EECE 231: RECURSION
READING: BIELAJEW, SECTION 9.3

ADDITIONAL REFERENCES: MAILK, CHAPTER 16

Objectives

- Learn recursive definitions
- Learn base and general cases in a recursive definition
- Appreciate recursive algorithms and the divide and conquer concept
- Learn recursive functions
- Learn how recursive functions implement recursive algorithms

Outline

Recursive definitions

- **Recursion:** solving a problem by reducing it to smaller versions of itself
- **factorial:** recursive definition
  - $0! = 1$ if $n = 1$ (1): the base case
  - $n! = n \times (n-1)!$ if $n > 0$ (2) : the general case

RECURSIVE DEFINITIONS — 2

- A recursive definition defines a structure in terms of a smaller version of itself.
- Every recursive definition must have at least one base case.
- The general case must eventually reduce the definition into the base case.
- The base case stops the recursion (reduction of the problem).

Recursive algorithms

- Recursive algorithms are solutions that reduce the problem into smaller versions of itself.
  - MUST have at least one base case.
  - The problem MUST eventually reduce to one of the base cases.
- A recursive function is an implementation of recursion definitions and algorithms.
  - It is a function that calls itself.

OUTLINE

Recursive functions

- Consider the execution of a recursive function as the execution of several copies of the recursive function.
- Each call to a recursive function has its own:
  - code,
  - parameters (argument),
  - local variables,
  - return value, and
  - control: knows where to return when done (who called it)

Recursive function control

- When one of the calls to the recursive function completes control returns to the calling function
  - could be another version of the function, or
  - could be the original call of the recursive function
- Execution in the calling function resumes from the point immediately following the call.

EXAMPLE FACTORIAL

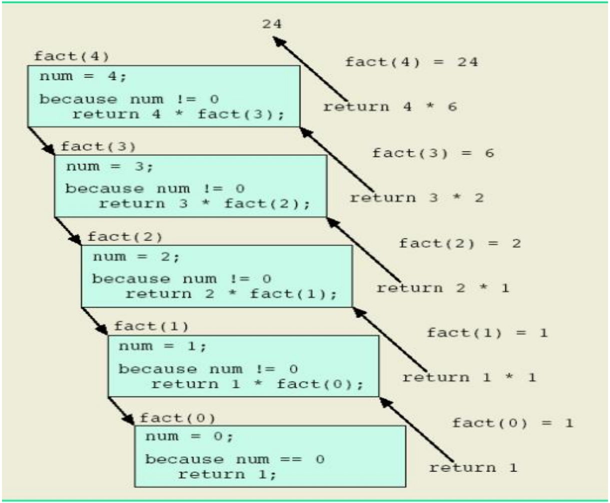
- *factorial*(n)
  - 1 if $n = 0$
  - $n * (n - 1)!$ if $n \geq 1$

```
int factorial(int n) {
  // n is a non-negative integer
  if (n == 0) {
    return 1;
  } else {
    int factNMinusOne = fact(n-1);
    return n*factNMinusOne;
  }
}
```

ALTERNATIVE *factorial* IMPLEMENTATION

```
int factorial(int n) {
  // n is a non-negative integer
  if (n == 0) {
    return 1;
  }
  return n*fact(n-1);
}
```

EXECUTION OF THE *factorial* IMPLEMENTATION



Execution of the expression fact (4)

Indirect recursion

- **Direct recursion:** a function calls itself.
- **Indirect recursion:** a function f calls other functions that eventually end up calling f.

