Solving Problems Recursively

- Recursion is an indispensable tool in a programmer’s toolkit
  - Allows many complex problems to be solved simply
  - Elegance and understanding in code often leads to better programs: easier to modify, extend, verify (and sometimes more efficient! See expotest.cpp)
  - Sometimes recursion isn’t appropriate, when it’s bad it can be very bad—every tool requires knowledge and experience in how to use it

- The basic idea is to get help solving a problem from coworkers (clones) who work and act like you do
  - Ask clone to solve a simpler but similar problem
  - Use clone’s result to put together your answer

- Need both concepts: call on the clone and use the result

Fundamentals of Recursion

- **Base case (aka exit case)**
  - Simple case that can be solved with no further computation
  - Does not make a recursive call

- **Reduction step (aka Inductive hypothesis)**
  - Reduce the problem to another smaller one of the same structure
  - Make a recursive call, with some parameter or other measure that decreases or moves towards the base case
    - Ensure that sequence of calls eventually reaches the base case
    - “Measure” can be tricky, but usually it’s straightforward

- **The Leap of Faith!**
  - If it works for the reduction step is correct and there is proper handling of the base case, the recursion is correct.

What row are you in?

Recursive example

```cpp
void WriteBinary(int n)
// Writes the binary representation // of n to cout.
{
    if (n < 0) {
        cout << "-";
        WriteBinary(-n);
    } else if (n < 2)
        cout << n;
    else {
        WriteBinary(n/2);
        cout << n % 2;
    }
}
```

Recursive example

```cpp
void mystery(int n)
{
    if (n <= 1)
        cout << n;
    else {
        mystery(n/2);
        cout << ", " << n;
    }
}
```
Classic examples of recursion

- For some reason, computer science uses these examples:
  - Factorial: we can use a loop or recursion (see facttest.cpp), is this an issue?
  - Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, ...
    - \( F(n) = F(n-1) + F(n-2) \), why isn’t this enough? What’s needed?
    - Classic example of bad recursion, to compute \( F(6) \), the sixth Fibonacci number, we must compute \( F(5) \) and \( F(4) \). What do we do to compute \( F(5) \)? Why is this a problem?
  - Towers of Hanoi
    - \( N \) disks on one of three pegs, transfer all disks to another peg, never put a disk on a smaller one, only on larger
    - Every solution takes “forever” when \( N \), number of disks, is large

Towers of Hanoi

- The origins of the problem/puzzle may be in the far east
  - Move \( n \) disks from one peg to another in a set of three

```
void Move(int from, int to, int aux, int numDisks)
// pre: numDisks on peg # from,
// move to peg # to
// post: disks moved from peg 'from' // to peg 'to' via 'aux'
{
    if (numDisks == 1)
    { cout << "move " << from << " to " << to << endl;
    }
    else
    { Move (from,aux,to,numDisks - 1);
      Move (from,to,aux, 1);
      Move (aux,to,from, numDisks - 1);
    }
}
```

Fibonacci: Don’t do this recursively

```
long RecFib(int n)
// precondition: 0 <= n
// postcondition: returns the n-th Fibonacci number
{
    if (0 == n || 1 == n)
    { return 1; }
    else
    { return RecFib(n-1) + RecFib(n-2); }
}
```

- How many clones/calls to compute \( F(5) \)?
- How many calls of \( F(1) \)?
- How many total calls?

```
See recfib2.cpp for caching code
```

Print words entered, but backwards

- Can use a vector, store all the words and print in reverse order
  - The vector is probably the best approach, but recursion works too

```
void PrintReversed()
{
    string word;
    if (cin >> word) // reading succeeded?
    { PrintReversed(); // print the rest reversed
      cout << word << endl; // then print the word
    }
}
```

- The function \texttt{PrintReversed} reads a word, prints the word only after the clones finish printing in reverse order
  - Each clone has its own version of the code, its own word variable

```
int main()
{PrintReversed();}
```
### Exponentiation

- Computing \(x^n\) means multiplying \(n\) numbers (or does it?)
  - What’s the easiest value of \(n\) to compute \(x^n\)?
  - If you want to multiply only once, what can you ask a clone?

```java
double Power(double x, int n)
// post: returns \(x^n\)
{
    if (n == 0)
        return 1.0;
    return x * Power(x, n-1);
}
```

- What about an iterative version?

