CS-202

Recursion

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	LAST Project	PASS Session		***FINAL PASS* Session**

Your 11th (& *Final* ☺) Project will be announced today Tuesday 5/2.

- ➢ A Final Sample has been announced since last week.

Your **Final** exam is next Thursday 5/9.

- ➢ A Final Sample has been announced since last week.
- ➢ Lectures, Labs, PASS sessions (with a Sunday extra), dedicated to recapitulation.

Today's Topics

Recursion

Definition of Recursion

A function is said to be Recursive, when it "calls its own self".

- Inside the Function definition, there exists a call to the function itself:

```
std::string & repeat(std::ostream & os, std::string & s){
    os << repeat(os, s) << std::endl;
    return s;
}
```

Note:
Intuition(s) about problems with this example recursive call ?

C++ allows recursion (as most high-level languages do)

- Can be useful as a programming technique.
- Has its own special limitations and considerations.

Recursive `void` Function(s)

Divide-and-Conquer is a basic Algorithm Design technique used in programming :

- Break large task into subtasks.

In certain cases, we can describe *subtasks* as smaller versions of the original task :

- When they can → Recursion.

Recursive `void` Function(s)

By-Example – Task: Find a Value

Task: Search through a list for a specific value

- Subtask 1: search 1st half of list for that specific value.
- Subtask 2: search 2nd half of list for that specific value.

Subtasks are smaller versions of original task:

- Same task description, but each operating on a *reduced range*.

When this occurs, a Recursive function can be used.

- Usually results in an "algorithmically elegant" solution.

Recursive void Function(s)

By-Example – Task: Digits of a Number

Output digits of `int` number 1-per-line.

- (i.e. 1-at-a-time, Most Significant Digit –to– Least Significant Digit)

Example call:

```
writeDigits(1234);
```

- Desired Output:

```
1
2
3
4
```

Recursive `void` Function(s)

By-Example – Task: Digits of a Number

Output digits of `int` number 1-per-line.

Break problem into two cases:

- The "Base" case: If the `int` number is `<10`, then output that.

Note:
The "simplest" possible case, where there is nothing more to be done.

- The "Recursive" case: If the `int` number is `>=10`, formulate two Subtasks:
  a) Output digits of *reduced* `int` number 1-per-line, (clipped - omitting the last digit).
  b) Output the last digit (which will always be `<10`).

Note:
The case where there is still processing to be done. It must be formulated in a way that matches original problem description.

Recursive `void` Function(s)

By-Example – Task: Digits of a Number

Output digits of `int` number 1-per-line.

- The "Base" case: If the `int` number is `<10`, then output that.
- The "Recursive" case: If the `int` number is `>=10`, formulate two Subtasks:
  a) Output digits of *reduced* `int` number (clipped - omitting the last).
  b) Output the last digit (which will always be `<10`).

Example argument: **`1234`**
Subtask a) : Will process for output **`123`**.
Subtask b) : Will write-out **`4`**.

Recursive void Function(s)

By-Example – Task: Digits of a Number

- The “Base” case: If the int number is <10, then output that.
- The “Recursive” case: If the int number is >=10, formulate two Subtasks:
  a) Output digits of *reduced* int number (clipped - omitting the last).
  b) Output the last digit (which will always be <10).

```
void writeDigits (int n) {
  if (n < 10)   // base case
    cout << n << endl;
  else {       // recursive step
    writeDigits( n / 10 );
    cout << ( n % 10 ) << endl;
  }
}
```

Recursive `void` Function(s)

By-Example – Task: Vertical Numbers

Tracing the Recursive call:

```
writeDigits(1234);
  writeDigits(123);            (1234/10)
    writeDigits(12);           (123/10)
      writeDigits(1);          (12/10)
      cout << 1 << endl;
    cout << 2 << endl;
  cout << 3 << endl;
cout << 4 << endl;
```

```
1
2
3
4
```

```
void writeDigits(int n) {
  if (n < 10)   // base case
    cout << n << endl;
  else {        // recursive step
    writeDigits( n / 10 );
    cout << ( n % 10 ) << endl;
  }
}
```

Note:

- ➢ Notice the first three calls again (Recursive).
- ➢ Last call (Base) displays **1** and "ends" Recursion sequence.

