Recursion

**Recursive Definition** – “An ancestor is a parent or an ancestor of a parent”

* Consists of something we are familiar with (a parent), and some sort of *recurrence relation* involving:
* The word we are trying to define & the familiar entity (ancestor of a parent)

\*Strength: the *recurrence relation* (the *repetitive* part of the definition) is used to briefly & clearly expand the scope of the definition

**Recursive Algorithm** –

* An algorithm defined at least *partially* in terms of itself
* Not all problems have recursive solutions
* The ones that do – provides an alternative to iterative algorithms (involving loops)

**Calculating n Factorial** –

* For n = 0, n! = 1
* For all (+) values of n, n! = n \* (n – 1) \* (n – 2) \* (n – 3) \* … \* 1
* 0! = 1 (by definition)

Coding The Algorithm 🡪 Tracing Its Execution Path –

* \*Any Java Method can implement a Recursive Algorithm\*
* One indication that a method implements a recursive algorithm 🡪 it invokes itself
* Directly Recursive: method invokes itself
* Indirectly Recursive: method invokes some other method that invokes it

**Formulating a Recursive Method** –

* Techniques for formulating Recursive Algorithms can be methodized
* Understanding of these methodized approaches:
* Further understanding of recursion
* Can take you far into the development of most recursive algorithms
* Reveals insights into the thought processes of those w/ the innate ability to think recursively

Definitions:

* **Base Case**:
* The known portion of the problem solution
* The nonrecursive portion of the solution (part “everyone knows”)
* The *Escape Clause* that allows the algorithm to escape from the recursive invocations
* **0! = 1**
* **Reduced Problem**:
* A problem very “close” to the original problem, but slightly closer to the base case
* When it is repeatedly reduced – it degenerates into the Base Case
* \*If it doesn’t eventually degenerate into the base class – the algorithm will enter an infinite loop 🡪 false base case
* **(n – 1)!**
* **General Solution**:
* The solution to the original problem (n!) expressed in terms of the reduced problem
* Uses the solution to the reduced problem to solve the original problem
* **n \* (n – 1)!**

n!

Triangular Numbers:

* the number of objects needed to form an equilateral triangle

Donnie’s Video:

* **Recursive Call** – the call where the method calls itself

\*\*Must alter the parameter value at each recursive call 🡪 or else results in stack overflow

* **Base Case** – the nonrecursive portion of the solution
* Examines the altered parameter value & determines if you’re done (time to unravel)

\*\*Must have a base case – or else results in stack overflow

* **Fibonacci Sequence** – famous sequence of numbers in mathematics where each number is the sum of the previous two numbers

Recursive Case 🡪 the General Case

Merge Sort

Merge Sort Algorithm:

* Recursive Method
* Divide-and-Conquer Algorithm
* Very fast & efficient for large data sets

Big Oh Analysis:

* Merge Sort does log n merge steps because each step doubles the list size
* It does n work for each merge step because it must look at every item
* Therefore 🡪 runs in **O(n log n)**

\*Given an array of items (integers), sort the items in ascending order

* Divide list into 2 halves & sort each list
* This gives you 2 smaller lists that are each in sorted order
* Merge the 2 lists back together:
* Compare the left most items in each list (bc these are the smallest)
* Move the smaller of the two into main list

Quick Sort:

Pivot – an item in the list used to compare every other item in the list to

* Move all items that are smaller than the pivot to the left of the pivot – X < Pivot
* Gives you the left partition
* Move all items that are larger than the pivot to the right of the pivot – X > Pivot
* Gives you the right partition

Steps:

* Compare the Left-most & Right-most values
* swap values so that the smallest are on the left & the largest are on the right
* Incrementally keep moving in and comparing each to values

Big-O Notation: (isn’t necessarily all about speed)

Measures how well a computer algorithm scales as the amount of data involved increases

O(1) –

* notation/algorithm that executes in the same amount of time regardless of the amount of data
* no matter how big the array is
* ex. adding a new list to the array – not influenced by how many items are already in the array