Recursion: A Deep Dive Into Recurrence Relations and Recursive Functions

Drake L. Austin
University of Central Florida
COT3100: Introduction to Discrete Structures
Dr. Arup Guha
December 2, 2024

Abstract

Recurrence relations and recursive functions play a critical role in both mathematics and computer science, defining sequences based on preceding terms. This paper investigates the foundations of recurrence relations, exploring their mathematical and computational significance. Beginning with a general overview, this research examines recurrence relations and their uses in combinatorics, counting, and other statistical problems, as well as their application in time complexity analysis and dynamic programming. This study aims to demonstrate how recursive methods generate systematic solutions, fostering a deeper understanding of algorithm design and problem-solving approaches in both theoretical and practical contexts.

1 Introduction

The definition of recursion is something that will probably differ depending on who you ask because of the complexities behind the topic. The way I define recursion is the process of solving a problem in terms of previous or alternate versions of the problem itself. On that note, in the realm of mathematics and computer science, recurrence relations are tools used to define functions in terms of itself. The concept of recurrence relations is vital to math and computer science because the development and management of algorithms, data structures, and many mathematical models rely on them. There are many applications of recurrence relations, including finding algorithmic time complexities, predicting financial growth, and other theoretical problem-solving tactics that will be seen throughout the entirety of this paper.

I was inclined to pick recurrence relations as my topic because of my experiences on my high school math competition team. I learned through competitions that recurrence relations can be used to simplify the steps it takes to solve complex processes, I always struggled to find the right recursive functions to use to make complex AIME (American International Mathematics Examination) counting problems simpler. It frustrated me that one of the major question types that prevented me from placing in multiple competitions were these counting problems I mentioned, so I eventually built up enough of a resolve to attempt to research these recurrence relations that were haunting me. Although I did halt my research due to fear from looking at the complexity of Bessel functions, the Nicholson–Bailey model [6], and other intimidating recurrence relations, a quote from an AoPS (Art of Problem Solving) handout that I came across recently rekindled my interest in the topic - "To understand recursion, we must first understand recursion" [7]. This quote took me by surprise because it made something that I struggled with in the past sound so trivial, and honestly I had a good laugh reading it.

Throughout this paper, I want to discover the uses and importance of recurrence relations for the purpose of discovering the connection between mathematical functions and computational practices, and finding uses for these relations in computer science and computational theory.

2 Diving Into Recurrence Relations

As I've mentioned previously, a recurrence relation is an equation that defines a sequence based on previous terms in that sequence. Recurrence relations are used to describe a process in terms of its past values, breaking down complex problems and making them simpler. Formally, a recurrence relation for a sequence $\{a_n\}$ is an equation that expresses a_n in terms of earlier terms such as a_{n-1}, a_{n-2}, etc., along with some initial conditions to fully define the sequence [1]. In mathematics, recurrence relations are fundamental in areas such as combinatorics and discrete mathematics, especially for problems that involve iterative structures or recursive processes.

One classic example of a recurrence relation is the one defining the Fibonacci sequence:

$$F_n = F_{n-1} + F_{n-2}$$

This recurrence describes each term in the sequence as the sum of the previous two terms. The sequence begins $0, 1, 1, 2, 3, 5, 8, \ldots$, and each term depends on the two terms before them. This relation is particularly useful because it provides a simple recursive structure for calculating terms in the Fibonacci sequence.

Below I've written a program that uses recursion to get an output of the first n terms of a periodic sequence with initial conditions $F_0 = 0$ and $F_1 = 1$ in hopes that the code will aid readers in visualizing where the recursion occurs and how it works.

```python
def fibonacci(n):
    fib_sequence = [0, 1]

    for i in range(2, n):
        next_term = fib_sequence[i - 1] + fib_sequence[i - 2]
        fib_sequence.append(next_term)

    return fib_sequence[:n]

n = int(input("Enter the number of terms: "))
print(f"The first {n} terms of the Fibonacci sequence are: {fibonacci(n)}")
```

Fibonacci numbers - numbers in the Fibonacci sequence described by my program - are often seen in nature and can describing natural phenomena such as the family tree of a bee colony and the spirals on a sunflower [3]. Additionally, the recurrence relation for the Fibonacci sequence is useful in various fields, from analyzing biological patterns, such as population growth in idealized settings, to algorithm analysis. For example, this recurrence is fundamental in algorithmic analysis when studying the complexity of recursive algorithms, particularly in dynamic programming. By understanding the recurrence structure of a problem, we can often derive closed expressions, allowing more efficient computations. [4, 5].

