Contents

1 Flowcharting

1.1 Introduction

Have you ever traveled somewhere new without studying a map? Have you given directions to someone without using a map? Road maps are useful tools that provide a general overview of roads to aid in our understanding of where a particular route will take us. To effectively accomplish this, the standard road map omits a level of detail - it does not mark where every stoplight, crosswalk, and trash receptacle is. If it did, the map would be so cluttered that we wouldn't be able to see the roads!

Like roadmaps, flowcharts serve as a big-picture layout of something more complex and detailed. Here, that 'something' is an *algorithm*, or a solution to a problem. Computer programs can be considered algorithms. From a flowchart of an algorithm, one can gain a sense of what goal the algorithm is accomplishing and how the algorithm accomplishes it.

1.1.1 Flowchart construction

Flowcharts consist of geometric elements, such as lines, arrows, and shapes, as well as text, to provide a general overview of a program. The key words here are *general overview*. When developing a flowchart, think about the big picture. Just as a house designer must first plan where the kitchen, living room, bedrooms, and bathrooms will go before placing the water and sewer lines, so we must first organize the stepping-stone blocks of our program before addressing the detailed procedures within each block.

What is meant by stepping-stone blocks? These blocks represent sections of code, where each section accomplishs a goal. That goal could be:

1. inputting or outputting information
2. completing a process
3. making a decision that will alter the outcome of the remaining code
4. connecting one piece of code to another
5. starting or stopping the code

Each type of goal is represented by a particular geometric shape. The basic shapes that we will use are shown in figure 1. The math or text describing each goal will appear inside its respective shape, but every detail of that goal or procedure will not be addressed. Again, the purpose of a flowchart is to provide a big picture of what a program is working toward, not all of the nit-picky details of how it works.

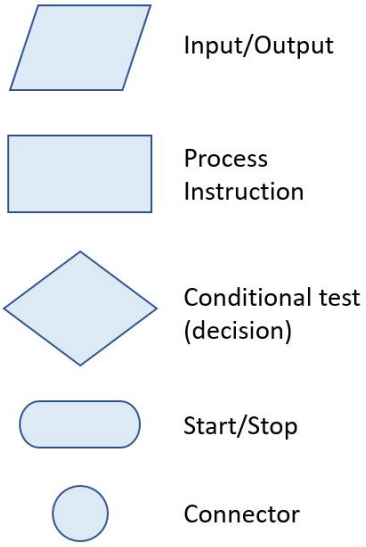

Figure 1: Basic flowchart symbols

There are three structures used to describe an algorithm, and thus, there are three structures from which a flowchart can be constructed:

1. sequential
2. selection
3. repetition

The *selection* and *repetition* structures can be modified to address certain aspects of an algorithm. We will explore these modifications in their respective sections.

1.2 Sequential structure

The sequential structure is the most straight-forward of the three structures. It describes a program in which the computer's processor executes the operations in each consecutive block of code - from start to finish. Think of it as the programming version of a four-course meal, where a person starts with the first course, finishes it, moves to the second course, finishes that, etc. until the meal comes to and end.

In a sequential structure, no decisions are made (no 'forks in the road'), and no steps or groups of steps are repeated. This parallels the four-course meal in that a person has no choice of what to and what not to eat, and that second helpings are not allowed.

A flowchart with the sequential structure is displayed in figure 2. Dashed lines represent repeated symbols which have been omitted. In other words, they imply that more steps and processes could be inserted there.

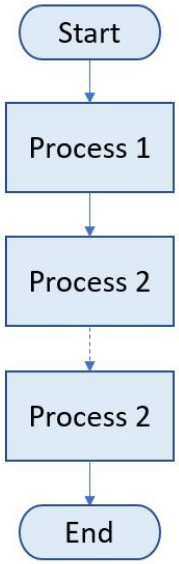

Figure 2: A flowchart illustrating the sequential structure

A specific example of a flowchart with a sequential structure is displayed in figure 3. Here, the program first takes in variables a, b, and c. Then it calculates m. After that, it outputs m and ends. No parts of the code are repeated, and no decisions are made. The processor will execute the same path regardless of what numbers are input for a, b and c.

