This document lays out the learning objectives for Chapter 1. Note that you generally will not meet all learning objectives on the first reading. Instead, more sophisticated learning objectives will require a second reading to fully understand. This document can also serve as an excellent outline for your notes on this chapter.

For each learning objective, the key verb (what you should be able to do) is marked/colored based on the level of Bloom's taxonomy at which that objective lies. For example, if you should be able to **define** at term, that is at the "remember" level. On the other hand if you should be able to **devise an algorithm** that is at the "create" level.

In particular, you will see learning objectives at the 6 levels of Bloom's taxonomy:

1. **Remember**
2. **Undertand**
3. **Apply**
4. **Analyze**
5. **Evaluate**
6. **Create**

Note that the gray backgrounds for some of these do not have any special meaning—those colors just do not show up well on a white background (and other colors don't show up well on a gray background).

1 First Reading: Understand Key Ideas

On your first reading, the goal is to get the big ideas, and build the mental scaffolding in which to hold the details on your second reading—the learning objectives for your first reading are generally **Remember** and **Understand** objectives. If you are missing a few details (but not completely lost), it is ok to move ahead, and come back to them the next time through—you might ask questions on the course forums too! Note you should **sleep at least one night between the first and second reading** to give your brain time to process the information.

- **What Is Programming?**
  - **Define** *metacognition.*
  - **Explain** metacognition's relation to programming.
  - **Explain** why planning before programming is important.
  - **Define** *algorithm.*
  - **Explain** the difference between a *problem* and a *class of problems.*
  - **Define** *parameter.*
- **How to Write a Program**

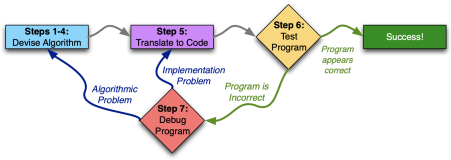


Figure 1.1: A high level view of writing a program

- **Explain** the high-level process for writing a program.
- **Explain** the difference between testing and debugging.
- **State** the amount of testing needed to ensure correct code.

- **Algorithms**
  - **Explain** how you know if an algorithm written in English is done well.
  - **State** the only type of information a computer can work with.
  - (Note: work through the example on the second reading)
  - **Explain** why some courses have an activity on writing instructions to make a PB&J sandwich.
  - **List** what parts of algorithm creation this sandwhich activity does not cover and **explain** those shortcomings.
  - **Name** one mechanism that guards against certain kinds of nonsense inputs and **explain** what types of nonsense inputs it protects against.
  - **Define** *error cases*.
  - **Explain** why starting by trying to write code is the wrong approach to designing an algorith.

- **Step 1: Work an Example Yourself**
  - **Name** the first step to write an algorithm.
  - **Explain** why you might need to do this step multiple times.
  - **Explain** what problem might arise for simple cases.
  - **Define** *ill-specified problem*.
  - **Explain** what you could do if you an ill-specified problem.
  - **Define** *domain knowledge*.
  - **Explain** what you could do if you lack domain knowledge.

- **Step 2: Write Down What You Just Did**
  - **Name** the second step to write an algorithm.
  - **Explain** why you might need to do this step multiple times.
  - **Explain** what the difficult part of Step 2 is.
  - **Relate** this step to the earlier PB&J sandwich example.

- **Step 3: Generalize Your Steps**

  - **Name** the third step to write an algorithm.
  - **List** the two primary activities in this step:
    *
    *
  - **Explain** stratgies for if I get stuck on Step 3.

- **Step 4: Test Your Algorithm**
  - **Name** Step 4 of creating an algorithm.
  - **Explain** why you want to use different values for the parameters (from the ones I used in Step 1).
  - **List** two common mistakes in Step 3 that we might detect in Step 4:
    *
    *
  - **Explain** how you might fix a problem in your algoritm if you find one in Step 4.
  - **Explainy** why you only need minimal "pencil and paper" testing.

