Don't Stop–Do It Again!

Paul S. Wang, Sofpower.com

January 22, 2023

The universe and the world around us work with rhythm and repetition. Sunrise and sunset; the seasons; you get out of bed, go to work or school, come back home and go to bed at night. Many things repeat regularly—songs and music have reprises, visual arts and architecture designs also use repetition of patterns, animals breath in and out, their hearts go on beating.

All that is not surprising. But, what if I tell you that repetition is also a powerful way to solve problems? This article focuses on repetition or "*iteration*" and formalizes it as a systematic way for problem solving through the lens of *computational thinking* (CT). By doing this we hope to gain a deeper understanding of the nature of iteration in procedures and to realize that iteration can be applied to solve problems in many ways and in different areas including computing and daily living.

This post is part of our CT blog (`computize.org`) where you can find many other interesting and useful articles.

A Simple Example

Often we need to arrange data items in a certain order to make them easy to use. For example words in a dictionary are in alphabetical order. Personnel records are ordered by names. Numbers are arranged in ascending or descending order, and so on.

To begin, let's see how iteration is applied to rearrange a list of, say, grades of eight students, into ascending order. Here is the list of grades:

$$67.5,\ \ 59.5,\ \ 82,\ \ 93.5,\ \ 77,\ \ 45.5,\ \ 100,\ \ 76.5$$

And we want to rearrange them into

45.5, 59.5, 67.5, 76.5, 77, 82, 93.5, 100

The method is simple, find the highest grade and move it to the last position of the list. Then, repeat the operation on the rest of the list (now with one less grade) and so on, until no more grades are left. The list is now in ascending order. This method is actually called *bubble sort* because in computing sorting means putting items into order.

Problem Solving with Iteration

Iteration is a powerful tool for problem solving. The tool can be applied if a problem, any problem, fits the following *iteration paradigm*.

1. *Do I know how to solve this problem if it is in its simplest form*?
2. *For that problem in general, do I have a way to reduce its size or complexity*?
3. *Is the reduced problem of the same exact nature*?

If the answer is yes to all three questions, you can apply iteration to solve that problem.

Walking to a destination, for example, can be solved by iteration because

1. The simplest form is when the destination is just one step away.
2. Otherwise, we can reduce the problem by one step.
3. And the reduced problem is still *walking to that destination*, the same exact nature.

Applying the Iteration Paradigm

Let's examine a couple of daily tasks that can be performed by iteration. First the difficult task of eating a bowl of soup. Here the simplest or trivial case is that the bowl is already empty or almost empty. We just need to do nothing or finish it directly and stop.

However, if the bowl is full, we can take a spoonful (surprise) thus reducing the size of the task. The resulting problem is a smaller task of the same nature, that is to finish the remaining soup. Because the problem fits the paradigm, we can be sure that repeatedly taking spoonfuls will work fine :-)

Another example is delivering newspapers on a *paper route* which is not unfamiliar to many of us. The task is to drop off today's newspaper at a list of locations in a neighborhood, usually by riding a bicycle.

Here is how that task fits the iteration paradigm. If the location list is

empty or of length one, then we simply finish the task directly. For a long list of locations, we can reduce the list by riding to one of the locations. The list is shortened and the remaining task is still of the same exact nature. Obviously by repeatedly riding to another location we can complete the task.

While the iteration stays the same, the order in which the locations are visited can make the task easier or harder. Ingenuity can lie in picking which location to visit next, especially when the number of locations increase and their distances vary.

Iteration Paradigm in Bubble Sort

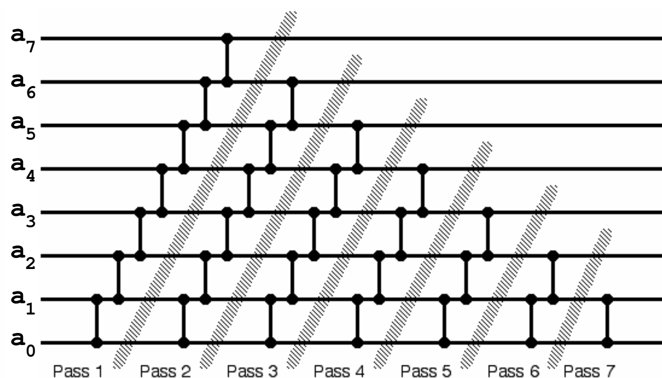
Now let's revisit bubble sort to reveal the iteration paradigm and the repeated operations in it.

If the list to be sorted contains just one element, then we are finished by doing nothing. In general, the list will have more than one element. Then, bubble sort pushes the largest (or smallest) element to the end of list for ascending (descending) order. The list is now reduced by one element and the smaller problem is still a list of elements to be ordered.

The actual *pushing of the largest (smallest) element to the end of the list* is again done by iteration. The pushing task fits the iteration paradigm (you can figure this out).

The iteration repeats a number of *compare-exchange* (CE) operations. Each CE operation compares two adjacent elements of the list, and exchanges (swaps) them when necessary so that the latter element is larger (or smaller for descending order).

Let's see how exactly it sorts a sequence of eight elements, a_0, a_2, ..., a_7.



Bubble sort makes multiple passes to complete its job.

Pass 1 repeats CE operation seven times: CE(a_0, a_1), CE(a_1, a_2), CE(a_2, a_3), CE(a_3, a_4), CE(a_4, a_5), CE(a_5, a_6), and CE(a_6, a_7) and moves the largest value to a_7.

