Memoization

Let's review why our first recursive implementation of **fib** was so dreadfully slow. Here's the code again, updated to make use of the **long long** data type so that much, much larger Fibonacci numbers can, in theory and given an infinite amount of time, be computed:

```
static unsigned long long fib(int n) {
   if (n < 2) return n;
   return fib(n - 1) + fib(n - 2);
}
```

The code mirrors the inductive definition, but because each call to **fib** usually gives birth to two more, the running time grows exponentially with respect to **n**.

One key observation: the initial recursive call leads to many (many, many) repeated recursive calls. The computation of the 40th Fibonacci number, for instance, leads to:

- 1 call to **fib(39)**
- 2 calls to **fib(38)**
- 3 calls to **fib(37)**
- 5 calls to **fib(36)**
- 8 calls to **fib(35)**
- 13 calls to **fib(34)**
- 21 calls to **fib(33)**

….

It's sad that **fib(33)** gets called 21 different times, because it builds the answer from scratch, even though the answer is always the same. The implementation is farcically slow because it spends virtually all of its time re-computing the same results over and over again.

One technique to overcome the repeated sub-problem issue is to keep track of all previously computed results in a **Map**, and to consult the **Map** to see if a partial result has been computed before moving on to the binary recursion.

The code that appears at the top of the next page is an extension of the above, save for the key addition that a **cache** has been threaded throughout the implementation so that previously computed results can be stored and retrieved very, very quickly:

```
static unsigned long long fib(int n, Map<int, unsigned long long>& cache) {
   if (cache.containsKey(n)) return cache[n];
   unsigned long long result = fib(n - 1, cache) + fib(n - 2, cache);
   cache[n] = result;
   return result;
}

static unsigned long long fib(int n) {
   Map<int, unsigned long long> cache;
   cache[0] = 0;
   cache[1] = 1;
   return fib(n, cache);
}
```

Notice the introduction of a **`Map<int, unsigned long long>`**. The base-case section of the recursive function now checks the **`cache`**, which initially houses the traditional base case results, but over time grows to include everything that's ever been computed during the lifetime of a single call.

All of a sudden, what used to be an exponential-time algorithm now runs in time that's proportional to n. This technique of caching previously generated results is called ***memoization***. It looks like the word memorization, but it's missing the r. Apparently the word is derived from memorandum, not memorization. At least that's what Wikipedia says. ☺

One key observation to point out: memoization is only useful when there are repeated sub-problems, but it doesn't do much when all or nearly all recursive calls are unique. That means that **`fib`** benefits from memoization, but functions like **`listPermutations`** and **`listSubsets`** (each of which produces output of length **`n!`** and $\mathbf{2^n}$, respectively) do not.