# Recursion

## Slides

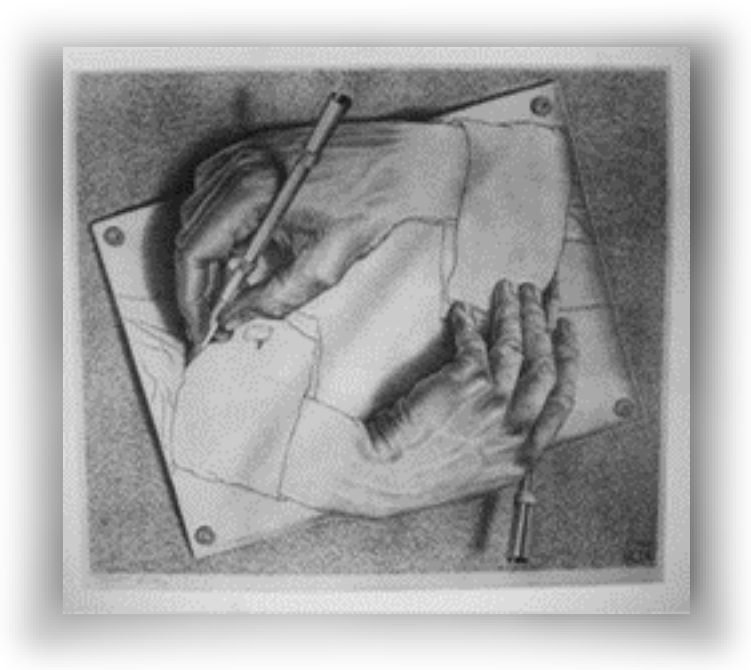
# Definitions

Recursion

- see **Recursion**
- a process in which the result of each repetition is dependent upon the result of the next repetition.
- Simplifies program structure at a cost of function calls

Hofstadter's Law

- *"It always takes longer than you expect, even when you take into account Hofstadter's Law."*



***Sesquipedalian***

- *a person who uses words like sesquipedalian.*

***Yogi Berra***

- *"Its déjà vu all over again."*

# Simple Recursion

A procedure or function which calls itself is a recursive routine.

Consider the following function, which computes

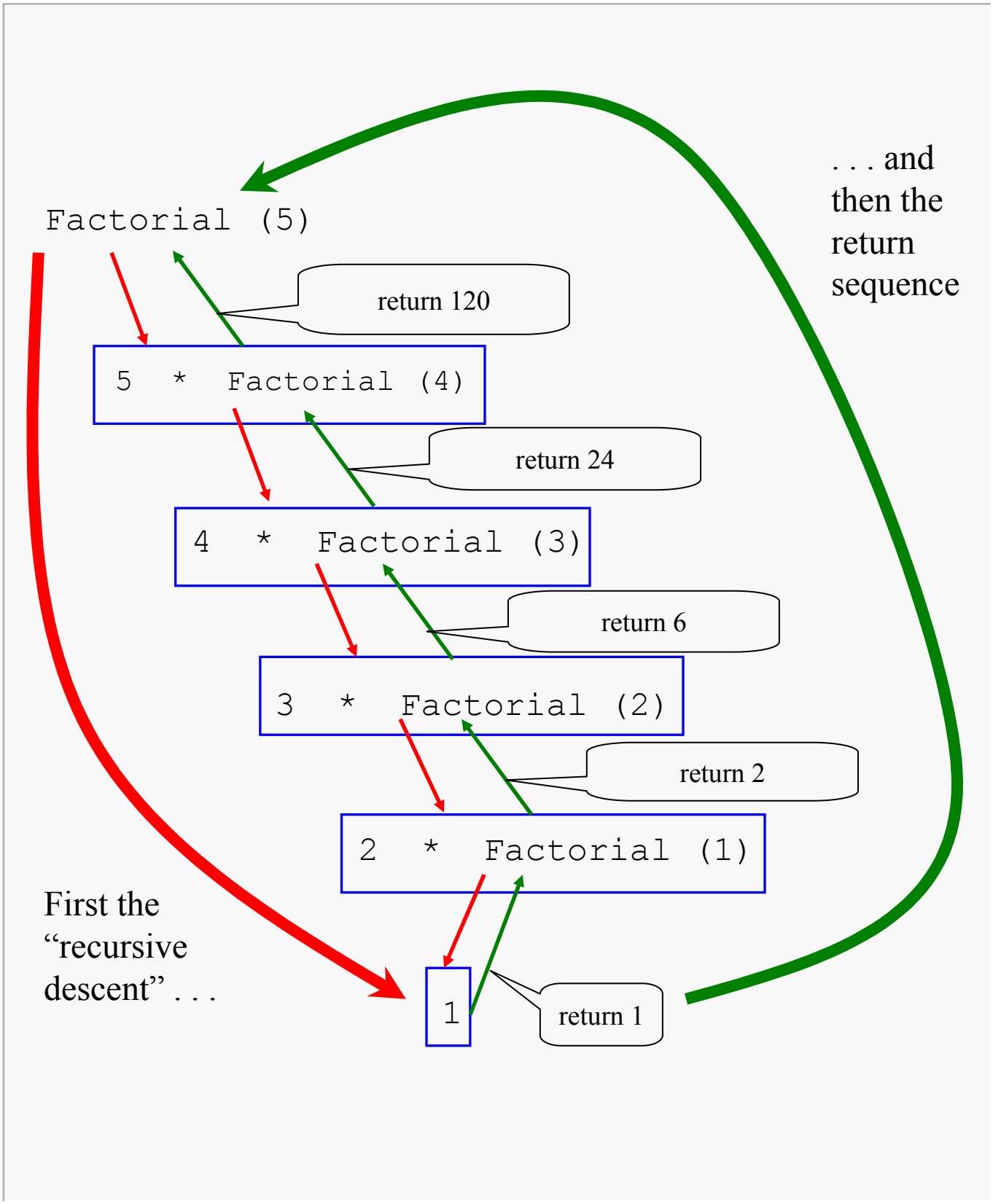
N! = 1 * 2 * ... * N

```
int Factorial(int n) {
  int Product = 1,
      Scan    = 2;

  while ( Scan <=  n ) {
    Product  =  Product  *  Scan ;
    Scan = Scan + 1 ;
  }
  return (Product) ;
}
```

