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 calls 21 different times, because it currently builds the answer from scratch, even though the answer is always the same. The implementation is farcically slow because it spends a large fraction 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 always consult the **Map** to see if a partial result has been computed before 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 out of length **`n!`** and **2^n**, respectively) do not.

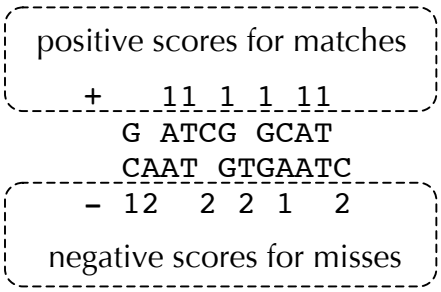
DNA Alignment[1]

There are several alignment methods for measuring the similarity of two DNA sequences (which for the purposes of this example can be thought of as strings over a four-letter alphabet: **`A`**, **`C`**, **`G`**, and **`T`**). One such method to align two sequences **`x`** and **`y`** consists of inserting spaces at arbitrary locations (including at either end) so that the resulting sequences **`x'`** and **`y'`** have the same length but do not have a space in the same position. Then you can assign a score to each position. Position **`j`** is scored as follows:

- +1 if **`x'[j]`** and **`y'[j]`** are the same and neither is a space,
- –1 if **`x'[j]`** and **`y'[j]`** are different and neither is a space,
- –2 if either **`x'[j]`** or **`y'[j]`** is a space.

[1] Drawn from Introduction to Algorithms, Cormen, Leiserson, Rivest, and Stein

The score for a particular alignment is just the sum of the scores over all positions. For example, given the sequences **GATCGGCAT** and **CAATGTGAATC**, one such alignment (though not necessarily the best one) is:



The positive scores are listed above the alignment, and negative scores are listed below. This particular alignment has a total score of –4.

The goal here is to write a function called **alignStrands**, which takes two legitimate DNA strings and returns the alignment score.

For instance, a call to **alignStrands("CACTCTGCA", "GTCCCCATT")** would return -5 because of the following alignment is optimal:

```
 +  1 1
  CACTCTGCA
  GTCCCCATT
- 11 1 1111
```

Calling **alignStrands("CTTGTGTGGCACTGCGA", "ACTGCCCTACCACCG")** would return -6 because the following alignment is optimal:

```
+   11   1  111  11
  CTTGTG TGGCACTGCGA
  ACTGCCCTACCAC  CG
- 11  112 11   22  2
```

We'll assume that the two strings passed in to **alignStrands** are each DNA strings containing only the four capital letters you'd expect. In order for the **alignStrands** function to return in a reasonable amount of time, we're going to need to cache the results of recursively generated results so that we don't unnecessarily repeat the same recursive call a second or a third time.

```
static int alignStrands(const string& one, const string& two,
                        Map<string, int>& cache) {
   if (one.empty()) return -2 * two.length();
   if (two.empty()) return -2 * one.length();

   string key = one + ":" + two;
   if (cache.containsKey(key)) return cache[key];

   if (one[0] == two[0]) { // two leading bases match
      int score = alignStrands(one.substr(1), two.substr(1), cache) + 1;
      return cache[key] = score;
   }

   int first = alignStrands(one, two.substr(1), cache) - 2;
   int second = alignStrands(one.substr(1), two, cache) - 2;
   int third = alignStrands(one.substr(1), two.substr(1), cache) - 1;
   return cache[key] = max(first, max(second, third));
}

static int alignStrands(const string& one, const string& two) {
   Map<string, int> cache;
   return alignStrands(one, two, cache);
}
```

Memoization brings the wildly exponential running time down to a polynomial one. With a few more optimizations (pre-computing strand lengths, replacing **`substr`** calls with additional string-index parameters, using a custom data structure instead of the **`Map`**), the running time is roughly proportional to the product of the two strand lengths.