Recursion

- Recursion is the process of defining something in terms of itself.
- Andrew Plotkin (interactive fiction writer, aka Zarf) said:
  
  "If you already know what recursion is, just remember the answer. Otherwise, find someone who is standing closer to Douglas Hofstadter than you are; then ask him or her what recursion is."

- Consider the following ‘dictionary’ definitions:
  
  **recursion**: See "recursion".
  
  **decoration**: n. any ornament or adornment used to decorate something.

- Though such recursive definitions are not always helpful when used in the dictionary, they can be very useful when representing and/or solving mathematical or computer programming problems.

Thinking Recursively

- To program recursively, we must learn to think recursively.
- Consider the following definition of a List:
  
  "a list is one or more numbers separated by commas".

- An alternate, equivalent recursive definition is:
  
  A List is a: number
  or a: number comma List

- The recursive definition defines each of the following lists:
  
  - 24, 45, 9
  - 2, 56
  - 34, 45, 3, 76, 94, 54, 1
  - 20
- The last of list containing a single element is described by the non-recursive part of the definition, that is, the base case.

Tracing the Recursive Definition of List

- Note the definition of List contains one option that is recursive and one that is not.
- The non-recursive part is called the base case.
- Without a base case, the recursion would never end, a problem referred to as infinite recursion.
Recursion in Math

• Consider the factorial function, $N!$, defined as the product of all integers between 1 and $N$.

\[
3! = 3 \times 2 \times 1 = 6 \\
N! = N \times (N-1) \times (N-2) \times \ldots \times 1
\]

• What is the base case? Look for the case that's not defined in terms of $N$: eg. $1! = 1$ or $0! = 1$.

• What is the recursive case? Define the problem in terms of itself, while reducing the problem size (i.e., reduce $N!$ to $(N-1)!$)

• The factorial function can be expressed recursively as

\[
1! = 1 \quad \text{(base case)} \\
N! = N \times (N-1)! \quad \text{for } N > 1 \quad \text{(recursive case)}
\]

• The base case is 1!. All other values of $N!$ are defined recursively as $N \times (N-1)!$

Recursive Programming

• Recursion is a programming technique in which a method calls itself, that is, a method is defined in terms of itself.

• Each call to a method creates a new environment in which all local variables and parameters are newly defined.

• Each time a method terminates, processing returns to the method that called it (which for recursive calls, is an earlier invocation of the same method).

• To ensure program termination, the method must define both

  – a base case AND
  – a recursive case

• A problem that can be solved in pieces is a good candidate for recursion.

Recursive Factorial Function

• Given here in pseudocode:

```java
function factorial(n) {
    if (n<=1)
        return 1;
    else
        return n * factorial(n-1);
}
```

• Eventually the base case will be reached:

\[
4! = 4 \times 3! \\
= 4 \times 3 \times 2! \\
= 4 \times 3 \times 2 \times 1! \quad \text{(base case reached)} \\
= 4 \times 3 \times 2 \times 1 \quad \text{(recursion is terminated)}
\]

Finding a Recursive Solution

• Given the following problem: Flip("recursion")

1. Find a smaller subproblem:

```
    Flip("ecursion")
```

2. Define a solution in terms of the subproblem:

```
    return Flip("ecursion") + r
```

3. Combine with original problem for a general solution:

```
    return Flip(s.substring(1)) + s.charAt(0)
```

• What is the base case?

```
    Flip("") = ""
```

• Flip("recur")

\[
= \text{Flip("ecur") + 'r'} \\
= (\text{Flip("cur") + 'e'}) + 'r' \\
= ((\text{Flip("ur") + 'c'}) + 'e') + 'r' \\
= (((\text{Flip("r") + 'u'}) + 'c')) + 'e') + 'r' \\
= (((((\text{Flip(""}) + 'r'}) + 'u')) + 'c') + 'e') + 'r'
\]
Subproblem

- The key is defining the subproblem
- The recursive step MUST reduce the problem size (or you will end up with infinite recursion).
- You must be able to split the problem to make a recursive call.
- When the subproblem is “obvious”, you have the base case.
- Every problem must end with a base case.