- **Some Examples**
  - **Explain** why it is important to understand there are *many* correct algorithms for a problem.
  - **Name** some real world problems where finding the closest point is useful.
  - Our core learning objectives are around developing algorithms, such as the ones in these examples—we will revisit those on the second reading. For now, we recommend writing the following points in your notes for each algorithm:
    * What were your key "takeaway" points from the example (what did you learn)?
    * What things were you unclear about from the example?
    * What key strategies for Step 3 (Generalize) were used in the example?

- **Next Steps**
  - **State** why novice programmers may try to skip a step-by-step approach.
  - **Explain** the disadvantages of skipping steps in The Seven Steps.
  - **Explain** what will change about your programming process as you get more experience.
  - **Explain** why reading code is an important skill, both in learning, and in industry.

2 Second Reading: Work Through Examples

On your second reading, read more carefully, ensuring you have a good understanding of the material. When you encounter an example, work the example in your notes. If you can't work it, that shows gaps in your understanding from your first reading—fill them in, and retry it. If there are things you don't understand in your second reading, ask questions about them! Note that the learning objectives for this reading are higher level (**Apply**, **Analyze**, **Evaluate**, and **Create**).

- **Execute** by hand and algorithm written in English. Note that we have reproduced the algorithm from 1.2 below for you and provided an outline for you to do it in your notes.

```
Given a non-negative integer N:
1: Make a variable called x, and set it equal to (N+2).
2: Count from 0 to N (include both ends),
   and for each number (call it i) that you count:
3:    Write down the value of (x * i).
4:    Update x to be equal to (x + i * N).
5: When you finish counting, write down the value of x.
```

When I execute it by hand for N=2, I do the following steps:

Before line:
N=

Before line:
N=
x=

Before line:
N=
x=
i=

On line 3, I write this output:

Before line:
N=
x=
i=

After line 4, before I count the next number:
N=
x=
i=

Now, I count my next number for i, which is

Before line:
N=
x=
i=

On line 3, I write this output:

Before line:
N=
x=

i=

After line 4, before I count the next number:
N=
x=
i=

Now, I count my next number for i, which is

Before line:
N=
x=
i=

On line 3, I write this output:

Before line:
N=
x=
i=

After line 4, before I count the next number: N=
x=
i=

The next number I count would be 3, but I am only counting to 2, so I am done counting, so now I go to Before line:
N=
x=

On line 5, I write this output:

Now, I am done.

- **Compare and constrast** steps in an algorithm to find repetition.
- **Transform** steps in an algorithm to make repetition clear. **Note:** the step from the algorithm in 1.3 where we transform

```
Multiply x by 3.
  You get 9.
Multiply x by 9.
  You get 27.
Multiply x by 27.
  You get 81.
81 is your answer.
```

into

```
Start with n = 3.
n = Multiply x by n.
n = Multiply x by n.
n = Multiply x by n.
n is your answer.
```

is a great example of what you should be able to do for these two learning objectives:

- **Transform** repetitive steps into a couting patter. **Note:** the step where we transform the above algorithm into the one below is a great example of this skill

```
Start with n = 3.
Count up from 1 to y-1 (inclusive), and for each number you count,
  n = Multiply x by n.
n is your answer.
```

- **Determine** what values in your algorithm depend on parameters, and **develop** a formula which expresses those relationships. **Note:** the step where we transform the above algorithm into the one below is a great example of this skill.

```
Start with n = x.
Count up from 1 to y-1 (inclusive), and for each number you count,
  n = Multiply x by n.
n is your answer.
```

3 Practicing with Key Learning Objectives

The following four learning objectives are the most key to this chapter.

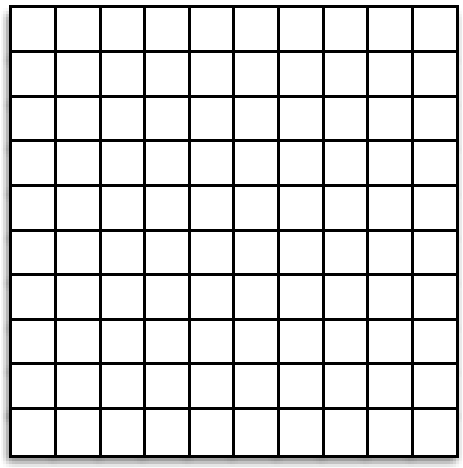
- **Compare and constrast** steps in an algorithm to find repetition.
- **Transform** steps in an algorithm to make repetition clear.
- **Transform** repetitive steps into a couting patter.
- **Determine** what values in your algorithm depend on parameters, and **develop** a formula which expresses those relationships.