### Faster exponentiation

- How many recursive calls are made to compute \(2^{1024}\)?
  - How many multiplies on each call? Is this better?

```java
double Power(double x, int n)
// post: returns \(x^n\)
{
    if (n == 0)
        return 1.0;
    double semi = Power(x, n/2);
    if (n % 2 == 0)
        return semi*semi;
    return x * semi * semi;
}
```

- What about an iterative version of this function?

### Recursive and Iterative log powers

- In the program expotest.cpp we calculate \(x^n\) using \(\log(n)\) multiplications (basically). We do this both iteratively and recursively using BigInt variables
  - We saw the iterative code in Chapter 5 with doubles
  - BigInt has overloaded operators so these values work like ints and doubles
  - We use the CTimer class to time the difference in using these functions (ctimer.h)

- The recursive version is faster, sometimes much faster
  - Using doubles we wouldn’t notice a difference
  - Artifact of algorithm? Can we “fix the problem”? Power
  - Natural version of both in programs, optimizing tough.

### What’s better: recursion/iteration?

- There’s no single answer, many factors contribute
  - Ease of developing code assuming other factors ok
  - Efficiency (runtime or space) can matter, but don’t worry about efficiency unless you know you have to
- In some examples, like Fibonacci numbers, recursive solution does extra work, we’d like to avoid the extra work
  - Iterative solution is efficient
  - The recursive inefficiency of “extra work” can be fixed if we remember intermediate solutions: static variables

- Static function variable: maintain value over all function calls
  - Local variables constructed each time function called
**Fixing recursive Fibonacci: recfib2.cpp**

```cpp
long RecFib(int n) // precondition: 0 <= n <= 30 // postcondition: returns the n-th Fibonacci number {
    static tvector<int> storage(31,0);
    if (0 == n || 1 == n) return 1;
    else if (storage[n] != 0) return storage[n];
    else {
        storage[n] = RecFib(n-1) + RecFib(n-2);
        return storage[n];
    }
}
```

- What does storage do? Why initialize to all zeros?
  - Static variables initialized first time function called
  - Maintain values over calls, not reset or re-initialized

**Thinking recursively**

- Problem: find the largest element in a vector
  - Iteratively: loop, remember largest seen so far
  - Recursive: find largest in [1..n), then compare to 0th element

```cpp
double Max(const tvector<double>& a) // pre: a contains a.size() elements, 0 < a.size() // post: return maximal element of a {
    int k;
    double max = a[0];
    for (k=0; k < a.size(); k++) {
        if (max < a[k]) max = a[k];
    }
    return max;
}
```

- In a recursive version what is base case, what is measure of problem size that decreases (towards base case)?

**Recursive Max**

```cpp
double RecMax(const tvector<double>& a, int first) // pre: a contains a.size() elements, 0 < a.size() // post: return maximal element a[first..size()-1] {
    if (first == a.size()-1) // last element, done
        return a[first];
    double maxAfter = RecMax(a,first+1);
    if (maxAfter < a[first]) return a[first];
    else                    return maxAfter;
}
```

- What is base case (conceptually)?
- We can use RecMax to implement Max as follows
  ```cpp
  return RecMax(a,0);
  ```

**Recognizing recursion:**

```cpp
void Change(tvector<int>& a, int first, int last) // post: a is changed {
    if (first < last)
        { int temp = a[first]; // swap a[first], a[last] a[first] = a[last]; a[last] = temp;
          Change(a, first+1, last-1);
        }
    // original call (why?): Change(a, 0, a.size()-1);
}
```

- What is base case? (no recursive calls)
- What happens before recursive call made?
- How is recursive call closer to the base case?
More recursion recognition

```c
int Value(const tvector<int>& a, int index)
// pre: ??
// post: a value is returned
{
    if (index < a.size())
    {
        return a[index] + Value(a,index+1);
    }
    return 0;
}
// original call: cout << Value(a,0) << endl;
```

- What is base case, what value is returned?
- How is progress towards base case realized?
- How is recursive value used to return a value?
- What if a is vector of doubles, does anything change?

