Jinping, S. (2013). Discussion on Writing of Recursive Algorithm. *International Journal of Hybrid Information Technology*, *6*(6), 127-134.

Introduction

This paper provides an introduction to recursive algorithms and techniques for writing them effectively. Recursion is an important programming concept where a function calls itself recursively to solve smaller instances of a problem. Mastering recursion allows programmers to write elegant solutions for complex problems by breaking them down into simpler sub-problems.

The paper is well-organized and starts with an explanation of the key features of recursive programs - the need for a recursive exit condition and dividing the problem into smaller sub-problems. It then covers two main techniques for developing recursive algorithms - using a mathematical formula and mathematical induction. Examples are provided to demonstrate these techniques for computing factorial and Fibonacci numbers. The strengths and weaknesses of simple linear recursion are analyzed, and tail recursion is presented as an optimization strategy. The paper concludes by summarizing the importance of learning recursive programming.

Explanation of Recursive Programming Concepts

The author does a good job of explaining what recursion is and how recursive functions work. Two key characteristics of recursion are highlighted:

1. Recursive exit condition: This defines the base case where the recursion stops. Without this, the function would call itself indefinitely.

2. Dividing into sub-problems: The main problem is broken down into smaller sub-problems, and the function calls itself recursively to solve these sub-problems.

Simple examples of computing factorial and Fibonacci sequence are used to illustrate these concepts. The factorial example shows how the exit condition handles the base case of n=1. The Fibonacci example divides the problem into computing the (n-1)th and (n-2)th numbers to build up the sequence.

These examples clearly demonstrate the conceptual framework underlying recursion - breaking a complex problem down into simpler cases and incrementally building up the solution. The author explains how recursion works at a high-level, avoiding complex details at this introductory stage.

Techniques for Writing Recursive Algorithms

The paper covers two major techniques for developing recursive algorithm:

1. Formula method: Derive a mathematical formula that defines the problem recursively. Implement this directly in code.

2. Mathematical induction: Use induction to prove the algorithm works for the base case and inductive step. Develop code mirroring these steps.

The formula method is illustrated by deriving the recursive formulas for factorial and Fibonacci numbers. The code directly implements these mathematical definitions. This technique is straightforward but only applicable when such formulas can be formulated.

Mathematical induction is demonstrated on the classic Tower of Hanoi problem. The base case of moving one disk is defined. The inductive step assumes we can move N disks and uses this to prove we can move N+1 disks. The code mirrors this reasoning. This structured approach helps methodically develop the recursive logic.

These techniques provide a general framework for designing algorithms recursively rather than thinking about the lower-level details. More examples demonstrating these methods on other problems could further strengthen this section. Overall, it equips readers with two major tools for tackling recursion in a systematic way.

Analyzing Recursive Algorithms

The advantages and disadvantages of linear recursion are examined. Its elegance and simplicity in expressing solutions is highlighted. However, large input sizes can result in excessive stack usage and poor performance.

To address these inefficiencies, tail recursion optimization is explained. Keeping state between calls avoids repeated calculations. Modifying the Fibonacci example to a tail recursive form shows how this works. The optimized version has constant stack space and can handle larger inputs.

This analysis provides important insights into the practical limitations and costs of naive recursion. Tail recursion is presented as a technique to optimize recursive code for better efficiency and scalability. Showcasing the stack usage and runtime improvements for the Fibonacci example could further illustrate the benefits. Additional examples of tail recursive functions would also be helpful for the reader.

Conclusion

The paper concludes by summarizing that recursive programming, despite its difficulties, is an important technique that allows elegant and concise algorithm expression. Mastering its concepts and methodologies is key for any programmer.

The author has provided a focused introduction to recursive programming. The concepts are explained clearly with simple examples. Mathematical formula derivation and induction are presented as systematic techniques for developing algorithms recursively. The costs of linear recursion and tail recursion optimization are analyzed. More examples and details could enhance some sections, but overall it covers the core aspects in a concise and approachable manner for the reader. This serves as a solid foundation for further exploring this important programming paradigm.

to Demonstrate Concepts

Sum of numbers from 1 to n:

Formula method:
$S(n) = 1 + 2 + 3 + ... + n$
$\quad = n + S(n-1)$

Code:

```
int sum(int n) {
  if (n == 1) return 1;
  return n + sum(n-1);
}
```

Mathematical induction:

Base case: $S(1) = 1$
Inductive step: Assume $S(k) = 1 + 2 + ... + k$
$S(k+1) = S(k) + (k+1)$

Code:

```
int sum(int n) {
   if (n == 1) return 1;
   int prev = sum(n-1);
   return prev + n;
}
```

Tower of Hanoi (additional example for mathematical induction):

```
Base case: Move 1 disk directly
Inductive step: To move N+1 disks -
        Move N disks to spare peg
        Move 1 disk to target peg
        Move N disks from spare to target
```

Code:

```
void hanoi(int n,Peg start,Peg end,Peg spare){
  if(n==1) {
    moveDisk(start,end);
    return;
  }

  hanoi(n-1,start,spare,end);
  moveDisk(start,end);
  hanoi(n-1,spare,end,start);
}
```