Today’s lecture

- Recursion: functions that call themselves.
- Examples: factorial and fibonacci.
- The long integer type.
- MergeSort (recursive version).
- Allocating memory dynamically with calloc.

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

We use `long` (meaning ‘long int’) rather than `int`, because as \( n \) increases, `factorial(n)` can get very large - for example \( 13! \) is too large for an `int` variable (on DICE).

On DICE, `long` uses 64 bits - can store values up to \( \pm 9.22337204 \times 10^{18} \).

**System dependent:** On old machines, `long` might be 32 bits only.
Factorial with recursion

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

The function `factorial` calls itself!
A function is a 'black box' to which you give arguments, and it returns a result. Recursive calls are no different from any other call!
`factorial(n)` needs to know what \((n-1)!\) is, so it just calls the black box `factorial(n-1)`.
`factorial(1)` doesn't need to call anything.

Fibonacci numbers

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

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

\[
\frac{F(n+1)}{F(n)} \text{ converges to the golden ratio } \frac{1 + \sqrt{5}}{2} = 1.618034.
\]

Execution of recursion

If you look at how the sequence of execution goes, it's like this:

```c
factorial(5)
    return 5 * factorial(4);
        return 4 * factorial(3);
            return 3 * factorial(2);
                return 2 * factorial(1);
                    return 1;
                    return 2 * 1;
            return 3 * 2
        return 4 * 6
    return 5 * 24
120
```

but just think in terms of the black box.

Recursive computation of Fibonacci numbers

```c
long 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)`?
▶ Running time?
  ▶ Is this faster/slower than the for and while versions from lecture 6?
  ▶ Interesting comparison using `clock()` :-).
    more interesting than comparing linear against binary search.
Idea:
Suppose we have two arrays $a$, $b$ of length $n$; and $m$ respectively, and that these arrays are already sorted. Then the merge of $a$ and $b$ is the sorted array of length $n+m$ we get by walking through both arrays jointly, taking the smallest item at each step.

```c
void merge(int a[], int b[], int c[], int m, int n) {
    int i=0, j=0, k=0;
    while (i < m || j < n) {
        /* In the following condition, if $j >= n$, $C$ knows the condition is true, and it *doesn't* check the rest, so it doesn't matter that $b[j]$ is off the end of the array. So the order of these is important */
        if (j >= n || (i < m && a[i] <= b[j])) {
            /* either run out of $b$, or the smallest elt is in $a$ */
            c[k++] = a[i++];
        } else {
            /* either run out of $a$, or the smallest elt is in $b$ */
            c[k++] = b[j++];
        }
    }
}
```

sorting using merge

- merge can create an overall sort from two smaller arrays which were individually sorted.
- Could we use merge as a helper function to perform the task of sorting a given array (where the initial arrays is not at all sorted)?
  - Divide-and-Conquer is the process of solving a big problem, by utilizing the solutions to smaller versions of that problem.
  - For sorting, we could divide our (unsorted) input array in two pieces, then sort those two smaller subarrays individually (recursion) and finally get the overall sort using merge.
sorting using merge

MergeSort through recursion

- Typical implementation of MergeSort is recursive:
  - The sort of the array key is the result of sorting each half of key and then merging those two sorted subarrays.
  - merge function takes two sorted arrays and creates the 'merge' of those arrays.
- C issues:
  - We will need to create a 'scratch array' to pass to merge (as the c parameter) to write the result of 'merging' the two smaller (already sorted) subarrays.
  - 'scratch' array needs same length as input array.
  - Need to dynamically allocate space.
  - In C, use calloc to get space of a 'dynamic' (not fixed) amount.
  - Returns a 'pointer to void' ... just means a pointer/address of no particular type. The pointer will be NULL if there was not enough available space.

wrap-up and reading

- Running-time of mergesort is proportional to \( n \lg(n) \). Compares favourably with BubbleSort \( (n^2) \).
- Read more about recursion in Sections 5.14, 5.15 of Kelley & Pohl.
- Read more about calloc in Section 6.8 of Kelley & Pohl.
- Our implementation of mergesort is on the course webpage. mergerec.c also has a wrt function for printing out small arrays, and a main for testing/timing on arrays of various sizes.
- Kelley & Pohl have a 'bottom-up' version of MergeSort for array lengths a power-of-2.
  - More troublesome/fiddly than the recursive version
  - Can adapt their 'bottom-up' version of MergeSort to work for general array lengths. If you want a challenge try this (but test it rigorously)