Infinite recursion

- **Infinite recursion:** every function call results in a recursive function call.
  - In theory it executes forever
- Because computer memory is finite:
  - Computer executes until it runs out of memory to make copies of function variables.
  - Unexpected results in terms of termination status of the program.
- **Important:** If not intended, it is usually the result of
  - missing base case
  - the smaller version of the problem does not reduce to the base case

TRACKING RECURSION

- Consider function:

```
void f(int n) {
  cout << n < " " ;
  if (n >= 1) {
    f(n-1);}
}
```

- Call function:

```
f(10);
```

TRACKING RECURSION

- Consider function:

```
void f(int n) {
  cout << n < " " ;
  if (n >= 1) {
    f(n-1);}
}
```

- Call function:

```
f(10);
```

- Result: 10 9 8 7 6 5 4 3 2 1 0

TRACKING RECURSION - 2

- Consider function:

```
void g(int n) {
  if (n >= 1) {
    g(n-1);}
  cout << n < " " ;
}
```

- Call function:

```
g(10);
```

Tracking recursion - 2

- Consider function:

```
void g(int n) {
  if (n >= 1) {
    g(n-1);}
  cout << n < " " ;
}
```

- Call function:

```
g(10);
```

- Result: 0 1 2 3 4 5 6 7 8 9 10

OUTLINE

REDUCING A PROBLEM TO TWO SIMILAR PROBLEMS SMALLER IN SIZE

- Fibonacci numbers:
  - $a_1 = 1$
  - $a_2 = 1$
  - $a_n = a_{n-1} + a_{n-2}$ if $n \geq 3$
- Given n compute the Fibonacci number of n.

FIBONACCI NUMBERS THE ITERATIVE WAY

```
int FibNum(int n)
{
  if(n==1 || n==2) { return 1;}
  int previous1=1,  previous2=1, current;
  for(int i=3; i<=n; i=i+1) {
    current = previous1+previous2;
    previous2 = previous1;
    previous1 = current;
  }
  return current;
}
```

RECURSIVE IMPLEMENTATION FOR FIBONACCI NUMBERS

- Fibonacci numbers:
  - $a_1 = 1$
  - $a_2 = 1$
  - $a_n = a_{n-1} + a_{n-2}$ if $n \geq 3$
- Given n compute the Fibonacci number of n.

```
int recFibNum(int n) {
  // the two base cases
  if(n==1 || n==2) { return 1;}
  else {
    // first recursive call
    int prev1 = FibNum (n-1);

    // second recursive call
    int prev2 =  FibNum(n-2);
    // merge the result
    return prev1 + prev2;
  }
}
```

ALTERNATIVE RECURSIVE FIBONACCI NUMBERS

- Fibonacci numbers:
  - $a_1 = 1$
  - $a_2 = 1$
  - $a_n = a_{n-1} + a_{n-2}$ if $n \geq 3$
- Given n compute the Fibonacci number of n.

```
int recFibNum(int n) {
// the two base cases
if(n==1 || n==2) { return 1;}

// Note that the else keyword
// can be omitted
return  FibNum (n-1) + FibNum(n-2);
}
```

TRACK THE RECURSION OF *recFibNum*

- Try *recFibNum*(5) and draw its *recursion execution tree*.
- Compare to *FibNum*(5).
- Which one is faster?
  - *recFibNum* is not efficient at all compared to *fibNum*.
  - Why?

recFibNum vs. *FibNum*

- The *recFibNum* repeats solving same smaller size problems several times.
- *recFibNum*(5) calls *recFibNum*(4) and *recFibNum*(3)
  - *recFibNum*(4) calls *recFibNum*(3) and *recFibNum*(2).
  - Notice *recFibNum*(3) will be solved at least twice.

Outline

DESIGNING A RECURSIVE FUNCTION

- Identify the recursive structure:
  - Define the solution of the general problem in terms of solutions of smaller versions of the problem.
  - Make sure the smaller solutions eventually reduce to one of the base cases.
- Identify base cases, and provide direct and simple solutions to each base case

BINARY SEARCH REVISITED

- Recall the array search problem given in Programming Assignment 4
- Given an array *a* with *n* elements and value *v*
  - Check if *a* contains an element with the same value of *v* and return its index,
  - Otherwise return -1.
  - *a* is sorted in non-decreasing order
- Sequential-search checks *v* against all elements of the array *a*. It does not use the fact that *a* is sorted.
- Binary search: a faster algorithm which uses the fact that *a* is sorted to eliminate half of the elements at each step
- Will do a recursive version of binary-search

