Algorithms: Recursion and Iteration

Darsh Jadhav
School of Computer Science and Electronic Engineering
Bangor University
Bangor, United Kingdom
darshjadhav7@gmail.com
3rd October 2019

***Abstract*—This report is a brief overview of the algorithmic concepts of recursion and iteration.**
***Index Terms*—Recursion, Iteration, Algorithms, NP/P**

I. Introduction

An algorithm is a set of instructions/rules that, especially if given to a computer, will assist in providing a solution to a problem. Examples of these include recursive, iterative, backtracking, divide and conquer, dynamic programming, greedy, brute force, stochastic, heuristic, and many more. I acknowledge the importance of these algorithms and the benefits of using different algorithms for different problems. Within this report, we will be analysing recursion and iteration as I feel that they are fundamental foundations for many other algorithms. A recursive algorithm is one that calls itself to solve a problem. An iterative algorithm uses a repeated set of instructions in order to solve a problem.

II. Background Literature

The history of recursion links back to 1888 where a German mathematician named Richard Dedekind used recursion in order to acquire functions to conduct an analysis on the concepts of natural numbers [1]. In 1989 and 1991, Italian mathematician called Giuseppe Peano used Dedekind's work for his five axioms for positive integers [2]. These five axioms are very important as it allowed the creation of an infinite set of numbers using a finite set of rules and symbols [3]. Dedekind's work introduced primitive recursion (so called since Rózsa Péter, 1934). This is what Peano used to prove the theorem that primitive recursion defines a function on the positive integers, whilst applying it on the development of functions $(m + n, m * n, m^2)$. In 1958, John McCarthy created a functional programming language called Lisp, this was implemented on an IBM 704 computer. This was known to be one of the first programming language to have used recursive functions. The history of iteration dates back to 1954 where John Backus invented a programming language known as Fortran. Fortran was a high-level programming language used as a digital code interpreter. The first sign of iteration was seen in documentation regarding programming conventions in Fortran for the IBM 360/67 (a mainframe computer which utilised the Fortran programming language). The concept of for loops were used to iterate through vectors containing data [4]. Iteration is still used in this way for iterating through different types of data structures.

III. Algorithm/Pseudo-Code

Algorithms can be deterministic or stochastic. A deterministic algorithm is one which produces the same solution to a problem every time that it is executed. A stochastic algorithm relies on randomness in order to solve a problem, meaning that upon calling the algorithm again to solve the same problem, it may provide a different solution. This is not particularly useful as we may not be provided with the best solution to the problem when running the algorithm. When looking at the following recursive and iterative algorithms, we can see that they are deterministic algorithms.

A. Recursion

Firstly, we will talk about recursive algorithms. Recursive algorithms work by calling itself to solve a problem. Recursion relies on the solutions of smaller, broken down instances of the same problem. This is commonly used when a solution to a problem can be solved in simpler cases of the same type of problem. The steps to a general recursive algorithm are as follows:

1) *Set up the recursive function*
    a) *Create an if statement*
        i) *This will contain an instruction which will call itself repeatedly to solve the problem whilst the condition is true.*
        ii) *Once condition is false, end the function.*

Using pseudo-code, we can see how a recursive algorithm is used to calculate the factorial of N. We can see in figure 1 that the factorial function contains an if statement which contains conditions to determine how the function will proceed. The algorithm will make a check to see whether N is more than or equal to 1, if the condition is met, then the function will call itself repeatedly. The algorithm will stop once the condition is no longer met. We can see that recursion is used in the if statement as it is calling itself in order to solve the problem.

Listing 1. Pseudo-code for a Recursive Algorithm to calculate factorial number

```
factorial(N equals any integer):
   if N is >= 1
      int factorialNum =N * factorial(N-1)
   else
      return factorialNum
```

B. Iteration

Iterative algorithms involve the execution of a repeated set of instructions in order to solve a problem. This means that the algorithm with keep on repeating an instruction until the condition is met. An iterative algorithm generally uses a for loop in order to repeatedly execute an instruction. The steps in an iterative algorithm are as follows:

1) *Setting up your iterative function*
   a) *Set up a for loop:*
      i) *This should initialise a counter which will iterate towards an end condition.*
      ii) *Insert your instructions into the for loop.*
   b) *When the for loop ends, print the solution to the problem.*

In listing 2, we can see that the iterative function consists of a for loop which will execute a set of instructions N times in order to solve the problem. This for loop will keep running until statement in the for loop is false. Once the for loop ends, the solution will be returned.

