Applications of Induction & Recursion in Computer Science and Mathematics

Donald Wheeler

IST 230 Section 002

Dr. Mathias Fonkam

December 12th, 2024

Abstract: The purpose of this paper is to relate the topics of induction and recursion to real world applications in the fields of mathematics and computing. These concepts provide knowledge that is foundational for designing efficient and elegant algorithms. This paper will provide definitions and histories of induction and recursion, explain why they are important to understand, and overview some benefits and drawbacks of recursive algorithms in a computing context.

Table of Contents

Introduction to Induction and Recursion

Induction and recursion are closely related concepts in the field of mathematics and computer science that allow for powerful computations that can be expressed simply. The terms have often been conflated, as they are very similar. Since induction was used first historically and can be seen as a precursor to recursion, it will be defined first. Mathematical induction works on sequences of numbers and aims to prove a theorem about those numbers. It starts by proving that the theorem is true for the smallest value in the sequence. This first proof is known as the 'base case'. Induction says that if you can prove the theorem for the next term in the sequence, then the theorem holds true for any number in the sequence. Proving that the theorem applies to the next term in the sequence is known as the 'inductive step'. Induction can be very useful in proving mathematical statements that would otherwise be very intense or time-consuming proofs. Rather than trying to prove that a large sequence of numbers all share the same property by tackling each number individually, you can prove it by mathematical induction using just a few steps.

Induction is the mathematical basis from which a recurrence relation is formed. A function defined by induction generates a term from the previous terms in the sequence. This sequence of terms generated by a function defined by induction is known as a recurrence relation. A classic example of such a sequence is the Fibonacci sequence, in which the current term in the sequence is defined as the sum of the previous two terms.

Recursion, a word stemming from the same term "recur" as recurrence relations, is a technique in computing that is built from the principles of induction. Since terms in a recurrence relation are dependent on previous terms, there must be at least one term defined, the base case. The base case can be defined without knowledge of any other cases, and it allows subsequent terms to be defined. In the case of the Fibonacci sequence, there are 2 base cases: The first term is 0, and the second term is 1. These allow any subsequent terms to be generated, such as the third term, 0+1=1, the fourth term, 1+1=2, and so on. The calculation for the cases beyond the base case(s), in this example sequence the third term and beyond, is called the recursive step. This recursive definition of the Fibonacci sequence can be programmed into a computer and will generate any number of terms of the Fibonacci sequence. If time and computing resources weren't limiting factors, the same algorithm could generate an infinite number of terms.

In computer science, recursion manifests as a program function that calls itself reflexively. The principles of mathematics are used to design a recursive algorithm. The function will reflexively call itself (the recursive step) until a defined base case is reached. If a base case is not defined or never reached, then the function will call itself over and over, in a concept called infinite recursion. Properly defined recursive functions allow programmers to break complex problems down into simple steps, allowing for elegant solutions to be crafted. Both induction and recursion are tools in the arsenal of mathematicians and computer scientists which can create simple, elegant solutions to daunting problems.

The History of Induction and Recursion

There are many people throughout history who have used the principles of mathematical induction, so it is difficult to credit any one person. The first known to use induction explicitly in a proof was Arab mathematician named al-Karaji, who used it to prove an early form of the binomial theorem. He also used it to generate a table of binomial coefficients that was a precursor to what we now know as Pascal's triangle (Yadegari, 1980). The popularization of the concept of induction in the West if often credited to an interaction between Swiss mathematician Jakob Bernoulli and English mathematician John Wallis. John Wallis coined the phrase "per modum inductionis" when investigating why the sum of the ratio of integer squares seemed to approach 1/3. He would continually use each next integer square ratio rather than variables. Bernoulli noted that Wallis' process of induction was incomplete, and that mathematical induction should start from a number n and then prove an argument holds for $n + 1$ as well. (Cajori, 1918)