Binary search

- Search a between indices *left* and *right*,

  $0 \leq left \leq right < n$
- Split in the middle $mid = \frac{left+right}{2}$
- if $v < a[mid]$

  limit search between *left* and $mid - 1$.
- if $v > a[mid]$

  limit search between $mid + 1$ and *rigth*.
- if $v == a[mid]$

  return *mid* (element is found)
- Repeat until either element is found or you reach an empty range.
- If the range is empty (i.e., $left > right$), return -1 (the element does not exist in the array)

ITERATIVE IMPLEMENTATION FROM LAB ASSIGNMENT

```
int binarySearch ( int a[], int n, int v) {
  // a is sorted and of size n,
  // initialize the search range [left ... right] to be [0 ... n-1]
  int left = 0, right = n-1;
  // loop terminates when search range becomes empty: left > right
  // and continues executing when range is still valid: left <= right
  while (left <= right ) {
  // split in the middle.
    int  mid = (left + right)/2;
    if (a[mid] < v) // check to eliminate left half of range
      left = mid + 1;// range becomes [mid+1 ... right]
    else if (a [mid] > v) // check to eliminate right half of range
      right = mid - 1;// range becomes [left ... mid-1]
    else   // i.e., if (a[mid] == v), then element found
      return mid;
  }
  // if we reach this point, then the range is empty
  return -1;// element not found
}
```

RECURSIVE BINARY SEARCH

General case:

- Split in the middle $mid = \frac{left+right}{2}$
- $v < a[mid]$
  - Recursively search between $left$ and $mid - 1$.
- $v > a[mid]$
  - Recursively search between $mid + 1$ and $right$.

Base cases:

- range is empty
  - v does not exist
  - return -1
  - happens when $left > right$
- $a[mid] = v$
  - found: return mid

RECURSIVE BINARY SEARCH PROTOTYPE

- We have *a*, *n*, and *v*
- For recursion we need *left* and *right*
- The first time $left = 0$ and $right = n - 1$,
  - so, *n* is redundant and can be eliminated from the argument list
- Declaration:

```
int recBinarySearch(int a[], int left, int right, int v);
```

- Initial call:
  - `recBinarySearch(a, 0, n-1, v);`

RECURSIVE BINARY SEARCH IMPLEMENTATION

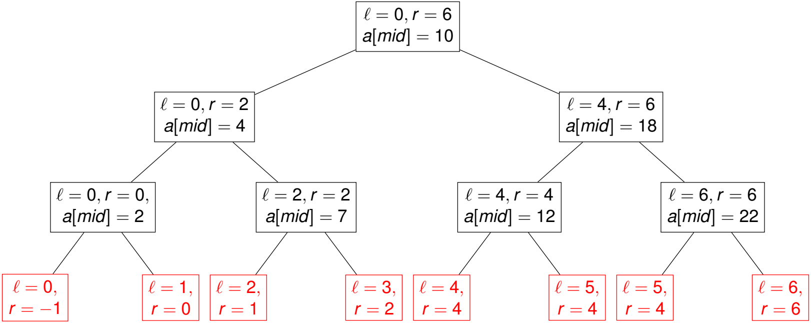
```
int recBinarySearch ( int a[], int left, int right, int v) {
// a is sorted and 0 <= left, right <= n-1
 if (left > right) // if range empty/invalid
   return -1;// element not found
// split in the middle.
 int mid = (left + right)/2;


 if (a[mid] < v) { // recursively search a[mid+1 ... right]
   // pay attention why we return directly here
   return recBinarySearch(a,mid+1,right,v);
 }
 else if (a [mid] > v) { // recursively search a[left ... mid-1]
   // pay attention why we return directly here
   return recBinarySearch(a,left,mid-1,v);
 }
 else // i.e., if(a[mid] == v) , element is found
   return mid;

}
```

