Recursion

- A recursive definition is one which is defined in terms of itself
- Example:
  - Compound interest: “The value after 10 years is equal to the interest rate times the value after 9 years.”
  - A phrase is a “palindrome” if the 1st and last letters are the same, and what’s inside is itself a palindrome (or is empty).

Computer Science Examples

- Recursive procedure: a procedure that invokes itself
- Recursive data structures: a data structure may contain a pointer to an instance of the same type

```c
struct Node {
    int data;
    Node *next;
};
```

- Recursive definitions: if \( A \) and \( B \) are postfix expressions, then \( A B + \) is a postfix expression

Factorial

\( n! (\text{ “n factorial”}) \) can be defined in two ways:

- Non-recursive definition
  \( n! = n \times (n-1) \times (n-2) \ldots \times 2 \times 1 \)

- Recursive definition
  \[
  n! = \begin{cases} 
  1 & , \text{if } n = 1 \\
  n \times (n-1)! & , \text{if } n > 1 
  \end{cases}
  \]

0! is usually defined to be 1

Factorial (2)

- How do we write a function that reflects the recursive definition?

```c
int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

- Note that the `factorial` function calls itself.
- How can this work?

Activation Records

- Remember that local variables and parameters are allocated when a function is entered, deleted when the function exits (automatic storage).
- Here’s how:
  - Whenever a function is called, a new activation record is pushed on the runtime stack, containing:
    - a separate copy of all local variables and parameters
    - control info (e.g. return address)
  - A recursive function call is no different in this respect
  - Each recursive call has its own copy of locals
Example

```cpp
int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
...
int main (void) {
    int x = factorial(4);
    cout << "4! = " << x << endl;
    ...
```

Infinite Recursion

- Must always have some way to make recursion stop, otherwise it runs forever:
  ```cpp
  int BadFactorial (n) {
      int x = BadFactorial (n-1);
      if (n == 1)
          return 1;
      else
          return n * x;
  }
  ```
- What is the value of BadFactorial(2)?

Using Recursion Properly

- For correct recursion (recursion that does something useful and eventually stops), need two parts:
  1. One or more **base cases** that are not recursive
     ```cpp
     if (n == 1) return 1;
     ```
  2. One or more **recursive cases** that operate on smaller problems that get closer to the base case(s)
     ```cpp
     return n * factorial(n-1);
     ```
- The base case(s) should always be checked before the recursive calls

Recursive Data Structures

- Many data structures are defined in terms of (pointers to) other instances of themselves
- Linked Lists and Trees (coming soon) are examples:
  ```cpp
  struct Node {
      int   data;
      Node  *next;
  };
  ```
- Recursive data structures suggest recursive algorithms

Printing a Linked List

```cpp
void print(Node* first) {
    if (first == NULL)
        return;
    else {
        cout << first->data << " ";
        print(first->next);
    }
}
```
Printing in Reverse Order
At first, seems difficult
All the pointers point only forward.
Recursion to the rescue!

```cpp
void print(Node* first) {
    if (first == NULL)
        return;
    else {
        print(first->next);
        cout << first->data << " ";
    }
}
```

Summing a List
```cpp
int listSum(Node* list) {
    if (list == NULL) // empty list has sum == 0
        return 0;
    else
        return list->data + listSum(list->next);
}
```

- How would you modify this to count the length of a list? Add N to each element of a list?
- What is the cost in time compared to a loop? How much space does it take?

List Remove
• Remove nodes with a given data value from list
```cpp
Node* ListRemove(Node *first, int v){
    if (first == NULL)
        return NULL;
    else if (first->data != v){
        Node* newNode = new Node;
        newNode->data = first->data;
        newNode->next = ListRemove(first->next, v);
        return newNode;
    }
    else
        return ListRemove(first->next, v);
}
```

Recursion: Not Just for Lists
```cpp
double sum (double iArray [], int from, int to) {
    //find the sum of all elements in the array between "from" and "to"
    if (from > to)
        return 0.0;
    return iArray[from] + sum (iArray, from+1, to);
}
```
```cpp
double CashValues[200];
...
double total = sum (CashValues, 0, 199);
```

Insist without Iterating
```cpp
char InsistOnYorN (void) {
    char answer;
    cout << endl << "Please enter y or n: ";
    cin >> answer;
    switch (answer) {
    case 'y': return 'y';
    case 'n': return 'n';
    default:
        return InsistOnYorN();
    }
}
```

What does this function do?
```cpp
int mystery (int x) {
    assert (x > 0);
    if (x == 1)
        return 0;
    int temp = mystery (x / 2);
    return 1 + temp;
}
How about this one

```c
int g(int n)
{
    if (n <= 1)
        return 1;
    else if (n % 2 == 0) // n even
        return 1 + g(n / 2);
    else       // n odd
        return 1 + g(3*n + 1);
}
g(7) = 1 + g(22) = 2 + g(11) = 3 + g(34) =
  4 + g(17) = 5 + g(52) = 6 + g(26) = 7 + g(13) =
  8 + g(40) = 9 + g(20) = 10 + g(10) = 9 + g(5) =
  10 + g(16) = 11 + g(8) = 12 + g(4) =
  13 + g(2) = 14 + g(1) = 15
```

Recursion vs. Iteration

- When to use recursion?
  - Processing recursive data structures
  - "Divide & Conquer" algorithms:
    1. Divide problem into subproblems
    2. Solve each subproblem recursively
    3. Combine subproblem solutions
- When to use iteration instead?
  - Nonrecursive data structures
  - Problems obvious iterative structure
  - Any iteration can be rewritten using recursion, and vice-versa (at least in theory)
  - Iteration generally uses less (stack) space and is somewhat faster

Which is Better?

- If a single recursive call is at the very end of the function:
  - Known as tail recursion
  - Easy to rewrite iteratively
- Recursive problems that are not tail recursive are harder to write nonrecursively
  - Usually have to simulate recursion with a stack
- Some programming languages provide no loops (loops are implemented through if and recursion)

Summary

- Recursion is something defined in terms of itself
  - Recursive procedures
  - Recursive data structures
  - Recursive definitions
- Activation records make it work
  - Two parts of all recursive functions
    - Base case(s)
    - Recursive case(s)
    - Base case always checked first
- Examples
  - Factorial (n!)
  - Recursive linked list manipulation