1 Magic Penny

1.1 Closed Form: Magic Penny

day	value
0	$0.01
1	$0.02
2	$0.04
3	$0.08
4	$0.16
5	$0.32
6	$0.64
...	...
31	$21,474,836.48

Let's start with a topic we are already familiar with. The magic doubling penny. It will double in value every day. After 31 days it will be worth several million dollars as partially shown in the table above. The astute developer, might recognize a mathematical pattern of powers of 2. Specific to this problem of pennies, we solve this in closed form as $\frac{2^n}{100}$ for some n number of days. In the interest of time, that code is provided below:

```
#include<cmath>
double magicPennyClosed( unsigned short days )
{
    return pow( 2, days ) / 100;
}
```

1.2 Loop: Magic Penny

Now of course, we might not recognize these powers of 2 immediately. Instead, we might solve this using a loop. In the interest of time, that code is provided below

```
double magicPennyLoop( unsigned short days )
{
    double result = 0.01;
    for ( int i = 0 ; i < days ; i++ )
    {
        result *= 2;
    }
    return result;
}
```

1.3 Recursion: Magic Penny

Now, let us try to solve this same problem a different way. Let us use recursion. We will work toward a function named *magicPennyRec*. The type signature shouldn't need to change:

```
double magicPennyRec( unsigned short days );
```

1. What is the base case? Put another way, for what values of *days* do we trivially know the final value of the magical doubling penny? SOLUTION: if *days* equals 0, then the penny is worth just that, a penny. NOTE: We can look at the table provided on the previous page and write out many base cases, but the goal isn't to change the function into a giant table, the goal is to think of the simplest possible case and document that case.

2. What can we do to a large value of *days* to guarantee that we arrive at the base case? SOLUTION: if $days \geq 1$, then we can make it decrement it. Since it's a natural number greater than 0, the only valid decrement available to guarantee we will reach the base case is to decrement by 1 each time.

3. Together, let us fill in the provided scaffolding: SOLUTION: see code below

```
double magicPennyRec( unsigned short days ) {
    // BASE CASE
    if ( days == 0 ) {
        return 0.01;
    }
    // RECURSIVE CASE
    else {
        unsigned short lessDays = days - 1;
        // wouldn't it be great...
        // if I had a function that:
        //     given a number of days
        //     it could find the value of a magic doubling penny
        //     that would be great
        //     it's called magicPennyRec
        double yesterdaysValue = magicPennyRec( lessDays );
        double todaysValue = yesterdaysValue * 2;
        return todaysValue;
}   }
// also valid
double magicPennyRec( unsigned short days ) {
    return ( days == 0 ) ? 0.01 : 2*magicPennyRec(days-1);
}
```

2 Famous recursive problems

A lecture on recursion is not complete without discussion of factorial and Fibonacci.

2.1 Factorial

Factorial is often easier to understand so we'll start there. Formally, the factorial of a natural number $n \in \mathbb{N}_0$, is denoted "n !" and is defined as follows for all

$$n\,! = \begin{cases} 1 & n \leq 1 \\ n \ * \ (\,(n-1)\,!\,) & n \geq 2 \end{cases}$$

4. What is the value of 0 !? SOLUTION: 1. it says so in the formula $n \leq 1$ if $n = 0$

5. What is the value of 1 !? SOLUTION: 1. it says so in the formula $n \leq 1$ if $n = 1$

6. What is the value of 2 !? SOLUTION: 2. take $n*((n-1)!)$ where n is 2 we get $2*((2-1)!)$ some easy math becomes $2 * (1!)$ we know that 1! is 1 so we get $2 * 1$ and that's easy math again

7. What is the value of 3 !? SOLUTION: 6. take $n*((n-1)!)$ where n is 3 we get $3*((3-1)!)$ some easy math becomes $3 * (2!)$ we just agreed that that 2! is 2 so we get $3 * 2$ and that's easy math again

8. What is the value of -7 !? SOLUTION: an error, negative numbers are not natural and not allowed.

9. let's turn it into code

```
unsigned int fact( unsigned short n )
{
        // BASE CASE
```

SOLUTION: if (n $\leq$ 1) return 1;

```
        // RECURSIVE CASE
```

SOLUTION: else return n * fact(n - 1);

```
}
```

10. let's walk through the code. To make the walkthrough consistent, I have pre-written one version of the solution. Let us construct and fill in a value table together to demonstrate that $fact(\ 5\ )$ will evaluate to 120

```
unsigned int fact( unsigned short n )
{
    unsigned int fn;
    // BASE CASE
    if ( n <= 1 )
    {
        fn = 1;
    }
    // RECURSIVE CASE
    else
    {
        unsigned int nm1, fnm1;
        nm1 = n - 1;
        fnm1 = fact( nm1 );
        fn = n * fnm1;
    }
    return fn;
}
```

SOLUTION: tricky to convey without a demonstration, let us break the table into two parts. First, drilling down to a base case, we fill in the table as follows:

n	nm1 = n - 1	fnm1 = fact(nm1)	fn = n * fnm1
5	4 (= 5 - 1)	(=fact(4))	
4	3 (= 4 - 1)	(=fact(3))	
3	2 (= 3 - 1)	(=fact(2))	
2	1 (= 2 - 1)	(=fact(1))	
1	N/A	N/A	1

NEXT, having found the base case, we complete the earlier rows that we previously couldn't. NOTE that we will fill in row $n = 2$ first, then $n = 3$ and so on.

