What is recursion?

- Generally, when something contains a reference to itself
- Math: defining a function in terms of itself
- Computer science: when a function calls itself:

```cpp
void message() {
    cout << "This is a recursive function.\n";
    message();
}
int main() {
    message();
}
```

What happens when this is executed?

How can a function call itself?

• Infinite Recursion:

```cpp
void message() {
    cout << "This is a recursive function.\n";
    message();
}
int main() {
    message();
}
```

Recursive message() modified

• How about this one?

```cpp
void message(int n) {
    if (n > 0) {
        cout << "This is a recursive function.\n";
        message(n-1);
    }
}
int main() {
    message(5);
}
```
Tracing the calls

• 6 nested calls to message:
  message(5):
  - outputs “This is a recursive function”
  calls message(4):
  - outputs “This is a recursive function”
  calls message(3):
  - outputs “This is a recursive function”
  calls message(2):
  - outputs “This is a recursive function”
  calls message(1):
  - outputs “This is a recursive function”
  calls message(0):
  - does nothing, just returns

• depth of recursion (#times it calls itself) = 5

Why use recursion?

• It is true that recursion is never required to solve a problem
  - Any problem that can be solved with recursion can also be solved using iteration.
• Recursion requires extra overhead: function call + return mechanism uses extra resources

However:
• Some repetitive problems are more easily and naturally solved with recursion
  - the recursive solution is often shorter, more elegant, easier to read and debug.

How to write recursive functions

• Branching is required (If or switch)
• Find a base case
  - one (or more) values for which the result of the function is known (no repetition required to solve it)
  - no recursive call is allowed here
• Develop the recursive case
  - For a given argument (say n), assume the function works for a smaller value (n-1).
  - Use the result of calling the function on n-1 to form a solution for n

Recursive function example

factorial

• Mathematical definition of n! (factorial of n)
  \[
  \begin{align*}
  &\text{if } n=0 \text{ then } n! = 1 \\
  &\text{if } n>0 \text{ then } n! = 1 \times 2 \times 3 \times \ldots \times n
  \end{align*}
  \]
• What is the base case?
  - n=0 (the result is 1)
• Recursive case: If we assume (n-1)! can be computed, how can we get n! from that?
  - n! = n \times (n-1)!
Recursive function example

factorial

```c
int factorial(int n) {
    if (n==0)
        return 1;
    else
        return n * factorial(n-1);
}

int main() {
    int number;
    cout << "Enter a number ";
    cin >> number;
    cout << "The factorial of " << number << " is " << factorial(number) << endl;
}
```

Tracing the calls

- Calls to factorial:
  - factorial(4):
    - return 4 * factorial(3); ≈ 4 * 6 = 24
    - calls factorial(3):
      - return 3 * factorial(2); ≈ 3 * 2 = 6
      - calls factorial(2):
        - return 2 * factorial(1); ≈ 2 * 1 = 2
        - calls factorial(1):
          - return 1 * factorial(0); ≈ 1 * 1 = 1
          - calls factorial(0):
            - return 1;

  - Every call except the last makes a recursive call
  - Each call makes the argument smaller

Recursive functions over ints

- Many recursive functions (over integers) look like this:

```c
type f(int n) {
    if (n==0)
        //do the base case
    else
        // ... f(n-1) ...
}
```

Recursive functions over lists

- You can write recursive functions over lists using the length of the list instead of n
  - base case: length=0 ==> empty list
  - recursive case: assume f works for list of length n-1, what is the answer for a list with one more element?
- We will do examples with:
  - arrays
  - strings
  - later: linked lists
Recursive function example

sum of the list

- Recursive function to compute sum of a list of numbers
- What is the base case?
  - length=0 (empty list) => sum = 0
- If we assume we can sum the first n-1 items in the list, how can we get the sum of the whole list from that?
  - sum (list) = sum (list[0..n-2]) + list[n-1]
  
  Assume I am given the answer to this

Recursive function example

sum of a list (array)

```c
int sum(int a[], int size) {  //size is number of elems
    if (size==0)
        return 0;
    else
        return sum(a,size-1) + a[size-1];
}
```

For a list with size = 4: sum(a,4)

sum(a,3) + a[3] =
sum(a,2) + a[2] + a[3] =

Recursive function example

count character occurrences in a string

- Recursive function to count the number of times a specific character appears in a string
- We will use the string member function substr to make a smaller string
  - str.substr (int pos, int length);
  - pos is the starting position in str
  - length is the number of characters in the result

string x = “hello there”;
cout << x.substr(3,5);  //lo th

- char access: x[1] is the second element (‘e’)
Three required properties of recursive functions

- **A Base case**
  - a non-recursive branch of the function body.
  - must return the correct result for the base case
- **Smaller caller**
  - each recursive call must pass a smaller version of the current argument.
- **Recursive case**
  - assuming the recursive call works correctly, the code must produce the correct answer for the current argument.