Now consider a recursive version of `Factorial`:

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

# Recursive Execution Trace



# Recursion Attributes

- Every recursive algorithm can be implemented non-recursively.

  recursion <==> iteration

- Eventually, the routine must not call itself, allowing the code to "back out".

- Recursive routines that call themselves continuously are termed:

  infinite recursion <==> infinite loop

- Problem with this recursive factorial implementation?

  Negative numbers!

- Recursion is inefficient at runtime.

# Recursive Array Summation

Here is a recursive function that takes an array of integers and computes the sum of the elements:

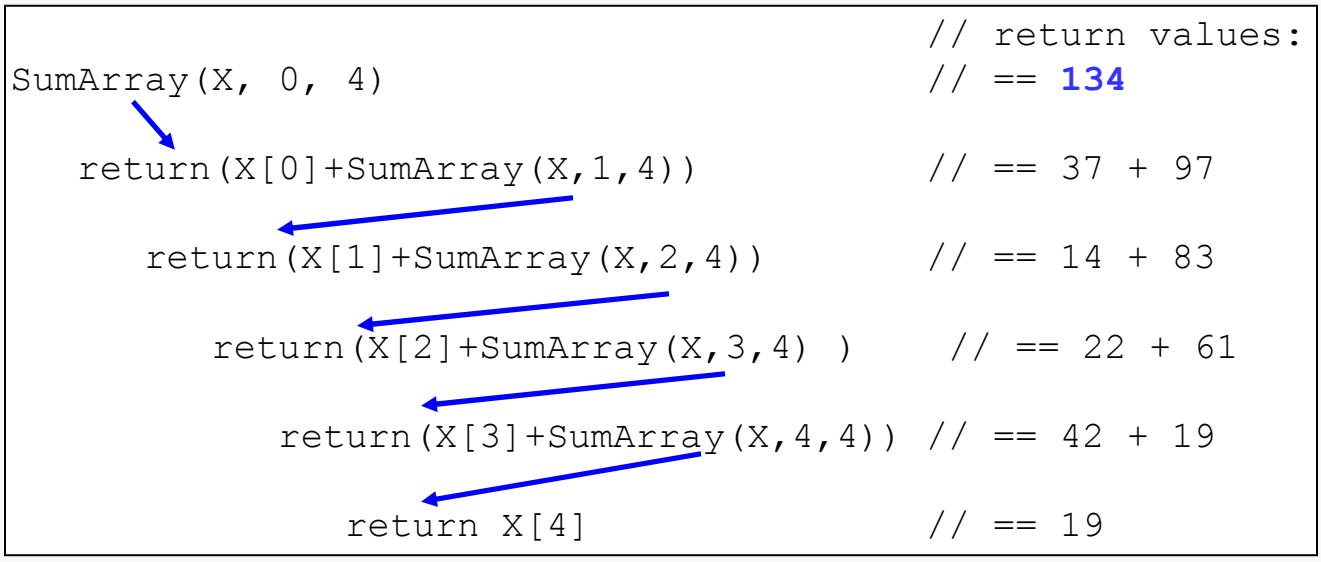
```
// X[]      array of integers to be summed
// Start    start summing at this index . . .
// Stop     . . . and stop summing at this index
//
int SumArray(const int X[], int Start, int Stop) {

                                          // error check
   if (Start > Stop || Start < 0 || Stop < 0)
      return 0;
   else if (Start == Stop)                // base case
      return X[Stop];
   else                                   // recursion
      return (X[Start] + SumArray(X, Start + 1, Stop));
}
```

# Recursive Array Summation Trace

The call:

```
const int Size = 5;
int X[Size] = {37, 14, 22, 42, 19};
SumArray(X,0,Size- 1);// note Stop is last valid index
```

would result in the recursive trace:



# Coding Recursively

## Mathematical Induction Model

- Solve the trivial "base" case(s).
- Restate general case in 'simpler' or 'smaller' terms of itself.

