

## Memoization

---

Let's review why our first recursive implementation of **Fibonacci** 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:

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

The code mirrors the inductive definition as closely as one could possibly imagine, but because each call to **Fibonacci**, in general, 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 40<sup>th</sup> Fibonacci number, for instance, leads to:

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

It's sad that **Fibonacci(33)** gets calls 21 different times, because it needs to build up the answer from scratch every single time, even though the answer is always the same. The overall implementation of **Fibonacci** is farcically slow because it spends a laughably large fraction of its time re-computing the same results over and over again.

One very common, simple, and clever 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 something's already been computed before committing to the time-consuming recursion.

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

```

unsigned long long
Fibonacci(int n, Map<unsigned long long>& previouslyComputed) {
    string key = IntegerToString(n);
    if (previouslyComputed.containsKey(key)) {
        return previouslyComputed[key];
    }

    unsigned long long result =
        Fibonacci(n - 1, previouslyComputed) +
        Fibonacci(n - 2, previouslyComputed);
    previouslyComputed[key] = result;
    return result;
}

unsigned long long Fibonacci(int n) {
    Map<unsigned long long> previouslyComputed;
    previouslyComputed["0"] = 0;
    previouslyComputed["1"] = 1;
    return Fibonacci(n, previouslyComputed);
}

```

Notice the introduction of the **Map<unsigned long long>**, which is initially populated with what in previous implementations have been the base cases. The base-case section of the recursive function now checks the **previouslyComputed Map**, which initially contains just the original base case results, but over time grows to include everything that's ever been computed during the lifetime of a single top-level 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 subproblems, but it doesn't do much when all or near all recursive calls are unique. That means that **Fibonacci** benefits from memoization, but functions like **ListPermutations** and **ListSubsets** (each of which produces a list of length  $n!$  and  $2^n$ , respectively) do not.

## DNA Alignment<sup>i</sup>

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.

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:

positive scores for matches										
+		1	1	1	1	1	1	1	1	
	G	A	T	C	G	C	A	T	C	
	C	A	A	T	G	T	G	A	A	T
	-	1	2	2	1	2	2	1	2	
negative scores for misses										

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								
	C	A	C	T	C	T	G	C	A	
	G	T	C	C	C	C	A	T	T	
-	1	1	1	1	1	1	1	1	1	

---

<sup>i</sup> Drawn from [Introduction to Algorithms](#), Cormen, Leiserson, Rivest, and Stein

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.

```
int AlignDNAstrands(string one, string two) {
    Map<int> previouslyComputed;
    return AlignDNAstrands(one, two, previouslyComputed);
}

static const int kMatchScore = 1;
static const int kMismatchScore = -1;
static const int kForcedInsertionScore = -2;
int AlignDNAstrands(string one, string two,
                    Map<int>& previouslyComputed) {
    // brute force checks when one or both DNA strands are empty
    if (one.empty()) return two.length() * kForcedInsertionScore;
    if (two.empty()) return one.length() * kForcedInsertionScore;

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

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

    int firstScore =
        AlignDNAstrands(one,
                        two.substr(1),
                        previouslyComputed) + kForcedInsertionScore;
    int secondScore =
        AlignDNAstrands(one.substr(1),
                        two,
                        previouslyComputed) + kForcedInsertionScore;
    int thirdScore =
        AlignDNAstrands(one.substr(1),
                        two.substr(1),
                        previouslyComputed) + kMismatchScore;
    return previouslyComputed[key] =
        max(firstScore, max(secondScore, thirdScore));
}
```

Once again, the caching makes the above implementation run in times that's proportional to the product of the two DNA strand lengths. Without it, the running time would be proportional to  $3^n$ , where  $n$  is the length of the shorter of the two strands.