Leone Learning Systems, Inc.
Wonder. Create. Grow.

Leone Learning Systems, Inc.
237 Custer Ave
Evanston, IL 60202
Email tj@leonelearningsystems.com

Phone 847 951 0127
Fax 847 733 8812

Magnitude and direction

TJ Leone

December 2004

Introduction

The movement made by the commands `fd` and `bk` is composed of two components. One component is the *direction* (either forward or back), and the other component is the *magnitude* (the number of turtle steps taken).

If we don't change the heading of the turtle (with, for example, `seth`, `rt`, or `lt`), then the two moves `fd 50` and `fd 20` are equivalent to (have the same direction and magnitude as) `fd 70`.

In this unit, we will discuss new ways to manipulate instructions lists like `[fd 50]` and `[bk 20]` as we develop methods for adding and subtracting turtle moves. We will then look at practical applications for this kind of addition and subtraction and see how it relates to addition and subtraction of positive and negative numbers.

List Operations

In the last unit, we saw that words can be combined to form lists which can be used as inputs to commands or operations. The following are examples of commands that accept lists as input. Try them out on your computer and make sure you understand what they do:

```
show [How are you?]
print [I'm fine]
if greater? 5 0 [rt 50]
repeat 8 [fd 90 rt 45]
```

In this section, we will discuss some special operations that take lists as input or produce a list as output.

First and Last

Here are some examples using the operations `first` and `last`. Since `first` and `last` are operations, they must be used as inputs to some other command or operation. In the examples below, I'll use them as inputs to the `show` command.

```
show first [How are you?]
show first [fd 50]
show last [fd 50]
show last [red yellow blue]
```

Try these instructions out on your computer. What does the `first` operation do? What does the `last` operation do?

Sentence

In the last section, you should have noticed that `first` and `last` both take a single input (one list is considered a single input even if it has lots of words in it). The `first` operation returned the first word in the list. The `last` operation returned the last word in the list. List operations like `first` and `last` that remove some part of a list are called *selectors*.

There are also list operations that put inputs together to create lists. These list operations are called *constructors*. One example of a constructor is the `sentence` operation. Here are a few examples for you to try on your computer:

```
show sentence "Hello "there
show sentence "red [blue yellow]
show se "fd 100
show (sentence "algebra "is "fun)
```

List Operations (continued)

Notice that you can use `se` as an abbreviation for `sentence`. Also notice the parentheses that were used in the last example. Let's try that example without parentheses:

```
show sentence "algebra "is "fun
[algebra is]
You don't say what to do with fun
```

What's going on here? Normally, `sentence` (or `se`) expects exactly two inputs. In the example above, Logo found two inputs for sentence (`"algebra` and `"is`) and gave them to sentence, which happily mushed them together into one list. The next thing Logo saw was the word `"fun`, so Logo looks for a command or operation that needs an input. However, at this point, there are no commands or operations that need inputs (`sentence` already has its two inputs, and the output of sentence is all the input needed by `show`). So Logo sends the error message.

When Logo sees an opening parenthesis before `sentence` (or `se`), Logo knows that the inputs to `sentence` (or `se`) are whatever follows se up to the closing parenthesis.

What's happening here?

```
show (se "apple)
[apple]
show se "apple
not enough inputs to se
```

Remember, under normal conditions, `se` expects *exactly* two inputs. In general, parentheses tell Logo, "Keep the procedure inside these parentheses together with the inputs inside these parentheses".

Copyright © 2004 Leone Learning Systems, Inc.

Run

In the last unit, we talked about a special kind of list called an *instruction list*.

An instruction list is just a list with instructions in it. Here are some examples of procedures that take instruction lists as inputs:

```
if greater? xcor 100 [rt 160 fd 40]
repeat 5 [fd 100 rt 144]
for [i 4 16 2] [show :i]
ifelse equal? heading 0 [seth 180] [seth 90]
```

All the lists above are instruction lists except for one. Which one is it? Try to find it before you read on.

In this section, we introduce a new procedure called `run` that takes an instruction list as an input. Try these:

```
run [fd 50]
show run [sum 4 7]
run [cs pd fd 100 rt 120 fd 100 rt 120 fd 100 rt 120]
```

The `run` procedure just executes whatever's in the input instruction list. The run procedure has a special property—*it can be used as a command or as an operation, depending on the input* (the same is true for `if` and `ifelse`). If the first item in the input list is the name of a command (like `fd` or `cs`), then `run` behaves like a command. If the first item in the input list is an operation (like `sum`), then `run` behaves like an operation.