We strongly recommend that on (or after) your second reading, you work through the examples in 1.7 to practice these skills. We give you a rough outline for what you might put in your notes as you practice on these.

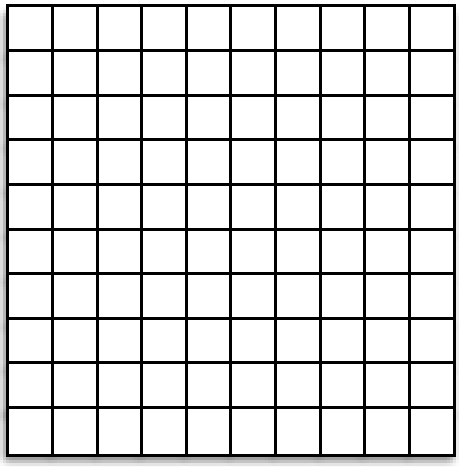
- Working the example in 1.7.1:
  - Doing Step 1 for N=5:
  - Doing Step 2 for N=5:
  - Doing Step 3:

        * How many numbers are there for each value of N?
        * What is the starting number for each value of N?
        * What is the formula for the ith number for each value of N?
        * My generalized algorithm is:
    - Doing Step 4 to test my algorithm with N=3:

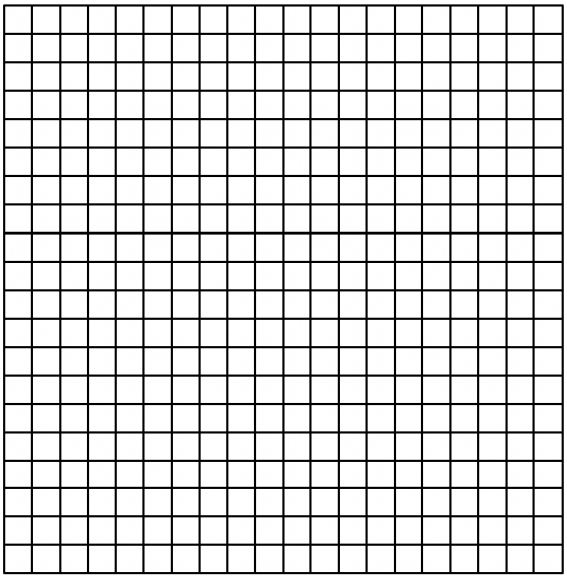
- Working the example in 1.7.2:
    - Doing Step 1 for N=2:



    - Doing Step 2 for N=2:
    - Doing Step 3:
        * How many columns do you count across?
        * What is the starting y-coordinate in each column?
        * How many squares are in the i^{th} column?
        * How do you determine if a square is blue or red?
            · Does this formula depend on N?
        * My generalized algorithm is:
    - Doing Step 4 to test my algorithm with N=4:

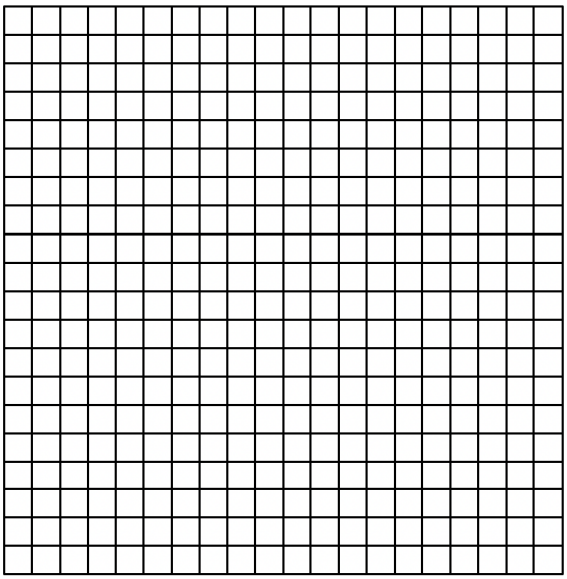


- Working the example in 1.7.3:
  - Doing Step 1 for x=2, y=1, width=3, height=4:

