HW 3

Due 23:00 November 27, 2017

CS ID: **SOLUTION**

Section (circle one):

Monday	L1J 9–11a	L1N 11a–1p	L1C 4–6p
Tuesday	L1A 11a–1p	L1F 4–6p	
Wednesday	L1B 9–11a	L1E 2–4p	L1M 6–8p
Thursday	L1D 10:30a–12:30p	L1G 4–6p	
Friday	L1K 1–3p	L1H 3–5p	

Please print out this document and write your solutions into the spaces provided. Show your work where necessary for full credit. If you require additional space, please indicate in the question space that you are writing on the last blank page, and also indicate on the blank page which question the work solves.

You must scan and upload the completed document, including this page and the last page, to GradeScope, using your 4- or 5-character CS ID.

1. **[Introductory Lucky Numbers - 3 points].**

   In this assignment, we will analyze various implementations of the Lucky numbers, to practice your skills in proofs and recurrence relations.

   The Lucky numbers are similar to the Fibonacci numbers. They are the numbers in the following integer sequence, called the Lucky sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones:

   $$2,\ 1,\ 3,\ 4,\ 7,\ 11,\ 18,\ 29,\ 47,\ \ldots$$

   For this assignment, the 0th Lucky number, denoted by L_0, is defined to be 0. The 1st Lucky number L_1 is 1, and for any other integer $k > 1$, $L_k = L_{k-1} + L_{k-2}$. Hence, the sequence above is the sequence $L_0, L_1, \ldots, L_8, \ldots$.

   (a) (1 point) By hand, calculate the 9th Lucky number L_9.

   Solution:

   $$L_9 = L_8 + L_7 = 47 + 29 = 76$$

   76

   (b) (2 points) By hand, calculate the 13th Lucky number L_{13}.

   Solution:

   $$\begin{aligned} L_{10} &= L_9 + L_8 && = 76 + 47 = 123 \\ L_{11} &= L_{10} + L_9 && = 123 + 76 = 199 \\ L_{12} &= L_{11} + L_{10} && = 199 + 123 = 322 \\ L_{13} &= L_{12} + L_{11} && = 322 + 199 = 521 \end{aligned}$$

   521

   As you can probably see, this is already too much work! How about we let the computer do the work for us? :)

2. **[Recursive Lucky Numbers - 14 points].**

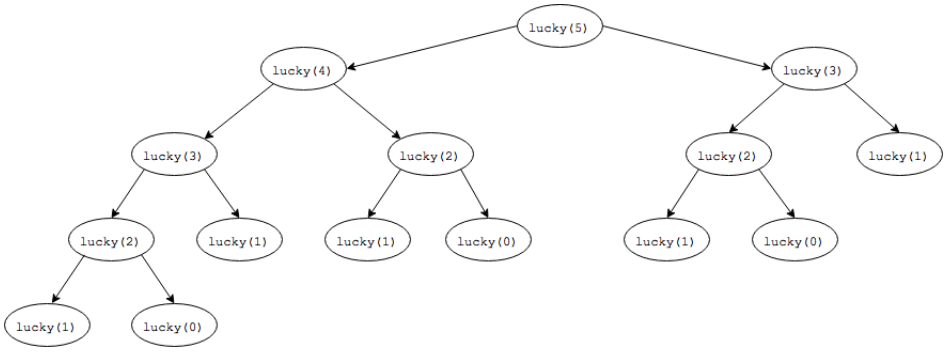
The Lucky numbers are inherently recursive. Hence, a (naïve) algorithm to calculate L_n could be the following:

```
unsigned long lucky(unsigned long n)
{
    if (n == 0)
        return 2;
    if (n == 1)
        return 1;

    return lucky(n-1) + lucky(n-2);
 }
```

(a) (3 points) Draw a tree to illustrate the recursion structure of `lucky` when $n = 5$. The root node of your tree should correspond to `lucky(5)`, and that node should have two children, one for each of `lucky(4)`, and `lucky(3)`, etc. down to the base cases, which are leaves. Inside each node, place the value of the argument to `lucky`.

Solution:



(b) (3 points) Write a recurrence relation $T_1(n)$ to describe the running time of the function `lucky`.

Solution: For constants b_1, b_2, c, we have:

$$T_1(n) = \begin{cases} b_1 & \text{if } n = 0 \\ b_2 & \text{if } n = 1 \\ T_2(n-1) + T_2(n-2) + c & \text{otherwise} \end{cases}$$

Note: Defining 3 different constants, or splitting the base cases into separate cases in the recurrence relation is not necessary.

