Recursion

- Divide and Conquer approach to problem solving
  - If you can't solve the big problem directly, maybe you can solve it with the solution to a smaller one
  - If you can't solve the smaller one directly, maybe you can solve it with the solution to a still smaller one …
  - Surely you can solve a trivial problem
  - Recursion is like reflections in mirrors opposite each other

- Recursion is kind of like induction in reverse
  - With induction, you prove a trivial base case
  - Then prove that you can get from one case to the next case
  - Voila - you have proved all cases, no matter how large

Recursive Concepts

- Base Cases
  - Problems that can be solved with primitive operators or are trivial
  - May be more than one base case

- Recursive Cases
  - Partition input of the problem
  - Solution uses results of solving the same problem on the partitions
  - Well founded recursion moves toward base case
### Programming Model

- if (base_case)
  - return simple_result;
- else
  - partition input into smaller cases;
  - call this function recursively on them;
  - return combined results;

- A recursive function calls itself
  - Each recursive call solves an identical, but smaller problem
  - Recursion eventually stops with the base case
  - Function may also call itself indirectly (through another function) – this is still recursion

### Constructing Recursive Solutions

- Questions to ask:
  - How can you define problem in terms of a smaller problem of the same type?
  - How does each recursive call diminish the size of the problem?
  - What is the base case?
  - As the problem size shrinks, will it reach the base case? (well founded recursion)
A Simple Example of Recursion

- The factorial of a number n, written n! is the product of all the numbers from 1 to n.
- Problem: calculate n!
  - If we knew the factorial of n-1, then we could multiply it by n to get n!
    - This partitions the problem for n into a smaller problem
  - Base case: The factorial of 1 is 1
  - Recursion is well formed since subtracting 1 each time will eventually lead to the base case

Factorial Computation by Recursion

- As an algorithm:
  \[
  n! = \begin{cases} 
  1 & \text{if } n == 1 \\
  n \cdot (n-1)! & \text{if } n > 1 
  \end{cases}
  \]
- As a program:
  ```java
  int factorial(int n) {
    int result;
    if (n <= 1)  result = 1;
    else {
      int partial = factorial(n-1);
      result = n * partial;
    }
    return result;
  }
  ```
Recursive Computation

\[

t_5 = 5! \\
t_4 = 5 \times 4! \\
t_3 = 4 \times 3! \\
t_2 = 3 \times 2! \\
t_1 = 2 \times 1! \\
1
\]

Recursive Function Calls

- The function \texttt{factorial} calls itself
- Each call has its own separate environment: an activation block containing
  - The local variable \texttt{result}
  - The local variable \texttt{partial}
  - Where to return, and value to return
  - This is like scratch space to work in
  - The "next" call must complete before current can return
  - And the next ... and the next ...
- Can be significant memory use for deep recursion
  - May exhaust memory space (stack overflow)
Recursive Data Structures

- Data structures can be recursively defined
- A list is:
  - an element (base case)
  - an element and a list (recursive case)
- Likewise, an array could be thought of this way
- A tree is a set of nodes
  - Nodes are either leaves or trees
- Recursive data structures may contain recursive methods or be processed by recursive methods

Template for a recursive list of integers in Java using inheritance

A list is defined as a value and a list, or is empty

```java
class List {
    private int first; // The element value
    private List rest; // The rest of the list

    int length() { // The number of items in the list
        return rest != null ? rest.length() + 1 : 1;
    }
    boolean isEmpty() { // Determine if the list is empty
        return rest == null;}
    boolean isMember(int n) { // Determine if value is in list
        return rest != null ? rest.isMember(n) : first == n;
    }
    List add(int n) { // Add a value to end of the list
        return new List(first == n ? rest : n, rest);}
    List remove(int n) { // Remove a value from the list
        return rest != null ? rest.remove(n) : null;
    }
}
```
Recursive List

- EmptyList is the base case
- Since it extends List, an EmptyList *is a* List

```java
class EmptyList extends List {
    int length() { return 0; }
    boolean isEmpty() { return true; }
    boolean isMember(int n) { return false; }
    List add(int n) { return new List(n); }
    List remove(int n) { return this; }
}
```

List Definition – Recursive Implementation

- Class defined recursively
- Most methods are recursive, too

```java
class List {
    private int first;
    private List rest;

    List(int n) { first = n; rest = new EmptyList(); }
    List(int n, List r) { first = n; rest = r; }

    protected List() {} // For EmptyList

    int length() { return 1 + rest.length(); }
    boolean isEmpty() { return false; }
    ...
```
List Definition (continued)

