# Assignment 1 - CS 3340B


### Name: Riley Emma Gavigan

### Student Number: 251150776

### Student ID: rgavigan


## Question 1

A complete binary tree of height 0 consists of 1 node, which is the root. h = 0, n = 1.

A complete binary tree of height h + 1 consists of two complete binary trees of height h, whose roots are connected to a new root.

Let T be a complete binary tree of height $h$. Prove that the number of leaves of the tree is $2^h$, and the number of nodes in T is $2^{h+1} - 1$.

**Proof:**

**Base case**: A complete binary tree of height 0 consists of 1 node. The number of leaves = $2^h = 2^0 = 1$. Since there is 1 node, it is evident that that node is also a leaf node. The number of nodes in T is $2^{0+1} - 1 = 2^1 - 1 = 2 - 1 = 1$. Thus, the base case is true.

**Inductive hypothesis**: Assume that the number of leaves for a complete binary tree T is $2^h$ and the number of nodes in T is $2^{h+1} - 1$. That is, P(T) is true.

**Inductive step**: We look to prove that using P(T), we can build a tree P(T') and prove P(T') is true. That is, $h_{T'} = 2^{(h+1)}, n_{T'} = 2^{(h+2)} - 1$.

Let T' be a complete binary tree composed of two complete binary trees T connected by a single node that acts as a new root. This tree will have height h + 1.

Thus, by IH the number of leaves of T' will be $2(2^h)$.

$2^1(2^h) = 2^{h+1} = IH$. Thus, it holds that for the tree T', the number of leaf nodes = $2^{h+1}$, which is true by the inductive hypothesis.

Furthermore, the number of nodes of tree T' will be $2(2^{h+1} - 1) + 1$.

$n = 2^1(2^{h+1} - 1) + 1 = (2^{h+2} - 2^1) + 1 = 2^{h+2} - 2 + 1 = 2^{h+2} - 1$.

Therefore, it holds that for the tree T', the number of nodes is $2^{h+2} - 1$ by the inductive hypothesis. Therefore, by induction we know that $h(T) = 2^h$ and $nodes(T) = 2^{h+1} - 1$ for all h $\geq$ 0 and n $\geq$ 1. Q.E.D.

## Question 2

$N_0 = 2;\ N_1 = 2;\ N_n = N_{n-1} + N_{n-2},\ \ n > 1.$

Prove that $N_n = 2F_{n+1}, n \geq 0$, where $F_i$ is the Fibonacci number.

Proof:

Fibonacci sequence is as follows: $F_0 = 1;\ F_1 = 1;\ F_n = F_{n-1} + F_{n-2}, n > 1$

**Base Case**: $F_0 = 0; F_1 = 1; F_2 = 1$

$N_0 = 2F_1 = 2(1) = 2.$

$N_1 = 2F_2 = 2(1) = 2.$

Thus, the base case holds for $N_0$ and $N_1$.

**Inductive Hypothesis**: Assume that $N_n = 2F_{n+1}$ for a value $n \geq 0$. Furthermore, we know that $N_n = N_{n-1} + N_{n-2}$ and $F_{n+1} = F_n + F_{n-1}$. Assume that $N_{n-1} = 2F_n,\ \ N_{n-2} = N_{n-2} + N_{n-3}$, and $F_n = F_{n-1} + F_{n-2}$.

**Inductive Step**: We seek to prove that $N_{n+1} = 2F_{n+2}$ by using the inductive hypothesis.

$N_{n+1} = N_n + N_{n-1} = 2F_{n+1} + 2F_n$

We know that $F_{n+2} = F_{n+1} + F_n$, thus since:

$N_{n+1} = 2(F_{n+1} + F_n)$, we know that $N_{n+1} = 2F_{n+2}$. Therefore it is true that if $N_n = 2F_{n+1}$ and $N_{n-1} = 2F_n$, then $N_{n+1} = 2F_{n+2}$.

Therefore, $N_n = 2F_{n+1}$ for all n, where $n \geq 0$. Q.E.D.