(c) (4 points) Consider the following similar recurrence relation, where b, c are constants:

$$T_2(n) = \begin{cases} b & \text{if } n = 0, 1 \\ 2T_2(n-2) + c & \text{otherwise} \end{cases}$$

Find a closed form solution for the recurrence relation $T_2(n)$. You should consider the two cases where n is even and n is odd separately.

Solution: For $n > 0$:

$$\begin{aligned} T_2(n) &= 2T_2(n-2) + c \\ &= 2(2(T_2(n-4) + c) + c) \\ &= 4(T_2(n-4)) + 2c + c \\ &= 4(2(T_2(n-6) + c)) + 2c + c \\ &= 8(T_2(n-6)) + 4c + 2c + c \\ &= \dots \\ &= 2^k(T_2(n-2k)) + \sum_{i=0}^{k-1} 2^i c \end{aligned}$$

If n is even:

$$\begin{aligned} T_2(n) &= 2^{\frac{n}{2}}(T_2(0)) + \sum_{i=0}^{\frac{n}{2}-1} 2^i c \\ &= 2^{\frac{n}{2}} b + (2^{\frac{n}{2}} - 1)c \\ &= 2^{\frac{n}{2}}(b + c) - c \end{aligned}$$

If n is odd:

$$\begin{aligned} T_2(n) &= 2^{\frac{n-1}{2}}(T_2(1)) + \sum_{i=0}^{\frac{n-1}{2}-1} 2^i c \\ &= 2^{\frac{n-1}{2}} b + (2^{\frac{n-1}{2}} - 1)c \\ &= 2^{\frac{n-1}{2}}(b + c) - c \end{aligned}$$

Write the closed form here: $\boxed{2^{\left\lfloor \frac{n}{2} \right\rfloor}(b + c) - c}$

Note: You may write the closed form using two separate cases for the even or odd cases, but the use of the floor function is preferred.

(d) (2 points) Compare the recurrence relations $T_1(n)$ and $T_2(n)$. Is the bound found for $T_2(n)$ an **upper-bound** or a **lower-bound** for $T_1(n)$? Circle one answer below:

Upper Bound **Lower Bound**

(e) (2 points) In 20 words or less, describe why the performance for `lucky` is not desirable.

Solution: Repeated recursion is not desirable, especially if solving the same large subproblems many times.

3. **[Dynamic Lucky Numbers] - 21 points.**

In labs, we discussed using memoization to speed up the performance of computing the Fibonacci numbers. In this assignment, we will explore another method: Dynamic Programming. The following is a dynamic programming solution which computes L_n:

```
unsigned long dp_lucky(unsigned long n)
{
    vector<unsigned long> dp(n+1);
    dp[0] = 2;
    dp[1] = 1;

    for(unsigned int j = 1; j < n; j++)
    {
        dp[j+1] = dp[j] + dp[j-1];
    }

    return dp[n];
}
```

(a) (3 points) For $n = 5$, illustrate the contents of the array `dp` before and after the iteration $j = 2$.

Solution:

Before:

2	1	3			

After:

2	1	3	4		

For the remainder of this problem, we will evaluate the following loop invariant, and prove that `dp_lucky` correctly computes L_n:

At the start of iteration j in the for-loop of `dp_lucky`, all the elements of the array `dp` from 0 to j are the Lucky numbers L_0 through L_j.

(b) (2 points) First, consider the base case. Show that the loop invariant holds when $j = 1$.

Solution: Before the start of the for-loop, `dp[0]` and `dp[1]` contain the values L_0 and L_1 respectively. Hence, at the start of the $j = 1$ iteration of the for-loop, all the elements of the array `dp` from 0 to j are the Lucky numbers L_0 through L_j.

(c) (6 points) Finish the following loop invariant proof by filling in the blanks:

Solution: Suppose, for some $k \geq 1$, that the loop invariant holds; in other words, at the start of iteration k in the for-loop, all the elements of the array `dp` from 0 to k are the Lucky numbers L_0 through L_k. We will show that the loop invariant holds at the start of iteration $k+1$, to prove the inductive step.

By the inductive hypothesis, all the elements of the array `dp` from $\underline{\mathbf{0}}$ to $\underline{\mathbf{k}}$ are the Lucky numbers $\underline{\mathbf{L_0}}$ through $\underline{\mathbf{L_k}}$. In particular, the Lucky numbers L_k and L_{k-1} are found in `dp[`$\underline{\mathbf{k}}$`]` and `dp[`$\underline{\mathbf{k-1}}$`]` respectively. So, inside the kth iteration of the for-loop, the line `dp[k+1] = dp[k] + dp[k-1]` will set `dp[k+1]` as the Lucky number $\underline{\mathbf{L_{k+1}}}$ by definition. Since the rest of the array `dp` has not been modified, at the start of iteration $k+1$, all the elements of the array `dp` from $\underline{\mathbf{0}}$ to $\underline{\mathbf{k+1}}$. are the Lucky numbers $\underline{\mathbf{L_0}}$ through $\underline{\mathbf{L_{k+1}}}$. Thus, the inductive step has been proven.