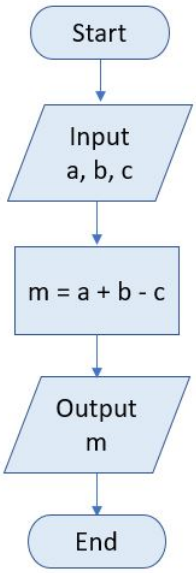

Figure 3: Flowchart with a sequential structure

1.3 Selection structure

The key features of the *selection* structure are *options*. That is, the program presents a 'fork in the road,' and the processor can execute only one of available the paths of code. The path is selected based on a conditional test, which may involve a yes/no or true/false answer or placing a number in a category. For example a conditional statement may pose the question of whether a particular variable has a value greater than three, and then either the 'true' path or the 'false' path is taken. Additionally, a different conditional statement may be used to determine whether a variable's value lies within a *range* of values, and then the path corresponding to that range will be taken. This process is illustrated in figures 4 and 5.

Each path of the selection structure is called a *branch*, and there can be as many branches as necessary. Each branch is composed of any number of steps (operations) with each operation having one of the five goals listed in section 1.1. It is also possible to have a branch with no operations, as seen in the flowchart in figure 11.

Once a decision is made, a path is taken and the rest of the paths related to that conditional statement are neglected. When the operations in the path are completed, the program continues executing the rest of the algorithm.