Recursive void Function(s)

By-Example – Task: Vertical Numbers

Tracing the Recursive call:

```
writeDigits(1234);
  writeDigits(123);
    writeDigits(12);
      writeDigits(1);
        cout << 1 << endl;
      cout << 2 << endl;
    cout << 3 << endl;
cout << 4 << endl;
```

```
void writeDigits (int n) {
 if(n < 10){ cout << n << endl; }
 else{ writeDigits( n/10 );
       cout << ( n%10 ) << endl; }
}
```

How the computer performs Recursive calls:

- Pauses current function.
  (Have to know results of new Recursive call before proceeding).
- Retains information needed for current call.
  (They will be used later, when nested Recursive call is finished and returns).
- Proceeds to evaluate new Recursive call.
  (When THAT call is complete, returns to “outer” computation).

Recursion in the Broader Sense

Outline of a successful Recursive function

- One or more cases where function accomplishes its task by making one or more recursive calls to solve smaller versions of original task.

  Called "Recursive" case(s).

- One or more cases where function accomplishes its task immediately (without any more recursive calls).

  Called "Base" case(s) – or "Stopping" case(s).

Infinite Recursion

Necessary consideration for a successful Recursive function

- The "Base" case has to be entered eventually.

  If the algorithm doesn't guarantee that → Infinite Recursion.
  (recursive calls keep getting created)

Remember:
In the `writeDigits` example, the Recursion stopped when a single-digit number was reached.

```
void writeDigits (int n) {
 if(n < 10){ cout << n << endl; }
 else{ writeDigits( n/10 );
       cout << ( n%10 ) << endl; }
}
```

Infinite Recursion

By-Example – Task: Vertical Numbers

Tracing the Recursive call:

```
writeDigits(1234);
  writeDigits(123);      (1234/10)
    writeDigits(12);     (123/10)
      writeDigits(1);    (12/10)
        writeDigits(0);  (1/10)
          writeDigits(0);  (0/10)
            writeDigits(0);  (0/10)
```

```
void writeDigits(int n) {
  if (n < 10)   // base case
    cout << n << endl;
  else {        // recursive step
    writeDigits( n / 10 );
    cout << ( n % 10 ) << endl;
  }
}
```

Note:

➢ Missing "Base" case → Recursion never stops.

The Program Stack

The Program / Execution Stack

- A specialized memory structure.
- Like stack of papers : Place new one on top,
  Remove one when possible from top.
- A Last-In-First-Out memory structure.

Program Function calls use the Program Stack.

- Each call is pushed onto the Stack.
- When the last-pushed call is completed, it is popped (removed) from the stack frame.

Program Stack & Recursion

The Program / Execution Stack

- Finite memory (often also intentionally limited by default).

Recursive calls are pushed onto the Program Stack.

- When (and only when) the "Base" case returns, sequential returns of each "outer" Function call are triggered.
- A long sequence of Recursive calls fills up the Stack.
- Until the "Base" case triggers sequential returns and Stack calls start getting popped.

Stack Overflow & Recursion

Recursive calls are pushed onto the Program Stack.

- A long sequence of Recursive calls keeps adding up, until the “Base” case (if such exists) triggers sequential popping.
- Finite memory.

If stack attempts to grow beyond the physical / virtual limit:

- Stack Overflow.
- Infinite recursion will always cause this.

Infinite Recursion:
Segmentation Fault

```
void recurse(){
  recurse();
}
```

Note: The Fork Bomb
(behavior depends on OS’ **`ulimit`**)

```
int main(){
  while(1) fork();
}
```

In a Shell:

```
:(){ :|:& };:
```

Recursion *vs* Iteration

Any task accomplished with Recursion can also be done without it.

- Iteration can be used in place of recursion
- Iterative Algorithm: Uses a looping construct.
- Recursive Algorithm: Uses a branching structure.

Recursive solutions are often less efficient (time & space).

Run slower, use more memory than Iterative solution(s).

Recursion can simplify the algorithmic solution of a problem.

- Shorter, more elegant and expressive code.

Recursive Function(s) that Return a Value

Recursion can also be implemented with non-`void` returning functions (value of any type).

Same Technique:

- One (or more) cases where the value returned is computed by recursive calls:
  Should follow the formulation of “smaller” Sub-problems.
- One (or more) cases where the value returned is computed directly, without any more recursive branching.
  The “Base” case.