(d) (2 points) Prove that the for-loop of `dp_lucky` terminates after a finite number of iterations, completing the proof of the loop invariant.

Solution: The for-loop starts at $j = 1$, and terminates when $j = n$, where n is a parameter passed into the function `dp_lucky`. Since the for-loop must terminate after n iterations, and n is finite, the for-loop is guaranteed to terminate.

(e) (2 points) Finally, use the loop invariant to prove that `dp_lucky` returns the correct value of L_n.

Solution: By the loop invariant, at the start of iteration $j = n$, all the elements of the array `dp` from 0 to n are the Lucky numbers L_0 through L_n. Since the loop terminates when $j = n$, and returns `dp[n]`, `dp_lucky` returns the correct value L_n.

(f) (2 points) Analyze the code of `dp_lucky`. Give a big-O bound on the running time for this function.

$O(n)$

(g) (2 points) In 20 words or less, describe a similarity between memoization and dynamic programming, and how both improve the performance of computing L_n over the recursive version.

Solution: Both DP and memoization store intermediate values of Lucky numbers so that they are not recomputed again.

(h) (2 points) In 20 words or less, describe a difference between the memoization solution found in lab, and the dynamic programming solution presented here.

Solution: DP fills memo of Lucky numbers from 0 to n (bottom-up approach), while memoization starts from n, and recurses down to fill in the memo (top-down approach).
Note: This was an optional question.

4. **[Golden Lucky Numbers] - 25 points.**

Define the following constant ϕ, which is also known as the Golden Ratio:

$$\phi = \frac{1+\sqrt{5}}{2}$$

(a) (3 points) Show that ϕ and $1-\phi$ are the roots to the equation $x^2 = x+1$.

Solution:

$$\begin{aligned} & x^2 = x+1 \\ \implies & x^2 - x - 1 = 0 \\ \implies & x = \frac{-(-1) \pm \sqrt{(1)^2 - 4(1)(-1)}}{2(1)} \quad \textit{by the quadratic formula} \\ \implies & x = \frac{1 \pm \sqrt{5}}{2} \\ \implies & x = \phi \text{ or } x = 1-\phi \end{aligned}$$

(noting that $\frac{1-\sqrt{5}}{2} = 1 - \frac{1+\sqrt{5}}{2} = 1-\phi$)

The following formula for the Lucky numbers is claimed to be true for all natural numbers n (where the natural numbers are the numbers $0, 1, 2, \ldots$):

$$L_n = (\phi)^n + (1-\phi)^n \tag{1}$$

In the remainder of this problem, we ask you to prove formula (1) by induction.

(b) (3 points) Prove that formula (1) holds for the base case(s).

Solution:
Base case: $n = 0$. Then,

$$\phi^0 + (1-\phi)^0 = 1 + 1 = 2 = L_0$$

Base case: $n = 1$. Then,

$$\begin{aligned} \phi^1 + (1-\phi)^1 &= \frac{1+\sqrt{5}}{2} + (1 - \frac{1-\sqrt{5}}{2}) \\ &= \frac{1}{2} + \frac{1}{2} \\ &= 1 = L_1 \end{aligned}$$

Hence, formula (1) holds for the two base cases presented.
To think about: why do we need two base cases here instead of one?

(c) (6 points) Complete the following induction proof by filling in the blanks:

Suppose that, for some integer $k \geq 1$, formula (1) is true for all $1 \leq m \leq k$; in other words, for any m between 1 and k:

$$L_m = (\phi)^m + (1 - \phi)^m.$$

Prove that formula (1) holds for $n = k + 1$.

Solution:

$$L_{k+1} = L_k + L_{k-1} \qquad \textit{by definition of Lucky Numbers}$$

$$= (\phi)^k + (1 - \phi)^k + (\phi)^{k-1} + (1 - \phi)^{k-1} \qquad \textit{by the inductive hypothesis}$$

$$= (\phi)^{k-1}(\phi + 1) + (1 - \phi)^{k-1}((1 - \phi) + 1) \qquad \textit{by factoring}$$

$$= (\phi)^{k-1}(\phi)^2 + (1 - \phi)^{k-1}(1 - \phi)^2 \qquad \textit{by the fact in part a)}$$

$$= (\phi)^{k+1} + (1 - \phi)^{k+1} \qquad \textit{by simplification}$$

Therefore, the inductive step is proven, and by the principle of mathematical induction, formula (1) is true for all natural numbers n.