Many mathematicians used recursion in the mathematical sense, that is to define a function by induction, well before it occurred as a concept in computing. Mathematicians Richard Dedekind and Giuseppe Peano worked extensively to develop and understand recursively defined functions. This work would serve as a foundation for recursion's eventual role in computer programming (Computability Source). Recursion was first used in programming with the advent of Backus-Naur Form (BNF) notation. This syntax, invented by Backus and modified by Naur to realize its full potential, allowed for much simpler definitions which could utilize recursion. This syntax was used in the programming language ALGOL60, and helped eliminate problems of tedious definitions in previous languages. For example, an

integer could be defined recursively as a digit (the base case), or an integer concatenated with a digit (Daylight, 2007). Computer scientist Dijkstra later implemented dynamic memory management into an ALGOL60 compiler, in an effort to reach his goal of giving any program procedure the ability to call any other program procedure. This in turn allowed the compiler to handle recursive procedure calls. At this point, recursion in programming was still very contentious, as it was a technique many saw as prioritizing the simplicity of language too much over the physical limitations of a machine (Daylight, 2007).

Applications

Induction is a powerful tool for proving statements in mathematics. It is used most often to prove properties that hold true for all natural numbers. One of the most common uses of induction is in proving summation statements. Perhaps the most popular proof by mathematical induction shows that the sum of all natural numbers from 1 to number *n* is equivalent to the formula '(n(n + 1)) / 2'. The equation reached is a useful shortcut for calculating such sums, as adding each term individually could quickly become tedious. Another common use of induction in mathematics is to prove inequalities. An example of a fact proven by this is that for any number *n* equal to 4 or greater, 2 to the power of *n* is greater than or equal to *n* squared (inequality source). Induction can also be used to prove divisibility statements. An example of this is that the equation '*n^2 + n*' is divisible by 2 for all natural numbers.

Recursion is a technique that has grown to be very popular in the field of computer science. Swiss computer scientist Niklaus Worth said on the matter, "The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions." (Wirth, 1976). One reason recursive algorithms gained popularity is because they allowed certain programs to be written in simple, elegant language. Recursion can actually replace control structures like 'for' and 'while' loops entirely. An example of this is the factorial function. One can approach the factorial function from an iterative standpoint, using a for loop to multiply every number from 1 to n. In a recursive approach, you would first define a base

case for factorial(*n)*, where 0! and 1! are equal to 1. For cases that *n* is greater than 1, you would simply multiply *n* by factorial(*n* - 1*)*. The factorial function will call itself until it reaches the base case.

A drawback of using recursive algorithms is that they can often lead to an inefficient use of machine resources. An example of this is if we revisit the Fibonacci function. A recursive function can be defined to generate numbers of the Fibonacci sequence up to a number *n,* with 2 base cases, where Fibonacci(0) = 0 and Fibonacci(1) = 1. The recursive call can then intuitively be written as equal to 'Fibonacci(*n* – 1) + Fibonacci(n-1)'. This function will produce the intended output but begins to consume a lot of resources as the number *n* grows. This is because for every recursive call branch into two more calls until a base case is reached. The program will end up redundantly calculating factorials. The runtime of the program will grow exponentially proportional to *n*, which is horribly inefficient. An iterative implementation of the Fibonacci function can utilize variables that cache previous values, eliminating the redundant calculations that cause such inefficiency.

A popular use for recursion in programming is as a medium for designing divide-and-conquer algorithms. These algorithms break down complex problems into simpler sub-problems of the same type. Once the sub-problems are simple enough to have reached a base case, they will be recursively solved and recombined in order to solve the more complex problems.

The most widely used divide-and-conquer algorithm is binary search. Binary search is a method of searching for an element within a sorted array, such as an array of numbers, or an array of strings ordered lexicographically. If the element is found, binary search returns the array index it was found at. A base case needs to be defined if the element is found, ending the recursion. The algorithm begins by checking the middle element of the sorted array, which will be at the index *n*/2 for an array of length *n*. If the element checked is the one being searched for, then the algorithm terminates, returning this index. If the element is not found, the algorithm can recurse one of two ways- if the element checked is