Recursive Function(s) that Return a Value

By-Example – Task: Power(s)

Raise a number to an `int n` power (exponent).

- The "Base" case: If the `int n` power is `0`, then result is `1`. ($x^0 = 1, x \in \mathbb{R}$)
- The "Recursive" case: If the `int n` number is `>0`, formulate Subtasks:
  a) Raise number to the `int n-1` power.
  b) Return the result of that, multiplied by the number itself.

Example call: `float power(float x, int n)` $\rightarrow$ `power(2,3);` (2^3)
Subtask a) : Will calculate `power(2,2);` (2^2) .
Subtask b) : Will return `2*power(2,2);` ($2 * 2^2 = 2^3$) .

Recursive Function(s) that Return a Value

By-Example – Task: Power(s)

- The "Base" case: If the `int n` power is `0`, then the result is `1`.
- The "Recursive" case: If the `int n` number is `>0`, formulate Subtasks:
  a) Raise number to the `int n-1` power.
  b) Return the result of that, multiplied by the number itself.

```
float power(float x, int n) {
  if (n<0){ cout << "Illegal argument";
            exit(1); }
  if (n>0) // recursive case
    return ( power(x, n-1) * x );
  else     // base case
    return 1;
}
```

Recursive Function(s) that Return a Value

By-Example – Task: Power(s)

Tracing the Recursive call:

```
power(2, 3);
  power(2, 2);          (2, 3-1)
    power(2, 1);        (2, 2-1)
      power(2, 0);      (2, 1-1)
      return 1;
    return 1 * 2;
  return 2 * 2;
return 4 * 2;
8
```

```
float power(float x, int n){
  if (n<0){ exit(1); }
  if (n>0) // recursive case
    return ( power(x, n-1) * x );
  else     // base case
    return 1;
}
```

Note:

- ➢ First three calls are Recursive cases.
- ➢ Then reaches the Base case and "ends" Recursion branching.
- ➢ Values are returned "up" as Stack gets "popped".

Recursive Function(s) that Return a Value

By-Example – Task: Factorial(s)

Calculate an `int x` number's Factorial ($n!$).

- The "Base" case: If the `int x` number is `0`, then result is `1`. ($0! = 1$)
- The "Recursive" case: If the `int x` number is `>0`, formulate Subtasks:
  a) Calculate the `int x-1` number's Factorial.
  b) Return the result of that, multiplied by the number itself.

Example call: `int factorial(int x)` → `factorial(4);` (4!)
Subtask a) : Will calculate `factorial(3);` (3!) .
Subtask b) : Will return `4*factorial(3);` ($4 * 3! = 4!$) .

Recursive Function(s) that Return a Value

By-Example – Task: Factorial(s)

- The "Base" case: If the `int x` number is `0`, then result is `1`.
- The "Recursive" case: If the `int x` number is `>0`, formulate Subtasks:
  a) Calculate the `int x-1` number's Factorial.
  b) Return the result of that, multiplied by the number itself.

```
int factorial (int x) {
      // base case
      if (x == 0) return 1;

      // recursive case
      return x * factorial (x - 1);
}
```

Recursive Function(s) that Return a Value

By-Example – Task: Factorial(s)

```
int factorial (int x) {
if (x == 0) return 1;
return x * factorial (x – 1);
}
```

Tracing the Recursive call:

```
factorial(4);
factorial(3);    (4-1)
factorial(2);    (3-1)
factorial(1);    (2-1)
factorial(0);    (1-1)
return 1;
return 1 * 1;
return 1 * 2;
return 2 * 3;
return 6 * 4;    24
```

Note: All important ideas of Recursion are here.

"Base" (or "Stopping") case :

- Code first tests for stopping condition.
- Provides a direct (non-recursive) solution.

"Recursive" case

- Expresses solution to problem in two (or more) smaller parts
- Invokes itself (factorial) to compute (at least one of) the smaller parts, which eventually reaches the "Base" case.

Thinking Recursively

Recursive Design Techniques

Often in designing the Recursive Algorithm, there is no need to try and trace entire recursive sequence.

Can design by checking 3 properties:

- No Infinite Recursion.
- Base (Stopping) cases return correct values.
- Recursive cases return correct values.