In a similar manner, pass 2 repeats CE operation six times and moves the largest value of a_0, a_1, ..., a_6 up into a_6; pass 3 repeats CE operation five times and moves the largest value of a_0, a_1, ..., a_5 up into a_5; and so on, until, finally, pass 7 repeats CE operation one last time and moves the largest value of a_0 and a_1 into a_1 to complete the sorting.

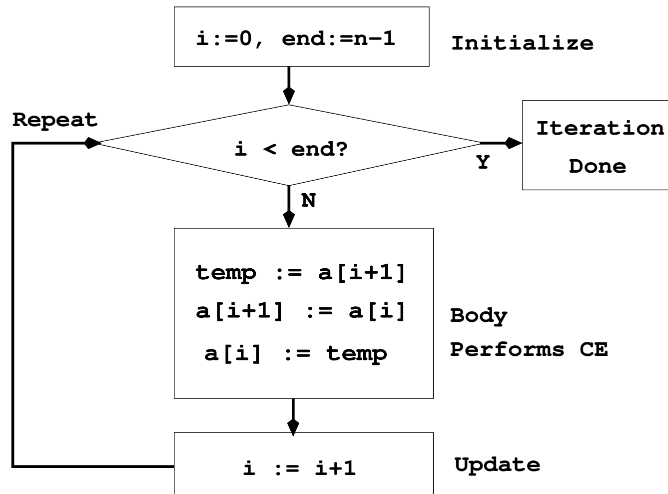
Iteration Control

To employ iteration in any application, we need to precisely control and execute all the necessary repetitions. We can look at an iteration abstractly as a construct, taking it out of the context of any particular application. We see that every iteration consists of the following elements:

- An **initialization** to set starting values in preparation for controlling and carrying out a number of repetitions of a certain operation
- An **end condition** to test if the iteration should terminate or to proceed to the next repetition
- A **body** of operations to be performed repeatedly
- An **update**, usually after completing a repetition, to set new values for controlling the next repetition

Let's put bubble sort in algorithmic form so as to clearly identify these elements. We assume the list of grades have been placed in an array of numbers `a[0]`, ... `a[n-1]` and the array length is `n` (the total number of grades).

Recall that bubble sort repeats a number of phases. and each phase repeats a number of CE operations. Here is a flowchart illustrating the procedure for each single phase:



One Single Phase of Bubble Sort

Every iteration requires control to manage the repetitions and to start, continue, and end the repetitions correctly. Here are the essential elements of iteration control in the flowchart.

1. **Initialization**: Set the control variables `i` to zero and `end` to `n-1` once and once only in the beginning.

2. **Repetition of iteration body**: Perform the iteration body once only if `i` is less than the value of `end`.

3. **Update**: Increase the value of the control variable by one, `i=i+1`.

Following the flowchart we see clearly one complete phase of CE operations and the largest value being pushed to the end. In the flowchart, CE uses a temporary variable `temp` to swap the values of two elements.

To complete the bubble sort, we need to perform all phases, starting with the full array of elements. By reducing the value of `end` by one, we ensure that the next phase will work on a shorter array.

Thus, **we have two iterations one nested within the other**. The outer iteration repeats the inner iteration to execute each phase. It updates `n`, `n=n-1`, just before the next phase. And each individual phase is performed by the inner iteration which repeats CE operations.

Iterations Are Not Created Equal

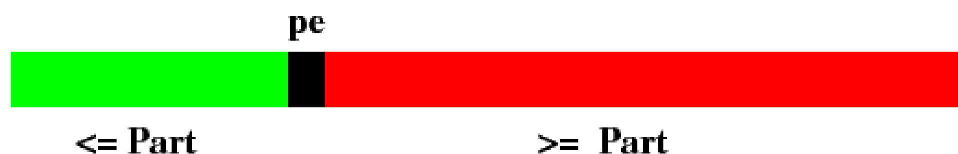
Even though the speed of modern computers can help us use brute-force to iterate over large amounts of data, the efficiency of an algorithm is still very important.

Bubble sort is inefficient and seldom used in practice. This is because the number of phases grows linearly as the length of the array to be sorted grows and the array in each phase is only shortened by one element.

A much better algorithm is *quicksort* which still applies iteration but much more cleverly.

With quicksort the idea is to split the array to be sorted into two parts, smaller elements to the left and larger elements to the right. This is called the *partition* operation, which first picks an arbitrary element of the array as the *partition element* `pe`. By exchanging elements, the array can be arranged so all elements to the right of `pe` are greater than or equal to `pe`. Also, all elements to the left of `pe` are less than or equal to `pe`. The location of `pe` is called the *partition point*.

After partitioning, we have cut the array into two parts, one to the left of `pe` and one to the right of `pe`. The same partition method is now repeated on each of the two parts. When the size of a part becomes less than 2, the partition stops. When all partitions stop, the task is done.



Quicksort is efficient in practice, because it often reduces the length of the array to be sorted quickly, basically cutting the array length in half after each partition.

Finally

We often hear the adage "*if it works do it again*." Iteration certainly follows that principle.

Iteration has wide applications, in computing and in daily living. Any problem that fits the *iteration paradigm* is a candidate.

Each iteration involves initialization, continuation, and termination. Controlling how these are performed is important when specifying an iteration for a particular application. While an iteration is straightforward, ingenuity often lies in the way to reduce a task to small ones. In the end, iteration is not only a method to solve problems but also a new way of thinking.

When we face a new problem or task, we may have various ways to tackle it. What iteration teaches us is that we can consider vastly simplified cases to get some insight which can lead us to ways to cut the task down. As soon as we reduced the problem to one or more smaller job of the same nature, **we have the problem solved already!**

Isn't that almost magical?