```
// __Recursion__ DEPTH: - depth does __not__ include
// the initial call to the function
// - the base case is the "simpler to solve problem"
// - recursion stops when the base case is reached
// - a recursive function __must always__ contain
//   a test to determine if another recursive call
//   should be made or if recursion should stop
// - each time a rec-fn is called, a new copy of the fn
//   runs with new instances of params and local vars
// - as each copy finishes executing, it returns to the
//   copy of the fn that called it
// - when the initial fn finished exec. it returns to
//   whichever part of the program it was called from
// __DIRECT RECURSION__ - fn calls itself
// __INDIRECT__ - fn A calls fn B - fn B calls fn A
```

```
// Recursive factorial fn
int factorial(int num) {
  if(num > 0) {
    return num * factorial(num - 1);
  } else {
    return 1;
  }
}
// if we were using 'tail-recursion' the above could be written as
int factorial(int num) {
  if(num <= 0) {
    return 1;     // early return of base case
  } else {
                  // recursive case last
    return num * factorial(num - 1);
  }
}
```

```
// RECURSIVE GCD - euclid's algorithm
int gcd(int x, int y) {
  if(x % y == 0) {
    return y;
  } else {
    return gcd(y, x % y);
  }
}
```

```
// FIBONACCI SEQUENCE
int fib(int n) {
  if(n <= 0) {
    return 0;
  } else if(n == 1) {
    return 1;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

```
// RECURSIVE LINKED LIST FNS
int NumberList::countNodes(ListNode *nodePtr) const {
  if(nodePtr == nullPtr) {
    return 0;
  } else {
    return 1 + countNodes(nodePtr->next);
  }
}
// calling numNodes
return countNodes(head);
void NumberList::showReverse(ListNode *nodePtr) {
  if(nodePtr !=) {
    showReverse(nodePtr->next);
    cout << nodePtr->value << " ";
  }
}
// calling
showReverse(head);
```

```
// LINEAR SEARCH
// SEQUENTIAL SEARCH
// - steps through each element
//   one by one until it finds a match
// - benefits: easy to understand
//              array can be any order
// - disadv:   inefficient - for array of N elements,
//              examines N/2 elements on average for
//              value in array, N elements for value not in array
int linearSearch(int arr[], int size, int value)
{
   int index = 0;      // Used as a subscript to search the array
   int position = -1;  // To record the position of search value
   bool found = false; // Flag to indicate if value was found

   while (index < size && !found)
   {
      if (arr[index] == value) // If the value is found
      {
         found = true; // Set the flag
         position = index; // Record the value's subscript
      }
      index++; // Go to the next element
   }
return position; // Return the position, or -1
}
```

```
// BINARY SEARCH
// - requires elements to be in order
//   divides arr into 3 sections
//    - middle
//    - side 1
//    - side 2
//   if middle = correct elem - done
//   else - repeat step 1 and divide again
//   until value is found
// - benefits: more efficient than linear search.
//   For array of N elements, performs at
//   most log2N comparisons
// - disadv: requires all elements be sorted

int binarySearch(int array[], int size, int value)
{
   int first = 0,             // First array element
       last = size - 1,       // Last array element
       middle,                // Mid point of search
       position = -1;         // Position of search value
   bool found = false;        // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value)      // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;           // If value is in upper half
   }
   return position;
}
```

```
// BUBBLE SORT
// - bene: easy to understand and implement
// - disadv: inefficient, slow for lg arrs
template<typename T>
void bubbleSort(T *array, int n) {
  if(n <= 1) {
    return;
  } else {
    for(int i = 0; i < (n - 1); ++i) {
      if(array[i] > array[i + 1]) {
        mySwap(array[i], array[i + 1]);
      }
    }
    bubbleSort(array, n - 1);
  }
}
```

```
// TOWERS OF HANOI
  const int NUM_DISCS = 3;
  const int FROM_PEG = 1;
  const int TEMP_PEG = 2;
  const int TO_PEG = 3;
  moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg) {
  if(num > 0) {
    moveDiscs(num - 1, fromPeg, toPeg, tempPeg);
    moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
  }
}
```

```
// QUICK SORT ALGORITHM
// - recursive algo that can sort an array
//   or linear linked list
// - determines an element/node to use as a
//   pivot value
// - once pivot value is determined, values
//   are shifted so elements in sublist1 are
//   < pivot and elements in sublist2 are
//   > pivot
// - algo then sorts sublist1 and sublist2
// - base case: sublist size == 1
// EXHAUSTIVE ALGORITHM
//    - search a set of combinations to find
//      an optimal one::change for certain amt
//      that uses fewest coins
//    - Uses the generation of all possible
//      combinations when determining the
//      optimal one
// RECURSION VS ITERATION
// RECURSION
// + Models certain algorithms most accurately
// + Results in shorter, simpler functions
// - may not execute efficiently
// ITERATION
// + executes more efficiently
// - often harder to code and understand
```

```
// __TEMPLATES__
// most c++ compilers require the complete def. of
// a template to appear in the client source-code
// file that uses the temp. -- temps are often def'd
// in headers -- for class temps, member fns are also
// def'd in header
template // keyword
<typeName T> // template parameters
template<typeName T> // fundamental type
template<class T> // or user def'd type
// printArray ex
template<typename T>
void printArray(const T * const array, int count) {
  for (int i = 0; i < count; ++i) {
    cout << array[i] ";
    cout << endl;
  }
}

//__If <T> is a user defined type, there MUST be an
// overloaded stream insertion operator for that type
// in order to use the stream operator in the fn temp.
```

```
// multiple fn-temp specializations are instantiated
// at compile time, despite the fact that the temps
// are written only once. These copies consume considerable
// memory - not normally an issue because generated code
// is sams size as code that would have been written
// as separate overloaded fns
```