Iteration vs. Recursion

Two Basic Algorithm Design Methodologies

Li Chen

University of the District of Columbia

Iteration and recursion are two essential approaches in Algorithm Design and Computer Programming. Both iteration and recursion are needed for repetitive processes in computing. An iterative structure is a loop in which a collection of instructions and statements will be repeated. A recursive structure is formed by a procedure that calls itself to make a complete performance, which is an alternate way to repeat the process.

Iteration and recursion are normally interchangeable, but which one is better? It DEPENDS on the specific problem we are trying to solve.

1. Understand Iteration and Recursion Through a Simple Example

Let us consider a very simple example as follows: Calculate *1+2+...+n*. What we need to do is define a variable *Sum* with the initial value of zero and repeat a calculation:

```
Sum = Sum + i ;

i = i+1;

Repeats above calculation while i <= n.
```

The above procedure is called an iteration.

On the other hand, we can define *f(n) = 1+2+...+n.* So, *f(n)= n + f(n-1)* where *f(n)* is called a recursive function. If one can write a program to perform such a recursive procedure, then the task can be done recursively. Using Java, C/C++, or Python languages, we can do that as follows:

```
Procedure f ( integer n)

   if (n is 0) then assign (f = 0);  // (meaning that we return 0 to the procedure)

   assign (f = n + call Procedure f(n-1));

End of the procedure
```

The above procedure calls itself and decreases the parameter by 1. When we need to calculate *Sum* for parameter *n*, we simply let *Sum=Procedure f(n)*.

2. Philosophy and the Graphical Representation

For iteration, we can use a sentence to present its philosophy: "The whole is a collection of all individuals." Iteration uses the idea of accumulation, which depends on human intuition. That is to say, the strategy solves one basic element of the problem at a time, then repeats and accumulates until the end.

We can say that iteration is a process that repeats and converges.

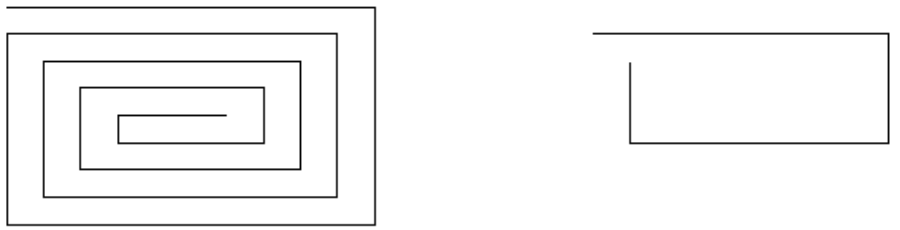

Fig. 1. A graphical explanation of iteration

An iterative structure has a basic type of element that repeats many times (for the whole). For example $f(n)= 1+2+...+n$, where $f(n)$ is "the whole" and $i=1,2,...$or n is a sample of the basic type of element. Collecting all elements is the whole $f(n)$ and elements or individuals are similar to each other. Fig.1 shows us that (a) can be formed by accumulating (b)'s together. **Iteration is a process that repeats and converges.**

For recursion, the philosophy is: "The whole can be represented by its parts." Recursion usually reduces the size of the original problem. It tries to find a solution for smaller parts and then merges the parts to the whole.