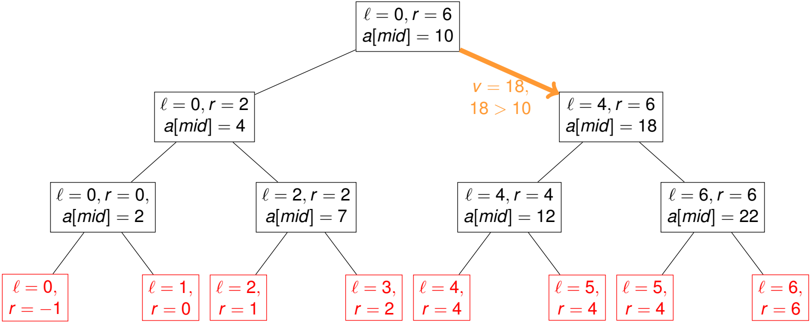
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



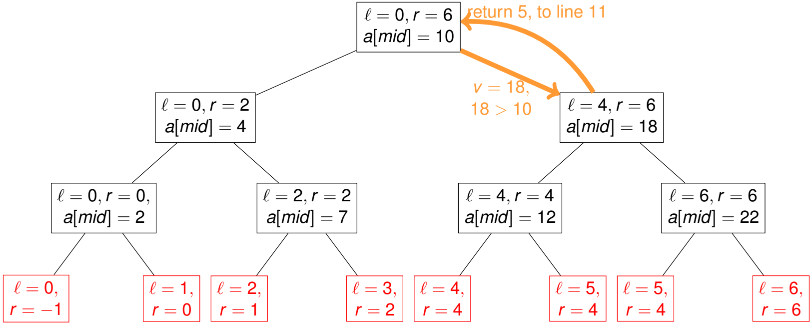
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



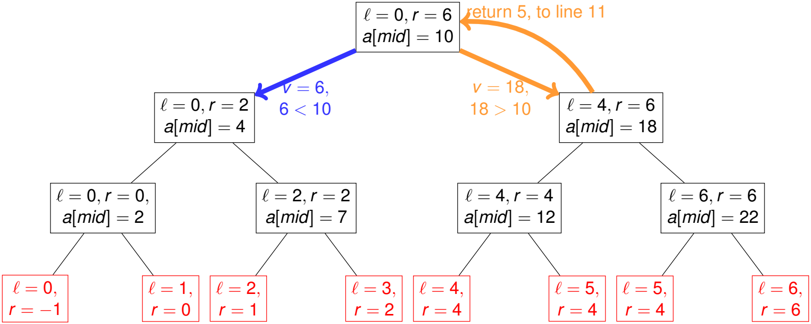
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



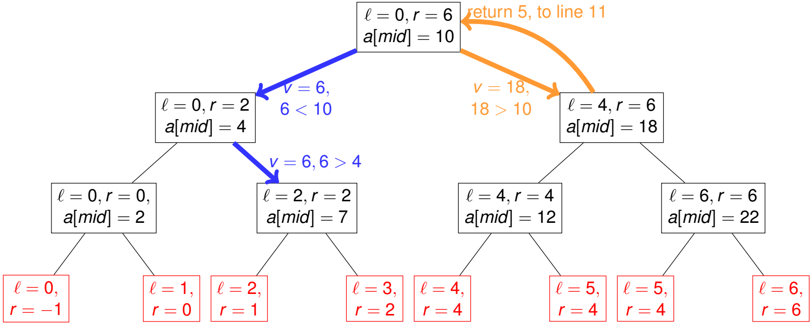
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



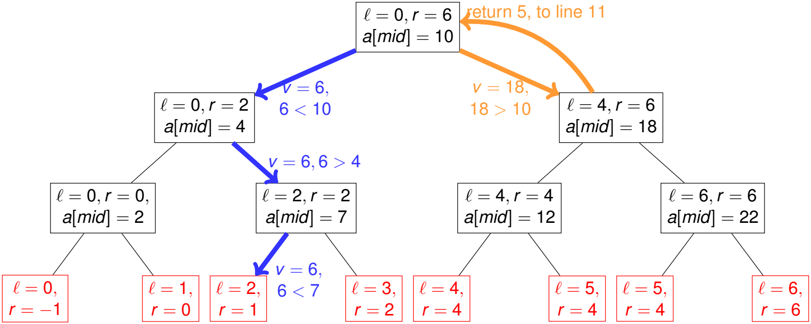
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