greater than the element searched for, recursion will occur on the left half of the array (from index 0 to index $n/2 - 1$). If the element checked is less than the element being searched for, recursion will occur on the right half of the array (from index $n/2$ to index n – 1). If a recursive call occurs in which the lower index exceeds the higher index, this means that all array space has been exhausted, therefore the element does not exist in the array. In this case, often times programmers will choose to return a dummy index like '-1' that indicates the sought element was not found. Binary search is much more efficient than linear search, with a time complexity of O(log n). The worst-case scenario for a search algorithm, the case in which the element isn't present in the array, is much worse for a linear search. For example, in an array of 1 billion elements, the linear search will end up iterating 1 billion times. The binary search algorithm will recurse a maximum of 30 times, proving to be impressively more efficient.

Sorting arrays of elements is extremely useful in programming, demonstrated by the fact that powerful functions like binary search only work on sorted arrays. Sorting is similarly important as a tool for efficient operation of many operating system utilities and search engines. Because of the importance of sorting, it has become a heavily researched topic in the field of computer science. The two most efficient general sorting algorithms computer scientists devised are merge-sort and quicksort. Both of the two are divide-and-conquer algorithms that utilize recursion to efficiently sort elements.

Merge-sort works by taking an array and breaking it into two sub-arrays, both about half the size of the original array. These sub-arrays are then recursively broken down into further sub-arrays, until a base case is reached where the length of the array is 1. When this occurs, merge-sort then merges the sub-arrays back together in sorted order, recurring until all elements have been merged back into the full array, now in sorted order. Merge-sort runs with a worst-case time complexity of O(n log n), as opposed to the other general sorting algorithms that run in worst-case O(n^2) time (Dasgupta et. al., 2006). This makes merge-sort on average significantly more efficient than the other general sorting algorithms.

The quicksort algorithm uses recursion similarly to merge-sort to divide arrays into smaller parts. In quicksort, first a pivot element is chosen from the array. There is a lot of debate about what pivot should be used, but many people prefer to use the median of the first, last, and middle elements of the array. Once the pivot is selected, a helper method is used to partition the array so that all of the elements less than the pivot appear to the left of it, and all of the elements greater appear to the right. This partition function iterates linearly through the list, meaning it runs in worst-case O(n) time. Recursive calls are then made on the new left and right partitions of the array. A base case is reached when the length of a partition is 1, meaning no elements are left to be sorted. (geeks quick) Unlike merge sort, in quicksort elements are sorted as the array is broken down rather than when it is merged back together. Quicksort runs with worst case O(n^2) time complexity, a figure that is worse than merge-sort, but it runs on average in O(n log n) time complexity and can perform more efficiently than merge-sort in many scenarios.

Recursion is also used in defining and traversing the data structure known as the 'tree'. The tree data structure is named so because its visual representation resembles the real-life big plant of the same name. The tree data structure can provide for very effective data storage and management for a variety of different purposes. A tree structure consists of points of data called nodes or vertices. The base step to defining a tree structure is to define the root node. This root node alone can be considered a tree. From here, you can then define nodes as 'children' of this root node. The root node will in turn have the property of being the 'parent' of the leaf nodes. The definition of child nodes is recursive, as each child can be considered its own tree, and in turn be the parent to its own child nodes. A child node that has no children of its own does not recur into another tree and is termed a 'leaf' node (Subero, 2020). An example of a practical example of a tree data structure is a computer OS file system. A file system consists of a 'root' folder, such as 'C:' on Windows or simply '/' on Unix based systems like MacOS and Linux. This root folder can contain child nodes in the form of files or subfolders. These subfolders themselves can recursively contain their own subfolders. Files, which do not contain subfolders trees, can be considered the leaf nodes.

The main benefit of storing data in tree structures is that algorithms exist to traverse tree structures with remarkable efficiency. The two most favored tree-traversal algorithms are Breadth-First Search (BFS), and Depth-First Search (DFS). These algorithms differ in the paths they take to visit all nodes of a tree structure. The two are generally the most efficient for traversing trees, and they outperform each other depending on the type of tree that is being traversed.