## Question 3

**Exercise 2.3-5**: Insertion sort can be written recursively. To sort A[1:n], recursively sort the subarray A[1:n - 1] and then insert A[n] into the sorted subarray A[1:n - 1]. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

```
def insertion_sort(array, n):
  # Return when size is 1 or 0 (nothing to sort)
  if n <= 1 then return

  # Sort A[1:n-1] recursively
  insertion_sort(array, n - 1)

  # Insertion element is the last element
  insert_val = array[n]
  i = n - 1

  while (i >= 1 and array[i] > insert_val) do:
    array[i + 1] = array[i]
    i -= 1

  array[i + 1] = insert_val
```

Recurrence For Worst-Case Running Time:

T(n) = T(n - 1) + n

T(1) = 1

- This recurrence shows that when n = 1, it is simply a constant operaton because it simply returns. In the recursive case, a recursive call is made to n - 1, and n is required for the while loop as it could, in the worst case, have to go through the entire array to find a place to insert the value (would be inserting at the start as it is smaller than everything). This worst case would occur when we have an array of values sorted in strictly descending order, such as [5,4,3,2,1].

## Question 4

**Problem 3-2**: Indicate, for each pair of expressions $(A, B)$ in the table below whether A is $O$, $o$, $\Omega$, $\omega$, or $\theta$ of $B$. Assume that $k \geq 1, \epsilon > 0$, and $c > 1$ are constants. Write your answer in the form of the table with “yes” or “no” written in each box.

| | A | B | O | o | Ω | ω | Θ |
|---|---|---|---|---|---|---|---|
| a. | lg^(k) * n | n^ε | yes | yes | no | no | no |
| b. | n^k | c^n | yes | yes | no | no | no |
| c. | sqrt(n) | n^(sin*n) | no | no | no | no | no |
| d. | 2^n | 2^(n/2) | no | no | yes | yes | yes |
| e. | n^(lg(c)) | c^(lg(n))] | yes | no | yes | no | yes |
| f. | lg(n!) | lg(n^n) | yes | no | yes | no | yes |

## Question 5

**Problem 4-6**: Professor Diogenes has $n$ supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

| Chip A Says | Chip B Says | Conclusion |
|---|---|---|
| B is good | A is good | both are good, or both are bad |
| B is good | A is bad | at least one is bad |
| B is bad | A is good | at least one is bad |
| B is bad | A is bad | at least one is bad |

Now you will design an algorithm to identify which chips are good and which are bad, assuming that more than n/2 of the chips are good. First, you will determine how to identify one good chip.

**b) Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size. That is, show how to use $\lfloor n/2 \rfloor$ pairwise tests to obtain a set with at most $\lceil n/2 \rceil$ chips that still has the property that more than half of the chips are good.**

**Algorithm using $\lfloor n/2 \rfloor$ pairwise tests to reduce problem to nearly half the size (recursive):**

- Initially: n chips, more than n/2 are good, less than n/2 are bad.
- Case 1: n is even
  - Divide the chips into $\lfloor n/2 \rfloor$ pairs evenly.
- Case 2: n is odd
  - Divide the chips into $\lfloor n/2 \rfloor$ pairs, and leave 1 chip out.
- Recursive Loop:
  - With divided chips into pairs, perform pairwise test on each pair:
    - Case 1: At least one chip reports the other is bad:
      - Ignore both chips (remove them from the pool)
    - Case 2: Both chips report the other chip as good:
      - Ignore one of the chips (remove it from the pool)
      - Add the other chip to a set 'f'
  - Combine all chips that aren't ignored into set 'f', along with the chip left out in the odd case (Case 2)
  - Repeat until only one chip is left → that chip is guaranteed to be a good chip

```
Pseudocode-like description of the algorithm:
def AlgorithmForChips(chips, n):
  if n == 1: return chips

  // create pairs
  if n is even:
    create floor(n/2) pairs and put into the set pairs
  if n is odd:
    create floor(n/2) pairrs and put into the set pairs
    put remaining chip into result_set

  // do pairwise tests
  for pair in pairs:
    perform pairwise test
    if one or more chips report the other as bad:
      remove both chips
    else if both chips report other as good:
      ignore one chip
```

```
      add the other chip into result_set

  // make recursive call on n-1
  if n > 1:
    AlgorithmForChips(result_set, n - 1)
```

**How to apply this algorithm in this exact case, ensuring $\lfloor n/2 \rfloor$ pairwise tests obtains a set with at most $\lceil n/2 \rceil$ chips, where more than half are still good:**