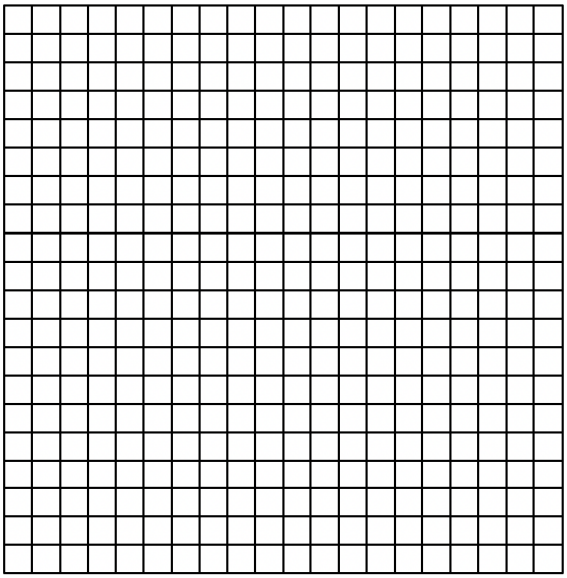


  - Doing Step 2 for x=2, y=1, width=3, height=4:
  - Doing Step 3:
    - How many horizontal lines did I draw?
    - How long was each horizontal line?
    - What is the starting x-coordinate of the ith line?
    - What is the starting y-coordinate of the ith line?
    - My generalized algorithm is:

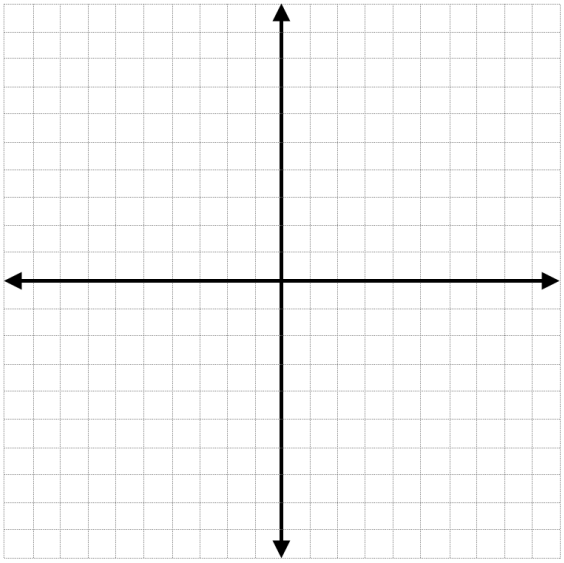
- I have a sub-problem of "Draw a horizontal line of length L starting at (X,Y)", Doing Step 1 of this sub-problem for x=2, y=1, length=3:
- Doing Step 2 of this sub-problem for x=2, y=1, length=3:
- Doing Step 3 of this sub-problem:
  * How many squares did I draw?
  * What was that x-coordinate of the i^{th} square?
  * What was the y-coordinate of the i^{th} square?
  * My generalized algorithm for this sub-problem is:
- Doing Step 4 for this sub-problem for x=3, y=5, length=2



- Doing Step 4 for the whole problem with x=4, y=3, width=2, height=5



- Working the example in 1.7.4:
  - Doing Step 1 for S=$\{(1,3)(5,2),(2,-2),(-3,-4)\}$ and P=$(1,-1)$



  - Doing Step 2 for S=$\{(1,3)(5,2),(2,-2),(-3,-4)\}$ and P=$(1,-1)$
  - Doing Step 3:
    - (we will let you think through what questions to ask to break things down this time)
    - My generalized algorithm is:
  - Doing Step 4 for S=$\{(-3,13),(3,9),(6,1),(1,6)\}$ and P=$(3,5)$