Breadth-First Search aims to visit every node on one level of a tree before visiting any nodes on deeper levels. BFS does not use recursive function calls, instead implementing a queue. The algorithm starts by enqueuing the root node of the tree. Next, all nodes in the queue are visited, checking to see if the target element is present. After visiting each node, its children are added to the queue, and the visited node is dequeued. These steps repeat until all nodes in the tree have been queued and dequeued. While this algorithm does not rely on recursive function calls, it takes advantage of the recursive definition of a tree to continually add child nodes to the queue so that every node is visited in the search. Breadth first search performs efficiently when traversing a tree that is shallow, meaning it does not have many levels, but contains many nodes on each level.

Depth-First Search aims to explore every path of edges in the tree as deep as they go before moving on to the next path. DFS typically is implemented recursively. Each node is treated as its own tree which the algorithm recurses. There are 3 different ways in which the DFS algorithm can traverse a tree, all utilizing different order of the same 3 steps: visiting the current node, recursing on the left subtree of the current node, and recursing on the right subtree of the current node. These 3 DFS types are known as In-Order, Pre-Order, and Post-Order traversal. Pre-Order traversal, for example has the root node visited first. After this, the algorithm recurses on the left subtree, and then finally the right subtree. Generally, the left subtrees are traversed before right subtrees and changing the order in which the root node is visited changes the DFS type. In-Order traversal has the root visited in between the recursive calls for the left and right sub-trees. Post-Order traversal has the root being visited after both the recursive calls for the left and right sub-trees. This allows a DFS algorithm to be changed quite simply. DFS is efficient for traversing trees that have many

levels, with relatively few nodes populating each level. The algorithm is very useful in finding paths between nodes, as unlike BFS the algorithm follows a chain of parent-child relationships.

The Depth-First Search algorithm has extensive uses beyond trees, also being used to explore other types of graphs. It can be used to solve puzzles such as mazes, as it explores every possible path until the end is found. Traveling through a maze exploring different paths until a dead end is hit can in fact be a good way to visualize how DFS functions. This explorative function can also be used in networking to find possible paths to send information. DFS can also be used for detecting cycles in graphs. This can help spot infinite loops of travel that can have consequences in real applications such as computer networks, where cycled information may never reach its intended destination (GeeksforGeeks, 2024).

Contributions and Challenges

Many mathematicians have made contributions to the concept of mathematical induction over the years. One of the earliest contributions to the concept of induction can be traced back to Plato, who used it implicitly in a recorded philosophical dialogue known as *Parmenides*. He uses it to prove that the number of contacts in a chain of terms must always be one less than the number of terms (Acerbi, 2000). As mentioned before, Persian mathematician al-Karaji contributed by using induction in a mathematical context to prove properties of arithmetic sequences. French mathematician Blaise Pascal formulated his own idea of induction while studying the binomial coefficients, a recurrent series of numbers he arranged into what is now known as ‘Pascal’s Triangle’ (Lodder, 2017). Bernoulli helped popularized the use of induction, as he used the technique extensively in his popular works. British mathematician Augustus De Morgan, known well for his eponymous laws used in propositional logic, was also the first to coin the term ‘mathematical induction’ and propose a formal definition in his article *Induction* in the 12th edition of the London *Penny Cyclopedia*. De Morgan proved as an example that the sum of all odd numbers up to a certain number always results in a square. He further says that the root of this square will be equal to half of the even number next following the final odd integer.

(Kolpas, 2019). Many people were introduced to the concept of induction as an investigative tool by De Morgan's article on the subject.

One of the big challenges in induction is applying the technique incorrectly. One of the ways people will do this is by starting with a base case that is a statement, rather than a value. Hungarian American mathematician George Polya raised the paradox that 'all horses are the same color', which he claimed to have used mathematical induction to have proved. Clearly in a set of one horse, all will be the same color (Rao & Sae, 2009). The problem is when the inductive step was used to prove the theorem for a set of two horses- the proof implicitly relied on the existence of a third horse. This improper use of the inductive step is what led to the fallacy. Induction in general can be a hard concept for students to learn. In particular the inductive step can be confusing, and for many solutions it takes a strong foundation of algebra to be able to finish a proof.