But so what? The examples above are pretty boring. It would be easier to do these:

```
fd 50
show sum 4 7
cs pd fd 100 rt 120 fd 100 rt 120 fd 100 rt 120
```

What's the big deal about `run`? Actually, the above examples were just meant to show you how `run` works. They are not necessary or even good uses of `run`. However, there are many cases where the `run` procedure can be useful. Later, we'll see a few of them.

Copyright © 2004 Leone Learning Systems, Inc.

http://www.leonelearningsystems.com

Adding with fd and bk

OK, try this:

```
? cs
? show ycor
0
? fd 50
? show ycor
50
? bk 30
? show ycor
20
```

I put a question mark in front of the instructions so you can tell the instructions apart from the outputs.

The `ycor` operation tells you how high the turtle is, measuring in turtle steps from the center of the screen. When you execute the `cs` command, you put the turtle in the center of the screen, so the `ycor` is 0 (so is the `xcor`, which tells you how far right the turtle is from the center of the screen). The `cs` command also makes sure that the turtle is pointed straight up. In Logo, this is done by setting the turtle's heading to 0 degrees.

After the turtle move `fd 50`, its `ycor` becomes 50. When it moves `bk 30`, it is 30 steps lower than it was before, so its `ycor` is now 50 – 30 = 20.

So a move of `fd 50` followed by a move of `bk 30` gives the same result as a move of `fd 20`. It might be useful to have an operation like the `sum` operation that adds `fd` and `bk` moves instead of adding numbers. Let's think a little more about how this operation should work. Let's call the operation `sum.v`. For the example above, we like to be able to do something like this:

```
? sum.v [fd 50] [bk 30]
fd 20
```

Adding with fd and bk (continued)

What would we want our outputs to be for the examples below? Try to figure out good answers, then use `cs`, `fd`, `bk`, and `ycor` to verify your answers. You'll notice that when the turtle is below middle of the screen, the number for the `ycor` will be preceded by a minus sign (e.g., -50, -20). The minus sign is basically telling you that you have moved `bk` from the middle of the screen, so, for example, -30 indicates a move of `bk 30` from the center of the screen. As we noticed above, a `ycor` of 30 without a minus sign indicates a move of `fd 30` from the center of the screen.

```
sum.v [fd 50] [fd 30]
sum.v [bk 50] [fd 30]
sum.v [bk 50] [bk 30]
```

Try figuring out the answers yourself before looking at the solutions on the next page. Write your answers on a separate sheet of paper or in the space below.

Adding with fd and bk (continued)

Here are the solutions to the problems posed on the previous page:

```
sum.v [fd 50] [fd 30]
```

We can get the solution by executing the following:

```
? cs
? show ycor
0
? fd 50
? show ycor
50
? fd 30
? show ycor
80
```

A move of `fd 50` followed by a move `fd 30` is equivalent to a move of `fd 80`.

```
sum.v [bk 50] [fd 30]
```

We can get the solution by executing the following:

```
? cs
? show ycor
0
? bk 50
? show ycor
-50
? fd 30
? show ycor
-20
```

A move of `bk 50` followed by a move `fd 30` is equivalent to a move of `bk 20`.

Adding with fd and bk (continued)

```
sum.v [bk 50] [bk 30]
```

We can get the solution by executing the following:

```
? cs
? show ycor
0
? bk 50
? show ycor
-50
? bk 30
? show ycor
-80
```

A move of `bk 50` followed by a move `bk 30` is equivalent to a move of `bk 80`.

Here's a summary of the results we want:

```
? sum.v [fd 50] [bk 30]
fd 20
? sum.v [fd 50] [fd 30]
fd 80
? sum.v [bk 50] [fd 30]
bk 20
? sum.v [bk 50] [bk 30]
bk 80
```

Solved Problems

1. *Problem:* Create a procedure `sum.v` that works as described in the section above.

   Solution:

   The first thing we need to is to bring up our editor so we can type in the procedure. Click on your Edall button or execute the command `edall`.

   We know the name of our procedure, and we know that it will have two inputs (two instruction lists that contain either `fd` or `bk` instructions). So we already have a start:

```
to sum.v :v1 :v2
end
```

   When we execute, for example, `sum.v [fd 50] [bk 30]`, the value in `:v1` will be `[fd 50]` and the value in `:v2` will be `[bk 30]`. So far, so good. Now we need to get the procedure to figure out what the sum of the two moves should be and output that sum.

