Today's lecture

- Recursion: functions that call themselves
- Flags: binary variables that take note of state in loops
- Recursive version of MergeSort

Computing Factorial
Task: write a function that computes factorial

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

Factorial with for loop

```c
int factorial( int n ) {
  int fact = 1;
  for( int i=2; i<=n; i++ ) {
    fact = fact * i;
  }
  return fact;
}
```

Nothing new here...
Factorial with recursion

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

The function `factorial` calls itself!

Execution of recursion

```
factorial(5)
    return 5 * factorial(4);
        return 4 * factorial(3);
            return 3 * factorial(2);
                return 2 * factorial(1);
                    return 2 * 1;
                        return 3 * 2
                            return 4 * 6
                                return 5 * 24
```

```
120
```

Fibonacci numbers

The Fibonacci numbers are the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
F(n) =
    \begin{cases}
        F(n-1) + F(n-2) & \text{if } n > 1 \\
        1 & \text{if } n = 1 \\
        0 & \text{if } n = 0
    \end{cases}
```

\[ \frac{F(n+1)}{F(n)} \] converges to the golden ratio 1.618034.

Recursive computation of Fibonacci numbers

```c
int fibonacci( int n ) {
    if (n==0)
        return 0;
    if (n==1)
        return 1;
    return fibonacci( n-1 ) + fibonacci( n - 2 );
}
```

▶ How many function calls does it roughly take to compute `fibonacci(10)` or `fibonacci(100)`?

▶ Could this be done faster?
Detecting events in a loop

We often loop through an array to detect a single event:

```c
#define FOUND 1
#define NOT_FOUND 0
char word[20] = "abracadabra!";

for(int i=0; word[i] != '\0'; i++) {
    if (word[i] == 'c')
        return FOUND;
}
return NOT_FOUND;
```

Here we use the trick of exiting a function at different places. This may not always be possible.

Flags

Use of a flag:

```c
#define FOUND 1
#define NOT_FOUND 0
char word[20] = "abracadabra!";
int flag = NOT_FOUND;
for(int i=0; word[i] != '\0'; i++) {
    if (word[i] == 'c')
        flag = FOUND;
    else if (word[i] == 'q')
        flag = FOUND;
}
```

Finding the longest streak

Input: String that encodes wins (W) and losses (L)

```c
char word[20] = "WLWWWLLWLLLLW";
char flag = 'X'; int length = 0, longest = 0;
for(int i=0; word[i] != '\0'; i++) {
    if (word[i] == flag) {
        length++;
        if (length > longest) longest = length;
    }
    else {
        length = 1;
        flag = word[i]; // indicates if tracking wins or losses
    }
}
printf("longest streak is %d games.\n",longest);
```
MergeSort through recursion

- Previously we saw a “bottom-up” version of MergeSort.
- Typical implementation of MergeSort is recursive:
  - merge function is identical - takes two sorted arrays and creates the “merge” of those arrays
  - “top-down” implementation - The sort of the array key is the result of sorting each half of key and then merge-ing those two sorted subarrays

This “declarative” way of thinking about a problem is often the best way of coming up with a recursive algorithm.

```c
void mergesort(int key[], int n){
    int j, *w;
    if (n>1) {
        w = calloc(n, sizeof(int)); /* space for temporary array */
        assert (w != NULL);
        j = n/2;
        mergesort(key, j);
        mergesort(key+j, n-j);
        merge(key, key+j, w, j, n-j);
        for (j = 0; j < n; ++j)
            key[j] = w[j];
        free(w); /* Free up dynamic memory no longer in use. */
    }
}
```

- simpler than before
- I thought it would be slower than before (not really)

Office hours

For weeks 9, 10, 11, in Informatics Forum, IF 5.16 (Mary’s office)

- Wednesday, 11am-12
- Thursday, 10am-11am
- responsive mode (ie, you bring the questions)

Start Wednesday 17th November.
Finish Thursday 2nd December.