- Put chips into pairs according to Case 1 and 2 → depending on whether even or odd. This results in $\lfloor n/2 \rfloor$ pairs.
- Perform the loop iteration one time (this performs $\lfloor n/2 \rfloor$ pairwise tests):
  - In case 1: at least one of the discarded chips is bad, so the property will still hold. This is taken from rows 2-4 of the options in the table (at least one is bad in all cases where 1+ reports of the other being bad exist).
  - In case 2: This will either add 1 good or 1 bad chip into the set that will be obtained. Since at least n/2 chips are good, this means that the set resulting will have more than half are still good because the other cases will offset / lead to more bad chips being discarded than good chips.
- After the loop, the remaining chip (in the odd case) is added to the set.
- There will be most $\lceil n/2 \rceil$ chips in the set:
  - Case 1 (Even):
    - If all chips are good, this is the maximal set amount as we add in each iteration. Since we add only one chip at a time, n/2 chips are added, which is $\lceil n/2 \rceil$
  - Case 2 (Odd):
    - If all chips are good, this is the maximal set amount as we add in each iteration. We add $\lfloor n/2 \rfloor$ chips into the set, add 1 for the odd remainder, retrieving $\lceil n/2 \rceil$ chips at most in the set after the algorithm is applied.

**c) Show how to apply the solution to part (b) recursively to identify one good chip. Give and solve the recurrence that describes the number of tests needed to identify one good chip.**

**Applying to identify one good chip:**

- The algorithm follows the same, but after performing the loop iteration a recursive call is made on the new set that is created, where a new set of $\lfloor n/2 \rfloor$pairs are created, pairwise tests are performed, and a new set of size $\lceil n/2 \rceil$ at max is left with at least half of the chips being good. This continues until the base case where $n = 1$, and this is guaranteed to be one good chip because the property of more good chips than bad chips existing will still hold.

**Recurrence:**

$T(n) \leq T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor$

$T(1) = c$

**Solution (Master Theorem):**

$T(n) \leq aT(n/b) + f(n)$

$f(n) = \theta(n^k log^p n)$

$a = 1, b = 2, k = 1, log_b a = log_2(1) = 0, p = 0$

Case 3: $log_b a < k (0 < 1)$

$p >= 0$, thus $T(n) = \theta(n^k log^p n) = \theta(\lfloor n/2 \rfloor) = \theta(n)$

**d) You have now determined how to identify one good chip. Show how to identify all the good chips with an additional $\theta(n)$ pairwise tests.**

To identify all the good chips with an additional $\theta(n)$ pairwise tests, we can simply do pairwise comparisons with the remaining $n-1$ chips after finding a good chip. We can store the good chips we find in each of these tests in a set and then return them after all good chips are found. The approach by doing this gives us $\theta(n)+n-1=\theta(n)$ total tests.

## Question 6

Suppose that the running time of a recursive program is represented by the following recurrence relation:

$T(2) \leq 2c$

$T(n) \geq 2T(n/2) + cn^2log_2(n)$

Determine the time complexity of the program using recurrence tree method (not using master theorem) and then prove your answer.

1 T(n)

2 T(n/2) T(n/2)

3 T(n/4) T(n/4) T(n/4) T(n/4)

… … … … … … … …

1 $cn^2log_2(n)$

2 $c(n/2)^2log_2(n/2)$ $c(n/2)^2log_2(n/2)$

3 $c(n/2^2)^2log_2(n/2^2)$ $c(n/2^2)^2log_2(n/2^2)$ $c(n/2^2)^2log_2(n/2^2)$ $c(n/2^2)^2log_2(n/2^2)$

… … … … … … … …

bottom: 2c + 2c + 2c + 2c… $a^i$ nodes = $2c * 2^i$.

Sums:

- Level 1: $cn^2log_2(n/2^0)/2^0$
- Level 2: $2(c(n/2)^2log_2(n/2)) = 2(cn^2log_2(n/2)/4) = cn^2log_2(n/2)/2^1$
- Level 3: $4(c(n/2^2)^2log_2(n/2^2)) = 4(cn^2log_2(n/4)/16) = cn^2log_2(n/2^2)/2^2$
- Level n: $cn^2log_2(n/2^{n-1})/2^{n-1}$

Levels: $log_bn + 1 = log_2n + 1$

$T(n) \geq cn^2log_2(n) + cn^2log_2(n/2)/2 + cn^2log_2(n/4)/4 + ...logn$ times.