Recursive function example greatest common divisor

- **Greatest common divisor of two non-zero ints is the largest positive integer that divides the numbers evenly (without a remainder)**
- **This is a variant of Euclid’s algorithm:**
  \[
  \text{gcd}(x,y) = \begin{cases} 
  y & \text{if } y \text{ divides } x \text{ evenly, otherwise:} \\
  \text{gcd}(y, \text{remainder of } x/y) & (\text{or } \text{gcd}(y,x\%y) \text{ in C++})
  \end{cases}
  \]
- **It’s a recursive definition**
  - If \( x < y \), then \( x\%y \) is \( x \) (so \( \text{gcd}(x,y) = \text{gcd}(y,x) \))
  - This moves the larger number to the first position.

Recursive function example greatest common divisor

- **Code:**

  ```cpp
  int gcd(int x, int y) {
    cout << "gcd called with " << x << " and " << y << endl;
    if (x % y == 0) {
      return y;
    } else {
      return gcd(y, x % y);
    }
  }
  
  int main() {
    cout << "GCD(9,1): " << gcd(9,1) << endl;
    cout << "GCD(1,9): " << gcd(1,9) << endl;
    cout << "GCD(9,2): " << gcd(9,2) << endl;
    cout << "GCD(70,25): " << gcd(70,25) << endl;
    cout << "GCD(25,70): " << gcd(25,70) << endl;
  }
  ```

- **Output:**

  
  *gcd called with 9 and 1*
  GCD(9,1): 1
  gcd called with 1 and 9
  gcd called with 9 and 1
  GCD(1,9): 1
  gcd called with 9 and 2
  gcd called with 2 and 1
  GCD(9,2): 1
  gcd called with 70 and 25
  gcd called with 25 and 20
  gcd called with 20 and 5
  GCD(70,25): 5
  gcd called with 25 and 70
  gcd called with 70 and 25
  gcd called with 25 and 20
  gcd called with 20 and 5
  GCD(25,70): 5*
Recursive function example
Fibonacci numbers

• Series of Fibonacci numbers:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

• Starts with 0, 1. Then each number is the sum of the two previous numbers
  \( F_0 = 0 \)
  \( F_1 = 1 \)
  \( F_i = F_{i-1} + F_{i-2} \)  (for \( i > 1 \))

• It’s a recursive definition

Recursive function example
Fibonacci numbers

• Code:

```cpp
int fib(int x) {
    if (x<=1)       //takes care of 0 and 1
        return x;
    else
        return fib(x-1) + fib(x-2);
}
```

```cpp
int main() {
    cout << "The first 13 fibonacci numbers: " << endl;
    for (int i=0; i<13; i++)
        cout << fib(i) << " ";
    cout << endl;
}
```

The first 13 fibonacci numbers:
0 1 1 2 3 5 8 13 21 34 55 89 144

Recursive functions over linked lists

• Recursive functions can be members of a linked list class
  - These functions take a pointer to the list (p) as a parameter
  - They compute the function for the list starting at the node p points to.

• The pattern:
  - base case: empty list, when p is NULL
  - recursive case: assume f works for list starting at p->next, what is the answer for a list with one more element (the list starting at p)?
Recursive function example
count the number of nodes in a list

class NumberList
{
    private:
        struct ListNode {
            double value;
            struct ListNode *next;
        };
        ListNode *head;
    int countNodes(ListNode *); //private version

    public:
        NumberList();
        NumberList(const NumberList & src);
        ~NumberList();
        void appendNode(double);
        void insertNode(double);
        void deleteNode(double);
        void displayList();
        int countNodes(); //public version, calls private
};

Recursive function example
count the number of nodes in a list

// the private version, needs a pointer parameter
// How many nodes are in the list starting at the pointer?
int NumberList::countNodes(ListNode *p) {
    if (p == NULL) {
        return 0;
    } else {
        return 1 + countNodes(p->next);
    }
}

// the public version, no arguments (Nodes are private)
// calls the recursive function starting at head
int NumberList::countNodes() {
    return countNodes(head);
}

Note that this function is overloaded

Recursive function example
display the node values in reverse order

// the private version, needs a pointer parameter
void NumberList::reverseDisplay(ListNode *p) {
    if (p == NULL) {
        //do nothing
    } else {
        //display the "rest" of the list in reverse order
        reverseDisplay(p->next);
        cout << p->value << " ";
    }
}

// the public version, no arguments
void NumberList::reverseDisplay() {
    reverseDisplay(head);
    cout << endl;
}

Recursive function example
calling the functions from main

int main() {
    NumberList list;
    for (int i=0; i<5; i++)
        list.insertNode(i);
    
    cout << "The number of nodes is " << list.countNodes() << endl;
    cout << "The values in reverse order are: ";
    list.reverseDisplay();
}