Principles of Recursive Program Design

Designing Recursive Algorithms

• Find the key step.
  – How can this problem be divided into parts?
  – How will the key step in the middle be done?
  – Avoid ending with multitude of special cases.

• Find a stopping rule.
  – This stopping rule is usually the small case that is easy to handle without recursion.

• Outline your algorithm.
  – Combine the stopping rule and the key step, using an if statement to select between them.
Designing Recursive Algorithms
(Continued..)

• Check termination.
  – Verify that the recursion will always terminate.
  – Be sure that your algorithm correctly handles extreme cases.

• Draw a recursion tree.
  – The height of the tree is closely related to the amount of memory that the program will require,
  – the total size of the tree relates to the number of times the key step will be done.

Implementing Recursion

• Implementation is separate from design.
  – Implementation can be in any language.

• Multiple Processors:
  – Processes that take place simultaneously are called concurrent.

• Single Processor:
  – can use multiple storage areas with a single processor.

• Re-entrant Programs
Improving Recursive Programs: The case of Tail Recursion
Tail Recursion

**Definition** Tail recursion occurs when the last-executed statement of a function is a recursive call to itself.

If the last-executed statement of a function is a recursive call to the function itself, then this call can be eliminated by reassigning the calling parameters to the values specified in the recursive call, and then repeating the whole function.

Removal of Tail Recursion

![Removal of Tail Recursion Diagram](image-url)
Can Tower of Hanoi be simplified?

/* Move: moves count disks from start to finish using temp for temporary storage. */
void Move(int count, int start, int finish, int temp)
{
    if (count > 0) {
        Move(count-1, start, temp, finish);
        printf("Move a disk from %d to %d.\n", start, finish);
        Move(count-1, temp, finish, start);
    }
}

Iterative Tower of Hanoi

void Move(int count, int start, int finish, int temp)
{
    int swap; /* temporary storage to swap towers */
    while (count > 0) {
        Move(count - 1, start, temp, finish);
        printf("Move %d from %d to %d.\n", count, start, finish);
        count--;
        swap = start;
        start = temp;
        temp = swap;
    }
}
Can n-queen be simplified?

void AddQueen(void)
{
    int col; /* column being tried for the queen */
    queencount++;
    for (col = 0; col < BOARDSIZE; col++)
        if (colfree[col] && upfree[queencount + col] &&
            downfree[queencount - col + DOWNOFFSET])
        {
            /* Put a queen in position (queencount, col). */
            queen[col][queencount] = col;
            colfree[col] = FALSE;
            upfree[queencount + col] = FALSE;
            downfree[queencount - col + DOWNOFFSET] = FALSE;
            if (queencount == BOARDSIZE-1) /* termination condition */
                Flashback:
            WriteBoard();
            else
                AddQueen(); /* Proceed recursively. */
        }
    colfree[col] = TRUE; /* Now backtrack by removing the queen. */
    upfree[queencount + col] = TRUE;
    downfree[queencount - col + DOWNOFFSET] = TRUE;
    queencount--;
}

Flashback: AddQueen()
Demo:
Time Comparison of two N-Queen Solutions

Queennr.exe

Should we Always use Recursion?
Case: Factorials

- Recursive Version:
  ```c
  /* Factorial: recursive version.*/
  int Factorial(int n)
  {
    if (n == 0)
      return 1;
    else
      return n * Factorial(n-1);
  }
  ```

- Iterative Version:
  ```c
  /* Function: iterative version.*/
  int Factorial(int n)
  {
    int count, product;
    for (product = 1, count = 2; count <= n; count++)
      product *= count;
    return product;
  }
  ```

Which one will work harder?
Fibonacci Numbers

- Fibonacci Number
  
  \[
  F_0 = 0 \\
  F_1 = 1 \\
  F_n = F_{n-1} + F_{n-1} \quad \text{when} \quad n \geq 2 
  \]

A Cute Solution

```c
int Fibonacci(int n)  
{  
  if (n <= 0)  
    return 0;  
  else if (n == 1)  
    return 1;  
  else  
    return Fibonacci(n-1) + Fibonacci(n-2);  
}
```

Efficiency?
**Recursion Tree**

The growth is nearly exponential!

---

**Iterative Fibonacci**

```c
/* Fibonacci: iterative version. */

int Fibonacci(int n) {
    int i;
    int twoback; /* second previous number, F_{i-2} */
    int oneback; /* previous number, F_{i-1} */
    int current; /* current number, F_i */
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else {
        twoback = 0;
        oneback = 1;
        for (i = 2; i <= n; i++) {
            current = twoback + oneback;
            twoback = oneback;
            oneback = current;
        }
        return current;
    }
}
```
Comparison: Iteration vs. Recursion

- Chain
- Duplicate Task
- Change Data Structures
- Recursion Removal

Guidelines

- If the recursion tree has a simple form, the iterative version may be better.
- If the recursion tree involves duplicate task, then data structures other than stacks will be appropriate.
- If the recursion tree appears quite bushy, with little duplicate tasks, then recursion is likely the natural solution.