Recursive Sum of Numbers

- Consider the sum of numbers:
  \[ \sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i \]
  \[ = N + N - 1 + \sum_{i=1}^{N-2} i \ldots \]
- What is the base case?
- What is the recursive case?

```java
public int sum (int num)
{
    int result;
    if (num == 1)
        result = 1;
    else
        result = num + sum (num-1);
    return result;
}
```

Recursion vs. Iteration

- The non-recursive solution to the summation problem compute the numbers between 1 and num in an iterative manner.

```java
sum = 0;
for (int i = 1; i <= num; i++)
    sum += i;
```
- This solution is more straight forward. The recursive solution however, demonstrates the concept of recursion using a simple problem.
- In many problems (though not all!), recursion is the shorter and more elegant solution.

Two Solutions to Powers

- Base case: \(x^0 = 1\)

```java
long pow(long x, long y)
{
    if (y==0) return 1;
    else return (x * pow(x, y-1));
}
```

- \(x^y = x^{y/2} \times x^{y/2}\)

```java
long pow(long x, long y)
{
    long tmp;
    if (y==0) return 1;
    else if (y%2==0) //even
        { 
            tmp = pow(x, y/2); return tmp * tmp;
        }
    else //odd
    { 
        tmp = pow(x, (y-1)/2); return tmp * tmp * x;
    }
}
```
Direct vs. Indirect Recursion

- Direct recursion occurs when a method invokes itself.
- Indirect recursion occurs when a method invokes another method, eventually resulting in the original method being invoked again.

Using Recursion—Traversing a Maze

- Solving a maze involves a great deal of trial and error: following a path, backtracking when you reach a wall, trying other possible paths. It’s an activity nicely solved using recursion.
- The Maze class is a two-dimensional array of integers representing a maze.

1110110001111
1011101111001
000001010100
1110111010111
1010000111001
1011111101111
1000000000000
1111111111111

- The goal is to move from the top left corner to the bottom right corner using only valid directions: down, right, up and left.
- Initially, a 1 indicates a clear path and a 0 indicates a blocked path.

Recursive Maze Strategy

- Think of the initial conditions and the final goal:
- The maze can be traversed successfully if it can be traversed from position (0,0), and therefore any position adjacent, namely (1,0), (0,1), (-1,0) or (0,-1).
- Picking a potential next step, say (1,0), we find ourselves in the same position: the maze can be traversed successfully if it can be traversed from position (1, 0), and from any adjacent position.
- This process is continued recursively.
The class Maze.java

```java
public class Maze {
    private final int TRIED = 3;
    private final int PATH = 7;

    private int[][] grid = {
        {1,1,1,0,1,0,0,0,1,1,1,1},
        {1,0,1,1,1,0,1,1,1,0,0,1},
        {0,0,0,1,0,1,1,0,1,0,1,0},
        {1,1,0,1,1,1,1,1,0,1,1,1},
        {1,0,1,0,0,0,0,1,1,0,1,1},
        {1,0,1,1,1,1,1,1,1,1,1,1},
        {1,1,1,1,1,1,1,1,1,1,1,1},
    };

    public boolean traverse(int row, int col) {
        boolean done = false;
        if (valid(row, col)) {
            grid[row][col] = TRIED;
            // base case;
            ...
            // recursive case;
            ...
            if (done) // this location is part of the final path
                grid[row][col] = PATH;
        }
        return done;
    }

    private boolean valid(int row, int col) {
        boolean result = false;
        // check if cell is in the bounds of the matrix
        ...
        // check if cell is not blocked and not previously tried
        if (grid[row][col] == 1)
            result = true;
        return result;
    }

    public String toString() {
        String result = "\n";
        for (int row = 0; row < grid.length; row++)
            for (int col = 0; col < grid[row].length; col++)
                result += grid[row][col] + " ";
        return result + "\n";
    }
}
```

