MATH 182 DISCUSSION 1 WORKSHEET SOLUTIONS

1. Arithmetic with extra effort

In some circumstances (for example, in ordinal arithmetic), addition of natural numbers is defined as repeated incrementation. That is, we define

$$m + n = m\overbrace{+ 1 + \cdots + 1}^{n \text{ times}}.$$

The following pseudocode implements this definition:

InductiveAdd(m, n)

```
1  sum = m
2  for j = 1 to n
3        sum = sum + 1
4  return sum
```

Let's start by tracing through an example. Suppose $m = 10$ and $n = 3$. Then:

(1) In line 1, we introduce the variable *sum* and assign it the value $m = 10$. Thus $sum = 10$.
(2) In line 2, we begin a **for** loop. The counter j takes the value $j = 1$. We still have $sum = 10$.
(3) In line 3, we compute the value $sum + 1 = 10 + 1 = 11$, and reassign *sum* to be 11. We now have $j = 1$ and $sum = 11$.
(4) We go back to line 2, and increment the counter j to $j = 2$. We still have $sum = 11$.
(5) We go to line 3, compute $sum + 1 = 11 + 1 = 12$, and reassign *sum* to 12. We now have $j = 2$ and $sum = 12$.
(6) We go back to line 2, and increment the counter j to $j = 3$. We still have $sum = 12$.
(7) We go to line 3, compute $sum + 1 = 12 + 1 = 13$, and reassign *sum* to 13. We now have $j = 3$ and $sum = 13$.
(8) We go back to line 2, and increment the counter j to $j = 4$. Since $4 > 3$, we do not go into the body of the loop again. We have $sum = 13$.
(9) We go to line 4 now. We return $sum = 13$. Thus InductiveAdd$(10, 3)$ returns $13 = 10+3$, as expected.

Having seen an example, we now prove carefully that InductiveAdd(m, n) computes $m + n$ correctly for all m and n. We now trace through the algorithm with an arbitrary input. First, line 1 sets $sum = m$. Line 2 initiates a **for** loop, Since this line will be executed multiple times, to determine the value of *sum* after line 2, we need to find a loop invariant. In our example, line 2 is run in items (2), (4), (6), and (8). After running line 2, the following occur:

- $j = 1$ and $sum = 10$
- $j = 2$ and $sum = 11$
- $j = 3$ and $sum = 12$
- $j = 4$ and $sum = 13$

The pattern that we see is that $sum = j + 9$. Since $9 = 10 - 1 = m - 1$, and *sum* was initialized to m, we can intuit that the pattern is actually $sum = m + j - 1$. If this were not clear from our example, we could have tried another example, with a different value of m, and it would have become clear. We therefore propose the following loop invariant:

(Loop Invariant) Each time line 2 is run, $sum = m + j - 1$.

Now we must prove that this is in fact a loop invariant.

(Initialization) The first time line 2 is run, $j = 1$ and $sum = m = m + j - 1$.

(Maintenance) Suppose that we have just finished executing line 2, that $j = j_0$ for some $1 \leq j_0 \leq n$, and that the loop invariant is true at this point. Then $sum = m + j_0 - 1$. We then execute line 3, which reassigns sum to be $(m + j_0 - 1) + 1 = m + j_0$. Then we return to line 2 and increment j, setting $j = j_0 + 1$. Then $sum = m + j_0 = m + (j_0 + 1) - 1$, so the loop invariant is still true.

(Termination) The last time that line 2 is run is when j becomes $n + 1$. At this point, the loop invariant guarantees that $sum = m + j - 1 = m + n$. Then line 4 is executed and the program returns $sum = m + n$. This concludes the proof of correctness for InductiveAdd.

Remark. The case $n = 0$ bears some additional scrutiny. In this case, line 2 becomes

2 **for** $j = 1$ **to** 0

How do you interpret a loop from 1 up to 0? From context, since a loop from 1 to k executes k tines, we would expect a loop from 1 to 0 to execute zero times. And this is indeed the case. To understand precisely why, we need to deconstruct the behavior of a **for** loop. A loop of the form

1 **for** $j = a$ **to** b
2 *body*

is interpreted as

1 $j = a$
2 **if** $j > b$
3 **goto** line 7
4 *body*
5 $j = j + 1$
6 **goto** line 2

That is, the **for** loop first initializes its iteration variable, then repeats a loop which begins by checking whether to exit the loop, then executing the body of the **for** loop, then incrementing the iteration variable. A loop for j from 1 to 0 will then initialize $j = 1$, then immediately discover that $1 > 0$ and exit without executing the body. In particular, the loop invariant proof above is still valid, but the maintenance step is not used, only the initialization.