There are some other noteworthy recurrence relations outside of the Fibonacci Sequence. Factorials can be represented by the recursive function $F(n) = F(n-1) * n, n \geq 1$ for $F(0) = 1$, and are often see across all of mathematics - from high school Pre-Calculus handouts to Poisson Distributions in advanced probability theory. Additionally, the Towers of Hanoi sequence [3] $(F(n) = 2F(n-1) + 1)$ is a well-known sequence in the mathematics community that's used by many instructors around the world to introduce people to the idea of recursion in mathematics.

Recurrence relations allow us to describe complex processes in terms of their foundational patterns, which is vital in both theoretical computation and applied mathematics. They facilitate problem-solving in structured steps, making them an essential tool in computer science for designing efficient algorithms, particularly those that optimize processes by reusing previously computed results (as seen in dynamic programming). This approach reduces redundant calculations, saving computational resources and time, especially in scenarios where resources are limited.

In summary, recurrence relations help enclose the recursive nature of many mathematical and computational problems, enabling simplification, optimization, and analysis in a range of practical applications.

3 Solving Recursion Problems

Now that I've introduced recurrence relations, I'd like to present some recursion problems from Dylan Yu's AOPS handout [7] and go through the process of solving each one.

Binary Sequences

How many binary sequences (sequences of 0's and 1's) of length 19 are there that begin with a 0, end with a 0, contain no two consecutive 0's, and contain no three consecutive 1's?

A.
Instead of saying that the binary sequences should be of length 19, we should generalize this problem and find out how many binary sequences of length n that there are.
Let a_n be the number of valid sequences of length n that there are.
We know that our sequence must end with a 0. Since we can't have two consecutive 0s in this sequence, we know that our sequence must end in "10". The digit before 10 must be either 0 (making our final digits "010") or 1 (making our final digits "110"). However, since there cannot be three consecutive 1s, our final digits of the "110"-ending sequence are actually "0110".
In the first scenario where the final digits of our sequence are "010", we know that the first $n-2$ digits are a valid sequence of length $n-2$. In the second scenario where the final digits of our sequence are "0110", we know that the first $n-3$ digits are a valid sequence of length $n-3$. Finally, we know that the length of these binary sequences must be at least 3 in order to have at least one valid combination, since our sequence must start and end with a 0 (2 digits) and there can't be any consecutive 0s (1 more digit in-between).
Using this information, we can make a recursive function $a_n = a_{n-2} + a_{n-3}, n \geq 6$, where $a_3 = 1$ because "010" is the only valid sequence of length 3, $a_4 = 1$ because "0110" is the only valid sequence of length 4, and $a_5 = 1$ because "01010" is the only valid sequence of length 5.
The terms of this sequence a_n are as follows:

a_3	a_4	a_5	a_6	a_7	a_8	a_9	...	a_{18}	a_{19}
1	1	1	1+1=2	1+1=2	1+2=3	2+3=5	...	21+28=49	28+37=65

Since $a_{19} = 65$, there are **65** valid binary sequences of length 19 that follow the conditions listed in the problem.
In order to help you all visualize how the recursive functions work, I have made computer programs for each problem in this section of paper. Here is the program for finding the amount of binary sequences of length n that exists and begin with a 0, end with a 0, contain no two consecutive 0's, and contain no three consecutive 1's.

```python
def binarySequences(n):
    binary_sequence = [0, 0, 0, 1, 1, 1]

    if n < len(binary_sequence):
        return binary_sequence[n]
    else:
        for i in range(6, n+1):
            next_term = binary_sequence[i - 2] + binary_sequence[i - 3]
            binary_sequence.append(next_term)

    return binary_sequence[n]

n = int(input())
print(f"The amount of valid binary sequences of length {n} that exists is {
    binarySequences(n)}.")
```

A and B Strings

Find the number of 9-letter strings that contains only the letters "A" and "B" and do not have more than 3 adjacent letters that are identical.