Recursive Design Techniques

By-Example – Task: Power(s)

3 Properties:

a) No Infinite Recursion:

- 2nd argument decreases by 1 each call.
- Eventually must get to Base case of 1.

b) Base case returns correct value:

- `power(x,0)` is Base case, correctly returns `1`.

c) Recursive calls correct:

- For `n>1`, `power(x,n)` returns `power(x,n-1)*x`
- Plug in some values → correct.

```
float power(float x, int n) {
  if (n<0){ cout << "Illegal argument";
            exit(1); }
  if (n>0) // recursive case
    return ( power(x, n-1) * x );
  else     // base case
    return 1;
}
```

Recursive Function(s) that Return a Value

By-Example – Task: Greatest Common Divisor

Calculate the `int a` and `int b` numbers' GCD (`int g`).

- Precondition: For parameters of function call `a > b` has to hold.
  Swap `a` and `b` if `a < b`.
- The "Base" case: If the residual of `a % b == 0`, then result is `b` (greatest).
- The "Recursive" case: If not, then formulate Subtasks:
  - Only possible for a number `r < b` to be the GCD (since it is not `b` itself -checked in the base case-, and if it is larger it can't be its divisor). Also, if `r` is an integer divisor, `b % r == 0` has to hold.

  a) Make `r` be the residual of `r = a % b`.
  b) Check to see if is the GCD of `b` and `r` (if it satisfies the Base case).

Recursive Function(s) that Return a Value

By-Example – Task: Greatest Common Divisor

- The "Base" case: If the residual of **a % b == 0** ,then result is **b** (greatest).
- The "Recursive" case: If not, formulate Subtasks:
  a) Make **r** be the residual of **r = a % b**.
  b) Check to see if is the GCD of **b** and **r** (if it satisfies the Base case).

Note:
Assumes **a > b**.

```
int gcd (int a, int b) {
  if (b == 0)   // base case
    return a;
  else{         // recursive case
    return gcd (b, a % b);
  }
}
```

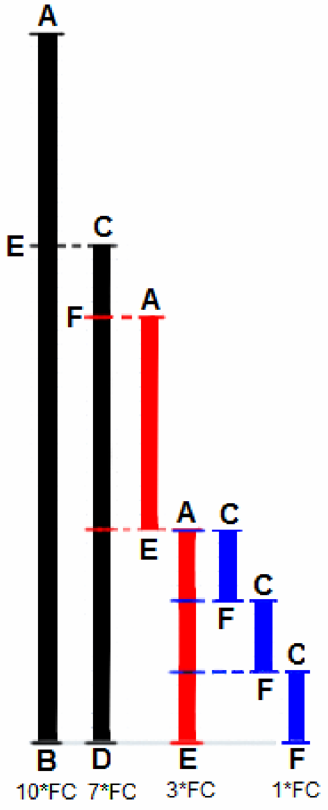
Note:
Base case formulation is written out to require 1 extra Recursive call.

Recursive Function(s) that Return a Value

By-Example – Task: Greatest Common Divisor

Tracing the Recursive call:

```
gcd(54, 28);
  gcd(24, 6);        <- (24, 54%24)
    gcd(6, 0);       <- (6, 24%6)
    return 6;
  return 6;
return 6;
```




```
int gcd(int a, int b) {
  if (b == 0)
    return a;
  else{
    return gcd(b, a % b);
  }
}
```

Note:
- Euclid's Algorithm.
- Find greatest common measuring unit between two lengths.

Recursive Function(s) that Return a Value

By-Example – Task: Fibonacci Sequence

Calculate the Fibonacci sequence of an `int n` number.

- Sequence is Recursive: $f(n) = \begin{cases} 0 & if\ n = 0\ \ (1) \\ 1 & if\ n = 1\ \ (2) \\ f(n-2) + f(n-1) & if\ n > 1\ \ (3) \end{cases}$

- The "Base" case: Two base cases, (1) and (2).
- The "Recursive" case: Multiple-branch (`f(…)` … `f(…)`) Recursive call, (3).

Recursive Function(s) that Return a Value

By-Example – Task: Fibonacci Sequence

- The "Base" case(s): If `n==0` → `0`, If `n==1` → `1`.
- The "Recursive" case: Otherwise, formulate Subtasks:
  a) Calculate `fib(n-2)`, calculate `fib(n-1)`.
  b) Return their sum.

Note:
Assumes `n > 0`.

```
int fib(int n) {
  if (n > 1)
    return fib(n-1) + fib(n-2);
  else
    return n;
}
```

Note:
Intuition(s) about problems with this example?