Finally, we analyze the running time of InductiveAdd. We can write down the following list of costs:

InductiveAdd(m, n)

1	$sum = m$	*cost*: c_1	*times*: 1
2	**for** $j = 1$ **to** n	*cost*: c_2	*times*: $n + 1$
3	$sum = sum + 1$	*cost*: c_3	*times*: n
4	**return** *sum*	*cost*: c_4	*times*: 1

The total cost is thus $(c_2 + c_3)n + (c_1 + c_2 + c_4) = an + b$, a linear function of n. Note that the running time does not depend on m at all.

The multiplication algorithm InductiveMult is extremely similar to InductiveAdd. The above analysis goes through in exactly the same way with sum replaced by prod, incrementation of *sum* replaced by adding m to *prod*, and the loop invariant replaced by $prod = m \cdot (j-1)$. The running

time analysis is also identical. We now turn to the suggested modification of InductiveMult which uses InductiveAdd for addition. We then have the following cost table:

InductiveMultV2(m, n)

1	$prod = 0$	*cost*: c_1	*times*: 1
2	**for** $j = 1$ **to** n	*cost*: c_2	*times*: $n + 1$
3	$prod =$ InductiveAdd$(prod, m)$	*cost*: $c_3m + c_4$	*times*: n
4	**return** *prod*	*cost*: c_5	*times*: 1

Adding these up, we find that the running time of InductiveMultV2(m, n) is $amn + bn + c$ for some positive constants a, b, c. This is significantly worse than the original InductiveMult, since it increases with m as well as n; if m and n are about the same size—a reasonably common situation—then the running time is roughly proportional to n^2.

Note that, since the running time of InductiveAdd is not symmetric in its arguments, using InductiveAdd$(m, prod)$ instead of InductiveAdd$(prod, m)$ would result in a different running time. This line has non-constant cost $c_3prod+c_4 = c_3m(j-1)+c_4$, so instead of simply multiplying the cost by the number of times the line is executed, we would need to sum the (different) costs of each execution. The reader may verify that the total running time is of the form $amn^2+bmn+cn+d$.

Now we'll write the standard algorithm for adding numbers written in decimal notation, which is much more efficient than the inductive addition algorithm. The standard algorithm works as follows:

(1) Add leading zeroes to both numbers until they are the same length and both begin with a 0. (When applying the algorithm by hand, these zeroes are usually invisible).
(2) Add the last two digits of the two numbers.
(3) If this sum has one digit, write that digit as the last digit of the result.
(4) If it has two digits (is greater than 9), write the second digit as the last digit of the result, and note that an additional 1 must be added in the second-to-last digit ("carried").
(5) Repeat this for the second-to-last digits, possibly including a carried 1.
(6) Repeat for the third-to-last digits, fourth-to-last, etc., until you reach the first digits.
(7) Remove any unnecessary leading zeroes from the result.

We convert this into pseudocode as follows:

```
DecimalAdd(a, b)
 1  m = len(a)
 2  n = len(b)
 3  r = max(m, n) + 1
 4  pad a with r − m zeroes on the left
 5  pad b with r − n zeroes on the left          // Extend the numbers with leading zeroes
 6  create a new array sum[0 .. r − 1]           // This array will hold the result
 7  carry = 0                                    // carry will track whether we are carrying a 1
                                                 // There is no carried 1 before we start adding
 8  for i = r − 1 downto 0                       // Iterate over the digits, starting with the last digit
 9      digitsum = a[i] + b[i] + carry           // Add the digits, and the carried 1 if there is one
10      carry = ⌊digitsum/10⌋                    // Find the next carry
11      sum[i] = digitsum mod 10                 // The next digit is the last digit of the digit sum
12  if sum[0] == 0
13      sum = sum[1 .. r − 1]                    // Delete a possible unnecessary leading zero
14  return sum
```

The proof of correctness for this algorithm is a direct proof. The key step is the **for** loop, for which we need to find a loop invariant. Tracing an example, or considering the design of the algorithm, we see can come up with the following loop invariant:

(Loop Invariant) After each time line 8 is run, $sum[i+1 \,..\, r-1]$ contains the last $r-1-i$ digits of the sum of numbers whose digits are a and b, and $10^{r-1-i}\,carry + \sum_{j=i+1}^{r-1} 10^{r-1-j}\,sum[j]$ is the sum of the last $r-1-i$ digits of those numbers.