## Array Sum Example

- Determine the size of a single linked list.

Base Case : array size =1, sum = the element

General Case : first element + Sum(Rest of Array)

```
// X[]      array of integers to be summed
// Start    start summing at this index . . .
// Stop     . . . and stop summing at this index
//
int SumArray(const int X[], int Start, int Stop) {

                                          // error check
   if (Start > Stop || Start < 0 || Stop < 0)
      return 0;
   else if (Start == Stop)                // base case
      return X[Stop];
   else                                   // recursion
      return (X[Start] + SumArray(X, Start + 1, Stop));
}
```

**Example of “tail recursion” (going up recursion).**
**Tail recursive functions are characterized by the recursive call being the last statement in the function, (can easily be replaced by a loop).**

# Recursive Design

Problem:

- Code a function void intComma( long ) that outputs the integer argument comma separated :
- e.g.,

  the call: intComma ( 123456789) ;

  displays: 123,456,789

Top-Down Design

```
void intComma ( long  num ) {
  if (num is less than 1000)
  display num
else
  display comma separated digits above 1000
  display comma
  display digits below 1000
}
```

Code

```
void intComma ( long num ) {
   if (num < 1000)
      cout << setw(3) << num;
   else {
      intComma(num / 1000);
      cout << ',' << setw(3) << num % 1000;
   }
}
```

**Consider:**

**intComma( 123456789 ) ;**
**intComma( 1001 ) ;**

# Avoiding Pitfalls

## General Solution

```
void intComma ( long  num ) {

   if (num < 0) {          // display sign for negatives
      cout << '-';
      num = -num;
   }

   if (num < 1000)
      cout << setw(3) << num;
   else {
      intComma(num / 1000);
      cout << ',';                    // display digits
      num = num % 1000;               // separately
      cout << (num / 100);            // for zeroes
      num = num % 100;
      cout << (num / 10) << (num % 10);
   }
}
```

**Example of "going down" (head) recursion**

## Trace intComma(9087605430);

intComma(9087605430)
= intComma(9087605) and • • •
= intComma(9087) and • • •
= intComma(9) and • • •
= intComma(9)

**output**

**9**

**9,087**

**9,087,605**

**9,087,605,430**

**alternatively**

```
string prefix =
       (num < 10 ) ? "00" :
       (num < 100) ?  "0" : "";
cout << prefix << num;
```

# Middle Decomposition

Problem:

- Given an array of integers of n+1 elements code a function to return the index of the maximum value in the array.

Solution:

- Check if the middle element is the largest if so return its index otherwise return the index of either the largest element in the lower half or the largest element in the upper half, whichever is the larger of the two.

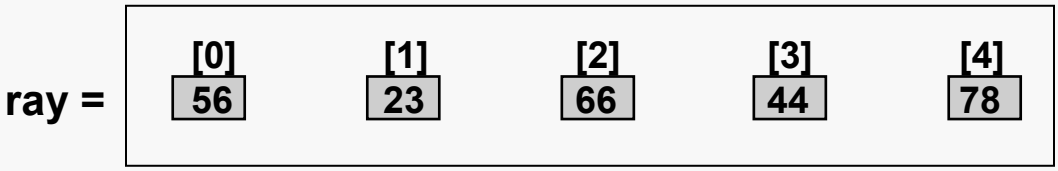
```
int rMax(const int ray[], int start, int end ) {

  const int Unknown = -1;

  int mid, h1max, h2max;

  if (end < start) return Unknown;

  mid = (start + end) / 2;
  h1max = rMax(ray, start, mid-1); //left half
  if (h1max == Unknown) h1max = start;
  h2max = rMax(ray, mid+1, end);   //right half
  if (h2max == Unknown) h2max = end;
  if ( (ray[mid] >= ray[h1max]) &&
       (ray[mid] >= ray[h2max]) )
   return mid;
 else
   return( (ray[h1max] > ray[h2max]) ?
                        h1max : h2max );
}
```