Note:
Both base cases formulated with a single statement.

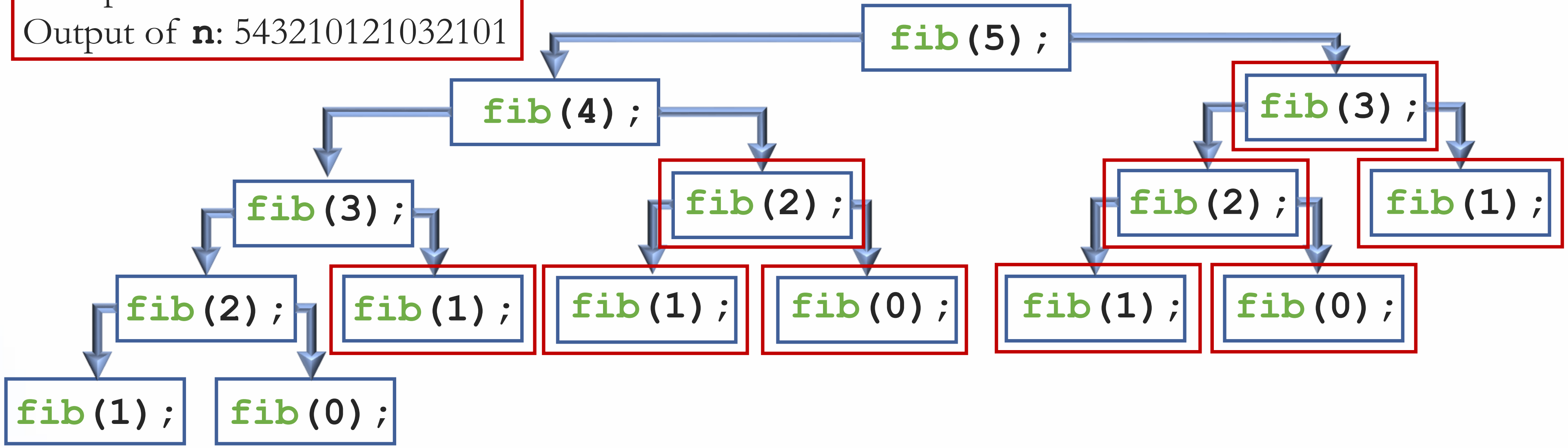
Recursive Function(s) that Return a Value

By-Example – Task: Fibonacci

Tracing the Recursive call:

Note:
Multiple redundant calls!
Output of **n**: 543210121032101

```
int fib(int n) {
  if (n>1) return fib(n-1) + fib(n-2);
  else return n;
}
```



Backtracking & Recursive Function(s)

By-Example – Task: Depth-First Search

Start at the root of a tree or graph, and explore as far deep) as possible along each branch before Backtracking.

- Algorithm follows one path as far as it can go.
- Backs up to the last point at which a different path could have been chosen.
- Follows that path as far as it can go.

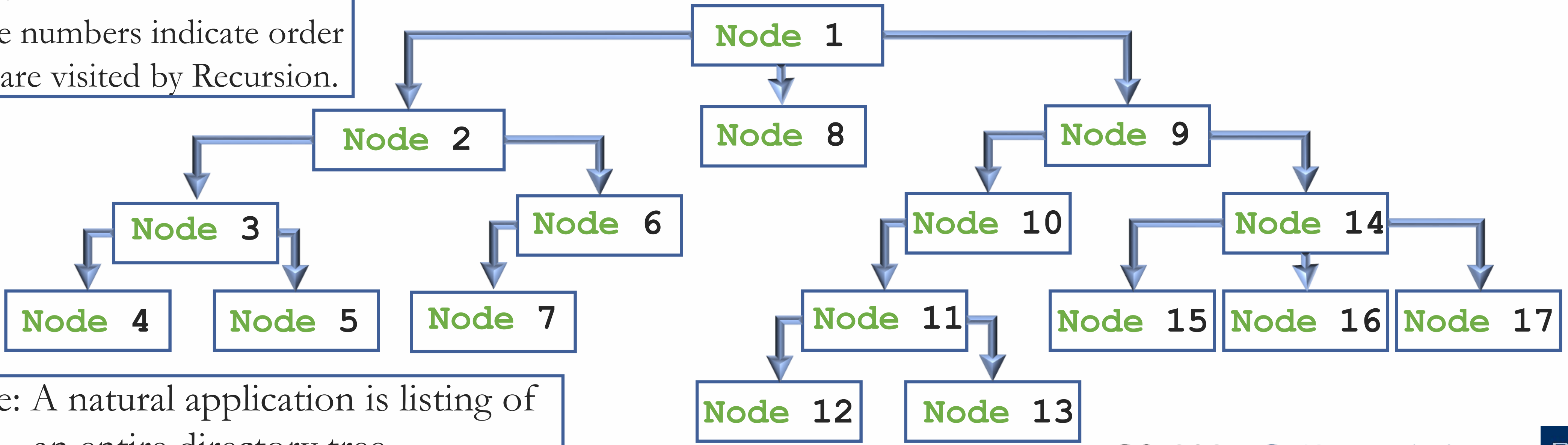
- The "Base" case: End-of path / no subsequent Nodes available.
- The "Recursive" case: Pick a branch / next Node, follow it for one step.