For $i = r-1$, "the last 0 digits" means zero, and the sum from r to $r-1$ is interpreted as 0. Proving Initialization for this loop invariant consists of verifying that an empty array contains zero digits and that $10^0 \cdot 0 + 0 = 0$. Proving Maintenance involves noticing that the last k digits of the sum of two numbers are the same as the last k digits of the sum of the last k digits of the two numbers, and verifying that a sum of two digits and a carry can either be at most 9 or between 10 and 19. For Termination, we made sure that the first entries of both a and b were zero, so $carry = 0$ after the last loop, and the loop invariant shows that *sum* is indeed the digits of the sum. The last **if** statement just cleans up a potential leading zero.

To analyze the running time of this algorithm, we note that the loop is executed r times, everything else is executed once, and each line has constant cost, so the total running time bounded by linear functions of $r = \max(\text{len}(a), \text{len}(b)) + 1$. That we don't know which of the options in the if/else is taken in each iteration doesn't matter, since each option has constant, positive cost, so the overall cost is bounded between the minimum and maximum of these.

We can also write the usual algorithm for multiplication by hand. There are (at least) two different "usual" algorithms which we could use here: "long multiplication", in which we multiply the first number by each digit of the second number separately, then shift and add the results, and "lattice multiplication", in which we multiply each pair of digits in a table, then add the diagonals, then perform carries. We'll write pseudocode for long multiplication. The first thing we need to be able to do is to multiply a many-digit number by a single digit. This can be accomplished by multiplying individual digits and doing carries. We then need to add up several many-digit numbers. This can be accomplished by repeated use of DecimalAdd to add them one at a time.

```
LongMultiply(a, b)
 1  m = len(a)
 2  n = len(b)
 3  let prods[0..n-1, 0..m+n-1] be a new array, filled with 0s
    // This array will store the intermediate products
 4  for j = n-1 downto 0                          // Iterate over the digits of b
 5      carry = 0                                 // Initial carry is zero
 6      for i = m-1 downto 0                      // Iterate over the digits of a
 7          digitprod = a[i] · b[j] + carry
 8          carry = ⌊digitprod/10⌋
 9          prods[j][i+j+1] = digitprod mod 10
10      prods[j][j] = carry                       // Put the final carry as the first digit
11  product = prods[0]
12  for j = 1 to n-1
13      product = DecimalAdd(product, prods[j])   // Add up the intermediate products
14  if product[0] == 0
15      product = product[1..r-1]                 // Delete a possible leading zero
16  return product
```

In order to prove the correctness of this algorithm, we would need a loop invariant for each of the **for** loops. Since the loop over i is executed multiple times for different values of j, we technically need a different loop invariant for each instance. However, these n loop invariants will have a very similar form which depends on j in a simple way, so we end up only writing one. By tracing examples, or by considering the intent of the algorithm, we can arrive at the following loop invariants. For the loop starting on line 4:

(Loop Invariant 1) For each $k = j+1, \ldots, n-1$, $prods[k]$ is the digits of the product of $b[k]$ and the number whose digits are in a, followed by $n-1-k$ zeroes, with enough leading zeroes to have length $m+n$. Also, all entries of $prods$ are in $\{0, 1, \ldots, 9\}$.

For the loop starting on line 6:

(Loop Invariant 2) $10^{m+n-1-i-j}\,carry + \sum_{k=0}^{m+n-1} 10^{m+n-1-k} prods[j][k]$ is equal to $b[j]$ times the number whose digits are $a[i+1\,..\,m-1]$. Also, $carry$ and all entries of $prods$ are in $\{0, 1, \ldots, 9\}$

For the loop starting on line 12:

(Loop Invariant 3) $product$ contains the digits of the sum of the numbers represented by $prods[0], \ldots, prods[j-1]$, with possibly some leading zeroes.

The second loop invariant is very similar to the loop invariant for DecimalAdd. The loop starting on line 4 adds a single row to an array, and the loop invariant is a statement about the rows of the array up to that point, so we need only check that the new row is correct, which is very similar to the proof of correctness for DecimalAdd (including the loop invariant—we need to make a loop invariant argument inside the maintenance step of a loop invariant argument, because we have nested loops). The last loop invariant is maintaining a running sum, and is straightforward to verify.