Traverse method for Maze.java

// Attempts to recursively traverse the maze. Inserts special
// characters indicating locations that have been tried and
// that eventually become part of the solution.

```java
public boolean traverse(int row, int col) {
    boolean done = false;
    if (valid(row, col)) {
        grid[row][col] = TRIED;
        // base case;
        ...
        // recursive case;
        ...
        if (done) // this location is part of the final path
            grid[row][col] = PATH;
    }
    return done;
}
```

Discussion of Maze

• The recursive method in the Maze class is called `traverse`. It returns a boolean indicating whether a solution was found.
• The initial call to `traverse` passes in the upper-left location (0,0).
• What is the base case?
• There are 3 possible base cases that will terminate any particular recursive path:
  1. an invalid move because the move is out of bounds
  2. an invalid move because the move has already been tried
  3. a move arrives at the final location
• The traverse method determines whether the maze has been completed by checking for the bottom-right location.
Towers of Hanoi

- The *Towers of Hanoi* puzzle was invented in the 1880s by Douard Lucas, a French mathematician.
- The puzzle consists of three pegs on which a set of disks, each with a different diameter, may be placed.
- Initially the disks are stacked on the leftmost peg, in order of size, with the largest disk on the bottom.

![Figure 3: The Towers of Hanoi puzzle.](image1)

Rules for Towers of Hanoi

- The goal of the puzzle is to move all the disks from the leftmost peg to the rightmost peg, adhering to the following rules:
  1. Move only one disk at a time.
  2. A larger disk may not be placed ontop of a smaller disk.
  3. All disks, except the one being moved, must be on a peg.

![Figure 4: Three-disk solution to the Towers of Hanoi puzzle.](image2)

Hanoi Strategy

- The rules imply that smaller disks must be “out of the way” to move larger disks from one peg to another.
- General strategy for moving *N* disks from the original peg to the destination peg:
  1. Move the topmost *N* − 1 disks from the original peg to the extra peg.
  2. Move the largest disk from original peg to destination peg.
  3. Move *N* − 1 disks from the extra peg to the destination peg.

![Figure 5: Reduced problem of the Towers of Hanoi puzzle.](image3)
Recursive Solution

• This strategy lends itself to a recursive solution.
• Step 1 and 3 are the same problems over and over again: move a stack of disks.
  – the problem size is reduced each time;
  – what’s referred to as the destination and extra peg may change each time;
• The base case for this problem occurs when we want to move a "stack" that consists of only one disk. This step can be accomplished without recursion.

Code for Towers of Hanoi

```java
public void solve()
{
    moveTower(totalDisks, 1, 3, 2);
}

private void moveTower(int numDisks, int start, int end, int temp)
{
    if (numDisks== 1)
        moveOneDisk(start, end);
    else
    {
        moveTower(numDisks-1, start, temp, end);
        moveOneDisk(start, end);
        moveTower(numDisks-1, temp, end, start);
    }
}

// prints instructions to move one disk
private void moveOneDisk(int start, int end)
{
    System.out.println("Move one disk from " + start + " to " + end);
}
```

Discussion of Tower Solution

• The initial call indicates all disks should be moved from peg 1 to peg 3, using peg 2 as the extra position.
• When the base case occurs in the moveTower method, one disk is moved.
• If the stack contains more than one disk, moveTower is called recursively to
  1. get \( N - 1 \) disks out of the way
  2. move the largest disk
  3. move \( N - 1 \) disks to their final destination
• Note the parameters to moveTower describing the original, destination, and extra peg are switched.
• This solution is actually inefficient. To solve for \( N \) disks, we must make \( 2^N - 1 \) individual disk moves. This is an example of exponential complexity.
• As the number of disks increases, the number of required moves increases exponentially.