$\geq cn^2 * \sum_{k=1}^{log_2n} \frac{log_2(\frac{n}{2^k})}{2^k}$

Simplifying: Let m = $log_2n$

$\sum_{k=1}^{m} \frac{log_2(n/2^k)}{2^k} = \sum_{k=1}^{m} \frac{log_2n-k}{2^k} = log_2n\sum_{k=1}^{m}\frac{1}{2^k} - \sum_{k=1}^{m}\frac{k}{2^k}$

$= log_2n(1) - 2$

$T(n) \geq cn^2 * (log_2(n) - 2) = cn^2log_2n * -2cn^2 = cn^2log_2n = n^2log_2n$

Therefore, the time complexity of the program is $\Omega(n^2log_2n)$

Proof by Substitution Method:

- The substitution method is a valid way to prove the time complexity because it utilizes mathematical induction to prove that the guess is correct, thus proving the recurrence tree is also correct.

$T(2) \leq 2c$

$T(n) \geq 2T(n/2) + cn^2log_2(n)$

We use $n^2log_2n$ as we found it from the recurrence tree method.

We show $T(n) \geq n^2log_2n$

Base Case: $2c > T(2) = 2^2 log_2(2) = 4(1) = 4, 2c > 4. c > 2$.

$T(n) \geq 2T(n/2) + cn^2 log_2(n)$

$\geq 2(c(n/2)^2 log_2(n/2)) + cn^2 log_2 n$

$\geq cn^2 log_2(n/2)/2 + cn^2 log_2 n$ Factor in the 2 and simplify

$\geq c(n^2 log_2(n/2)/2 + n^2 log_2 n)$

$\geq n^2 log_2(n/2)/2 + n^2 log_2 n$ Take out constant c (insignificant term - a constant > 2)

$\geq n^2 log_2 n$ Inequality holds by removing first term

$T(n) \geq \Omega(n^2 log_2 n)$ Q.E.D.

## Question 7

$N_n = N_{n-1} + N_{n-2}, n > 1;\ N_1 = 2;\ N_0 = 2.$

**a) Write a recursive function to compute Nn using the above definition directly. Implement your solution and print $N_{i*5}$, where 0 ≤ i ≤ 10, as output.**

> See **Question_7a.java** for the program, below is the same code

```
/*
 *  Name: Riley Emma Gavigan
 *  Student ID: 251150776
 *  UWO ID: rgavigan
 *  Assignment: CS 3340B Assignment 1
 *  Question: 7a)
 */

public class Question_7a {
    // Function to calculate fib-like numbers
    public static long recursiveFibLike(int n) {
        // Base case: return 2 for n = 0 and n = 1
        if (n <= 1) {
            return 2;
        }
        // Recursive case: Nn = N(n - 1) + N(n - 2)
        return recursiveFibLike(n - 1) + recursiveFibLike(n - 2);
    }

    public static void main(String args[]) {
        // Slower recursive function
        for (int i = 0; i <= 10; i++) {
            // Get i * 5, calculate it's fibonacci-like value, then print result
            int val = i * 5;
            long res = recursiveFibLike(val);
            System.out.println("N(" + val + "): " + res);
        }
    }
}
```

**b) Write a recursive function/procedure to compute $N_n$ with time complexity O(n) (more precisely, the time complexity should be O(nA(n)) when n is large, where A(n) is the complexity of adding $N_{n-1}$ and $N_{n-2}$). Implement your solution and print $N_{i*20}$, where 0 ≤ i ≤ 25, as output. This program must be able to compute $N_n$ precisely for n ≤ 500.**

