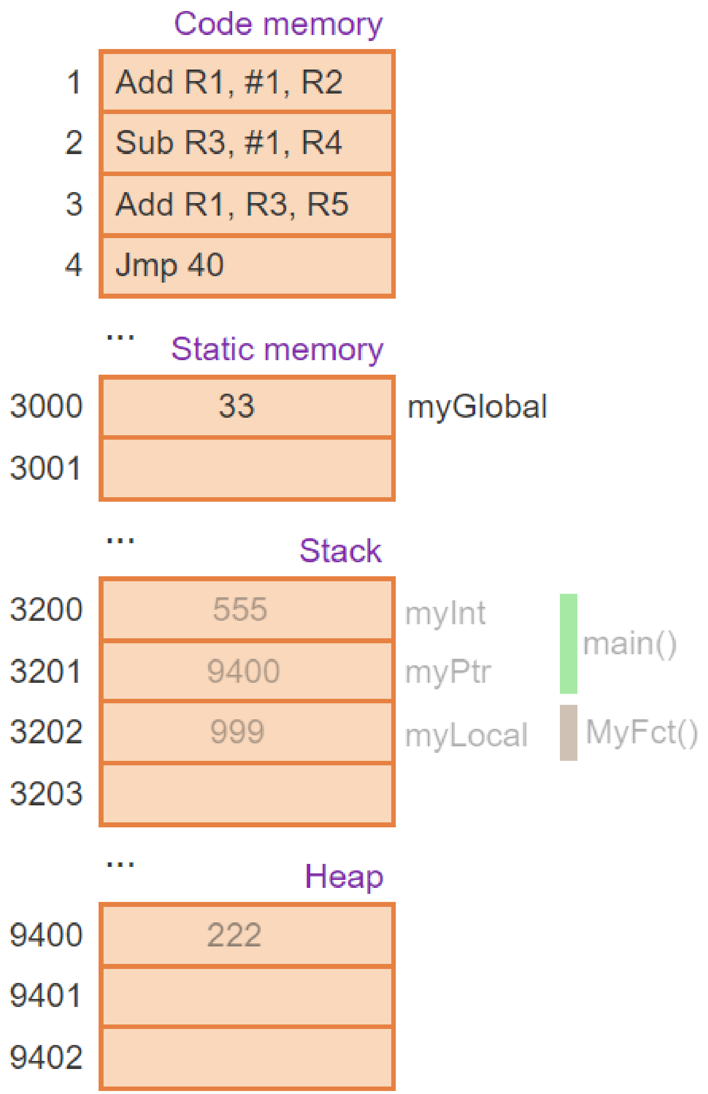
Memory Layout

- Code/Text
  - Program instructions
- Static memory
  - Global variables (variables declared outside any function)
  - Static local variables (variables declared inside functions starting with the keyword "static")
  - Are allocated once and stay in the same memory location for the duration of a program's execution.
- Stack
  - Function's local variables are allocated during a function call
  - A function call adds local variables to the stack, and a return removes them, like adding and removing dishes from a pile
- Heap
  - The region for the dynamic memory allocation. (will be introduced later)

Memory Layout



```
#include <iostream>
using namespace std;

// Program is stored in code memory

int myGlobal = 33;     // In static memory

void MyFct() {
   int myLocal;        // On stack
   myLocal = 999;
   cout << " " << myLocal;
}

int main() {
   int myInt;                // On stack
   int* myPtr = nullptr; // On stack
   myInt = 555;

   myPtr = new int;          // In heap
   *myPtr = 222;
   cout << *myPtr << " " << myInt;
   delete myPtr; // Deallocated from heap

   MyFct(); // Stack grows, then shrinks

   return 0;
}
```



Code memory

1 Add R1, #1, R2

2 Sub R3, #1, R4

3 Add R1, R3, R5

4 Jmp 40

...

Static memory

3000 33 myGlobal

3001

...

Stack

3200 555 myInt

3201 9400 myPtr

3202 999 myLocal

3203

main()

MyFct()

...