Merge Sort

• Now that we’ve seen recursion, we can look at another sorting algorithm called merge sort, which is more efficient than insertion and selection sort.
• It is usually defined recursively, where the recursive calls work on smaller and smaller parts of the list.
• Idea:
  – Split the unsorted list into two sublist of about half the size
  – Recursively sort each half
  – "Merge" the two sorted halves into a single sorted list
Merge Sort Example

- Based on a divide and conquer strategy:
  - list is divided into two halves (divide)
  - each half is sorted independently (conquer).
  - two halves are merged into a sorted sequence
- Split original list recursively until base case is reached

![Figure 6: Recursive sort.](image)

Merge Step

- Must put the next smallest element into the merge list at each point.
- Each next smallest could come from either list.
- Merging requires checking the smallest element of each half, requiring just one comparison step. This is where the recursive call saves time.

Merge Sort Algorithm

- Pseudocode

```pseudocode
mergeSort(array, first, last):
    // sort array[first] to array[last-1]
    if last - first <= 1:
        return // length 0 or 1 already sorted
    mid = (first + last)/2
    mergeSort(array, first, mid) // recursive call 1
    mergeSort(array, mid, last) // recursive call 2
    merge(array, first, mid, last)
```

- Java code:

```java
public static void mergesort(int[] array, int first, int last)
{
    if (last-first > 1)
    {
        int mid = (first + last)/2;
        mergesort(array, first, mid); // recursive call 1
        mergesort(array, mid, last); // recursive call 2
        merge(array, first, mid, last);
    }
}
```

- Merge Algorithm

```java
merge(array, first, mid, last):
    // merge array[first to mid-1] and array[mid to last-1]
    leftpos = first
    rightpos = mid
    for newpos from 0 to last-first:
        if array[leftpos] <= array[rightpos]:
            newarray[newpos] = array[leftpos]
            leftpos++
        else:
            newarray[newpos] = array[rightpos]
            rightpos++
    copy newarray to array[first to (last-1)]
```

- What is wrong with the above code?
  - The algorithm starts correctly but has an error as it finishes:
    - Eventually one of the halves will be empty and the ‘if’ will compare against the element past one of the halves.
  - Solution: compare only until we reach the end of one half, then just copy the rest over.
Corrected Merge Algorithm

merge(array, first, mid, last):
//merge array[first to mid-1] and array[mid to last-1]
leftpos = first
rightpos = mid
newpos = 0
while leftpos<mid and rightpos<=(last-1):
    if array[leftpos] <= array[rightpos]:
        newarray[newpos] = array[leftpos]
        leftpos++; newpos++
    else:
        newarray[newpos] = array[rightpos]
        rightpos++; newpos++
while leftpos<mid: //copy the rest left half (if any)
    newarray[newpos] = array[leftpos]
    leftpos++; newpos++
while rightpos<=last-1 //copy the rest of the right half (if any)
    newarray[newpos] = array[rightpos]
    rightpos++; newpos++
copy newarray to array[first to (last-1)]

Running Time

- What is the running time for merge sort?
- recursive calls × work per call? But work per call changes...
- We know the merge algorithm is $O(N)$ to merge $N$ elements.

Merge sort recursive calls

- FIX: include figure here.
- Each level has a total of $N$ elements.
- Steps to merge each level: $N$
- Number of levels: $\log N$
- Total Running Time: $O(N \log N)$
- Much faster than selection/insertion sort which both take $O(N^2)$
- In general, no sorting algorithm can do better than $O(N \log N)$ (though some are faster for limited cases).

Difference in Memory Requirements?

- Merging requires extra storage: an extra array with $N$ elements which can be re-used by all merges.
- Insertion sort only requires a few storage variables
- An algorithm that uses at most $O(1)$ extra storage is called “in-place”.
- Insertion/section sort are both in-place algorithms.
- Merge sort is not.
Stable Sort

- In a **stable** sorting algorithm, equal elements are kept in order.
- Insertion sort and merge sort have this property, selection sort does not.