   Let's review how we summed two `fd` or `bk` instructions in the last section. First, we cleared the screen. Next, we executed the instructions for moving the turtle forward or back. After that, we checked the `ycor`. If it was bigger than zero, we output a list with `"fd` and the ycor. Otherwise, we output a list with `"bk` and the `ycor` without the minus sign.

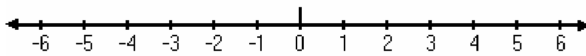
```
to sum.v :v1 :v2
cs
run :v1
run :v2
ifelse ycor > 0 [
     op se "fd ycor
] [
     op se "bk abs ycor
]
end
```

The `abs` operation takes a number as input and outputs the same number without a minus sign. If the input number had no minus sign to begin with, the same number is output. Here are examples:

```
show abs -405
405
show abs 0
0
show abs -7
7
show abs 7
7
```

The procedure name `abs` stands for "absolute value". In algebraic notation, the absolute value of *a* is denoted by $|a|$. The absolute value is often referred to as the distance of *a* from 0 on the number line.

We could imagine moving `fd` and `bk` along a number line like the one below:



The turtle's heading would have to be 90. Execute `seth 90` to set the turtle's heading to 90 and see what that looks like. You can always set it back to zero with `seth 0`. The `cs` command always sets the turtle's heading to 0 along with clearing the screen and putting the turtle in the middle of screen.

If we start at 0 (with heading 90) and move `fd 6`, we are six units away from zero at the 6 marker. If we start at 0 (with heading 90) and move `bk 6`, we are six units away from zero at the -6 marker. So $|6| = 6,$ and $|-6| = 6.$

Notice that we can move `fd` and `bk` along a horizontal line just as we can move fd and bk along a vertical line. In fact, we can move fd and bk along any straight line that can be drawn on our turtle screen.

This makes our `sum.v` solution problematic. What if we're drawing some shape and we need to calculate our `sum.v` but we don't want to change our turtle's heading or position?

Copyright © 2004 Leone Learning Systems, Inc.

Solved Problems (continued)

Here's an example that shows how our current version of `sum.v` can cause trouble:

```
fd 100
rt 120
fd run sum.v [fd 150] [bk 50]
rt 120
fd 100
rt 120
```

The output to `sum.v [fd 150] [bk 50]` is `[fd 100]`, so we should have drawn a triangle. What happened?

Since we move the turtle around in `sum.v`, we mess up our drawing. In general, it's a bad idea to move the turtle inside an operation. You should only move the turtle with commands.

From now on, we will refer to our `sum.v` inputs as *vectors*. A vector is a quantity that has *magnitude* and *direction*. For the vector `[fd 50]`, the magnitude is 50 and the direction is `fd`. Vectors can be described in much more general ways, for example, with directions that correspond to turtle headings, but we'll stick with `fd` and `bk` descriptions in this course.

We can write procedures to extract the direction and magnitude from a vector like this:

```
to direction :vector
op first :vector
end

to magnitude :vector
op last :vector
end
```

After you've written the procedures, try executing the following:

```
show direction [fd 50]
show magnitude [bk 80]
```

Solved Problems (continued)

How can we write a `sum.v` that doesn't move the turtle? A good general strategy for puzzling out problems like this is to start with one or two easy cases and then move on to include more cases. Our `sum.v` would be easy to write if the two inputs were both `fd` moves. If we move `fd 50` and then `fd 30`, we've gone `fd 80`. A sum.v procedure that only works for this case would look like this:

```
to sum.v :v1 :v2
op (se "fd sum magnitude :v1 magnitude :v2)
end
```

The idea is to keep the `"fd` direction and just add the two magnitudes.

However, the case when both vector inputs have a `fd` direction is only one case. The direction of either of the vectors could also be `bk`. So we need to check to make sure that both vectors have a direction of `fd` before we output anything:

```
to sum.v :v1 :v2
if (and equal? "fd direction :v1 equal? "fd direction :v2)
[
      op (se "fd sum magnitude :v1 magnitude :v2)
]
op []
end
```

Now we output the correct answer if both inputs are vectors with a `fd` direction. Otherwise, we output an empty list. Now let's fill in the other cases.

