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

How can a function call itself?

- What happens when this function is called?

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

Infinite Recursion:

This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
...

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
  - Iterative solution may be unreadable to humans

Why use recursion?

• Recursion is the primary method of performing repetition in most **functional** languages.
  - Implementations of functional languages are designed to process recursion efficiently
  - Iterative constructs that are added to many functional languages often don’t fit well in the functional context.

• Once programmers adapt to solving problems using recursion, the code produced is generally 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

Mathematical definition of n! (factorial of n)

if n=0 then  n! = 1
if n>0 then  n! = 1 x 2 x 3 x ... x (n-1) x n

What is the base case?

If we assume (n-1)! can be computed, how can we get n! from that?

Recursive function example

```
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:

  factorials(4):
  return 4 * factorial(3);
calls factorial(3):
  return 3 * factorial(2);
calls factorial(2):
  return 2 * factorial(1);
calls factorial(1):
  return 1 * factorial(0);
calls factorial(0):
  return 1;

- Each return statement must wait for the result of the recursive call to compute its result.

Recursive functions over ints

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

  ```
  type f(int n) {
    if (n==0) // do the base case
      return 1;
    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
  - vectors
  - linked lists
  - strings
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

```cpp
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)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

Pass by value ==> v is a copy of x, so x is unchanged

Recursive function example
sum of a list: vector

```cpp
int sum(vector<int> v) {
  if (v.size()==0)
    return 0;
  else {
    int x = v.back();
    v.pop_back();
    return x + sum(v);
  }
}
```

Pass by value ==> v makes a copy of x, for EACH recursive call

v.pop_back() creates the shorter vector
Aren’t we changing x each time (size = 0 at end)?
  - No (why not?)
  - But something else bad is happening each time.

Aren’t we changing x each time (size = 0 at end)?
  - No (why not?)
  - But something else bad is happening each time.

v.back() returns the last element
v.pop_back() creates the shorter vector
Aren’t we changing vector size each time (size = 0 at end)?
  - No (why not?)
  - But something else bad is happening each time.
Recursive function example
sum of a list: vector without copying

```cpp
int sumRec(vector<int> & v) {
    if (v.size()==0)
        return 0;
    else {
        int x = v.back();
        v.pop_back();
        return x + sumRec(v);
    }
}
```

- Sometimes an auxiliary or driver function is needed to set things up before starting recursion.

Recursive function example
sum of a list: linked list

- Add a sum function to List_3358_LL.h

```cpp
// this is the public one
int List_3358::sum() {
    return sumNodes(head);
}
```

```cpp
// this one is private
int List_3358::sumNodes(Node *p) {
    if (p==NULL)
        return 0;
    else {
        int x = p->value;
        return x + sumNodes(p->next);
    }
}
```

- `sumNodes(p)` will sum the Nodes starting with the one p points to until the end of the list (NULL)
- Advances p to next Node, (makes the shorter list)

Summary of the list examples

- How to determine empty list, single element, and the shorter list to perform recursion on.

<table>
<thead>
<tr>
<th>Array</th>
<th>Vector</th>
<th>Linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base case</td>
<td>size==0</td>
<td>p==NULL</td>
</tr>
<tr>
<td>last(or first) element</td>
<td>v.size()==0</td>
<td>p-&gt;value</td>
</tr>
<tr>
<td>shorter list (recursive call)</td>
<td>a[size-1]</td>
<td>p-&gt;next</td>
</tr>
<tr>
<td>shorter list (recursive call)</td>
<td>use size-1</td>
<td>v.pop_back()*</td>
</tr>
</tbody>
</table>

*may need to copy original vector

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

```cpp
string x = "hello there"; 
cout << x.substr(3,5);  // lo th
```
Recursive function example  
count character occurrences in a string

```cpp
int numChars(char target, string str) {
    if (str.empty()) {
        return 0;
    } else {
        int result = numChars(search, str.substr(1,str.size()));
        if (str[0]==target)
            return 1+result;
        else
            return result;
    }
}

int main() {
    string a = "hello";
    cout << a << numChars('l',a) << endl;
}
```

Recursive function example  
greatest common divisor

- Greatest common divisor of two non-zero ints is the largest positive integer that divides the numbers without a remainder
- This is a variant of Euclid’s algorithm:
  \[ \text{gcd}(x,y) = y \quad \text{if} \quad y \text{ divides } x \text{ evenly, otherwise:} \]
  \[ \text{gcd}(x,y) = \text{gcd}(y, \text{remainder of } x/y), \text{ or } \text{gcd}(y, x\%y) \text{ in c++} \]
- 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.

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

- 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;
  }
  ```
Recursive function example

greatest common divisor

Output:

gcd called with 9 and 1
GCD(9,1): 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 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

F₀ = 0
F₁ = 1
Fᵢ = Fᵢ₋₁ + Fᵢ₋₂ (for i > 1)

It's a recursive definition

Recursive function example

Fibonacci numbers

Code:

```cpp
int fib(int x) {
    if (x<=1)
        return x;
    else
        return fib(x-1) + fib(x-2);
}
```

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

Recursive function example

Fibonacci numbers

Modified code to count the number of calls to fib:

```cpp
int fib(int x, int &count) {
    count++;
    if (x<=1)
        return x;
    else
        return fib(x-1, count) + fib(x-2, count);
}
```

```cpp
int main() {
    cout << "The first 40 fibonacci numbers: 
    for (int i=0; i<40; i++) {
        int count = 0;
        int x = fib(i, count);
        cout << "fib (" << i << ") = " << x
             << " # of recursive calls to fib = " << count << endl;
    }
}
```
Recursive function example  
Fibonacci numbers