**Example of "splitting into halves" recursion**

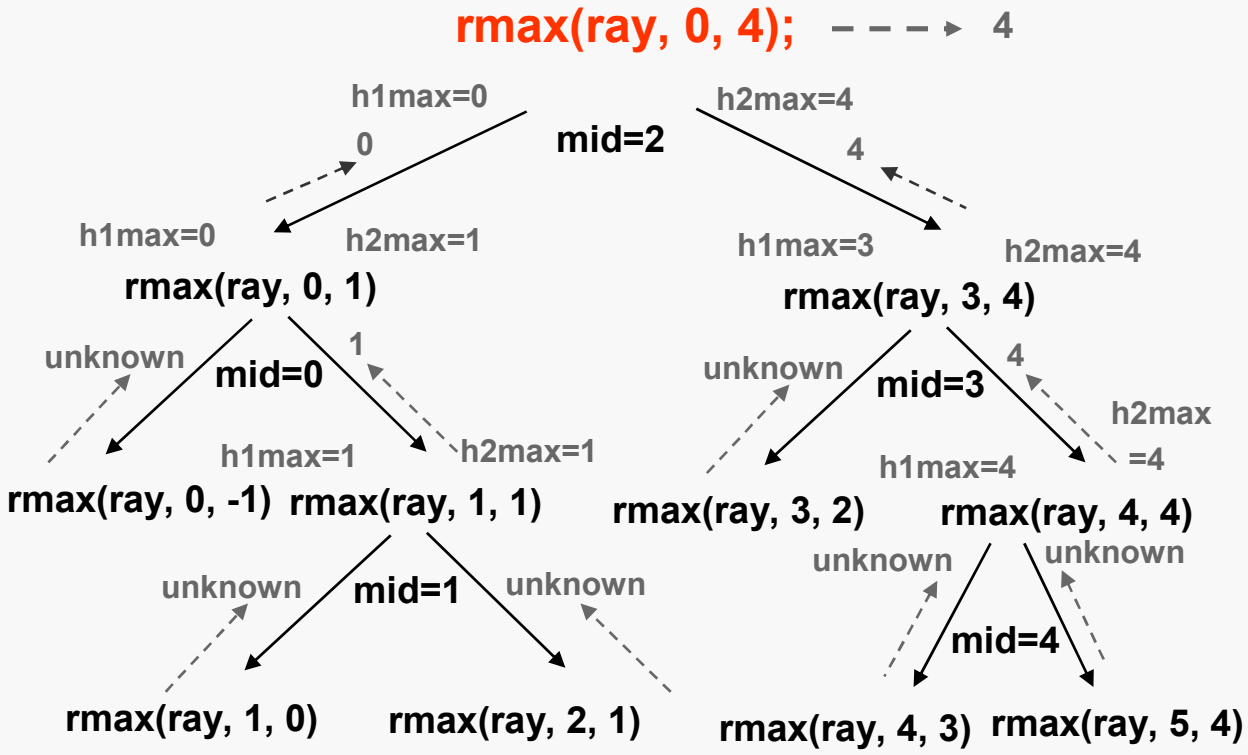
**"Unknown" checks ensure that indices are within array subscript range**

# Call Tree Traces

Given:




Call Tree Trace of




**Middle decomposition (splitting problem into halves), recursive functions are best traced with tree diagrams**

# Edges & Center Decomposition

Problem:

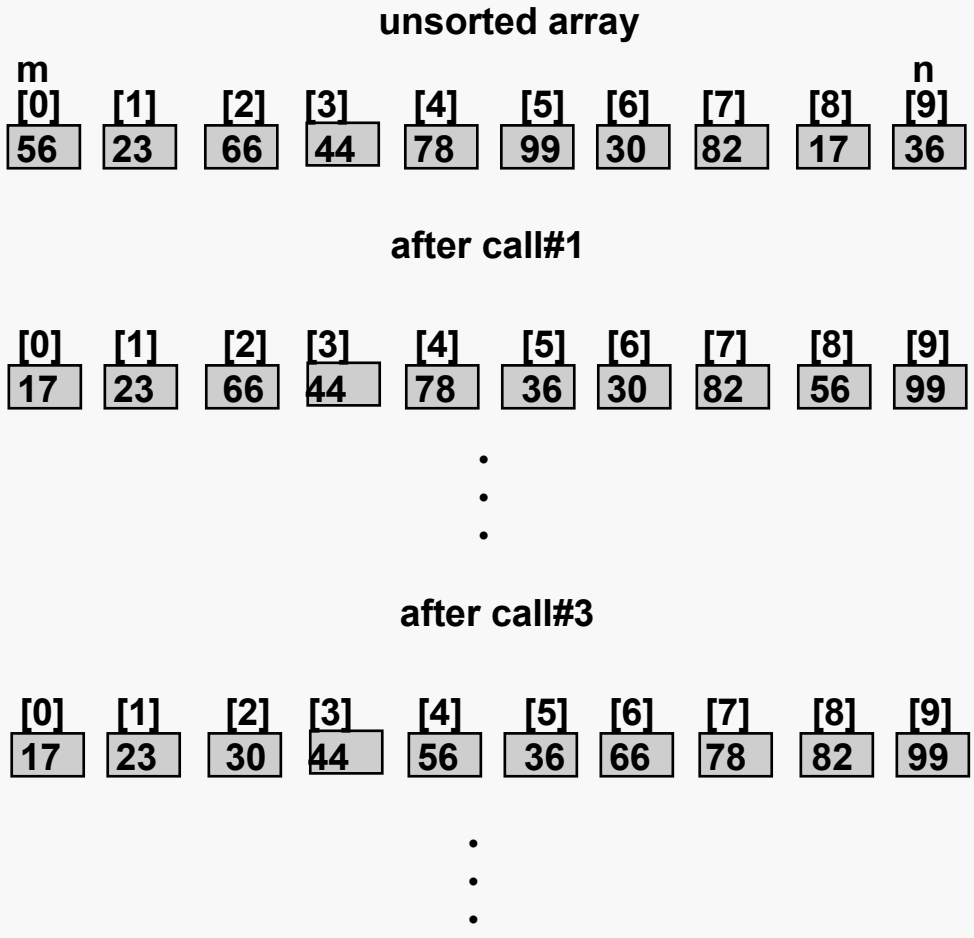
- sort a subset, (m:n), of an array of integers (ascending order)