One more recognition

```c
void Something(int n, int& rev)
// post: rev has some value based on n
{
    if (n != 0)
    {
        rev = (rev*10) + (n % 10);
        Something(n/10, rev);
    }
}
int Number(int n)
{
    int value = 0;
    Something(n,value);
    return value;
}
```

- What is returned by Number(13) ? Number(1234) ?
  - This is a tail recursive function, last statement recursive
  - Can turn tail recursive functions into loops very easily

Non-recursive version

```c
int Number(int n)
// post: return reverse of n, e.g., 4231 for n=1234
{
    int rev = 0;    // rev is reverse of n's digits so far
    while (n != 0)
    {
        rev = (rev * 10) + n % 10;
        n /= 10;
    }
}
```

- Why did recursive version need the helper function?
- Where does initialization happen in recursion?
- How is helper function related to idea above?
- Is one of these easier to understand?
- What about developing rather than recognizing?

Blob Counting: Recursion at Work

- Blob counting is similar to what’s called Flood Fill, the method used to fill in an outline with a color (use the paint-can in many drawing programs to fill in)
  - Possible to do this iteratively, but hard to get right
  - Simple recursive solution

- Suppose a slide is viewed under a microscope
  - Count images on the slide, or blobs in a gel, or ...
  - Erase noise and make the blobs more visible

- To write the program we’ll use a class CharBitMap which represents images using characters
  - The program blobs.cpp and class Blobs are essential too
Counting blobs, the first slide

prompt> blobs
enter row col size 10 50
# pixels on: between 1 and 500: 200

+--------------------------------------------------+
| *  *    *    *  *  *  ***     *  ****   * *      |
|    * *        ***   **   **  * * *       *  * * *|
|  * *** * * *** *  *    *   *    * *    *   *   **|
|   *       **  ** *         **  * *  * ***    * * |
| *   *   ***** ***  *         *       **     ** * |
|* *    *        * *    * **   *     ***  *  ***  *|
|* *            *** * **     * *  *    * *      ** |
|*  *    **  *    *   *     *  *** **    *         |
| ****    *  *   **  ****    *   ***    * *      **|
|**  *  *       *    ** **** **        *  * **    *|
+--------------------------------------------------+

How many blobs are there? Blobs are connected horizontally and vertically, suppose a minimum of 10 cells in a blob
  - What if blob size changes?

Identifying Larger Blobs

blob size (0 to exit) between 0 and 50: 10

# blobs = 3
  - The class Blobs makes a copy of the CharBitMap and then counts blobs in the copy, by erasing noisy data (essentially)
    - In identifying blobs, too-small blobs are counted, then uncounted by erasing them

Identifying smaller blobs

blob size (0 to exit) between 0 and 50: 5

# blobs = 8
  - What might be a problem if there are more than nine blobs?
    - Issues in looking at code: how do language features get in the way of understanding the code?
    - How can we track blobs, e.g., find the largest blob?

Issues that arise in studying code

  - What does static mean, values defined in Blobs?
    - Class-wide values rather than stored once per object
    - All Blob variables would share PIXEL_OFF, unlike myBlobCount which is different in every object
    - When is static useful?

  - What is the class tmatrix?
    - Two-dimensional vector, use a[0][1] instead of a[0]
    - First index is the row, second index is the column

  - We’ll study these concepts in more depth, a minimal understanding is needed to work on blobs.cpp
Recursive helper functions

- Client programs use 
  **Blobs::FindBlobs** to find blobs of a given size in a CHARBITMAP object
  - This is a recursive function, private data is often needed/used in recursive member function parameters
  - Use a helper function, not accessible to client code, use recursion to implement member function

- To find a blob, look at every pixel, if a pixel is part of a blob, identify the entire blob by sending out recursive clones/scouts
  - Each clone reports back the number of pixels it counts
  - Each clone “colors” the blob with an identifying mark
  - The mark is used to avoid duplicate (unending) work

Conceptual Details of BlobFill

- Once a blob pixel is found, four recursive clones are “sent out” looking horizontally and vertically, reporting pixel count
  - How are pixel counts processed by clone-sender?
  - What if all the clones ultimately report a blob that’s small?

- In checking horizontal/vertical neighbors what happens if there aren’t four neighbors? Is this a potential problem?
  - Who checks for valid pixel coordinates, or pixel color?
  - Two options: don’t make the call, don’t process the call

- Non-recursive member function takes care of looking for blobsign, then filling/counting/unfilling blobs
  - How is unfill/uncount managed?