- Recursion uses partitioning by `first` and `rest`

  ```java
  boolean isMember(int n) {
      return first==n || rest.isMember(n);
  }
  List remove(int n) {
      if (first == n) return rest;
      else return new List(first, rest.remove(n));
  }
  List add(int n) {
      return new List(first, rest.add(n));
  }
  ```

Recursive List Implementation

- Partitioning done by ElementList data
  - a single element
    and
  - a smaller list
- Base case is EmptyList
- Selection of base or recursion done by dynamic binding of polymorphism
  - Abstract List object can be either base case
    (EmptyList) or non-empty list (ElementList)
Proving Correctness

- Prove the correctness of recursive methods using induction
  - Show true for base case type
  - Assume true for up to size n-1
  - Show true for size n
- Prove the length() method is correct
  - Length of an EmptyList object is zero by definition
  - Assume length() returns n-1 for list of size n-1
  - Then for list of size n, length() returns 1 + (n-1), which is n
- Proof for isMember, ...

Recursion on Arrays

- Partition the array by passing beginning and ending indices
- Initially entire Array, e.g.,
  \[ \text{recFunc}(a, 0, a.\text{length}-1, \ldots) \]
- General form
  \[ \text{recFunc}(a, \text{low}, \text{high}, \ldots) \]
- Base case: when low is not less than high
- Must make sure partitions get smaller
Recursion on Arrays

- Recursive helper method `recIsMember` for `isMember` of an unordered array

```java
private E[] a;
public boolean isMember(E o) {
    return recIsMember(0, a.length-1, o);
}
private boolean recIsMember(int low, int high, E o) {
    if (low > high)  return false;
    else if (low == high) return a[low].equals(o);
    else   return recIsMember(low,low,o) ||
                recIsMember(low+1,high,o);
}
```

Proving Correctness

- Prove the correctness of `recIsMember()`
- Base cases - range of size 0 or 1
  - No elements in range, so object can't be member or
  - One element in range - compare to target
- Recursive case
  - Assume `recIsMember` is correct for range of size n-1
  - For a range of size n, partition into two ranges, the first element and the remaining n-1
  - We are progressing toward base case with this partition
  - `recIsMember` works correctly for each partition by induction assumption
  - Logical OR is the correct way to combine results
Another Partitioning

- Partition in half

private boolean recIsMember(int low, int high, E o) {
    if (low > high)  return false;
    else if (low == high) return a[low].equals(o);
    else   return recIsMember(low,(low+high)/2,o) ||
               recIsMember((low+high)/2 +1,high,o);
}

- Correctness proof – still two smaller partitions

Binary Search

- Assume array is sorted, elements comparable

public boolean isMember(E o) {
    return recBinSearch(0, a.length-1, o);
}

private boolean recBinSearch(int low, int high, E o) {
    if (low > high)  return false;
    int mid = (low + high)/2;
    if (a[mid].equals(o))
        return true;
    else if (a[mid].lessThan(o))
        return recBinSearch(mid+1, high, o)
    else
        return recBinSearch(low, mid-1, o);
}
Proving Correctness of recBinSearch

- Base cases – range of size 0 or 1
  - No elements in range, so object can’t be member or
  - Target matches single (midpoint) element otherwise zero length case

- Recursive case
  - Assume recBinSearch is correct for range of size less than n
  - For a range of size n, partition into two ranges, excluding the midpoint
  - We are progressing toward base case with this partition
  - Works correctly for each partition by induction assumption
  - Logical OR (if-elseif-else) is the correct way to combine results taking sorting into consideration

Towers of Hanoi

- Goal: move disks from first to last peg
  - Only one disk can be moved at a time
  - Cannot place a larger disk on top of a smaller one
  - Disks have stay on a peg (except moving one)
Partitioning Towers of Hanoi

- Suppose we have 4 disks
  - If we could get the top 3 disks moved to the center peg, we would have:

  ![Diagram of 4 disks](image1)

  - Then we could move the big disk to the last peg

  ![Diagram with big disk moved](image2)

- Now if we knew how to move a stack of 3 ...

Towers of Hanoi

- Putting it together: moving the top 3 requires first moving the top 2 which requires moving the top 1 ...