What if both the vectors `:v1` and `:v2` have a direction of `bk`? If we go back 40 and then go back 80, we've gone back a total of 120. This is like the case when both vectors have direction `fd`, because in both cases, we add the magnitudes of the vectors. However, when both vectors have direction `bk`, the sum of the vectors has a direction `bk`. We can cover both the `fd-fd` and `bk-bk` cases (both vectors with direction `fd` or both vectors with direction `bk`) like this:

```
to sum.v :v1 :v2
if equal? direction :v1 direction :v2 [
      op (se direction :v1 sum magnitude :v1 magnitude :v2)
]
op []
end
```

Solved Problems (continued)

How do we know this will work for both cases? Our test checks to see if the direction of :v1 is the same as the direction of :v2. If the directions are the same, then we have either fd-fd or bk-bk. If we have fd-fd, then the direction of the sum is fd, which is the same as the direction of either input vector. If we have bk-bk, the direction of the sum is bk, which again is the same as the direction of either input vector.

In this line from the code above,

```
op (se direction :v1 sum magnitude :v1 magnitude :v2)
```

I arbitrarily chose direction :v1 (I also could have used direction :v2, since both vectors have the same direction) to give the direction of the sum. The magnitude of the sum is just the sum of the magnitudes of :v1 and :v2.

What if the input vectors don't have the same direction? Let's consider a fd-bk pair. Previously, we gave an example of the result we wanted for sum.v with one of these pairs:

```
? sum.v [fd 50] [bk 30]
fd 20
```

One way to code this case would be:

```
(1)  op (se direction :v1 difference magnitude :v1 magnitude :v2)
```

But consider this case:

```
? sum.v [fd 30] [bk 50]
bk 20
```

In this case, we should code like this:

```
(2)  op (se direction :v1 difference magnitude :v1 magnitude :v2)
```

So it looks like there are two cases for fd-bk pairs. In one case, the magnitude of the fd vector is greater than the magnitude of the bk vector. In that case, we've gone farther forward than back, so our sum is a fd vector as in (1) above.

In the other case, the magnitude of the bk vector is greater than the magnitude of the fd vector, so we've gone farther back than forward and need to use (2) above.

Copyright © 2004 Leone Learning Systems, Inc.

Solved Problems (continued)

With `if`, we can do a test to see which vector has the greater magnitude and then output the appropriate result:

```
to sum.v :v1 :v2
if equal? direction :v1 direction :v2 [
      op se direction :v1 sum magnitude :v1 magnitude :v2
]
if greater? magnitude :v1 magnitude :v2 [
      op se direction :v1 difference magnitude :v1 magnitude :v2
]
op se direction :v2 difference magnitude :v2 magnitude :v1
end
```

Let's see how this works. One of the most important things to understand here is that *as soon as Logo sees the op command, it outputs a value and goes no further*.

If the two vectors have the same direction, we output a vector with that direction and the sum of the magnitudes and we're done. Logo will not try to execute any of the instructions on the lines that follow.

So the first if statement takes care of `fd-fd` and `bk-bk` pairs of input. For other pairs, we go to the next line in the procedure, which compares the magnitudes of the vector. If you read closely and think about it, you will see that the remaining lines cover the cases of both `fd-bk` and `bk-fd`.

If `:v1` has a greater magnitude, then we give the output vector `:v1`'s direction and subtract `:v2`'s magnitude from `:v1`'s magnitude to get the magnitude for the output vector. Then we output the vector and we're done. Otherwise, we know we have vectors with two different directions with the magnitude of `:v2` greater than the magnitude of `:v1` (that's the only case left).

In the last case, the output gets the direction of `:v2` and we subtract the magnitude of `:v1` from the magnitude of `:v2` to get the magnitude for the output vector.

Copyright © 2004 Leone Learning Systems, Inc.

Solved Problems (continued)

2. *Problem:* Write a predicate `vector?` to test to see if an input is a vector. Use the predicate to check inputs in `sum.v` and output an error message if it finds an input that is not a vector.

   Solution:

   For an input to be a vector (of the kind described on page 11), (a) it must be a list, (b) it must have exactly two items, (c) its first item must be either `"fd` or `"bk`, and (d) its second item must be a number, and (e) that number must be zero or greater.

   Now we just need to put the conditions into code:

```
to vector? :v
if not list? :v [op "false]
if not equal? 2 count :v [op "false]
if not (or equal? first :v "fd equal? first :v "bk) [op "false]
if not number? last :v [op "false]
if less? :v 0 [op "false]
op "true
end
```

   The operator `count` outputs the number of items in a list.