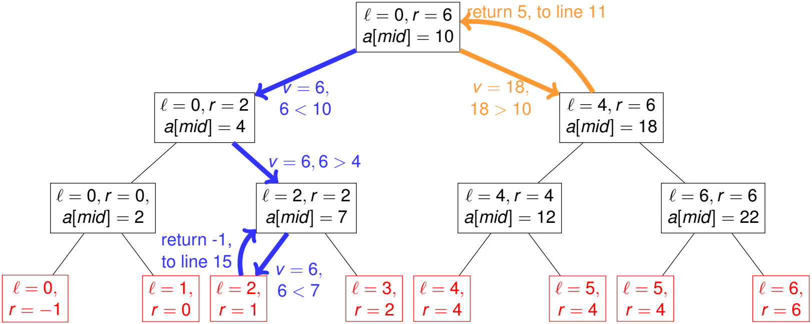
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



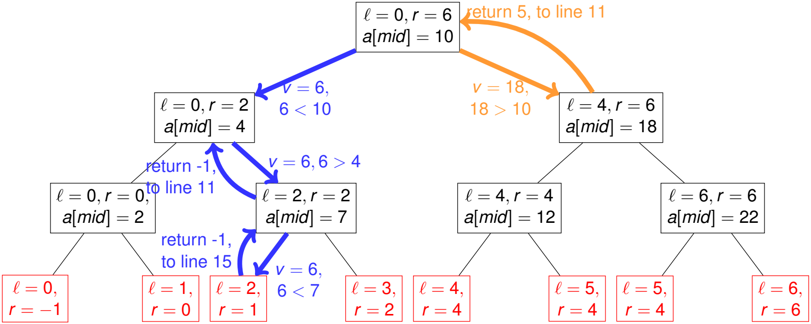
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



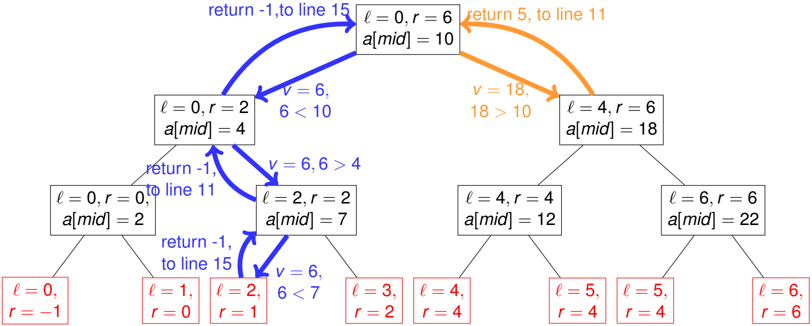
TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



TRACE OF EXECUTION OF *recBinarySearch*

```
int A[]={2,4,7,10,12,18,22};
int idx = recBinarySearch(A,0,6,18);// return 5
idx = recBinarySearch(A,0,6,6); // return -1, not found
```



FYI: Array parameters in recursive functions

- **C++:** Typically Arrays are passed by reference and are NOT copied to recursive calls.
  - All the copies of the function work on the same copy of the array.
- **Matlab:** The interpreter tries to do the “smart” thing
  - For instance, if the array is not changed in the function then it is not copied.

WHY RECURSION?

- Some of the examples so far are better done iteratively.
- We used them for illustration purposes.
- It is often more elegant to define a recursive solution.
  - Typical divide and conquer algorithms.
- More interesting examples will follow

OUTLINE

THE TOWER OF HANOI PROBLEM

- Given 3 needles and n disks of increasing sizes.
- The n disks are originally stacked on needle 1 in increasing size order with largest at the bottom.
- Target is to move the disks to needle 3.
- Constraint:
  - Move one disk at a time.
  - The removed disk must be directly placed on one of the needles.
  - A larger disk can not be placed on top of a smaller disk.