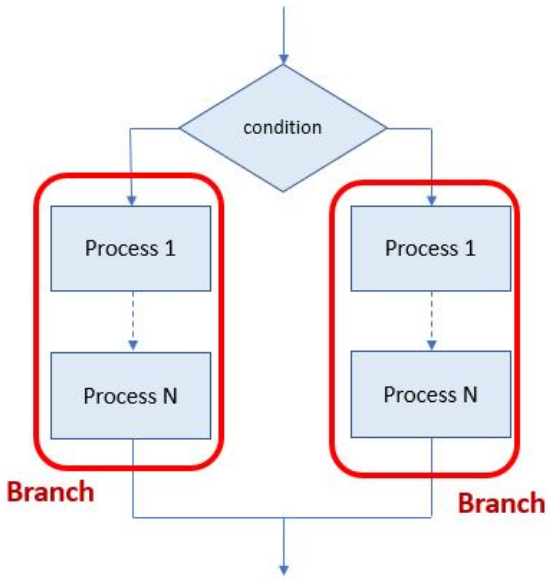

Figure 4: Selection structure with a yes/no or true/false condition

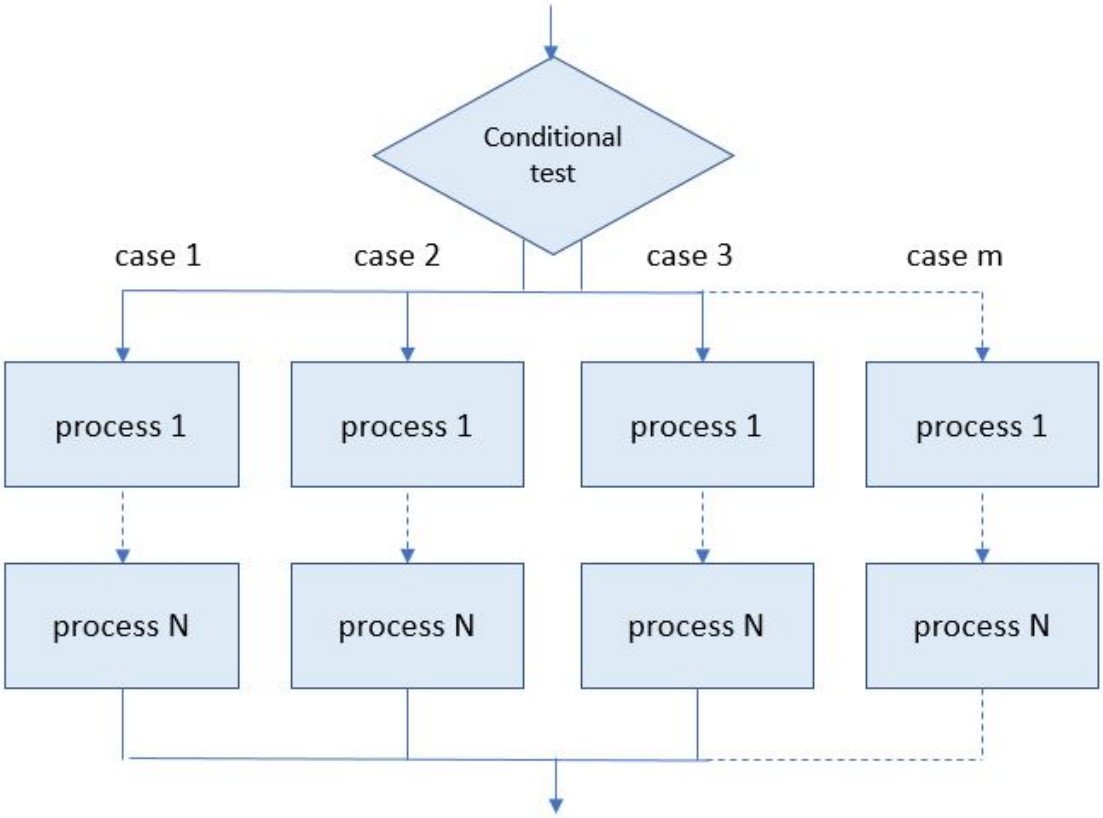

Figure 5: Selection structure with a category-placement conditon

Figure 6 shows a specfic example of a selection structure with a yes/no conditon. From this flowchart, we know that the program will check to see if variable a is greater than 3. If it is a is multiplied by 3, stored as b, and both a and b are returned as outputs. If a is not greater than 3, the right hand path is taken, 10 is added to a, and both a and b are returned as outputs. After this the program executes the remaining code. From this selection structure we see that the rest of the code will run differently depending on which value of b is output.

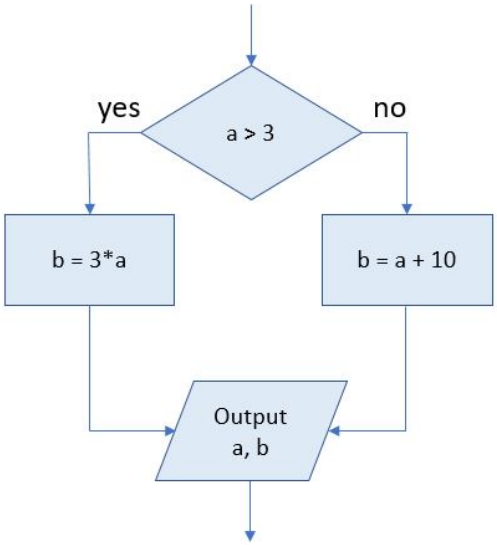

Figure 6: case-structure

Figure 8 provides a specific example of a case-structure flowchart that describes the process used to calculate the load under which a slender column will buckle. If you take mechanics courses in the future, you will learn that the buckling load of a slender column is calculated differently depending on how the column is attached to the top and bottom surfaces. The column can be fixed on both ends (F-F), fixed at one end and hinged on the other (F-H) or hinged on both ends (H-H). Figure 7 displays the three possible end conditions for the slender column.

Depending on which case is used in the algorithm, one of the three paths will be taken, which will assign a specific value to n, and the buckling force will be calculated. The buckling force is also dependent on the modulus of elasticity (E), cross sectional area (A), Length, and the least moment of gyration (r) of the column, which do not change with the end conditons. Thus, these variables would be defined at some point prior to the conditional statement.

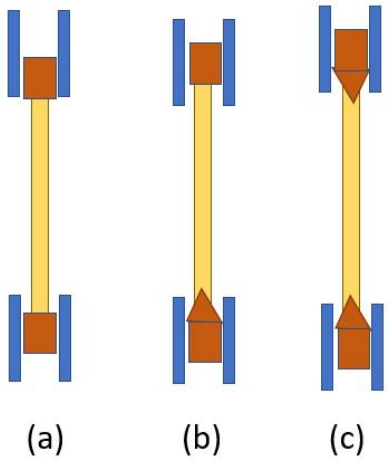

Figure 7: Possible end conditions for a column subject to buckling. (a) fixed at both ends (F-F); (b) fixed at one end, hinged on the other (F-H); (c) hinged at both ends (H-H).