Solution:

- Find the smallest and largest values in the subset of the array (m:n) and swap the smallest with the $m^{th}$ element and swap the largest with the $n^{th}$ element, (i.e. order the edges).
- Sort the center of the array (m+1: n-1).

**Variation of the "selection" sort algorithm**

Solution Trace



# Recursive Sorting

```
void duplexSelection(int ray[], int start,
                                          int end ) {
  int mini = start, maxi = end;

  if (start < end) { //start==end => 1 element to sort

      findMiniMaxi( ray, start, end, mini, maxi);
      swapEdges(ray, start, end, mini, maxi);
      duplexSelection( ray, start+1, end-1);
  }
}
```

```
void findMiniMaxi( const int ray[], int start, int end,
                                int& mini, int& maxi) {

  if (start < end) { //subset to search exists

      if (ray[start] < ray[mini]) mini = start;
      else if (ray[start] > ray[maxi]) maxi = start;
      findMiniMaxi( ray, start+1, end, mini, maxi);
  }
}
```

```
void swapEdges(int  ray[], int start, int end,
               int mini, int maxi)
{
  //check for swap interference
  if ( (mini == end) && (maxi == start) ) {
      swap( ray[start], ray[end] );
  } //check for low 1/2 interference
  else if (maxi == start) {
      swap( ray[maxi], ray[end] );
      swap( ray[mini], ray[start] );
  } // (mini == end) || no interference
  else {
      swap( ray[mini], ray[start] );
      swap(ray[maxi], ray[end] );
  }
}
```

```
void swap
( int& x, int& y)
{
  int tmp= x ;
  x = y ;
  y = tmp ;
}
```

# Backtracking

## Knapsack Problem (*very weak form*)

- Given an integer total, and an integer array, determine if any collection of array elements within a subset of the array sum up to total.
- Assume the array contains only positive integers.

## Special Base Cases

- total = 0 :
  - † solution: the collection of no elements adds up to 0.
- total < 0 :
  - † solution: no collection adds to sum.
- start of subset index > end of subset index :
  - † solution: no such collection can exist.

## Inductive Step

- Check if a collection exists containing the first subset element.
  - † Does a collection exist for total - ray[ subset start ] from subset start + 1 to end of subset?
- If no collection exists containing ray[ subset start ] check for a collection for total from subset start + 1 to the end of the subset.

