Rob Henderson
mystery

Once the input string is converted to an integer with atoi, Control jumps to L14. At L14, If the input number is less than or equal to 199, control then goes to L15. At L15, each slot in the cached_nums array is set to -1. From there, control goes back to L14. In L14, Mystery is called. Inside Mystery, If the fibonacci sequence number (cached_nums[num]) is -1 (not calculated yet), we need to calculate it. Basically, this is a Recursive fibonacci function utilizing memoization. Here, we need to realize that most calls can be avoided by saving the value of previous calls. The 1st time that mystery is called in the else, only ½ of the calculations actually needed to be made since some had already been produced from the previous call. After we complete fib(n-1), all the values for fib(n-2) are already computed, so as we go through the fib sequence the 1st time, there is no reason to recalculate the number (just pull it out of a table, or array in this case). fib(n-2) is finally added to fib(n-1) to get the final value of fib(n).

Here the fibonacci was calculated by
int n1=mystery(num-1);
int n2=mystery(num-2);
returnValue=n1+n2;

A slightly more efficient way to calculate the fibonacci would be
int n1=mystery(num-1);
int n2=cached_nums[num-2]; <-Already calculated completely, just a lookup, dont need to call mystery
returnValue=n1+n2;

I went about figuring this out by using the debugger and stepping through the assembly figuring out what it was doing. It was good practice in looking into the registers and following along with what was going on. From there i had to put that all together and put it into c.

Optimizations

Changes that the compiler made in the assembly code had to do with the control path of the recursive loop. It seems as though, to optimize, the assembler tried to keep all of the loop instructions inside of the one label. The unoptimized version was broken up more and the recursive self call was in L8, which was called by mystery instead of inside mystery like it was in the optimized version.

Inside main:

The optimized version uses strtol instead of atoi. In researching it, it seems as though atoi is older and has been replaced by strtol. atoi is still around because of its use in existing code. strtol is better because atoi doesnt perform any of the error checking that strtol does.

Optimized Version:	Unoptimized Version:
cmpl $200, %edx	cmpl $199, 28(%esp)

jne .L8 jle .L3

Here, it looks as though in the optimized version, in addition to using a register instead of the stack to hold the variable in the comparison, when it actually does the comparison, it is smart enough to realize that the whats in %edx (cached_nums index) will never be greater than 200 (due to the loop structure). Since it realizes this it can just say less than 200 jump and initialize the next slot in the array to -1. This is the same thing as jumping if the index of cached_nums is less than or equal to 199. Not equal to 200 seems like a lot less work than checking for less than or equal to 199.

Inside mystery:

1.

In the optimized version, it uses callee save instead of caller save utilizing edi, esi, and edx registers. In callee save, the calllee must save the values before they can be used. The caller is then able to assume that the register will be restored when the function is finished. Callee-save seems to be the way to go for recursion.

2.The optimized version utilizes jbe instead of jle and a jne inside of the mystery function. Jbe and Jle are working the same way here. Jbe is used for unsigned comparisons. In the optimized version, using jbe, if 0 or 1 are the input to mystery cached_nums[0]=0 and cached_nums[1]=1 for the respective inputs. In the unoptimized version, the compiler is doing a comparison of the input with 0 and doing a jump if equal to, then it is comparing it to 1 and doing a jump not equal to. jbe only has to check to see if CF is 1 or ZF=1 and jump if below or equal. In analyzing it, the control flows are much different at this point in particular (shown below).

Optimized:

```
cmpl    $1, %ebx

jbe     .L5#unsigned less than equal to 1./*fib(0)=0, fib(1)=1 cached_nums[0]=0, cached_nums[1]=1*/

.L5:

movl    %eax, cached_nums(,%ebx,4)
```

Unoptimized:

```
cmpl    $0, 8(%ebp)
        je      .L7#fib(0)
        cmpl    $1, 8(%ebp)
        jne     .L8#not equal to 1
.L7:
        movl    8(%ebp), %eax
        movl    %eax, -24(%ebp)
        jmp     .L9
.L8:
        movl    8(%ebp), %eax
        subl    $1, %eax
        movl    %eax, (%esp)
        call    mystery
        movl    %eax, -16(%ebp)
        movl    8(%ebp), %eax
        subl    $2, %eax
        movl    %eax, (%esp)
        call    mystery
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        movl    -16(%ebp), %edx
        leal    (%edx,%eax), %eax
```

movl %eax, -24(%ebp)

3. In my c code i actually just did this right away instead of assigning it to some temporary variable.

fibonacciNumValue = cached_nums[num];

if (fibonacciNumValue == -1)/*hasnt been filled yet*/

....

I noticed that gcc checked to see if that number was -1 twice, but at first I didnt know if the compiler just made a temporary variable and then did the assignment to the fibonacciNumValue later on. Since it is doing the checking twice, I just condensed it (not effecting the inner workings of the function). Perhaps if i would have left the temporary variable and later comparison to -1, the optimized version (compared to its unoptimized version) would have done something similar in assembly as to what i did in the c.