A.
Let a_n be the number of n-letter strings in which each letter is either an A or a B and no more than 3 adjacent letters are identical.
We know that at the end of each n-letter string, there must be a substring of length 1,2, or 3 of a the same letter - either A or B - at the end of a valid string of length $n-1$, $n-2$, or $n-3$, respectively. So we can determine that $a_n = a_{n-1} + a_{n-2} + a_{n-3}, n \geq 4$. We know that $a_1 = 2$ (2^1 string possibilities of length 1), $a_2 = 4$ (2^2 string possibilities of length 2), and $a_3 = 8$ (2^3 possibilities of length 3).
Now we can use our recursive function to determine our answer.

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
2	4	8	2+4+8=14	4+8+14=26	8+14+26=48	14+26+48=88	162	298

Since $a_9 = 298$, we can conclude that there are **298** 9-letter strings where each letter is an "A" or a "B" and no more than three consecutive letters are the same.
Below is the code for finding number of valid n-letter strings:

```python
def abstring(n):
    str = [1, 2, 4, 8]

    if n < len(str):
        return str[n]
    else:
        for i in range(4, n+1):
            next_term = str[i-1] + str[i - 2] + str[i - 3]
            str.append(next_term)

    return str[n]

```

```python
n = int(input())
print(f"The amount of valid strings of length {n} is {abstring(n)}.")
```

Mail Carrier

A mail carrier delivers mail to the twenty-two houses on the east side of Elm Street. The carrier notices that no two adjacent houses ever get mail on the same day, but that there are never more than two houses in a row that get no mail on the same day. How many different patterns of mail delivery are possible?

A.
Let's create a sequence a_n which represents the number of ways the mailman can delivery mail to n Elm Street houses on a given day (also known as the number of valid arrangements of the n-character binary string that we just created).
There are two scenarios to this problems. Let's start by assuming the first of the n houses receives mail. If the first house receives mail, then the second house will not, since no two consecutive houses can both receive mail. Now make the mailman make deliveries to the last $n-3$ houses on the east side of Elm Street. We can see that the option of mail being delivered to the third house (which we skipped over) depends on the fourth house (if the fourth house received mail then the third house didn't receive mail, and if the fourth house did not receive mail then the third house must've received mail). There are a_{n-3} to determine the number of ways that the mail is delivered from the fourth house to the final house in this scenario.
Now let's assume that the first house does not receive mail. Make the mailman deliver mail to the last $n-2$ houses on the east side of Elm Street. We can see that the option of mail being delivered to the second house (which we skipped over) depends on the third house (if the third house received mail then the second house didn't receive mail, and if the third house did not receive mail then the second house must've received mail). There are a_{n-2} to determine the number of ways that the mail is delivered from the third house to the final house in this scenario.
Adding the two scenarios together, we can confirm a recursive function of $a_n = a_{n-2} + a_{n-3}, n \geq 4$, where $a_1 = 2$ (first house gets mail or it doesn't), $a_2 = 3$ (first house gets mail and second doesn't, second house gets mail and first doesn't, or neither house gets mail), and $a_3 = 4$ (first house gets mail while second and third don't, first and third house get mail while second doesn't, first and third house don't get mail while second does, or first and second don't get mail while third does).
Using our recurrence relation, we eventually get that there are $a_22 =$ **816** ways that the mailman can deliver mail to 22 houses on the east side of Elm Street.
Below is the code to find number of valid ways that the mailman can deliver to n houses on the east side of Elm Street:

```python
def mailRoutes(n):
    mailways = [0, 2, 3, 4]

    if n < len(mailways):
        return mailways[n]
    else:
        for i in range(4, n+1):
            next_term = mailways[i - 2] + mailways[i - 3]
            mailways.append(next_term)

    return mailways[n]

n = int(input())
print(f"The amount of ways that the mailman can deliver to {n} houses is {
    mailRoutes(n)}.")
```

Ring Painting

There is a ring made of seven small sections which you are to paint on a wall. You have four paint colors available and will paint each of the six sections a solid color. Find the number of ways you can choose to paint each of the six sections if no two adjacent section can be painted with the same color?