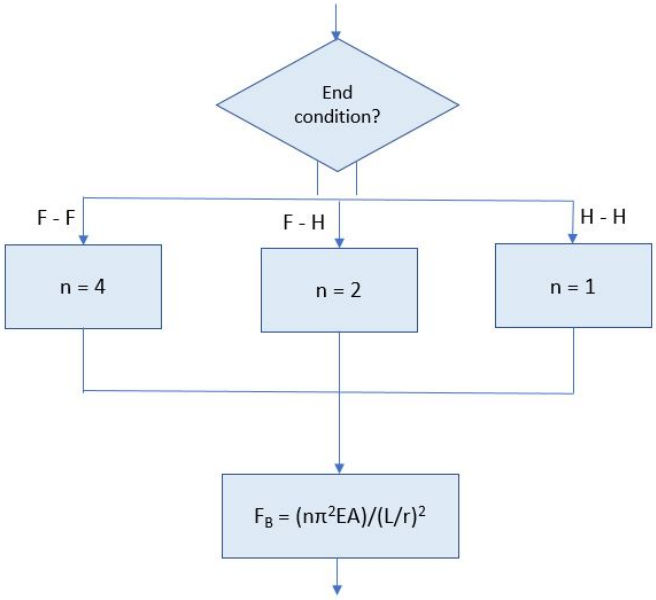

Figure 8: Selection structure with a case-structure conditon

1.4 Repetition Structure

A *repetition* structure, also known as a *looping* structure, describes a step or block of steps that are repeated until specific criteria is met. Sometimes we know the exact number of times we want to repeat a set of setps, and other times we need the steps to be repeated until a condition is met. The former structure is known as a *for* loop, and the latter is a *while* loop. Each of these repetiton

structures will be addressed in more detail in a later chapter. The purpose of this section is to give the reader an idea of how to lay out a repetition process.

In repetition structures, the conditional statement can be at the beginning or at the end of the steps that are potentially repeated. Figure 9 illustrates a repetition structure where the condition is placed at the end of process, whereas figure 11 shows a structure with the condition at the beginning.

Note: Figure 9 is currently designed such that a true condition will cause the steps to be repeated and a false condition will cause the program to disregard the repeat and continue executing the remaining code. The true/false (or yes/no) answers to the conditional statement can be reversed to fit the purpose that the loop is accomplishing. This principle holds true regardless of where the conditioal statement lies relative to the block of steps. Using figure 9 as an example, the programmer could have a false condition causing the steps to be repreated and a true condition causing the program to disregard the repeat and execute the remaining code immediately.

1.4.1 Conditional test is last

As mentioned before, the flowchart in figure 9 has the conditional statement at the end of the repetition structure. Thus, there are a number of processes placed in the *forward* section of the structure, which must be executed before the condition is considered.

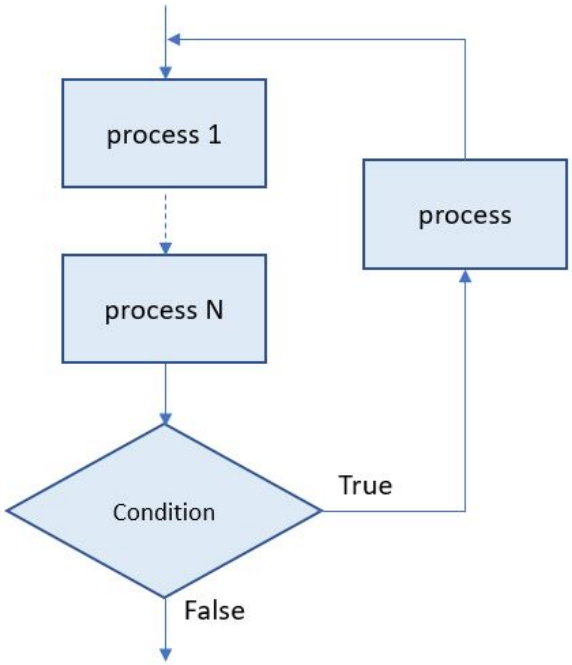

Figure 9: repetition structure with conditional statement at the end

This type of structure contains a *reverse loop*. In figure 9, the the reverse loop is the path taken if the conditional statement is true and the steps are about to be repeated. As seen in the figure, the reverse loop contains a process block. The purpose of this block is to alter or increment some piece or pieces of information each time before the steps in the forward section are repeated. Depending

on what the steps in the forward section accomplish and account for, the reverse-process block may not be needed.