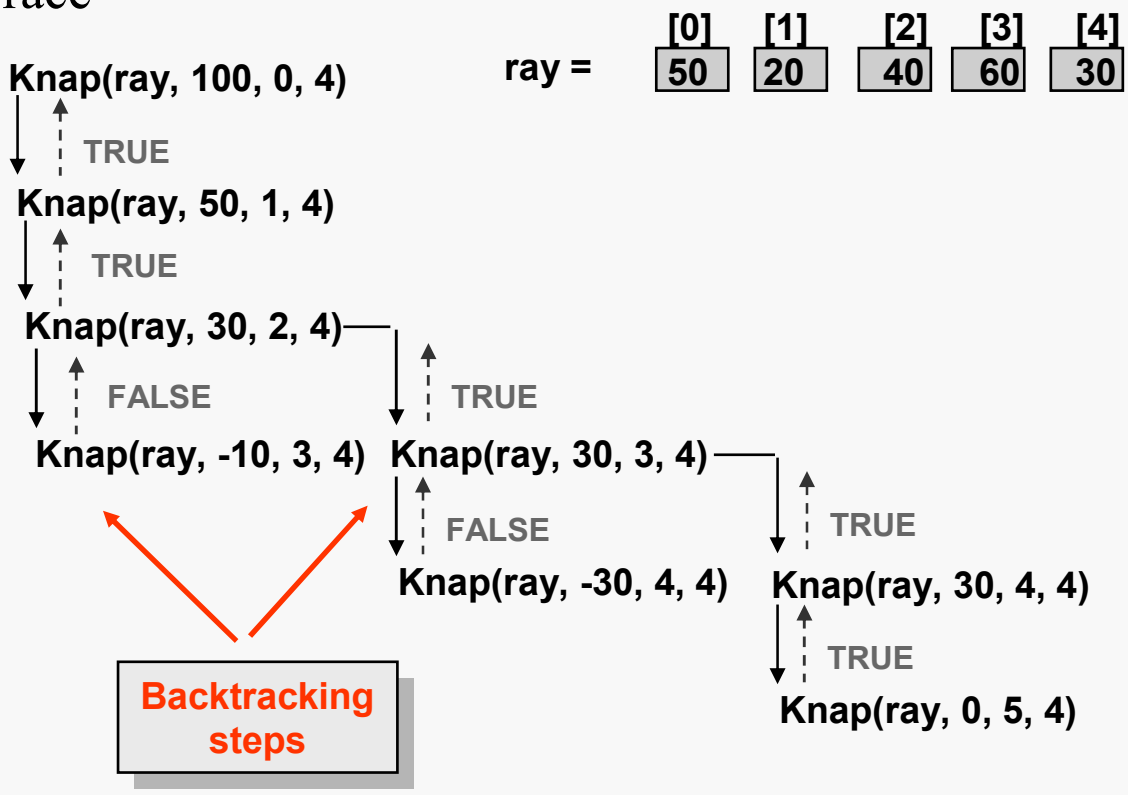
**Backtracking step. Function searches for alternative solution "undoing" previous possible solution search work.**

# Knapsack Solution

## Knap **backtracking** function

```
bool Knap (const int ray[], int total, int start, int end)
{
  if (total == 0)           // empty collection adds up to 0
     return true;
  if ( (total < 0) || (start > end) ) //no such
     return false;                    //collection exists

  //check for collection containing ray[start]
  if (Knap(ray, total-ray[start], start+1, end))
     return true;

  // check for collection w/o ray[start]
  return (Knap(ray, total, start+1, end));
}
```