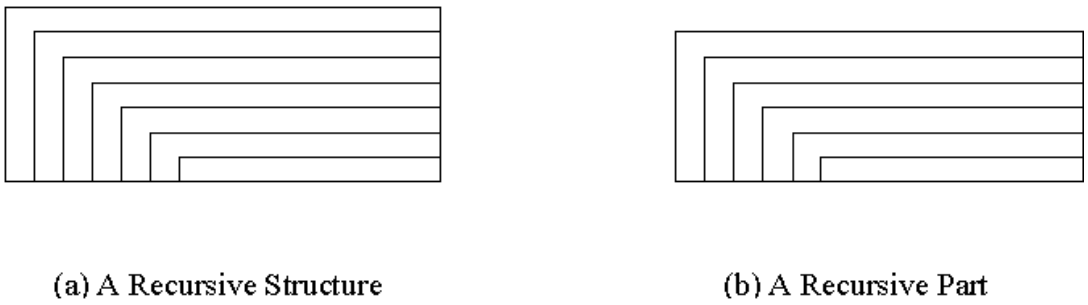

Fig. 2. The graphical explanation of recursion

A recursive structure has parts that are almost the same as the whole (see Fig. 2 (a) and (b)). Fig. 2 (a) can be viewed as the whole, and Fig. 2 (b) is a part of (a) with the same structure. For example, $f(n)= f(n-1) + n$, where $f(n)$ *(e.g. Fig.2 (a))* is "the whole" and $f(n-1)$ (e.g. Fig.2 (b)) is a part that is similar to $f(n)$. **Recursion is a process that calls itself in the manner of reducing the size until the base (ending) case is reached** (such as $f(0)=0$).

3. Iterative Thinking and Recursive Thinking

Iterative thinking follows human's nature, i.e., goes through each individual to reach the whole. For a simple problem, this way may be more efficient, such as in the above example.

If we want to calculate Fibonacci numbers: $f(n) = f(n-1) + f(n-2)$, where $f(0)=0$ and $f(1) =1$, it is very hard to represent $f(n)$ by listing all numbers in linear order using iteration. A more elegant way is to calculate $f(n-1)$ and $f(n-2)$ first and then add them together to get $f(n)$. In such a case, recursion would be the best choice in terms of coding simplicity. (Please note that when n is relatively a big number, we need a sophisticated strategy to avoid exponential times of calls, such a strategy is called dynamic programming.)

4. Examples: Sorting and Searching

Sorting

Sorting is to put a set of numbers into an order. The easiest way is to find the largest one and put it last, then find the second largest one and put it second to last, and so on. This method is called Bubble-Sort. The basic element of the process (Fig. 1 (b)) is to find the largest number in an unsorted number set (or list).

Another way to sort this set of numbers is to partition it into two parts, then sort the first part and second part separately. After that, merge the two parts into one. This algorithm is called the merge sort algorithm (Merge-Sort) and the methodology is called Divide-and-Conquer. For Merge-Sort, Fig. 2 needs to be modified since there are two parts (the first half and the second half) that need to be sorted before getting a solution for the whole set. Please think about it in detail if you like to program it.

Searching

Search usually means finding a specific number (can be called a search number) in a set of numbers. If the set of numbers are randomly arranged, we must check each of them. However, if this set of numbers is put in an order, then the search task can be done easily if we use a smart method.

Let us see if we can find a number by comparing it with each element in a set (called sequential search algorithm). We are sure we will eventually figure out if the search number is in the set or not. However, when the set is already sorted in an ascending order, we can do the job much faster using the following method: We can first check if the search number is smaller or bigger than the number placed at the middle of the set (as a list). Then, we only need to search the first

or second part that is separated by the middle element in the list. What do you think? Can you design this algorithm in detail?

The best hint here is to think about total student records in the university's computer system, which are listed in an alphabetical order. If we know the middle element of the list is named "Norton," a student whose last name is "Freeman" must be in the first half of the list. We do not need to search for it in the second half.

This algorithm is called the binary search algorithm (a type of the bisection method), and we present the Python code for both recursion and iteration in the Appendix.

5. Iteration vs. Recursion: Which is Better?

What Kind of Algorithms Can be Called Good Algorithms?

A good algorithm usually means one that is fast and uses less (memory) space. How fast an algorithm can be achieved to solve a specific problem not only depends on the design technology of the algorithm, but also depends on the problem itself. Some problems are hard to solve or have no way of being solved, such as an NP-Hard problem or "The Halting Problem."

At the same time, some problems are much easier. Sorting and searching are two popular problems where simple algorithms exist. For instance, for the sorting problem, we use two loops in Bubble-Sort. For the search problem, even though we compare every number, we only need one loop. Finding the best algorithm for a specific problem is always the goal for computer scientists and programmers.