   The procedure will output the word `"false` and stop if any of the tests fail. If none of the tests fail, the line op “true is executed, and the procedure outputs `"true`.

Copyright © 2004 Leone Learning Systems, Inc.

Solved Problems (continued)

We can use this predicate `vector?` in our `sum.v` procedure like this:

```
to sum.v :v1 :v2
if not vector? :v1 [
      show (se [sum.v doesn't like] :v1 [as input])
      op []
]
if not vector? :v2 [
      show (se [sum.v doesn't like] :v2 [as input])
      op []
]
if equal? direction :v1 direction :v2 [
      op se direction :v1 sum magnitude :v1 magnitude :v2
]
if greater? magnitude :v1 magnitude :v2 [
      op se direction :v1 difference magnitude :v1 magnitude :v2
]
op se direction :v2 difference magnitude :v2 magnitude :v1
end
```

3. *Problem:* Use integer addition to express the addition of the following vectors:

(a) `? sum.v [fd 40] [bk 20]`
`[fd 20]`

(b) `? sum.v [fd 10] [bk 50]`
`[bk 40]`

(c) `? sum.v [bk 20] [bk 30]`
`[bk 50]`

Solution:

We can think of `fd` as a positive direction and `bk` as a negative direction. So, for example, we could write `[fd 40]` as 40 and `[bk 20]` as -20.

(a) $40 + (-20) = 20$
(b) $10 + (-50) = -40$
(c) $-20 + (-30) = -50$

Copyright © 2004 Leone Learning Systems, Inc.

Solved Problems (continued)

4. *Problem:* Solve the following word problems and explain how they are related.

(a) The top of Mt. Tremblant is 3,150 feet above sea level. The top of Mt. Jacques Cartier is 1,277 feet above sea level. How many feet higher is the top of Mt. Tremblant than the top of Mt. Jacques Cartier? Show your work.

(b) The lowest point of the St. Lawrence River is 294 feet below sea level. The top of Mt. Jacques Cartier is 1,277 feet above sea level. How many feet higher is the top of Mt. Jacques Cartier than the lowest point of the St. Lawrence River? Show your work.

Solution:

(d) We can think of this problem in terms of a turtle. Suppose the turtle starts with a `ycor` of 1,277, and after moving forward *y* steps reaches a `ycor` of 3,150. Then $y = 3{,}150 - 1{,}277 = 1{,}873$. So Mt. Tremblant is 1,873 feet higher than Mt. Jacques Cartier.

(e) In this case, we start at 294 feet *below* sea level. If we were doing this problem with a turtle, we could use `cs` followed by `bk 294`, or just directly set the `ycor` to -294 with `sety -294`. To get up to a ycor of 1,277, we would have to first go `fd 294` to get to `ycor` 0 and then go `fd 1277`, to get the rest of the way. Altogether we would be going forward by $294 + 1{,}277 = 1{,}571$. So the top of Mt. Jacques Cartier is 1,571 feet higher than the lowest point of the St. Lawrence River.

Both of these problems require that we find differences in height.

Supplementary Problems

1. *Problem*: Write a Logo operation `difference.v` that outputs the difference between two vectors. Explain how it works.

Solution:

```
to difference.v :v1 :v2
if not equal? direction :v1 direction :v2 [
    op (se direction :v1 sum magnitude :v1 magnitude :v2)
]
if greater? magnitude :v1 magnitude :v2 [
    op (se direction :v1 difference magnitude :v1 magnitude :v2)
]
op (se opposite.d direction :v1 difference magnitude :v2 magnitude :v1)
end
```

We can use Solved Problem 4 to analyze subtraction of vectors. Consider first the case of two mountain peaks in Solved Problem 4a. We can imagine two vectors that start at sea level and end at one of the two mountain peaks. When we want to find the difference between two quantities like 3150 and 1277, we ask, "1277 plus what equals 3150?" Another way to ask this is, "What should be the second input to `sum` to make it output 3150, if the first input is 1277?"

When finding the difference between vectors `[fd 3150]` and `[fd 1277]`, we ask "What should be the second input to `sum.v` to make it output `[fd 3150]`, if the first input is `[fd 1277]`?" If we've already gone forward 1277, we need to forward 1873 to make fd 3150.

For Solved Problem 4a, our difference operation should look like this:

```
? show difference.v [fd 3150] [fd 1277]
[fd 1873]
```

We can check this with:

```
? show sum.v [fd 1277] [fd 1873]
[fd 3150]
```

In general, if the vectors `:v1` and `:v2` have the same direction and the magnitude of `:v1` is greater than the magnitude of `:v2`, the difference of the vectors has the same direction as the vectors and its magnitude is the magnitude of `:v1` minus the magnitude of `:v2`.