Heap

9400 222

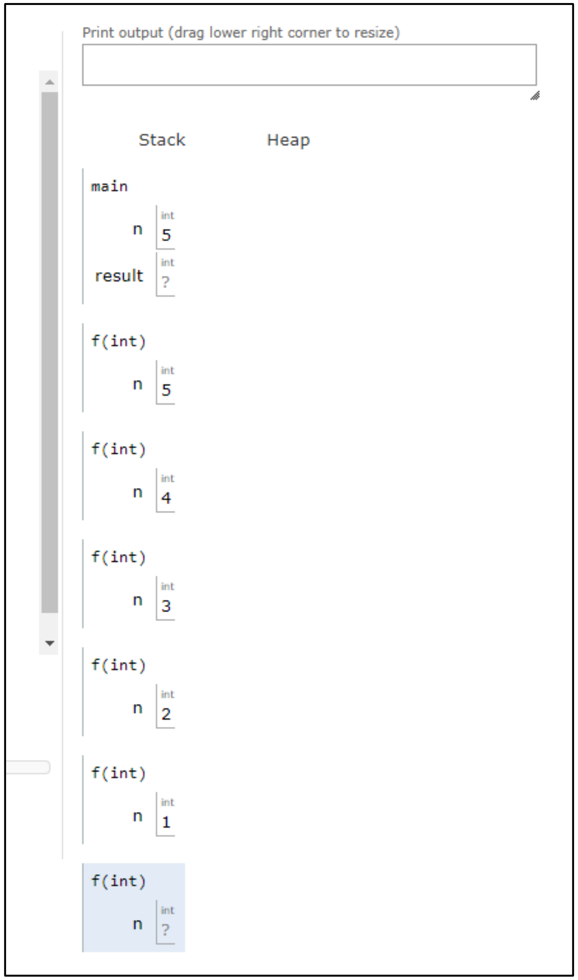
9401

9402

Memory Allocation in Recursion Functions

- Execute the following code at https://pythontutor.com/visualize.html

```
#include <iostream>
using namespace std;


unsigned long f(int n)
{
    if (n == 0)
        return 1;
    else
        return n * f(n-1);
}

int main()
{
    int n = 5;
    int result = f(n);

    cout << result;

    return 0;
}
```



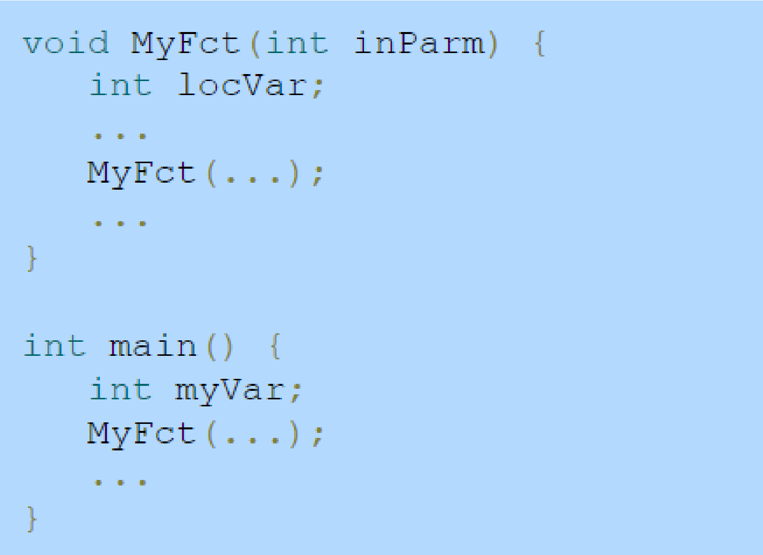
Print output (drag lower right corner to resize)

```
Stack        Heap

main
       n   int 5
  result   int ?

f(int)
       n   int 5

f(int)
       n   int 4

f(int)
       n   int 3

f(int)
       n   int 2

f(int)
       n   int 1

f(int)
       n   int ?
```

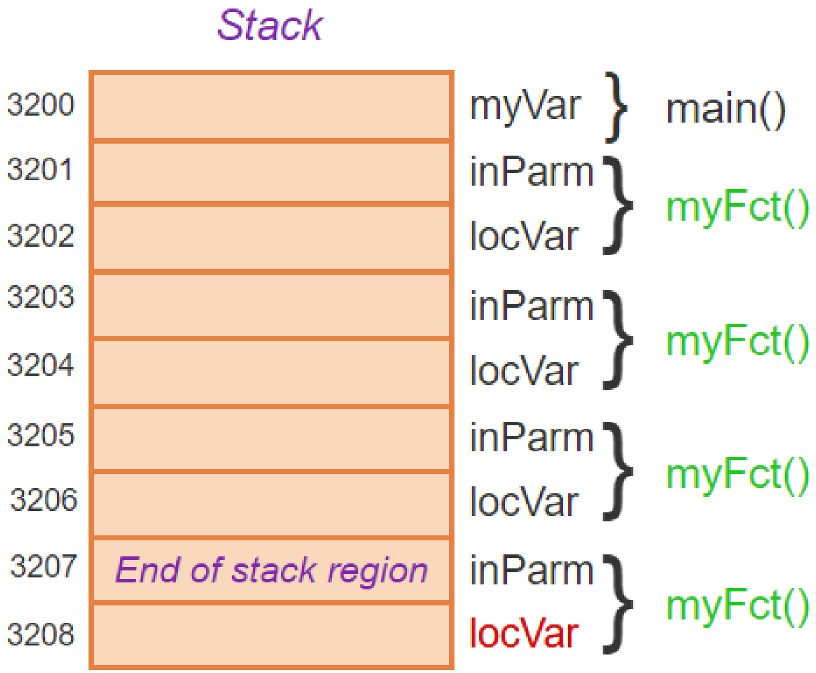
Participation Activity 7.8.1

Stack Overflow

- A stack frame extends beyond the memory region allocated for stack
- Usually causes the program to crash and report an error like: segmentation fault, access violation, or bad access.



```
void MyFct(int inParm) {
   int locVar;
   ...
   MyFct(...);
   ...
}

int main() {
   int myVar;
   MyFct(...);
   ...
}
```



Stack
3200
3201
3202
3203
3204
3205
3206
3207
3208
End of stack region
myVar
main()
inParm
locVar
myFct()
inParm
locVar
myFct()
inParm
locVar
myFct()
inParm
locVar
myFct()