Listing 2. Pseudo-code for an Iterative Algorithm to calculate factorial number

```
factorial(N equals any integer):
   int i equal to 1
   int factorialNum equal to 1
   for i <= N, post increment i:
      factorialNum = factorialNum * i
   return factorialNum
```

IV. Mathematics (NP/P)

A problem that an algorithm is trying to solve can be classified into P, NP, NP-complete, and NP-hard time complexities. We will primarily focus on understanding P and NP. A problem with the time complexity of P means that an algorithm can complete a problem in polynomial time (meaning that the algorithm can provide a solution to the problem in finite time). NP stands for non-deterministic polynomial time. A problem that has a time complexity of NP is one which an algorithm cannot provide a solution to in polynomial time (the algorithm may go on indefinitely until solved), therefore we can say that the algorithm will spend exponential time in order to complete the problem [5] [6]. In our examples, we can classify that the recursive and iterative factorial algorithms are P, meaning that the solution can be solved in polynomial time. This can be explained by looking at time complexity of these algorithms. All algorithms have a time complexity, this tells us how much time it takes for an algorithm to run. We can use the big O notation to do this. The big O notation allows us to assess the efficiency of an algorithm, which gives us an idea as to how long it would take for an algorithm to solve a problem. We can calculate the time complexities for the algorithms in listings 1 and 2. The way we calculate time complexity is as follows. We calculate the number of operations in the algorithm. We then find a general formula for this by reducing the expression into a generic expression of T(n). This allows us to evaluate the relationship of T(n) to n which will give us the time complexity.

$$
\begin{aligned}
T(n) &= T(n-1) + 3 \\
T(n) &= T(n-2) + 6 \\
&\approx T(n-k) + 3k \\
T(0) = 1 \Rightarrow n - k = 0 &\Rightarrow k = n \\
T(n) &= T(0) + 3n \\
T(n) &= 1 + 3n \\
T(n) &\propto n \Rightarrow O(n)
\end{aligned}
$$

V. Diagrams

Figure 1 is showing the individual steps in the recursive factorial algorithm. The first step is to call the function, we then have a condition whether N is more than or equal to 1. If the case is true, then the function will call itself with the parameter N-1. This will keep occurring until the condition is false, upon which the function will return the solution.

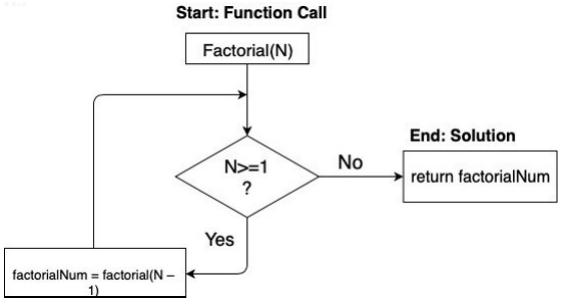

Fig. 1. Recursive Factorial Algorithm Diagram

Figure 2 is a visualisation of how the iterative factorial algorithm solves the problem. Upon calling the function with parameter N, we will go through a for loop. This for loop will assign the integer i to 1, check if i is less than or equal to N, and then increment i. If the condition in the for loop is true, we will multiply the factorialNum variable by i. The for loop will continue looping until the condition is no longer met, upon which the solution will be returned.

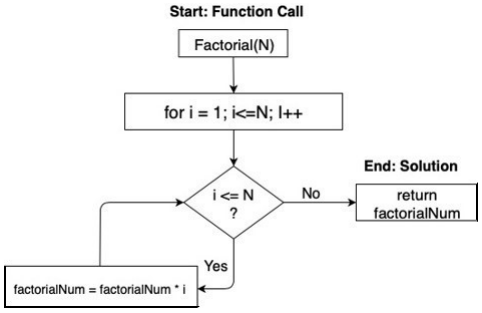

Fig. 2. Iterative Factorial Algorithm Diagram

VI. Applications

The recursion and iteration algorithms are used as a base model/method for many other algorithms. Recursive algorithms tend to be more elegant when an algorithm is much bigger. This means that the code is smaller as opposed to an iterative algorithm, making it much easier when debugging a program. Recursion is particularly useful and favoured when searching through specific types of data structures like trees whereas iterative algorithms tend to search through data structures like arrays. Iterative algorithms are generally favoured over recursive algorithms as they tend to execute faster, and do not require extra memory. Recursion stores all its function calls in a stack which means that more memory needs to be allocated, making recursion a much slower algorithm to use.

VII. Similar To

Many different algorithms tend to build on the concept of recursion and iteration. Divide and conquer is an algorithm that utilises recursion to split a larger problem into smaller, independent tasks. The solutions of these tasks are then combined to solve the initial problem. Dynamic programming algorithm is similar to divide and conquer but utilises overlapping sub-problems to solve the main problem. This tends to be favourable when we do not know the clear line between the sub-problems. This is more of a bottom-up approach to a problem as opposed to divide and conquer starting from the problem and breaking it down. Brute force is an algorithm that utilises iteration to try all possible solutions to a problem in order to find the solution to the problem, this algorithm relies on vast amounts of computing power to produce the solution.

References

[1] Odifreddi, Piergiorgio and Cooper, S. (2019). Recursive Functions (Stanford Encyclopedia of Philosophy). [online] Plato.stanford.edu. Available at: https://plato.stanford.edu/entries/recursive-functions/#1.3 [Accessed 27 Oct. 2019].

[2] Kleene, S. (1981). Origins of Recursive Function Theory. IEEE Annals of the History of Computing, 3(1), pp.52-67.

[3] Partee, B., Wall, R. and Meulen, A. (1993). Mathematical methods in linguistics. Dordrecht: Kluwer Academic Publishers.

[4] Lawson, C., Hanson, R., Kincaid, D. and Krogh, F. (1977). Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathematical Software, 5(3), pp.308-323.

[5] Cook, S. (2003). The P vs NP Problem.

[6] Ralston, A., Reilly, E. and Hemmendinger, D. (2003). Encyclopedia of computer science. 4th ed.