## Trace



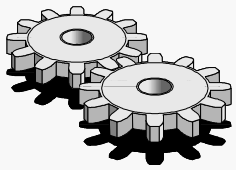
# Runtime Stack

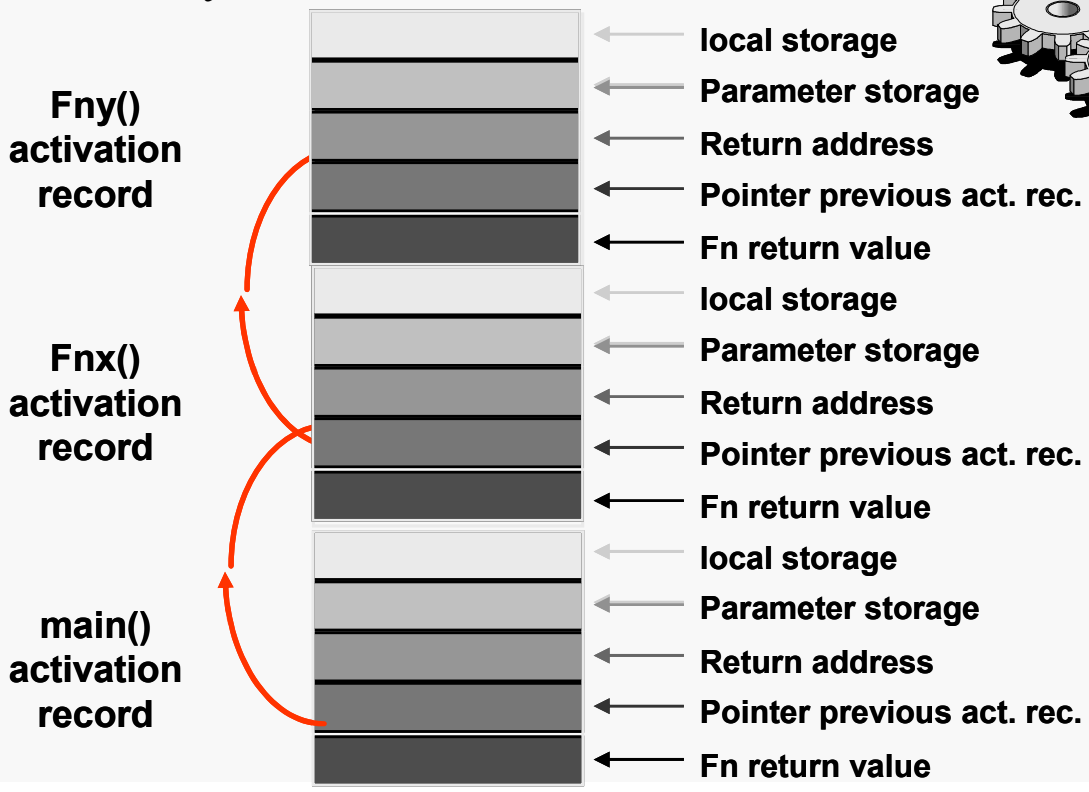
## Recursion Underpinnings

- Every instance of a function execution (call) creates an **Activation Record, (frame)** for the function.
- Activation records hold required execution information for functions:
  - † Return value for the function
  - † Pointer to activation record of calling function
  - † Return memory address, (calling instruction address)
  - † Parameter storage
  - † Local variable storage

## Runtime Stack

- Activation records are created and stored in an area of memory termed the "**runtime stack**".