![Diagram of towers of Hanoi](image3)
Towers of Hanoi Recursive Solution

- Base case: 1 disk
  - Just move it to desired peg
- Recursive case: move n disks from start peg to end peg, using other peg
  - Move n-1 disks to other peg (using end peg)
  - Move remaining disk to end peg
  - Move n-1 disks to end peg (using start peg)

```java
void solveTowers(int count, int start, int end, int tmp)
{ // Invariant: no larger disk on top of smaller
  if (count == 1)
    print("Move top disk from peg " + start
           + " to peg " + end);
  else {
    solveTowers(count-1, start, tmp, end);
    solveTowers(1, start, end, tmp);
    solveTowers(count-1, tmp, end, start);
  }
}
```

Towers of Hanoi Recursive Solution

- Implemented in Java
Towers of Hanoi Proof

- Prove solveTowers works by induction
  - Base case of one disk obviously works
  - Assume it works for n-1 or fewer disks
  - Partition into 3 smaller problems: n-1, 1, n-1
  - Combination of results is correct since bottom disk is moved to end peg, and n-1 disks moved after
- Note that each call to solveTowers may have different values for the pegs
  - Initial call looks like:
    \[
    \text{solveTowers}(n, 1, 3, 2);
    \]

Recursion and Complexity

- To determine complexity of iterative algorithm, we determine complexity of the iteration and the complexity of the work to be done in each iteration
  - Essentially, we are counting iterations
  - Multiply Big-O of looping times Big-O of work performed in loop
- But how to determine complexity of a recursive algorithm?
  - Need to know how deep is recursion, e.g., how many times a function calls itself
Recurrence Relations

- Find a recurrence relation for the time in terms of the size of the problem
- Determine the Big-O of the recurrence relation
- Basically, we want a function $T(n)$, where
  - We know $T(0)$, $T(1)$ - the base case(s)
  - $T(n)$ defined in terms of $T(k)$ for $k < n$
  - $T(n)$ reflects structure of algorithm
- How to find $T(n)$?
  - Substitute $T$ for function name
  - Arguments to function become "size" of problem
  - Determine how many times function is called and how much work is done at that level

Solving Recurrence Relations

- Unroll (as in list, tree)
- Look for general patterns
- Use general theorems
- Ask a mathematician

- Solving can be an art
Palindrome Example

- Method to determine if a string is a palindrome (reads same forwards and backwards)
  - Algorithm: compare ends, working toward middle

```java
boolean isPalindrome(String s) {
    return recIsPalindrome(s, 0, s.length()-1);
}

boolean recIsPalindrome(String s, int low, int high) {
    if (low >= high)  return true;
    else  return ( (s.charAt(low) == s.charAt(high))
                     && recIsPalindrome(s, low+1, high-1) );
}
```

Palindrome Algorithm Analysis

- Base cases - zero or one character in range
  - $T(0) = T(1) = c$
- Recurrence relation
  - $T(n) = c + T(n-2)$
- Unroll recurrence relation
  - $T(n) = c + T(n-2) = c + c + T(n-4) = ...$
    - $c + c + ... + c = c \cdot (n/2)$
- Conclusion:
  - $T(0) = T(1) = O(1)$
  - $T(n) = O(n/2) = O(n)$
  - So, recIsPalindrome is $O(n)$
Analysis of Recursion on Arrays

Recall method `recIsMember` for unordered array

```java
boolean recIsMember(int low, int high, E o) {
    if (low > high) return false;
    else if (low == high) return a[low].equals(o);
    else return recIsMember(low,low,o) ||
           recIsMember(low+1,high,o);
}
```

- $T(0) = c$, $T(1) = c$
- $T(n) = T(1) + T(n-1) = c + T(n-1) = c + c + T(n-2) = \ldots = c \cdot n$
- By unrolling, we see that algorithm is $O(n)$

Find minimum for unordered array of comparable elements

```java
E recMinElt(int low, int high, E o) {
    if (low == high) return a[low];
    else return minOf( recMinElt(low, (low+high)/2)),
                     recMinElt(1+(low+high)/2, high) );
}
```

```java
E minOf(E e1, E e2) {
    if (e1.lessThan(e2)) return e1; else return e2;
}
```

- $T(0) = T(1) = c$
- $T(n) = c + 2 \cdot T(n/2) = c + 2 \cdot c + 2 \cdot 2 \cdot T(n/4) = \ldots$
  
  $= c \cdot (1+2+4+\ldots+2^{\log n}) = 2^{1+\log n} - 1 = 2n-1$
- By unrolling, we see that algorithm is $O(n)$
Analysis of Recursion on Arrays

- Binary search of an ordered array of comparable elements

```java
boolean recBinSearch(int low, int high, E o) {
    if (low > high)  return false;
    int mid = (low + high)/2;
    if (a[mid] == o) return true;
    else if (a[mid] < o) return recBinSearch(mid+1, high, o);
    else return recBinSearch(low, mid-1, o);
}
```

- \( T(0) = T(1) = c \)
- \( T(n) = c + T(n/2) = c + c + T(n/4) = \ldots \)
  \[= c \cdot (1+1+1+\ldots+1) \] (for \( \log_2 n \) times)

- By unrolling, we see that algorithm is \( O(\log n) \)