Often the reverse-process block serves as a *counter*. A counter is a user-defined variable that gets incremented each time the steps in the loop are repeated. Thus, when the loop is done repeating, the last value of the counter reveals how many times the loop was repeated. Consider the turnstiles at the entrance of an amusement park. Each time a person walks through the turnstile, the metal pronged structure rotates and the counter inside increases by one. At the end of the day, the counter reveals how many people walked through the turnstile. Counters work the same way in looped structures.

It is also possible to have negative increments such that the counter counts down. This is known as a decrement.

Figure 10 provides a specific example of how we can use a repetitive structure with the conditional statement at the end. This flowchart calculates the surface area (A) and volume (V) of different-sized spheres. Before the looping structure begins, we see the radius (R) is defined to be 1. Then the loop begins. The first time through the loop, a value of R=1 is used to calculate A and V. Once these steps are executed, the values for R, A, and V are outputted. Then the conditional statement is addressed. Is the outputted value of R equal to 10? No - the outputted value of R is still 1 because R was not changed in the forward section. Since R does not equal 10, the reverse loop is taken and R is incremented. In other words, the value of R becomes one increment greater than the old value. Then the surface area and volume are calculated with a value of R=2, and all three variables are outputted again. Then the condition is checked: is the value of R equal to 10? No. The reverse loop is taken and R is assigned a new value of 3. This loop is repeated until the value of R *is* equal to 10. Under this condition, the loop is exited the remaining steps in the program are executed.

We now have the surface area and volume values for ten different spheres. Think about how much longer it would have taken for you to do those same calculations by hand. Loops are looking pretty good, yeah?

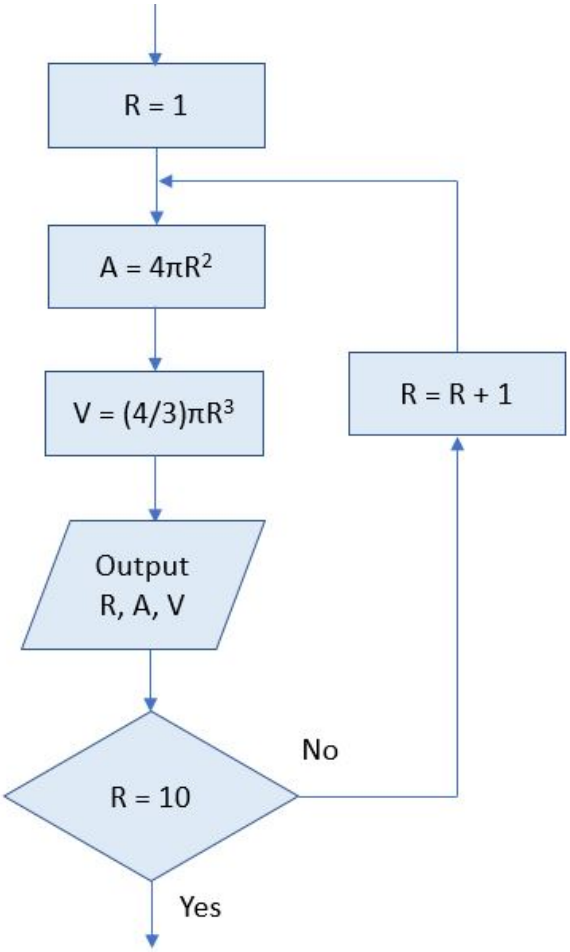

Figure 10: repetition structure with a the conditional statement at the end of the loop

1.4.2 Conditional test is first

In a repetition structure, if the conditional test is not last, it is first. Figure 11 illustrates this.

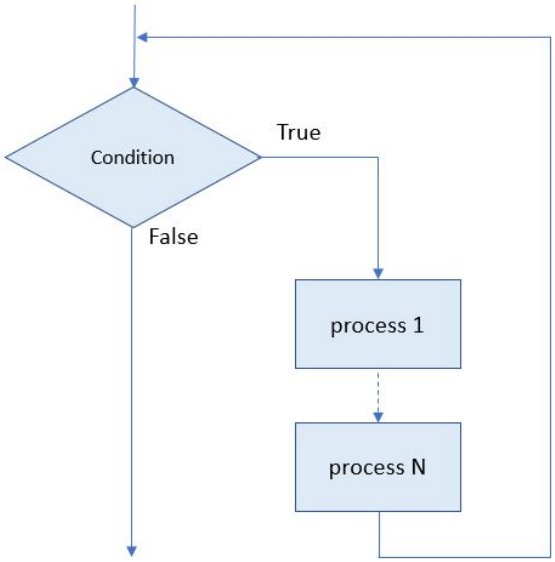

Figure 11: repetition structure with conditional statement at the beginning of the loop

Figure 12 provides the same surface area and volume calculations as figure 10, only here the conditional statement is at the beginning of the loop.

