Caio Brighenti
COSC 302 - Analysis of Algorithms – Spring 2019
Lab 4

1. Prove binary addition algorithm correctness using induction
   The base case for this proof is that we have two one bit numbers we are adding, along with a carry bit. As the value of each of these can be 0 or 1, we have the case where all values are 0, the case where one value is 1, the case where two values are 1, and finally the case where all values are 1.

   - **Base case 1:** All three bits are 0.
     In this case, as all three bits are 0, the sum $y = 0$ and the carry bit $c_y = 0$. This is as we would expect, as the sum of 0 and 0 with no carry is 0.
   - **Base case 2:** Exactly one bit is 1. In this case, either the sum of x_1 and x_2 would result in 0, and the single carry bit would be added resulting in 1, or the addition itself would sum up to 1, which would remain unchanged with a 0 carry bit. Either way, the output is 1, which is what we expect since $0 + 0 + 1 = 1$.
   - **Base case 3:** Exactly two bits are 1. In this case, the resulting sum must be 0 with a carry bit of one, or in other words 10 (in binary). This makes sense, as 10binary = 2. Either the sum will result in the carry bit from the two inputs, or it will result in the sum of one input and the carry bit.
   - **Base case 4:** All three bits are 1.

   As the cases above cover all possible combinations for three inputs of size one bit (two inputs and the carry bit), then the algorithm is correct for $n = 1$ where n is the length of the inputs. Next, we proceed by induction.
   Inductive hypothesis: We assume that the algorithm works, meaning that it successfully adds two input numbers of length 1 to $n - 1$, outputting the correct value and the appropriate carry bit. Alternatively, we can consider that the sum of a n bit length is simply the sum of two 1 bit numbers n times. With this framing, we assume that the addition has been performed correctly for all indices from 1 to $n - 1$. As we have reduced the the addition to a set of one-bit additions, the cases are identical to the base cases.
   The first case is that we have all bits (the carry, the current digit of the first input, and the current digit of the second input) equal to 0. In this case the resulting sum and the carry bit are equal to 0, which is what we expect.
   In the case that one of the bits is equal to 1, the resulting sum will be 1 and the carry bit will be 0. This is as we expect.
   In the case that two of the bits are equal to 1, the resulting sum is 0, and the carry bit is 1. This is what we expect.
   Finally, in the case that three of the bits are equal to 1, the resulting sum and carry bit are both 1. As 11(binary) $= 1 + 1 + 1 = 3$, this is correct.
   Clearly, our algorithm works for any given pair of corresponding indices in two input binary numbers, and a carry bit. As the inductive hypothesis shows that the algorithm holds for all indices from 0 to $n - 1$, then the algorithm mst work for all values of n.

2. Prove the recursive binary division algorithm
   The claim is that a call to divide(x,y) properly returns the quotient and remainder of this division. We proceed by the induction. First, we must show the base case.
   Base case: For this algorithm, the base case is x=0. In this scenario, the algorithm returns 0 for the quotient and the remainder. As 0 divided by any number is equal to 0 with remainder 0, then the base case is true.
   Inductive Hypothesis: We assume that the claim holds for all recursive calls up to n. Thus, we assume that divide($\lfloor \frac{x}{2} \rfloor, y$) returns the correct quotient and remainder.
   Given the inductive hypothesis, then the following forms for the quotient and remainder must be true.

$q = \frac{\lfloor \frac{x}{2} \rfloor}{y}$,
$r = \lfloor \frac{x}{2} \rfloor - qy$, and
$\lfloor \frac{x}{2} \rfloor - qy = qy + r$
These are simply definitions of quotient, remainder, and division. Next, we split into two cases: the case where x is even, and where x is odd.

Odd case

When x is odd, the algorithm returns $q_x = 2q$ and $r_x = 2r + 1$. Since x is odd, by the property of floor we must have:
$r = \lfloor \frac{x}{2} \rfloor - qy = \frac{x}{2} - \frac{1}{2}$. This is because an odd number divided by 2 will always have remainder $\frac{1}{2}$.
We start with the form below.

$$\lfloor \frac{x}{2} \rfloor = qy + r \qquad \text{inductive hypothesis} \tag{1}$$
$$\frac{x}{2} - \frac{1}{2} = qy + r \qquad \text{floor for odd numbers} \tag{2}$$
$$x - 1 = 2qy + 2r \qquad \text{multiply both sides by 2} \tag{3}$$
$$x = 2qy + 2r + 1 \qquad \text{algebra} \tag{4}$$

The final form of this equation shows that x is equal to y multiplied by the quotient $2q$, plus the remainder and 1. This is precisely what our algorithm returns in the odd case. Thus, the claim holds when x is odd.