The running time can be calculated in the usual way. Since we have nested loops, lines 7–9 are executed m times each time the inner loop is called, and it is called n times, so they are executed a total of mn times (and line 6 is executed $m(n+1)$ times). We also need to deal with DecimalAdd having potentially non-constant running time. In this case, it is always called with arguments of length $m+n$, so has running time $\Theta(m+n)$. It is run n times. Adding up all of the costs, we find that the total running time is $\Theta(mn+n^2)$.

2. Fun with floor functions

Let $x, y \in \mathbb{R}$. Then $\lfloor x \rfloor + \lfloor y \rfloor$ and $\lceil x \rceil + \lfloor y \rfloor$ are integers, and $\lfloor x \rfloor \le x \le \lceil x \rceil$ and $\lfloor y \rfloor \le y < \lfloor y \rfloor + 1$. Then

$$\lfloor x \rfloor + \lfloor y \rfloor \;\le\; x + y \;<\; \lceil x \rceil + \lfloor y \rfloor + 1,$$

so by Fact 1.4.1(7) and (10) from the lecture notes, we have

$$\lfloor x \rfloor + \lfloor y \rfloor \;\le\; \lfloor x + y \rfloor \;<\; \lceil x \rceil + \lfloor y \rfloor + 1.$$

For integers a and b, the inequalities $a < b$ and $a \le b - 1$, are equivalent, so this gives us the first two inequalities in the desired chain. The last two can be proved very similarly. Alternatively, if we replace x and y by $-x$ and $-y$ in the first two inequalities, use the identity $\lfloor -a \rfloor = -\lceil a \rceil$, and then negate the quantities and reverse the inequalities, we obtain the last two inequalities.

If we take x and y to be integers, then all of the inequalities become equalities. If we take $x = y = 1/3$, then the second and fourth inequalities are strict. If we take $x = y = 2/3$, then the first and third inequalities are strict.

For three numbers x, y, z, by applying the above twice we have

$$\lfloor x \rfloor + \lfloor y \rfloor + \lfloor z \rfloor \;\le\; \lfloor x \rfloor + \lfloor y + z \rfloor \;\le\; \lfloor x + y + z \rfloor \;\le\; \lceil x + y \rceil + \lfloor z \rfloor \;\le\; \lceil x \rceil + \lceil y \rceil + \lfloor z \rfloor,$$

and

$$\lfloor x + y \rfloor + \lceil z \rceil \;\le\; \lfloor x \rfloor + \lfloor y \rfloor + \lceil z \rceil \;\le\; \lceil x + y + z \rceil \;\le\; \lceil x \rceil + \lceil y + z \rceil \;\le\; \lceil x \rceil + \lceil y \rceil + \lceil z \rceil.$$

This is the best that can be done; it is not the case that, for example, $\lfloor x + y + z \rfloor \le \lceil x \rceil + \lfloor y \rfloor + \lfloor z \rfloor$ (take $x = y = z = 2/3$).

The floor function is constant on intervals between integers, and jumps at integers. Thus $\lfloor 2x \rfloor$ has jumps when $2x$ is an integer, i.e. when either x is an integer or $x + \frac{1}{2}$ is an integer. These are the same places that $\lfloor x \rfloor + \lfloor x + \frac{1}{2} \rfloor$ has jumps, so the claimed formula is plausible. This also tells us that we should analyze the identity separately on each interval between an integer and a half-integer. Therefore, let $k = \lfloor x \rfloor$ be an integer, so that $x \in [k, k + 1)$. If $x \in [k, k + \frac{1}{2})$, then $k \le x < x + \frac{1}{2} < k + 1$, so $\lfloor x \rfloor = \lfloor x + \frac{1}{2} \rfloor = k$, and $2k \le 2x < 2k + 1$, so $\lfloor 2x \rfloor = 2k$. If $x \in [k + \frac{1}{2}, k + 1)$, then $k \le x < k + 1 \le x + \frac{1}{2}$, so $\lfloor x \rfloor = k$ and $\lfloor x + \frac{1}{2} \rfloor = k + 1$, and $2k + 1 \le 2x < 2k + 2$, so $\lfloor 2x \rfloor = 2k + 1$. In either case, $\lfloor 2x \rfloor = \lfloor x \rfloor + \lfloor x + \frac{1}{2} \rfloor$.