As mentioned in the history section, John Backus and Peter Naur contributed greatly to the popularity of recursion by utilizing it to define terms in Backus-Naur Form. Two programming languages from the late 1950's, ALGOL 60 and LISP were the first to allow recursion, with LISP being designed specifically around it (Dimoulas, 2016). Dutch computer scientist Edsger Dijkstra coined the concept of dynamic programming in 1956 and designed compilers with dynamic memory management that allowed for functions call themselves recursively. In 1959 he also published the widely studied recursive algorithm known simply as 'Dijkstra's Algorithm' that can be used to find the shortest path from one vertex to another on a graph. This algorithm is used widely throughout applications reliant on networking to find efficient routes for information to travel (CWI Amsterdam, n.d.). Hungarian computer-scientist John von Neumann invented the aforementioned merge-sort algorithm in 1945 (Alake, 2022). British computer scientist Tony Hoare is credited with devising the Quicksort algorithm while studying at Moscow State university (Stanford, 2008). These discoveries and those of many others led to recursion becoming the important and widely studied computer science topic that it is today.

While recursion can be an incredibly powerful tool in a computer programmer's arsenal, recursive programs can still lead to many problems. Recursion, like induction, can be a difficult concept to grasp for many of those new to computer programming. Analysis of the LISP

language, which lacks unbounded iteration, has proven that any program can be designed recursively, that is to say recursive-only programming is Turing-complete (Old Dominion University, n.d). However, it can quickly become cumbersome to accomplish certain tasks with recursion that may be trivial to implement iteratively. Debugging recursive functions is often more challenging as well, as they require navigating function calls in the stack. Iterative programming generally comes more intuitively to new students.

Another problem that plagues many recursive functions is that they can often cause an inefficient use of computer resources. As demonstrated by the Fibonacci sequence recursive implementation, recursive functions can end up recalculating solutions that have already been solved, wasting stack memory with redundant function calls. Many recursive variants of algorithms share the problem of relatively inefficient use of the stack. This can lead to a stack overflow, a dangerous error in which most compilers will terminate the program for security reasons. Tail-recursion, where the recursive statement is the very last statement executed in the function, can be used to optimize many functions to combat the inefficient use of the call stack. However, this makes the process of designing the recursive function more complicated. As mentioned in the introduction, infinite recursion is another example of a common error made in implementation, caused by the lack of a base case or a failure to move towards the base case. Most programming environments have adapted by setting a guard rail known as maximum recursion depth to limit resource waste caused by such errors.

Conclusion

This report has demonstrated the worth of both induction and recursion as powerful tools to be used by mathematicians, logicians and computer scientists. Induction has a long and ambiguous history and serves as the foundation upon which the more recent concept of recursion is based. As somebody studying a computer science adjacent major in cybersecurity, I have encountered recursive algorithms many times already and so I am particularly interested in furthering my study of this topic. Understanding recursion can reveal the inner workings of many programs and lead to a better comprehension of the program stack as well.

References

Acerbi, F. (2000). Plato: Parmenides 149a7-c3. A Proof by Complete Induction? Archive for History of Exact Sciences, 55, 57–76. https://www.academia.edu/8016024/Plato_Parmenides_149a7_c3_A_Proof_by_Complete_Induction

Alake, R. (2022, March 31). *Merge Sort Explained: A Data Scientist's Algorithm Guide*. NVIDIA Developer. https://developer.nvidia.com/blog/merge-sort-explained-a-data-scientists-algorithm-guide/

Cajori, F. (1918). Origin of the Name "Mathematical Induction." *The American Mathematical Monthly*, *25*(5), 197–201. https://doi.org/10.2307/2972638

Chilimath. (n.d.). Mathematical Induction for Summation. Chilimath. https://www.chilimath.com/lessons/basic-math-proofs/mathematical-induction/

ChiliMath. (n.d.). *Mathematical Induction for Divisibility*. ChiliMath. https://www.chilimath.com/lessons/basic-math-proofs/mathematical-induction-for-divisibility/

CWI Amsterdam. (n.d.). *Edsger W. Dijkstra: Brilliant, colourful, and opinionated*. CWI Amsterdam. https://www.cwi.nl/en/about/history/e-w-dijkstra-brilliant-colourful-and-opinionated/

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2006). *Algorithms (Chap. 2)* (1st ed.). McGraw-Hill Education. https://people.eecs.berkeley.edu/~vazirani/algorithms/chap2.pdf

Daylight, E.G. (2007). The Advent of Recursion in Programming, 1950s-1960s. The Journal of Neuroscience.