(d) (2 points) What is the difference between weak induction and strong induction? Which form of induction did we use in the above inductive proof?

Solution: We used strong induction in the above proof. In strong induction, the inductive hypothesis is assumed to be true for interval of values of k. In weak induction, the inductive hypothesis is assumed to be true for one single value of k (this is the form of induction we used in our loop invariant proof for question 2).
To think about: Why was strong induction necessary for this question's proof, but weak induction was sufficient for the loop invariant proof?

We have now proven a simple formula for L_n! Let's put this into code. The following code snippet uses a small variation to the function `recpow` as seen in HW2:

```
#include <math.h>

float recpow(float x, unsigned long n)
{
   if (n == 0)
       return 1;
   if (n == 1)
       return x;

   float half = recpow(x, n/2);
   if (n % 2 == 1)
       return x*half*half;

   return half*half;
}

unsigned long gold_lucky(unsigned long n)
{
    float gold = (1 + sqrt(5))/2;
    return recpow(gold, n) + recpow(1 - gold, n);
}
```

(e) (2 points) In 20 words or less, describe why are we using `float` as the return type for `recpow` instead of `unsigned int` or `unsigned long`.

Solution: `float` supports numbers with decimal places, which is required to compute $\sqrt{5}$ and its powers.

(f) (3 points) Write a recurrence relation to describe the running time of the function `recpow`.

Solution: For constants b, c, we have:

$$T(n) = \begin{cases} b & \text{if } n = 0, 1 \\ T\left(\frac{n}{2}\right) + c & \text{otherwise} \end{cases}$$

Note: It is possible to split the base cases into two cases, and to split the two recursive cases into two cases by even and odd

$$n$$

.

(g) (4 points) Find a closed form solution for the recurrence relation found in part (e). You may assume that n is a power of two; however, you should explain (in 20 words or less) how your solution changes if n is not a power of two.

Solution: For ease of notation, let n be a power of 2. If n were not to be a power of 2, we would have to use the floor function at each iteration of $\frac{n}{2}$.

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + c \\
&= T\left(\frac{n}{4}\right) + c + c \\
&= T\left(\frac{n}{8}\right) + c + c + c \\
&\cdots \\
&= T\left(\frac{n}{2^k}\right) + ck && \text{by extrapolation} \\
&= T\left(\frac{n}{2^{log_2(n)}}\right) + clog_2(n) && \text{solving } \tfrac{n}{2^k} = 1 \text{ for } k \\
&= T(1) + clog_2(n) \\
&= b + clog_2(n)
\end{aligned}$$

Write the closed form here: $b + clog_2(n)$

(h) (2 points) What is a big-O bound for the running time for `gold_lucky`? You may assume that the assignment operator, addition, subtraction, multiplication, division, and the `sqrt` function take constant time.

$O(log(n))$

5. **[Running Lucky Numbers] - 7 points.**

   On the course website download the file `hw3code.zip`. This file contains all the code snippets from this document. To run all versions of `lucky` presented here, run the following to compile your code, and replace `13` with a (luckier) number of your choice.

```
make
time ./lucky 13    # Use naive recursive formula
time ./lucky 13 -d # Use dynamic programming
time ./lucky 13 -g # Use Golden Ratio formula
```

   (a) (2 points) For each function, write down the **(real) execution time**, in seconds, for that function for each value of n in the table below.

   Solution: On an Early 2015 MacBook Pro, with a 2.7 GHz Intel Core i5 processor:

n	`lucky`	`dp_lucky`	`gold_lucky`	Expected Value
5	0.005s	0.004s	0.004s	11
10	0.005s	0.004s	0.004s	123
50	106.42s (!)	0.004s	0.004s	28143753123
100	*Too slow!*	0.004s	0.004s	792070839848372253127

   (b) (1 point) Do the running times of the functions above match with your big-O runtime bounds that you have found in this assignment?

   Solution: Yes! The naive recursive solution is much slower than either the Dynamic Programming solution or the Golden Ratio formula solution.

   (c) (2 points) For which functions, and for which inputs n does the function **not** produce the correct value L_n? For example, if `lucky(5)` produced 10 instead of 11, include `lucky(5)` in your answer here.

   Solution: `gold_lucky(50)`, `gold_lucky(100)` and `dp_lucky(100)` did not produce the correct value of L_{50}, L_{100}, and L_{100} respectively.

   (d) (2 points) Can you describe one reason why some of the functions presented do not always produce the expected return value, even though we have proven that these functions return L_n correctly?

   Solution: The value of L_{100} is too large for `unsigned long` and overflows. In addition, `float` causes undesired roundoff errors, hence `gold_lucky` does not work very well for larger Lucky numbers.