TODO: FIX CARRY SECTION

> See **Question_7B.java** for the program, below is the same code

```
/*
 *  Name: Riley Emma Gavigan
 *  Student ID: 251150776
 *  UWO ID: rgavigan
 *  Assignment: CS 3340B Assignment 1
 *  Question: 7b)
 */

public class Question_7b {
    // Need a size large enough to handle the result from N(500)
    private static final int SIZE = 150;
    private static int[][] res = new int[2][];
    // Id matrix that won't be changing, is used to multiply with Nn-1
    private static int[][][] Id = new int[2][2][];


    // Function to calculate fibonacci-like numnbers
    public static int[][] recursiveFibLike(int n) {
        int[] two = new int[SIZE];
        int[] one = new int[SIZE];
        int[] zero = new int[SIZE];
        initializeVal(two, 2);
        initializeVal(one, 1);
        initializeVal(zero, 0);

        // Initialize res with [2, 0], where values are integer arrays
        res[0] = two;
        res[1] = zero;

        // Initialize Id array with {{1, 1}, {1, 0}}
        Id[0][0] = one;
        Id[0][1] = one;
        Id[1][0] = one;
        Id[1][1] = zero;

        // Base case: N0 and N1
        if (n == 0) {
            return res;
        }

        // Recursive case: Calculate the values of N_n and N_n-1 by using matrix powers
        calcMatrix(n - 1);
        return res;
    }

    // Function to initialize value to 2 (N0 and N1 cases)
    public static void initializeVal(int[] val, int num) {
        val[SIZE - 1] = num;
    }

    // Function to calculate the powers of a matrix for determination of fib-like numbers
    public static void calcMatrix(int n) {

        // Loop to multiply (1 1; 1 0) with Gn-1, n-1 times
        for (int i = 0; i <= n; i++) {
            multMatrix();
        }
    }

    // Helper function to perform matrix multiplication for powers
    public static void multMatrix() {
        // Do 2x2 matrix multiplication [res[0] = res[0] + res[0]. res[1] = res[1] + 0
        int[] a = numAddition(res[0], res[1]);
        res[1] = res[0];
        res[0] = a;
    }

    // Method to add two numbers together in integer arrays
    public static int[] numAddition(int num1[], int num2[]) {
        int[] sum = new int[SIZE];
        int carry = 0;

        // Loop backwards
        for (int i = SIZE; i > 0; i--) {
            int tempSum = 0;

            // Handle carry at start of addition
            if (carry == 1) {
                tempSum = carry;
```

```
                carry = 0;
            }

            // Calculate sum of i-1'th indices in nums
            tempSum += num1[i - 1] + num2[i - 1];

            if (tempSum > 9) {
                carry = 1;
            }
            // Handle when there is a carry (get just one's place)
            sum[i - 1] = tempSum % 10;
        }
        return sum;
    }

    // Method to print out resulting values that are stored as integer arrays
    public static void printRes(int[] val) {
        boolean removeLeadZeros = true;

        for (int j = 0; j < SIZE; j++) {
            while (removeLeadZeros) {
                // Loop past leading zeros
                if (j < SIZE && val[j] == 0) {
                    j++;
                }
                else {
                    removeLeadZeros = false;
                }
            }

            if (j < SIZE) {
                System.out.print(val[j]);
            }
            else {
                System.out.print("0");
            }
        }

        System.out.println();
    }
    public static void main(String args[]) {
        // Faster recursive function
        for (int i = 0; i <= 25; i++) {
            int val = i * 20;
            int[][] res = recursiveFibLike(val);
            System.out.print("N(" + val + "): ");
            printRes(res[0]);
        }
    }
}
```

**c) Use the Unix time facility (bash time command) to track the time needed for each algorithm. Compare the results and state your conclusion in two or three sentences**