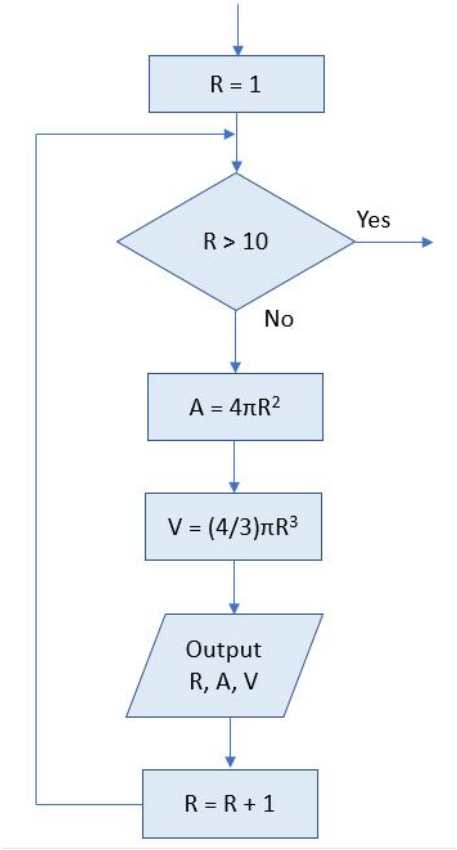

Figure 12: repetition structure with conditional statement at the beginning of the loop

Note: while a loop is repeating, the increment or decrement does not need be increasead by exactly 1. For example, calculating the surface area and volume for a sphere as the radius changes from 1 to 1,000 in increments of 1 would be unnecessary if you didn't need all 1,000 values. In that case, incrementing the radius by, say, 10, would yield a more reasonably sized data set.

1.4.3 Nested loops

Multiple *selection* or *repetition* structures can be combined in a process called *nesting*. This means that a loop is contained within another loop, as seen in figure 13.

In the nesting structure, the condition of the innermost structure must be satisfied before the processor moves to the *next outer* structure. As the program progresses toward meeting condition 1, the processor will execute the inner loop as many times as necessary until condition 2 is met. Once condition 1 is satisfied, the processor continues with the rest of the code.

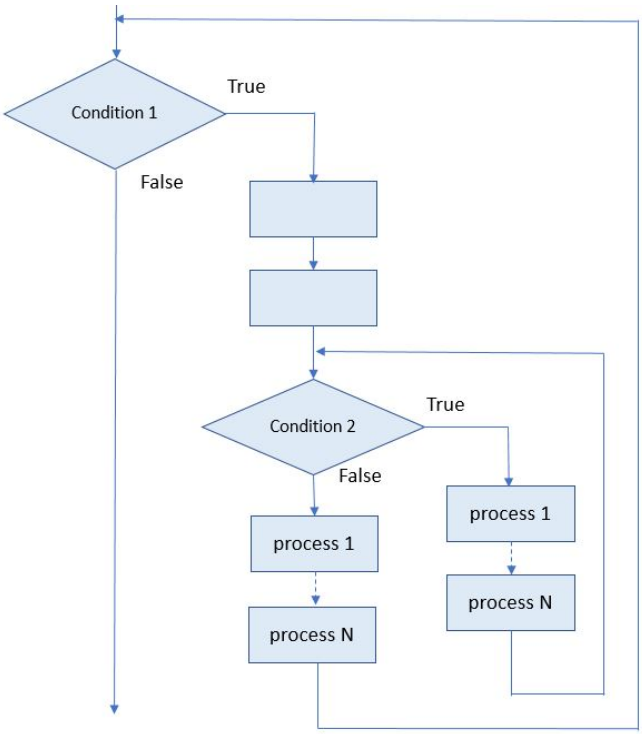

Figure 13: Nested loops

1.4.4 Nested loop example

Nested loops can be difficult to grasp, so let's use a physical example as an illustration. Say you are at a family picnic and you are participating in a most unusual race. At the start line you see four shirts: red, blue, yellow, and green. Off in the distance, you see three hats, each with a big letter on the front: A, B, and C. Next to those hats is a potato sack; yes, a burlap potato sack. Finally, at the end of the field is a tree. So what are the rules of the race? As a participant, you must hop down to the tree and back, in the burlap potato sack, while wearing every combination of shirt and hat. How do you do this?