Dimoulas, C. (2016). *From LISP and ALGOL 60 to Interpreters and Abstract Machines*. Harvard CS 252. https://www.cwi.nl/en/about/history/e-w-dijkstra-brilliant-colourful-and-opinionated/

Downey, A. B. (2012). *How to Think Like a Computer Scientist: C++ Version* (1st ed.). Green Tea Press. https://runestone.academy/ns/books/published/thinkcpp/Chapter4/InfiniteRecursion.html

GeeksForGeeks. (2023, January 10). *Why is Tail Recursion optimization faster than normal Recursion?*. GeekforsGeeks. https://www.geeksforgeeks.org/why-is-tail-recursion-optimization-faster-than-normal-recursion/

GeeksForGeeks. (2024, December 12). *Quick Sort*. GeekforGeeks. https://www.geeksforgeeks.org/quick-sort-algorithm/

GeeksForGeeks. (2024, May 15). *Applications, Advantages and Disadvantages of Depth First Search (DFS)*. GeekforGeeks. https://www.geeksforgeeks.org/applications-of-depth-first-search/?ref=lbp

GeeksForGeeks. (2024, November 13). *Factorial of a Number*. GeeksforGeeks. https://www.geeksforgeeks.org/program-for-factorial-of-a-number/

GeeksForGeeks. (2024, September 23). *Applications of Dijkstra's shortest path algorithm*. GeekforGeeks. https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/

GeeksforGeeks. (2024, September 4). *Binary Search*. GeekforGeeks. https://www.geeksforgeeks.org/binary-search/

GeeksForGeeks.(2024, February 19). *BFS vs DFS for Binary Tree*. GeekforGeeks. https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/

Kolpas, S. J. (2019). *Mathematical Treasure: Augustus De Morgan and Mathematical Induction*. Mathematical Association of America. https://old.maa.org/press/periodicals/convergence/mathematical-treasure-augustus-de-morgan-and-mathematical-induction

Lodder, J. (2017). *Pascal's Triangle and Mathematical Induction*. Digital Commons @ Ursinus College. https://digitalcommons.ursinus.edu/cgi/viewcontent.cgi?article=1005&context=triumphs_number

MyTutor. (n.d.). *Prove by induction that n! > n^2 for all n greater than or equal to 4.*. MyTutor. https://www.mytutor.co.uk/answers/20541/A-Level/Further-Mathematics/Prove-by-induction-that-n-n-2-for-all-n-greater-than-or-equal-to-4/

Old Dominion University. (n.d.). *Turing Completeness CS390, Fall 2024*. Old Dominion University. https://www.cs.odu.edu/~zeil/cs390/latest/Public/turing-complete/index.html

Rao, S., & Tse, D. (2009). *CS 70 Fall 2009 Course Notes - Induction*. Stanford. https://old.maa.org/press/periodicals/convergence/mathematical-treasure-augustus-de-morgan-and-mathematical-induction

Stanford. (2008). *Quicksort*. Stanford. https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/tony-hoare/quicksort.html

Subero, A. (2020). *Codeless Data Structures and Algorithms* (1st ed.). Apress Berkeley, CA. https://link.springer.com/book/10.1007/978-1-4842-5725-8

Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice Hall.

Yadegari, M. (1980). The binomial theorem: A widespread concept in medieval Islamic mathematics. Historia Mathematica, 7(4th ed.), 401-406. https://www.sciencedirect.com/science/article/pii/031508608090004X\