Supplementary Problems (continued)

But suppose we wanted to subtract the height of the higher peak from the height of the lower peak, to get from the top of Mt. Jacques Cartier from the top of Mt. Tremblant? Then the question would be "What should be the second input to `sum.v` to make it output `[fd 1277]`, if the first input is `[fd 3150]`?" In this case, if we've already gone forward 3150, we need to go back 1873 to make `fd 1277`:

```
? show difference.v [fd 1277] [fd 3150]
[bk 1873]
```

In this case, `[fd 1277]` is `:v1` and `[fd 3150]` is `:v2`. The magnitude of `:v2` is greater than the magnitude of `:v1`. The ouput gets the magnitude of `:v2` minus the magnitude of `:v1` and the opposite direction of both `:v1` and `:v2`.

Something similar happens looking at points that are both below sea level (i.e., `bk` vectors).

Solved Problem 4b illustrates the case in which the input vectors have opposite directions. In this case, we sum magnitudes and take the direction of the vector with the larger magnitude:

```
? show difference.v [fd 1277] [bk 294]
[fd 1571]
```

In this case, `:v1` is `[fd 1277]` and `:v2` is `[bk 294]`. The directions of `:v1` and `:v2` are different. The output has the direction of `:v1`. Its magnitude is the sum of the magnitudes of `:v1` and `:v2`. This works for `bk-fd` pairs as well as `fd-bk` pairs.

Copyright © 2004 Leone Learning Systems, Inc.

Supplementary Problems (continued)

2. *Problem:* Write a Logo operation `minus.v` that outputs a vector with the same magnitude but opposite direction of its input.

   Solution:

   This operation is easier to write if we first write a helper operation `opposite.d` and use it to change vector direction `minus.v`:

```
to opposite.d :d
if equal? :d "fd [op "bk]
op "fd
end

to minus.v :v
op (se opposite.d first :v last.v)
end
```

3. *Problem:* Use integer subtraction to express the subtraction of the following vectors:

   (a) `? difference.v [fd 40] [bk 20]`
   `[fd 60]`
   (b) `? difference.v [fd 10] [bk 50]`
   `[fd 60]`
   (c) `? difference.v [bk 20] [fd 30]`
   `[bk 50]`
   (d) `? difference.v [bk 20] [bk 30]`
   `[fd 10]`
   (e) `? difference.v [fd 60] [fd 20]`
   `[fd 40]`

   Solution:

   (a) 40 - (-20) = 60
   (b) 10 - (-50) = 60
   (c) -20 - 30 = -50
   (d) -20 - (-30) = 10
   (e) 60 – 20 = 40

Copyright © 2004 Leone Learning Systems, Inc.

Supplementary Problems (continued)

4. *Problem:* Solve the problems below.

(a) Monday morning, it was 25 degrees outside. By the afternoon, it was 32 degrees. What was the change in temperature?

(b) Tuesday afternoon, it was 25 degrees. Overnight, the temperature reached 7 degrees. What was the change in temperature?

(c) Wednesday morning, it was 5 degrees below zero. In the afternoon, the temperature rose 10 degrees. What was the temperature in the afternoon?

(d) Thursday morning, it was 5 degrees below zero. In the afternoon, the temperature fell by 4 degrees. What was the temperature in the afternoon?

Solution:

(a) $32 - 25 = 7$. The change in temperature was a rise of 7 degrees.

(b) $7 - 25 = -18$. The change in temperature was a drop of 18 degrees.

(c) $-5 + 10 = 5$. The temperature in the afternoon was 5 degrees.

(d) $-5 - 4 = -9$. The temperature in the afternoon was 9 degrees below zero.

Copyright © 2004 Leone Learning Systems, Inc.

The Author

TJ Leone owns and operates Leone Learning Systems, Inc., a private corporation that offers tutoring and educational software. He has a BA in Math and an MS in Computer Science, both from the City College of New York. He spent two years in graduate studies in education and computer science at Northwestern University, and six years developing educational software there. He is a former Montessori teacher and currently teaches gifted children on a part time basis at the Center for Talent Development at Northwestern University in addition to his tutoring and software development work. His web site is http://www.leonelearningsystems.com.

Copyright © 2004 Leone Learning Systems, Inc.