A.
Let a_n be the number of ways you can choose to paint n sections if no two adjacent sections can be painted with the same color.
If there's one section on the ring, there are 4 different colors that section could be painted, so $a_1 = 4$. If there's two sections on the ring, we can choose $2! \times \binom{4}{2} = 12$ ways to paint both sections, so $a_2 = 12$. If there's three sections on the ring, we can choose $3! \times \binom{4}{3} = 24$ to paint the sections, so $a_3 = 24$.
Now that we've discovered the starting terms of a_n, let's develop a recurrence relation for the sequence. We will color in the first $n-1$ sections in clockwise order.
If the first section and the $n-1$th section are the same color, we could say that there are a_{n-2} ways to color the first $n-1$ sections of the ring (it's a_{n-2} rather than a_{n-1} because the first and the $n-1$th section are the same color so we must remove one of the section to make a circle with no identical colors next to each other). In this situation, there are 3 possible colors that the nth section could be.
If the first section and the $n-1$th section are different color, we could say that there are a_{n-1} ways to color the first $n-1$ sections of the ring. In this situation, there are 2 possible colors that the nth section could be.
From these observations, we can determine the sequence to be $a_n = 3 * a_{n-2} + 2 * a_{n-1}$ for $n \geq 4$. Using the sequence, the number of ways to color n sections of the ring without any bordering colors being the same is $a_7 = \mathbf{2184}$.
Below is the code to find the number of valid ways to color n sections on this ring:

```
def ringColors(n):
    sections = [0, 4, 12, 24]

    if n < len(sections):
        return sections[n]
    else:
        for i in range(4, n+1):
            next_term = 2*sections[i - 1] + 3*sections[i - 2]
            sections.append(next_term)

    return sections[n]

```

```
n = int(input())
print(f"The number of valid ways that there are to color {n} sections of this
    ring is {ringColors(n)}.")
```

Dominos

How many ways are there to tile an 13 by 2 board with 1 by 2 dominoes such that each domino covers exactly two squares and no domino overlaps?

A.
Let's determine a_n, the number of ways that there are to tile a n by 2 board with 1 by 2 dominos. We'll start by determining some base cases. There is only one way to tile a 1 by 2 board with 1 by 2 dominos, so $a_1 = 1$. There's two ways to tile a 2 by 2 board with 1 by 2 dominos (either horizontally or vertically), so $a_2 = 2$.
There are two scenarios that can occur for this sequence: Either you fill up the first $n-1$ by 2 squares of the board with dominos so you can finish the board with one vertically placed 1 by 2 domino, or you fill up the first $n-2$ by 2 squares of the board with dominos so you can finish the board with two horizontally placed 1 by 2 dominos.
Using this information, we find that $a_n = a_{n-1} + a_{n-2}$ for $n \geq 3$. Therefore there are $a_{13} = 377$ ways to tile a 13 by 2 board without any overlaps.
Below is the code for the ways to fill up the n by 2 board with 1 by 2 dominos without overlap:

```
def dominoBoard(n):
    dominos = [0, 1, 2]

    if n < len(dominos):
        return dominos[n]
    else:
        for i in range(3, n+1):
            next_term = dominos[i - 1] + dominos[i - 2]
            dominos.append(next_term)

    return dominos[n]

n = int(input())
print(f"The number of valid ways that there are to tile an {n} by 2 with 1 by
    2 dominos is {dominoBoard(n)}.")
```

4 Recursion in Computer Science

As someone that has dealt with these equations in AIME and other various math competitions, it was hard not to what other purposes that these insanely vast equations had in the real world besides figuring out answers to combinatorics and counting problems. Lucky for me, I found in my research that my field of computer science is a field that heavily utilizes recursive functions.