Backtracking & Recursive Function(s)

By-Example – Task: Depth-First Search

- The "Base" case: End-of path / no subsequent Nodes available.
- The "Recursive" case: Pick a(nother) branch / next Node, follow one step.

Note:
Node numbers indicate order they are visited by Recursion.

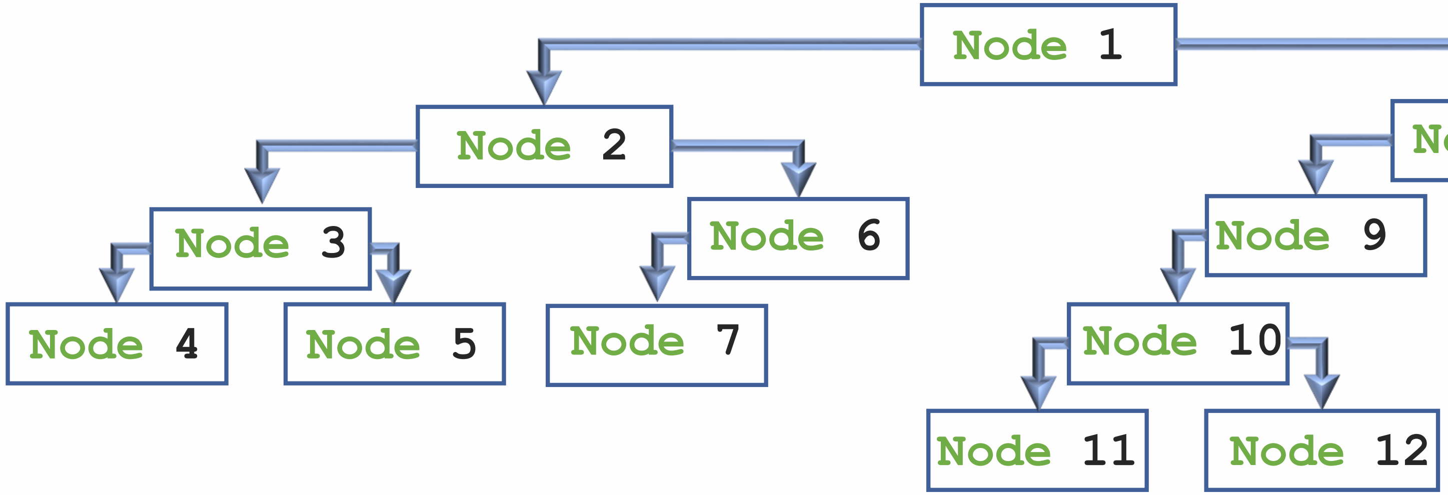


Note: A natural application is listing of an entire directory tree.

Backtracking & Recursive Function(s)

By-Example – Task: Maximum Depth of Binary Tree

- The "Base" case: End-of path: Start a counter (will increase upwards).
- The "Recursive" case: Maximum downwards Depth of that Node.
  a) Follow **`left`** & **`right`** Nodes and get their corresponding Max Depth.
  b) Return the Max of these two, incremented by one (the current Node).



Backtracking & Recursive Function(s)

By-Example – Task: Maximum Depth of Binary Tree

- The "Base" case: End-of path: Start a counter (will increase upwards).
- The "Recursive" case: Maximum downwards Depth of that Node.
  a) Follow **left** & **right** Nodes and get their corresponding Max Depth.
  b) Return the Max of these two, incremented by one (the current Node).