Even case

When x is even, $\lfloor \frac{x}{2} \rfloor = \frac{x}{2}$. This is because there will be no remainder, thus nothing to round down to. In the even case, our algorithm returns that $q_x = 2q$ and that $r_x = 2r$. We show that each of these work below. First, we show the quotient as follows.

$$q = \frac{\lfloor \frac{x}{2} \rfloor}{y} \qquad \text{inductive hypothesis} \tag{5}$$
$$q = \frac{\frac{x}{2}}{y} \qquad \text{floor when x is even} \tag{6}$$
$$2q = \frac{x}{y} \qquad \text{multiply both sides by 2} \tag{7}$$
$$\tag{8}$$

Thus, it must be that the quotient of x divided by y is $2q$. We now show the remainder.

$$r = \lfloor \frac{x}{2} \rfloor - qy \qquad \text{inductive hypothesis} \tag{9}$$
$$r = \frac{x}{2} - qy \qquad \text{floor when x is even} \tag{10}$$
$$2r = x - 2qy \qquad \text{multiply both sides by 2} \tag{11}$$
$$\tag{12}$$

As the definition of remainder takes the form $r = x - qy$ where r is the remainder, x is the numerator, y is the denominator, and q is the quotient, then we have shown that the remainder r_x is equal to the $2r$, where $x = x$, $y = y$, and $q_x = 2q$. Thus, the even case is true for both the quotient and remainder.

Overflow case

As a final note, the algorithm may also increment an r such that $r \geq y$. As no division could have a remainder greater than the numerator, the algorithm resolves this case by incrementing the quotient by one , and subtracting the remainder by y. In practice, this means that y fits into x one more time, thus the remainder should be decreased by $1y$ and the quotient incremented by 1. With this final correction, the claim is true in all cases.

3. Show the running time of the recursive binary division algorithm
We proceed by the method of finding the running time for each recursive time, followed by finding the

total number of recursive calls. The total running time is thus the number of recursive calls time the running time per call.

Running time per call

This section is straightforward. Each line in the code performs a single comparison or operation, and is thus in constant time. However, the recursive call passes a parameter of $\lfloor \frac{x}{2} \rfloor$, which must be calculated. The running time for this is n, as a division by two simply shifts the bits to the right, truncating the last bit. As an example:

$$101010 = 42 \tag{13}$$

$$010101 = 21 \tag{14}$$

$$001010 = 10 \tag{15}$$

Thus, the running time for each call is $T(n) = 1 + n = n$, as the n dominates.

Total number of calls

As the base case of the algorithm is $x = 0$, we will make recursive calls until $\lfloor \frac{x}{2} \rfloor$ goes to 0. Let us consider the same bitwise shift strategy. Each call to $\lfloor \frac{x}{2} \rfloor$ adds a 0 to the left, shifts all bits to the right one, truncating the last bit. The value of x will go to 0 once all bits become 0. In the worst case, we have a 1 in the leftmost position of x. In this case, the algorithm must run n times, where n is the bit length of x, each time shifting to the right until $x = 0$. Thus, the total number of calls is equal to n, where n is the bit length of x.

Total running time

We found the running time for each call to be n, where n is the binary length of x. We found the total number of calls to be n, where n is the binary length of x. Thus, the total running time must be $T(n) = n * n = n^2$.

4. Show the running time of the recursive Fibonacci algorithm

   To show the running time of a recursive algorithm, we must find the running time per recursive call, and the number of recursive calls.

   Running time per call

   The running time per call is straightforward, as the algorithm performs only a single operation: addition. As such, the running time of each call is constant $T(n) = 1$.

   Number of calls

   The number of calls for the Fibonacci algorithm is less straightforward to show. This is because each call makes two calls, and each of these calls are of different lengths. We thus proceed by representing the calls as a binary tree, where each node is a recursive call and each child is recursive calls made from the node.

   If we start the tree with call n, this call will have two children, $n - 1$ and $n - 2$. Thus, the first level will have 1 node, and the 2nd 2. Each of the two nodes in the 2nd level also make two calls, thus the third level has 4 nodes $(n - 2, n - 3, n - 4, n - 5)$. This pattern continues until n goes to 0, thus there are n levels total, as the largest n in each level is 1 less than the previous level.

   As we know the number of levels the tree has is n, we can now determine how many nodes each level has, and thus sum them to obtain the total number of nodes. Each level of the three contains 2^i nodes (1,2,4,etc) where i is the level of the tree. As we know the tree has n levels, we have the following sum: total calls $= 2^0 + 2^1 + 2^2 + ...2^n$

   This series is know to simplify to $2^{n+1} - 1$. This is thus the total number of recursive calls.

   Total running time

   As we know the running time per recursive call, and the total number of recursive calls, we multiply the two to obtain the total running time of the Fibonacci algorithm as a function of the input n. The total running time is thus $2^{n+1} - 1 - \Theta(1) = \Theta(2^n)$, as the exponential aspect dominates. Thus, the recursive Fibonacci algorithm runs in exponential time.