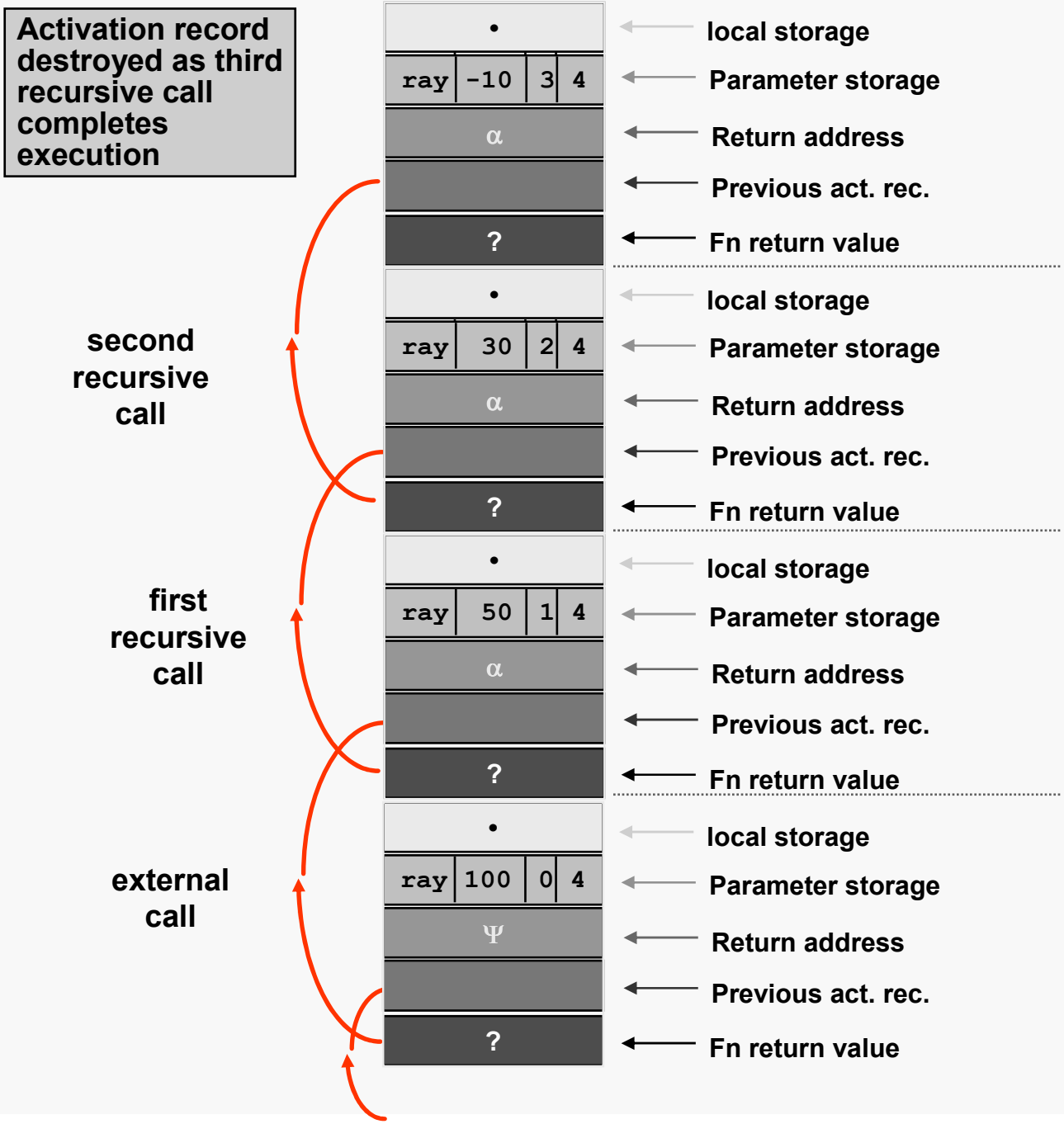




# Knap Runtime Trace Snapshot

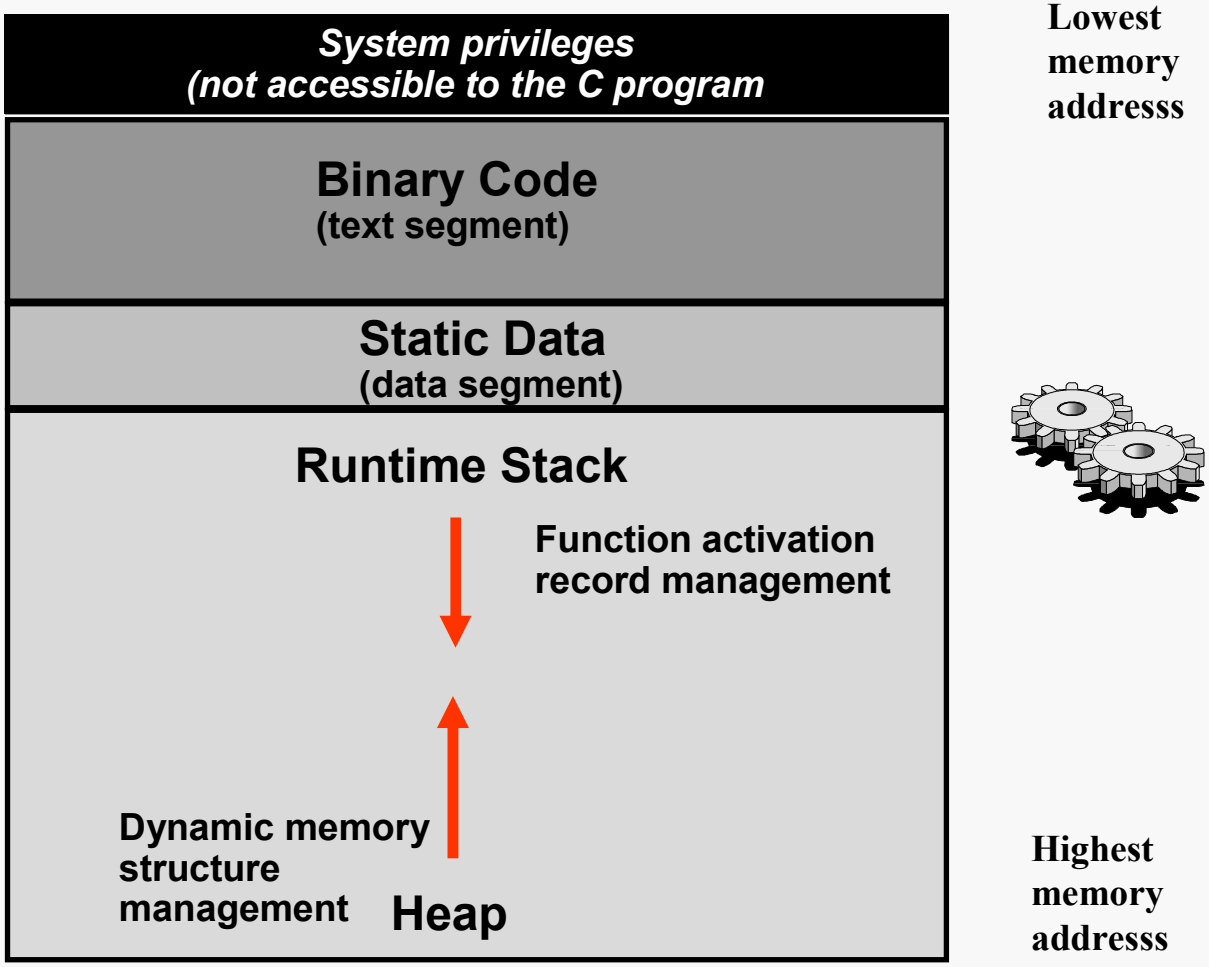
First backtrack step(during fourth call)

- Let first recursive call in knap be at address α
- Let second recursive call in knap be at address β



# Storage Organization

## Typical C++ program execution memory model




## Storage Corruption

- Infinite regression results in a collision between the "run-time" stack & heap termed a "run-time" stack overflow error.
- Illegal pointer de-references (garbage, dangling-references) often result in memory references outside the operating system allocated partition, (segment) for the C program resulting in a "segmentation error" (GPF - access violation) and core dump.