**See asn1_script.nice** for thetypescript file. The output is also displayed below.

> Script started on 2023-01-25 20:11:27-05:00 [TERM="xterm-256color" TTY="/dev/pts/40" COLUMNS="80" LINES="24"]
> 0;rgavigan@compute:~/cs3340/asn1[rgavigan@compute asn1]$ who am i
>
> 0;rgavigan@compute:~/cs3340/asn1[rgavigan@compute asn1]$ date
>
> Wed 25 Jan 2023 08:11:33 PM EST
> 0;rgavigan@compute:~/cs3340/asn1[rgavigan@compute asn1]$ pwd
>
> /home/rgavigan/cs3340/asn1
> 0;rgavigan@compute:~/cs3340/asn1[rgavigan@compute asn1]$ whoami

```
rgavigan
0;rgavigan@compute:~/cs3340/asn1[rgavigan@compute asn1]$ time java Question_7a

N(0): 2
N(5): 16
N(10): 178
N(15): 1974
N(20): 21892
N(25): 242786
N(30): 2692538
N(35): 29860704
N(40): 331160282
N(45): 3672623806
N(50): 40730022148

real  1m4.953s
user 1m4.578s
sys   0m0.046s
0;rgavigan@compute:~/cs3340/asn1[rgavigan@compute asn1]$ time java Question_7b

N(0): 2
N(20): 21892
N(40): 331160282
N(60): 5009461563922
N(80): 75778124746287812
N(100): 1146295688027634168202
N(120): 173400147970158973161038 42
N(140): 26230240268816379067306864 9732
N(160): 396784842812383886449561214 8392122
N(180): 60021642909926907815061334295658979762
N(200): 907947388330615906394593939394821238467652
N(220): 13734520083255583906104114706164126578641192042
N(240): 20776208439145982941702103676555080336028407 3551682
N(260): 31428170368550927563356933170483722962668906019751 01572
N(280): 47541393108744903733630203389969690960078450775793287927962
N(300): 71915865041316712192353133034437819810473442861853446451117 9602
N(320): 10878712857258585944592354700489205612760626741634120068867325911492
N(340): 16456228867259197917068142763076888295985180261596490566322157455195 9882
N(360): 24893337298715860116563120111782861920444726054110743863334326893801712235 22
N(380):
37656151167205192925733052622412507596287854142201519626100930629032275546257412
N(400):
56962459621697922351597787536292199123076017756460978197295439129193854280806464780

N(420):
```

> 8616711229318093546921004394881868338935201609731998029703361450972223708025318381
>
> N(440):
> 1303449901962702048672948099654001470001508035196557566307129666959024367393604483
>
> N(460):
> 1971728658082268159709475043425603628789412865906631020820797017505554709584081794
>
> N(480):
> 2982633928046548025565502411460429612729730142241880393230043985309355939597596855
>
> N(500):
> 4511830323238726617450253900721441440920226498275163811772777328369492554773737668
>
> real 0m0.112s
> user 0m0.084s
> sys 0m0.027s
> 0;rgavigan@compute:~/cs3340/asn1[rgavigan@compute asn1]$ exit
>
> exit
>
> Script done on 2023-01-25 20:13:08-05:00 [COMMAND_EXIT_CODE="0"]

- **Comparison of Results**
  - The first program, the slower fibonacci-like number calculator, is very slow in comparison to the second program that is $O(n)$. The first program takes just over a minute to calculate up to $N_{50}$ whereas the second program calculates up to $N_{500}$ in only approximately 0.1 seconds. This time difference is extremely significant and shows a huge difference in efficiency, where the second program is much more suited to calculate fibonacci numbers, and can store numbers up to whatever size is desired, so long as there is enough memory to store a integer arrays of the desired number of digits.

**d) Can you use your program in a) to compute $N_{50}$ if int type of 4 bytes is used? Briefly explain your answer. Explain why your program in b) can compute $N_{500}$ precisely?**

- I can not use my program in a) to compute $N_{50}$ if an int type of 4 bytes is used. When calculating $N_{50}$, the result is 40,730,022,148. However, an int type of 4 bytes can only store values in the range of -2,147,483,647 to 2,147,483,647 in a signed type, and a range of 0 to 4,294,967,295 in an unsigned int type. This value is much smaller than the result obtained (by a magnitude of 10: 4 billion vs 40 billion.), meaning in order to store this result, we would need 36 bits, which is 4.5 bytes → so 5 bytes would be the minimum if we were to round up to the next byte in order to store the result $N_{50}$.
- My program in b) can compute $N_{500}$ precisely because it works just as it would if we were to calculate regularly, with a few modifications. First, all addition operations are simply the using long addition as we would on pen and paper, going from right-to-left and using the carry bit for all additions ≥ 10. Second, the operation performed in the algorithm essentially replaces the first entry of $G_n$ with $G_{n-1} + G_{n-2}$, as we take the two values from $G_{n-1}$, which are actually just the values $G_{n-1}$ and $G_{n-2}$ and we add them together, putting that result into the top row entry of $G_n$ while replacing the bottom entry with $G_{n-1}$ so that in the next iteration, the n+1'th iteration, it will be adding $G_n + G_{n-1}$ into its top entry and replacing the bottom with $G_n$, and this will continue. This is a completely accurate way to add as it is equivalent to adding fibonacci numbers regularly, but instead does not require you to make a lot of recursive calls to retrieve the $n-1$ and $n-2$ values, as they are instead stored by the previous calculation of the n-1'th number.