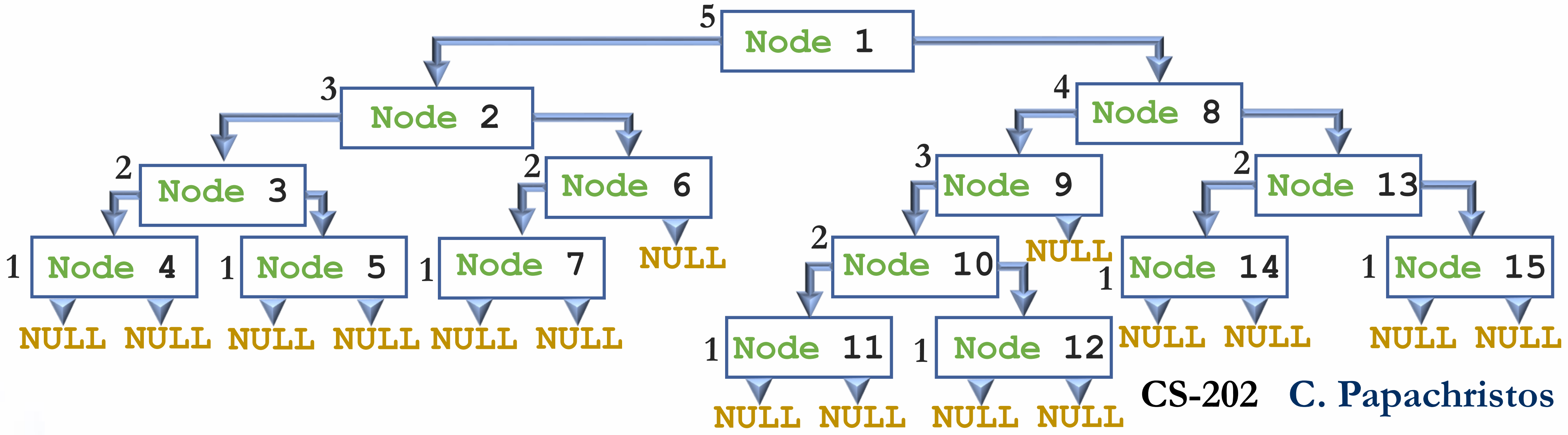
```
int depth (Node * node) {
  if (node == NULL)  // base case
    return 0;

  // recursive case
  return (1 + max( depth(node->left), depth(node->right) ));
}
```

Backtracking & Recursive Function(s)

By-Example – Task: Maximum Depth of Binary Tree

Note:
Returned numbers are evaluated from the bottom and upwards (backtracking).

```
int depth (Node * node) {
  if (node == NULL)
    return 0;
  return (1 + max( depth(node->left), depth(node->right) ));
}
```



Recursive Function(s) that Return a Value (continued)

By-Example – Task: Binary Search

Find (`int i` index of) a given `int v` Value inside a *Sorted* List.

- The "Base" case: If number at `int i` index is the same as `v`, found it!
- The "Recursive" case: If not, formulate Subtasks:
  a) Split List in two halves, and determine in which one it lies.
     (The List is *Sorted* !)
  b) Search only that half to Find (`i` index of) the `v` Value Recursively.
     (Until Base case is encountered !)

Note: Binary Search → Extremely fast compared with Sequential Search
- Half of the array eliminated at start, then a quarter, then 1/8, etc…

For a Sorted Array of 100 elements Binary Search never needs more than 7 comparisons !
- Logarithmic efficiency `O(log n)`

Recursive Function(s) that Return a Value (continued)