Following a similar idea, $\lfloor nx \rfloor$ jumps at multiples of $\frac{1}{n}$, which are of the form $k + \frac{i}{n}$ for $k \in \mathbb{Z}$, $i \in \{0, \ldots, n - 1\}$. We therefore guess the formula

$$\lfloor nx \rfloor = \sum_{i=0}^{n-1} \left\lfloor x + \frac{i}{n} \right\rfloor.$$

To prove this, fix an x, and let $k \in \mathbb{Z}$ and $r \in \{0, \ldots, n - 1\}$ be such that $x \in [k + \frac{r}{n}, k + \frac{r+1}{n})$. Then $nk + r \le nx < nk + r + 1$, so $\lfloor nx \rfloor = nk + r$. Also, for $i < n - r$, $k \le x + \frac{i}{n} < k + 1$, while for $n - r \le i \le n - 1$, $k + 1 \le x + \frac{i}{n} < k + 2$. The summation therefore has $n - r$ terms equal to k and r terms equal to $k + 1$, so sums to $(n - r)k + r(k + 1) = nk + r = \lfloor nx \rfloor$.

If $k \in \mathbb{Z}$ and $x \in (k - \frac{1}{2}, k + \frac{1}{2})$, then k is the nearest integer to x, so Round-half-up$(x) = k$. If $x = k - \frac{1}{2}$, then x is rounded up to k, so Round-half-up$(x) = k$. Since the intervals $[k - \frac{1}{2}, k + \frac{1}{2})$,

$k \in \mathbb{Z}$, cover $\mathbb{R}$, we have that Round-half-up$(x) = k$ iff $k - \frac{1}{2} \leq x < k + \frac{1}{2}$. Adding $\frac{1}{2}$ to each term, we find the Round-half-up$(x) = k$ iff $\lfloor x + \frac{1}{2} \rfloor = k$, i.e. Round-half-up$(x) = \lfloor x + \frac{1}{2} \rfloor$. The proof for round-half-down is essentially identical.

For round-half-even, we again look at where the functions change value. The first term $\left\lfloor \frac{x}{2} + \frac{1}{4} \right\rfloor$ jumps when $\frac{x}{2} + \frac{1}{4}$ is an integer, i.e. when $x = 2k - \frac{1}{2}$ for some integer k. Similarly, the second term jumps when $x = 2k + \frac{1}{2}$ for some $k \in \mathbb{Z}$. If $2k - \frac{1}{2} \leq x \leq 2k + \frac{1}{2}$, then either $2k$ is the closest integer to x, or x is a half-integer and $2k$ is the closest even integer, so Round-half-even$(x) = 2k$. We also have

$$k \leq \frac{x}{2} + \frac{1}{4} \leq k + \frac{1}{2} < k + 1 \quad \text{and} \quad k - 1 < k - \frac{1}{2} \leq \frac{x}{2} - \frac{1}{4} \leq k,$$

so $\left\lfloor \frac{x}{2} + \frac{1}{4} \right\rfloor + \left\lceil \frac{x}{2} - \frac{1}{4} \right\rceil = k + k = 2k$. If instead $x \in (2k + \frac{1}{2}, 2k + \frac{3}{2})$, then Round-half-even$(x) = 2k + 1$ and

$$k < k + \frac{1}{2} < \frac{x}{2} + \frac{1}{4} < k + 1 \quad \text{and} \quad k < \frac{x}{2} - \frac{1}{4} < k + \frac{1}{2} < k + 1,$$

so $\left\lfloor \frac{x}{2} + \frac{1}{4} \right\rfloor + \left\lceil \frac{x}{2} - \frac{1}{4} \right\rceil = k + (k + 1) = 2k + 1$. These two cases cover all real numbers, so the identity is proved.

We can show that truncation cannot be written in a similar way with only rational constants to add and multiply by. Let $f(x)$ be such an expression. If we choose a number $n > 0$ which is divisible by the denominators of all the constants involved, then $f(x + n) = f(x) + n$ for all x. But truncation does not have this property, since $-\frac{1}{2}$ truncates to 0 but $n - \frac{1}{2}$ truncates to $n - 1$.

It seems that allowing irrational constants would not allow us to do better, since in order to avoid a similar argument or having a jump in the wrong place, the irrationals would need to somehow "conspire" to act in a very regular way except at a single point. Proving the nonexistence of such conspiracies, however, is a rather difficult endeavor—it is a major difficulty in, for example, transcendental number theory.

If we allow the sign function, then we can fairly easily write truncation as $\text{sign}(x) \cdot \left\lfloor |x| \right\rfloor$.