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

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

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

```java
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, 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:

```java
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
```

but just think in terms of the black box.

---

Recursive computation of Fibonacci numbers

```java
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.
Merge

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.

```
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

MergeSort through recursion

- Typical implementation of MergeSort is recursive:
  - The sort of the array \textit{key} is the result of sorting each half of \textit{key} and then merging those two sorted subarrays
  - \textit{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 \textit{merge} (as the \textit{c} parameter) to write the result of ‘merging’ the two smaller (already sorted) subarrays.
  - ‘scratch’ array needs same length as input array.
  - Need to \textit{dynamically} allocate space.
  - In C, use \textit{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 \texttt{NULL} if there was not enough available space.

Wrap-up and Reading

- Running-time of \texttt{mergesort} is proportional to \textit{n} \textit{lg}(\textit{n}).
- Compares favourably with \texttt{BubbleSort} (\textit{n}^2).
- Read more about recursion in Sections 5.14, 5.15 of Kelley & Pohl.
- Read more about \texttt{calloc} in Section 6.8 of Kelley & Pohl.
- Our implementation of \texttt{mergesort} is on the course webpage. \texttt{mergerec.c} also has a \texttt{wrt} function for printing out small arrays, and a \texttt{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. \textit{If you want a challenge try this}
    (but test it rigorously)

---

\texttt{int mergesort(int key[], int n)}

\begin{verbatim}
int j, *w;
if (n <= 1) { return 1; } /* base case, sorted */
w = calloc(n, sizeof(int)); /* space for temporary array */
if (w == NULL) { return 0; } /* calloc failed */
j = n/2;
/* do the subcalls and check they succeed */
if ( 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 the dynamic memory no longer in use. */
  return 1;
} else { /* a subcall failed */
  free(w);
  return 0;
}
\end{verbatim}