Efficiency---The Time Complexity of an Algorithm

In the bubble sort algorithm, there are two kinds of tasks. The first is to find the maximum number in a set (an array) that has n elements. It takes $(n-1)$ number of comparisons. Then the second is to repeat the first task. Since after the first round we have a fixed maximum number, the size of the unsorted set is reduced by 1 and we only need (n-2) comparisons to find the second largest number. We will repeat this process ..., until the end. Then, the total number of comparisons is $(n-1)+(n-2)+...+1 = n(n-1)/2$, where n is called the problem size and $n(n-1)/2$ is called the time complexity of the algorithm. The divide-and-conquer sorting algorithm (called Merge-Sort) only needs $C\ (n \log n)$ number of comparisons where C is a constant. For simplicity, we use $O(n \log n)$ to represent $C\ (n \log n)$. $O(.)$ is called *big O-Notation.*

For the sequential search algorithm, check that every element only needs n comparisons, so the complexity is $O(n)$. However, Binary-Search only requires at most $\log(n) +1$ comparisons. Thus, the binary search algorithm is much faster and its complexity is $O(\log n)$.

Iteration vs. Recursion

For any problem, if there is a way to represent it sequentially or linearly, we can usually use iteration to solve it, such as *1+2+...n*. This is because we use less memory. In a recursive process, a computer needs to store intermediate results. For example, for the purpose of merging in Merge-Sort, we need to store the sorted results for the first half and the second half.

On the other hand, if there is no easy way to have linear representation, then we can consider using recursion. If a problem can be divided into two or more similar components, recursion can be used to speed up the process by a great deal, generally. This also refers to tree structures and operations.

Again, the drawback of recursion is that it needs extra space to store intermediate results. This can be considered a trade-off between "Time" and "Space."

Appendix: The Python Code for the Binary Search Algorithm

```
1.  # BinarySearch.py  use one of the following Python compilers
2.  # https://www.programiz.com/python-programming/online-compiler/
3.  # https://www.tutorialspoint.com/execute_python_online.php for Python v2.7.13
4.
5.  A=[1,2,3,4,5,6,10,15,16,18,20]
6.
7.  ##### The function of recursion
8.  def binarySearchRec(Start,End,SearchVal,array):
9.      k= (Start + End)//2; #k = mid; using // to get an integer for division
10.     if (Start==End):
11.         if(SearchVal == array[k]):
12.            return 1;
13.         else:
14.            return 0;
15.     # Recursive-call for first part or second part of the array
16.     if (SearchVal <= A[k]):
17.         return  binarySearchRec(Start, k, SearchVal,array);
18.     else:
19.         return  binarySearchRec(k+1, End, SearchVal,array);
20.     # SearchVal is not in the array
21.     return 0
22.
23.
24. ##### The function of iteration
25. def binarySearchItr(Start,End,SearchVal,array):
26.     #using while-loop to iterate
27.     while (Start<=End) :
28.         k= (Start + End)//2; #k = mid; using // to get an integer for division
29.         #Change the search interval when iterating.
30.         if (SearchVal < array[k]):
31.             End= k-1;
32.         elif (SearchVal > array[k]):
33.             Start= k+1
34.         else:    # SearchVal == array[k]
35.            return 1
36.     # SearchVal is not in the array
37.     return 0
38.
39. ##########--main-- #########
40. Val=16;
41. beg=0;
```

```
end=len(A)-1;
#call the recursive function
if (binarySearchRec(beg, end, Val,A)==1):
    print (" Yes, the 16 number is in the array");
else:
    print (" No, the number 16 is not in the array");

Val=12;
#call the recursive function
if (binarySearchItr(beg, end, Val,A)==1):
    print (" Yes, the number 12 is in the array");
else:
    print (" No, the number 12 is not in the array");


#======= Result Display:

# Yes, the 16 number is in the array
#  No, the number 12 is not in the array
# >
```