• Counting calls to fib: output

The first 40 fibonacci numbers:

\[
\begin{align*}
fib(0) &= 0 \quad \# \text{ of recursive calls to } fib = 1 \\
fib(1) &= 1 \quad \# \text{ of recursive calls to } fib = 1 \\
fib(2) &= 1 \quad \# \text{ of recursive calls to } fib = 3 \\
fib(3) &= 2 \quad \# \text{ of recursive calls to } fib = 5 \\
fib(4) &= 3 \quad \# \text{ of recursive calls to } fib = 9 \\
fib(5) &= 5 \quad \# \text{ of recursive calls to } fib = 15 \\
fib(6) &= 8 \quad \# \text{ of recursive calls to } fib = 25 \\
fib(7) &= 13 \quad \# \text{ of recursive calls to } fib = 41 \\
fib(8) &= 21 \quad \# \text{ of recursive calls to } fib = 67 \\
fib(9) &= 34 \quad \# \text{ of recursive calls to } fib = 109 \\
fib(10) &= 55 \quad \# \text{ of recursive calls to } fib = 177 \\
fib(11) &= 89 \quad \# \text{ of recursive calls to } fib = 287 \\
fib(12) &= 144 \quad \# \text{ of recursive calls to } fib = 465 \\
fib(13) &= 233 \quad \# \text{ of recursive calls to } fib = 753 \\
\ldots \\
fib(40) &= 1,023,344,155 \quad \# \text{ of recursive calls to } fib = 331,160,281
\end{align*}
\]

Recursive function example  
Fibonacci numbers

• Why are there so many calls to fib?

\[
\begin{align*}
\text{fib}(n) \text{ calls } \text{fib}(n-1) \text{ and } \text{fib}(n-2)
\end{align*}
\]

• Say it computes fib(n-2) first.

• When it computes fib(n-1), it computes fib(n-2) again

\[
\begin{align*}
\text{fib}(n-1) \text{ calls } \text{fib}((n-1)-1) \text{ and } \text{fib}((n-1)-2) = \text{fib}(n-2) \text{ and fib } (n-3)
\end{align*}
\]

• It’s not just double the work. It’s double the work for each recursive call.

• Each recursive call does more and more redundant work

Recursive function example  
Fibonacci numbers

• Trace of the recursive calls for fib(5)

Recursive function example  
Fibonacci numbers

• The number of recursive calls is

- larger than the Fibonacci number we are trying to compute
- exponential, in terms of n

• Never solve the same instance of a problem in separate recursive calls.

- make sure f(m) is called only once for a given m
Binary Search

- Find an item in a list, return the index or -1
- Works only for SORTED lists
- Compare target value to middle element in list.
  - if equal, then return index
  - if less than middle elem, search in first half
  - if greater than middle elem, search in last half
- If search list is narrowed down to 0 elements, return -1
- Divide and conquer style algorithm

**Iterative version**

```cpp
int binarySearch(const int array[], int size, int value) {
    int first = 0,         // First array element
    last = size - 1,      // Last array element
    middle,               // Mid point of search
    position = -1;        // Position of search value
    bool found = false;    // Flag
    while (!found && first <= last) {
        middle = (first + last) / 2;     // Calculate mid point
        if (array[middle] == value) {    // If value is found at mid
            found = true;
            position = middle;
        } else if (array[middle] > value)  // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1;           // If value is in upper half
    }
    return position;
}
```

**Recursive version**

- Convert the iterative version to recursive
- What is the base case?
  - empty list: result = -1 (not found)
- What is the recursive case?
  - split list into: middle value, first half, last half
  - if middle value equals target, then return its index
  - if less than middle elem, search in first half
  - if greater than middle elem, search in last half
- Need to add parameters for first and last index of the current subpart of the list to search.
- Two base cases
- Two recursive cases
**Binary Search**

**Recursive version**

```c
int binarySearchRec(const int array[], int first, int last, int value) {
    int middle; // Mid point of search
    if (first > last)           //check for empty list
        return -1;
    middle = (first + last)/2;  //compute middle index
    if (array[middle]==value)
        return middle;
    if (value < array[middle])    //recursion
        return binarySearchRec(array, first,middle-1, value);
    else
        return binarySearchRec(array, middle+1,last, value);
}
```

```c
int binarySearch(const int array[], int size, int value) {
    return binarySearchRec(array, 0, size-1, value);
}
```

---

**Binary Search**

**Running time efficiency**

- What is the Big-O analysis of the running time?
- **N** is the length of the list to search
- Worst case: keep dividing **N** by 2 until it is less than 1.
- This is equivalent to doubling 1 until it gets to **N**.

\[
\begin{align*}
1*2 &= 2 \\
2*2 &= 4 \\
4*2 &= 8 \\
8*2 &= 16 \\
16*2 &= 32 \\
32*2 &= 64
\end{align*}
\]

After 6 steps we have \(2^6\)

After \(k\) steps we have \(2^k\)

**How many steps does it take to double 1 and get to \(N\)?**

\[
2^k = N
\]

**How do we solve that for \(k\)?**

**Definition of logarithm (see math textbook):**

\[\log_B N = k \text{ if } B^k = N\]

The logarithm is the exponent

**So solving for \(k\):**

\[
k = \log_2 N
\]

---

**Binary Search**

**Running time efficiency**

- **How many steps does it take to repeatedly double 1 and get to \(N\)?**
  \[\log_2 N\]

- **How many steps does it take to repeatedly divide \(N\) by 2 and get to 1?**
  \[\log_2 N\]

- Since (worst case) binary search repeatedly divides the length of the list by 2, until it gets down to one, its running time is \(O(\log N)\)