THE TOWER OF HANOI ANIMATION

Use the tower-of-Hanoi slides by Hofman and Damman

TOWER OF HANOI ALGORITHM IDEA

- In general to move n disks from needle 1 to needle 3
  1. if $n \geq 2$ move top $n-1$ disks recursively **from** needle 1 **to** needle 2, using needle 3 as an **intermediate** needle.
  2. Move disk from needle 1 to needle 3
  3. If $n \geq 2$, move top $n-1$ disks recursively **from** needle 2 **to** needle 3, using needle 1 (which is now empty) as an **intermediate** needle.
- Base case:

TOWER OF HANOI ALGORITHM IDEA

- In general to move n disks from needle 1 to needle 3
  1. if $n \geq 2$ move top $n - 1$ disks recursively **from** needle 1 **to** needle 2, using needle 3 as an **intermediate** needle.
  2. Move disk from needle 1 to needle 3
  3. If $n \geq 2$, move top $n - 1$ disks recursively **from** needle 2 **to** needle 3, using needle 1 (which is now empty) as an **intermediate** needle.
- Base case:
  - Do second step only.

TOWER OF HANOI ALGORITHM IDEA

- In general to move n disks from needle 1 to needle 3
  1. if $n \geq 2$ move top $n-1$ disks recursively **from** needle 1 **to** needle 2, using needle 3 as an **intermediate** needle.
  2. Move disk from needle 1 to needle 3
  3. If $n \geq 2$, move top $n-1$ disks recursively **from** needle 2 **to** needle 3, using needle 1 (which is now empty) as an **intermediate** needle.
- Base case:
  - Do second step only.
- Two-way recursion: the function calls itself twice

RECURSIVE IMPLEMENTATION OF TOWER OF HANOI

The following algorithm prints instructions to move *n* disks from needle 1 to needle 3 when called with `moveDisks(n,1,3,2);`

```
void  moveDisks(int n, int from, int to, int intermediate)
{
  if(n>=2) modeDisks(n-1, from,intermediate,to);
  // first recursive call

  cout << "Move disk " << n << " from " << from <<" to " << to << "." << endl;
  // If n ==1, only the count will be executed, hence this is the base case

  if(n>=2) modeDisks(n-1, intermediate,to,from);
  // second recursive call
}
```

Recursive vs. iterative solutions

- There are usually two ways to solve a particular problem
  - Iteration (looping)
  - Recursion
- Which method is betteriteration or recursion?
- In addition to the nature of the problem, the other key factor in determining the best solution method is *efficiency*.

RECURSION MEMORY COSTS

- Every recursive call has its own set of parameters and local variables
- Whenever a function is called
  - Memory space for its formal parameters and local variables is allocated
- When the function terminates
  - That memory space is then deallocated
- Overhead associated with recursive functions:
  - Memory space
  - Computer time

EFFICIENCY: ITERATIVE VS. RECURSIVE

- The choice between the two alternatives depends on the nature of the problem.
- For problems such as mission control systems
  - Efficiency is absolutely critical and dictates the solution method.
- An iterative solution is often more obvious and easier to understand than a recursive solution.
- If the definition of a *problem* is inherently recursive, sometimes it is a good idea to consider a recursive solution.
- If the idea of the *algorithm* is inherently recursive, consider a recursive solution, e.g.,
  - Binary Search, Tower of Hanoi

SUMMARY

- Recursion is the process of solving a problem by reducing it to smaller versions of itself.
- A recursive definition or algorithm has one or more base cases.
- Recursive algorithms are implemented using recursive functions
- A function is called recursive if it calls itself.
- The solution to a base case is typically obtained directly.
  - The base case stops the recursion
- The solution of a general case breaks the problem into smaller versions of itself.
- A general case must eventually be reduced to a base case.
- Directly recursive: a function calls itself
  - Indirectly recursive: A function calls another function that eventually calls the original.

HANOI -

EECE 231: RECURSION
READING: BIELAJEW, SECTION 9.3

ADDITIONAL REFERENCES: MAILK, CHAPTER 16