Pseudocode for Binary Search

```
int a[Some_Size_Value];
```

ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
//Precondition:
//a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
```

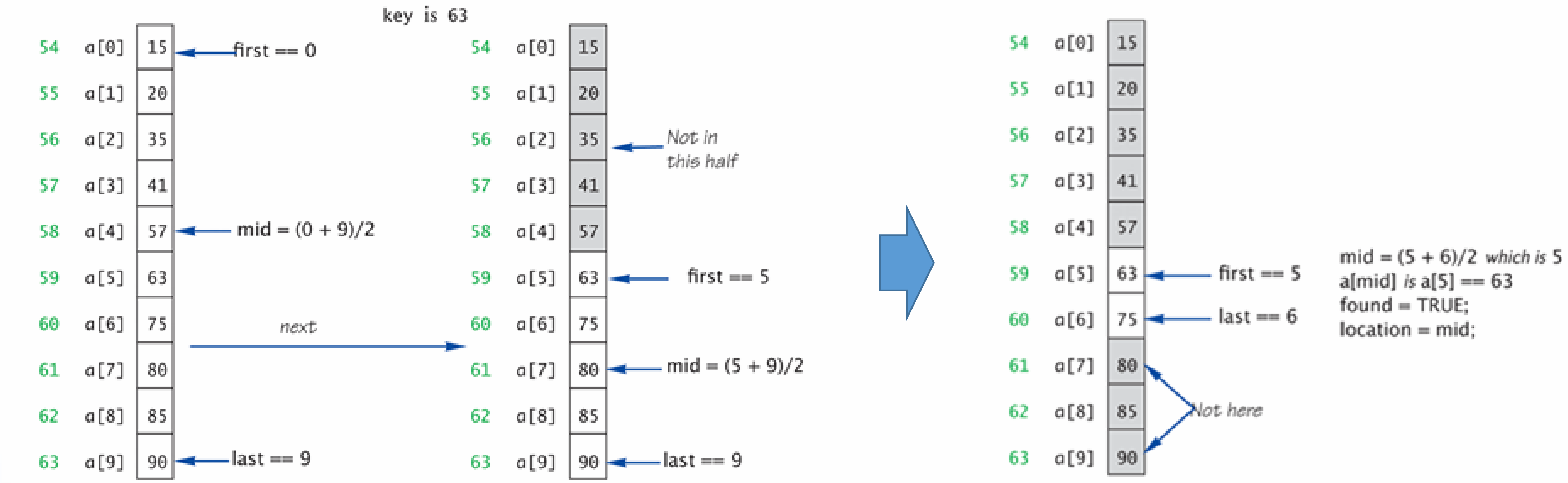
TO LOCATE THE VALUE KEY:

```
if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = false;
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        search a[mid + 1] through a[last];
}
```

Note:
Array/List/Set etc… has to be *Sorted* !

Recursive Function(s) that Return a Value (continued)

Execution of the Function search



Tail Recursion

Tail-Recursive methods

- There exists one recursive call at the very end of the method.
  Examples: Recursive Factorial, Recursive Euclid's algorithm.

C++ compilers can detect and optimize Tail-Recursion by refactoring it into iterative code

- More efficient since it does not create new instances onto the Call Stack and in many cases can avoid redundant calculations.
- Called Tail-Call Optimization, or TCO.

Extra(s)

Task: Towers of Hanoi

"At the beginning of time a group of monks in Hanoi were tasked with moving a set of 64 disks of different sizes between three pegs, according to these rules:

- No disk may ever be placed above a smaller disk
- The starting position has all the disks, in descending order of size, stacked on the first peg.
- The ending position has all the disks in the same order, stacked on the third peg.

An optimal solution for n disks requires 2n-1 moves (=18,446,744,073,709,551,616)

https://en.wikipedia.org/wiki/Tower_of_Hanoi

Extra(s)

Task: Towers of Hanoi

Recursive solution:

For any `n > 1`, the problem can be solved optimally in this way:

- Number the disks from **1** (smallest on the top) to **n** (largest at the bottom).
- Solve the problem for **n-1** disks, starting at the start post and ending at the "extra" post.
- The remaining disk will be the largest one. Move it to the finish "target" post.

Then solve the problem for the **n-1** disks, moving from the "extra" post to the "target" post.

Apply Recursively until `n = 0`.

Base case:
Move **0** disks

Extra(s)

Task: Compile-time Factorial with Recursion & Template Metaprogramming

```
template <int N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0> {
    enum { value = 1 };
};

int main()
{
    int x = Factorial<4>::value; // == 24
    int y = Factorial<0>::value; // == 1
}
```

Note:
Recursive templated Compile-Time evaluation of **`value`** (has to be of *integral*-type e.g. **`enum`**)

Note:
Template Specialization for Base case, (there is no "Compile-time **`if`**" to check for Base case condition).

At least not until C++17, **`static if`** is coming…

CS-202

Time for Questions !