You step up to the starting line, put on the red shirt, and run to the hat station, where you put on the A hat. Jumping into the potato sack, you hop down to the tree and back to the hat station, where you take off hat A and put on hat B. Then it's down to the tree and back in the burlap sack. You switch hats from B to C and once again hop to the tree and back. When you return to the hat station, you rejoice as you realize you have finished cycling through all three hats. At this point, you wriggle your way out of the burlap potato sack and run (and oh, how glorious it feels to *run*) back to the starting line where you take off the red shirt and put on the blue. Now wearing the blue shirt, you run back to the hat station, where you wear each of the three hats during your three hopping trips down to the tree and back.

After completing that segment of the race, you run back to the starting line where you exchange your blue shirt for the yellow shirt and then it's time to go through each of the hats while hopping to the tree and back. After this, you cycle through the hats one last time while wearing the green shirt and then you make your way to the finish line (which is really the starting line in the opposite direction). The last shirt is green to camouflage the grass stains you are expected to incur

as you cross the finish line and dramatically crumble to the ground in a heap of exhaustion. Though your legs feel like flubber, you are very 'hoppy' to be done with the *nested loop* race.

Figure 14 displays the nested loop race in a flowchart. Of course, there are other ways to arrange the elements of the flowchart, since the conditional statements can be placed in different locations, as we discussed in previous sections.

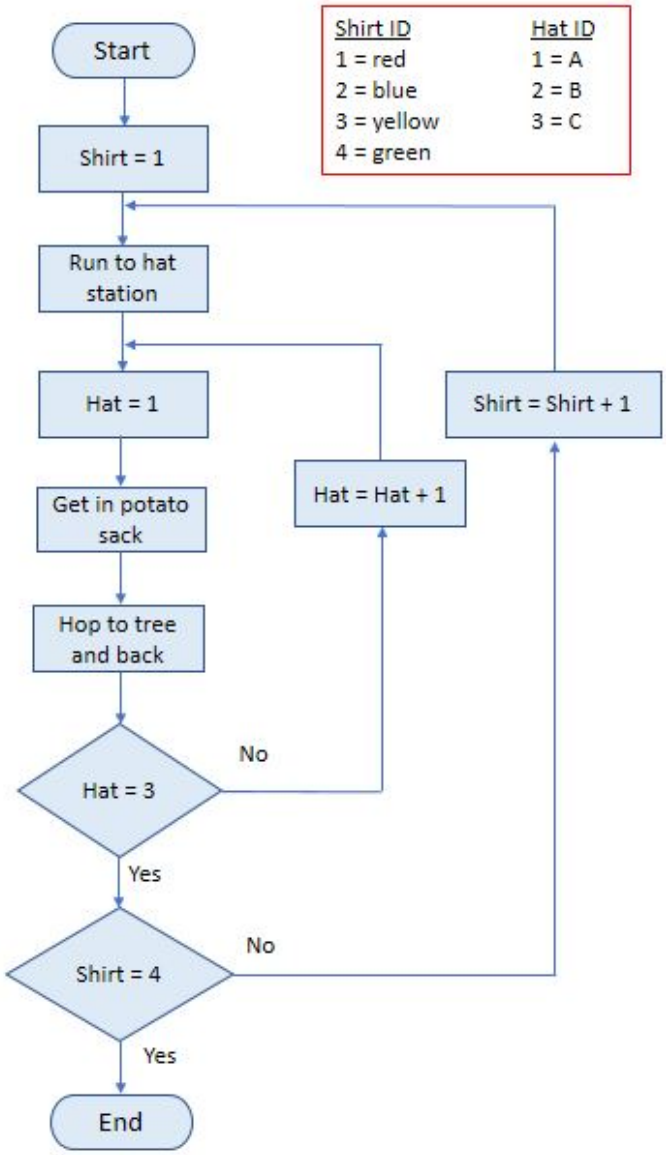

Figure 14: Nested loops