n	nm1 = n - 1	fnm1 = fact(nm1)	fn = n * fnm1
5	4 (= 5 - 1)	24 (=fact(4))	120 (= 5 * 24)
4	3 (= 4 - 1)	6 (=fact(3))	24 (= 4 * 6)
3	2 (= 3 - 1)	2 (=fact(2))	6 (= 3 * 2)
2	1 (= 2 - 1)	1 (=fact(1))	2 (= 2 * 1)
1	N/A	N/A	1

2.2 Fibonacci

It would appear that we have already solved this for recitation this week, regardless, let us review. The Fibonacci Sequence is a famous sequence that occurs in the natural world. Much like the factorial value of a natural number, the Fibonacci sequence is easily described using recursion. Below is a mathematic definition for the Fibonacci sequence value for any $x \in \mathbb{N}_0$ (note that some prefer to use $y \in \mathbb{N}_1$)

$$fib(x) = \begin{cases} x & x \leq 1 \\ fib(x-1) + fib(x-2) & n \geq 2 \end{cases}$$

11. What is the value of $fib(-5)$? SOLUTION: an error, negative numbers are not natural and not allowed.

12. What is the value of $fib(0)$? SOLUTION: 0. it says so in the formula x when $x \leq 1$. here $x = 1$

13. What is the value of $fib(1)$? SOLUTION: 1. it says so in the formula x when $x \leq 1$. here $x = 1$

14. What is the value of $fib(2)$? SOLUTION: 1. since $x \geq 2$ when $x = 2$ that means that we apply the longer case $fib(x-1) + fib(x-2)$, here $fib(2-1) + fib(2-2)$. Some simple math gives $fib(1) + fib(0)$. we already agreed on the values of each of these $1+0$. easy math.

15. What is the value of $fib(3)$? SOLUTION: 2. since $x \geq 2$ when $x = 3$ that means that we apply the longer case $fib(x-1) + fib(x-2)$, here $fib(3-1) + fib(3-2)$. Some simple math gives $fib(2) + fib(1)$. we already agreed on the values of each of these $1+1$. easy math.

16. let's turn it into code

    SOLUTION: see code below

```
// your code here
unsigned int fib( unsigned short x ) {
    if ( x < 2 ) return x;
    else
    {
        unsigned int fxm1, fxm2;
        fxm1 = fib( x - 1 );
        fxm2 = fib( x - 2 );
        return fxm1 + fxm2;
    }
}
```

3 Merge Sort

Let us end our discussion of recursion with a re-visiting of the merge-sort algorithm. We will write a few functions.

17. first, describe a function named *foo* given 2 vectors of numbers that we assume are sorted in ascending order, return a single vector of all of these numbers in ascending order. You don't need to write full code, just describe it.

SOLUTION: see code below, one of many valid options.

```
vector<int> foo( vector<int> v1, vector<int> v2 ) {
    vector<int> res;
    unsigned int x = 0, y = 0;  // index trackers
    while ( x < v1.size() && y < v2.size() )  // are we inside either vector?
    {
        if ( v1.at( x ) < v2.at( y ) ) {  // take from v1
            res.push_back( v1.at( x ) );
            x++;
        } else {
            res.push_back( v2.at( y ) );  // take from v2
            y++;
    }   }
    // consume the remainder of either array. one or neither will run.
    for ( ; x < v1.size( ) ; x++ ) {  // yes, you can leave the first sectoin blank
        res.push_back( v1.at( x ) );
    }
    for ( ; y < v2.size( ) ; y++ ) {
        res.push_back( v2.at( y ) );
    }

    return res;
}
```

18. Is the above function recursive? SOLUTION: No, the solution above is not recursive. Note that there are valid recursive solutions to this problem. Could be a fun thing to try on your own.

19. What is the complexity of the above function in terms of the result vectors length n? SOLUTION: O(n). worst case there the last two for loops don't really execute and the while loop causes $2 * n$ comparisons. The constant 2 is ignored.

20. Review the following code and then describe in your own words what the insert function does for us here.

```
vector< int > x = { 0, 100 };
vector< int > y = { 5, 12, 2, 18, 7, 9 };
x.insert( x.begin() + 1, y.begin() + 2 , y.begin() + 5);
// x is now { 0, 2, 18, 7, 100 }
```

SOLUTION: Given a vector v and locations x, y, z, the $v.insert(x, y, z)$ start at location x of v. Inject data from location y until location z to the vector v. This will shift existing data in v. This supposes that y to z are contiguous accessed memory. NOTE "until" means that z itself will not be included.

21. finally, we will use the provided scaffolding to implement mergeSort. We should assume that the *foo* function has been written for us and we can call it as we see fit. We can tackle this in any order we want.

```
vector<int> mergeSort( vector<int> v )
{
    // base case
```

SOLUTION:
if (v.size( ) ≤ 1) return v;

```
    // split
```

SOLUTION:
unsigned int split = v.size( ) / 2;
// name them what you want, but left and right are good semantic names
vector⟨ int ⟩ left, right;
left.insert(left.begin( ), v.begin( ), v.begin( ) + split);
right.insert(right.begin( ), v.begin( ) + split, v.end( ));

```
    // recurse
```

SOLUTION:
left = mergeSort(left);
right = mergeSort(right);

```
    // merge
```

SOLUTION:
// order doesn't really matter here, but it's nice to be consistent
return foo(left, right);

```
}
```