Recursion is a fundamental concept in computer science - leveraging a divide-and-conquer approach and allowing for solutions to complex problems. Recursive algorithms are defined in terms of base cases, which initiate the recursion, and recursive cases, which break down the problem as we have seen throughout this paper. One of the primary strengths of recursion is its ability to provide clear and concise solutions for problems involving hierarchical or self-similar structures. According to [1], recursion is particularly powerful in algorithm design for problems like divide-and-conquer algorithms, such as Merge Sort and Quick Sort, exploring tree-like structures in search algorithms, and other problems that involve searching for items within a large data structure.

Furthermore, recursion is widely used in dynamic programming. Problems solved using dynamic programming often involve overlapping sub-problems that can initially be expressed recursively and later optimized using memoization or tabulation [5]. This recursive approach is key to solving complex optimization problems efficiently. Recursion also supports many real-world applications, such as graph traversal algorithms (e.g., depth-first search) and parsing in compilers. The uses of recursion in handling self-referencing structures like linked lists and trees is also highlighted in [4], making it a crucial tool to use for data structure manipulation.

Despite its elegance, recursion must be used with caution in programming. Poorly designed recursive functions can lead to excessive memory usage and stack overflow errors. Understanding and analyzing recurrence relations, as discussed in [2], is critical to ensuring the efficiency and correctness of recursive algorithms.

In conclusion, recursion is a versatile and powerful tool in computer science, with applications spanning algorithms, mathematical computation, and real-world systems. Its ability to simplify complex problems makes it an essential concept for both theoretical and practical domains.

5 Conclusion

In conclusion, recurrence relations and recursive functions are the keys to bridging theory and mathematical computations. By defining sequences and solutions based on preceding terms, they provide a systematic and elegant approach to problem-solving. This paper has demonstrated their foundational role in diverse areas such as combinatorics, statistical analysis, and algorithm design. Through these applications, recurrence relations not only simplify complex problems but also reveal patterns and structures that might otherwise remain hidden. Their use in dynamic programming and time complexity analysis shows their ability to optimize processes, ensuring both efficiency and clarity in computational tasks.

Recurrence relations, when applied in fields like combinatorics, enable counting and enumeration methods that are vital for statistical modeling. Meanwhile, in computer science, they offer insights into the behavior and efficiency of algorithms, guiding developers toward creative solutions in programming and systems design. Recursive methods teach us to decompose problems, a skill that is valuable in theoretical pursuits and real-world applications ranging from scheduling to data analysis.

Future research could go into advanced forms of recursion, including non-linear and multi-variable recurrence relations, to address increasingly complex challenges in computational fields. Additionally, the integration of recursive techniques in emerging areas such as artificial intelligence and big data analytics holds promise for driving innovation. As technology evolves, the principles of recursion will undoubtedly continue to inspire new ways of thinking, problem-solving, and advancing the fields of both mathematics and computer science.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed., MIT Press, 2009.

[2] GeeksforGeeks. "Recurrence Relations - A Complete Guide." Accessed November 12, 2024. `https://www.geeksforgeeks.org/recurrence-relations-a-complete-guide/`.

[3] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. 2nd ed., Addison-Wesley, 1994.

[4] Massachusetts Institute of Technology. "Mathematics for Computer Science - Chapter 10." MIT OpenCourseWare, 2010. Accessed November 12, 2024. `https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-fall-2010/18b5495f7408055f9679e3afebb108ab_MIT6_042JF10_chap10.pdf`.

[5] Stanford University. "Guide to Dynamic Programming." CS 161: Algorithms. Accessed November 12, 2024. `https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/handouts/140%20Guide%20to%20Dynamic%20Programming.pdf`.

[6] Wikipedia. "Nicholson-Bailey Model." Accessed November 12, 2024. `https://en.wikipedia.org/wiki/Nicholson%E2%80%93Bailey_model`.

[7] Dylan Yu. "Recursion in the AMC and AIME." Accessed November 12, 2024. `https://yu-dylan.github.io/euclid-orchard/Handouts/Recursion_in_the_AMC_and_AIME.pdf`.