Chapter 4

Recursion

Principle of Recursion

- A recursion is a process of defining a problem (or solution to a problem) in terms of (a simple version of) itself.
- A **recursive algorithm** is one expressed in terms of itself. In other words, at least one step of a recursive algorithm is a “call” to itself.
- In programming, a **recursive function** is one that calls itself. We use recursive function to implement recursive algorithms in programming languages
- A recursive algorithm is more elegant and easier to understand but it is less efficient (extra calls consume time and space).

Principle of Recursion

- Given a recursive algorithm, how can we sure that it terminates?
- The algorithm must have:
  - one or more “easy” cases or stopping conditions
  - one or more “hard” cases or recursive calls
- In a stopping condition, the algorithm must give a direct answer without calling itself.
- In a recursive call, the algorithm may call itself, but only to deal with an “easier” case.

Example: Factorial

- **Algorithm:**

n! = 1 if n = 0

n! = n * (n – 1)! If n > 0

- **C-Function:**

```
int factorial(int n)
{
        if(n == 0)
                return 1;
        else
                return n * factorial(n - 1);
}
```

Example: Fibonacci Number

- **Algorithm:**

fib(n) = 0 if n = 1

fib(n) = 1 if n = 2

fib(n) = fib(n – 1) + fib(n – 2) If n > 2

- **C-Function:**

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

Example: Greatest Common Divisor

- **Algorithm:**

gcd(p, q) = q if q exactly divides p

gcd(p, q) = gcd(q, p mod q) if q does not exactly divides p

- **C-Function:**

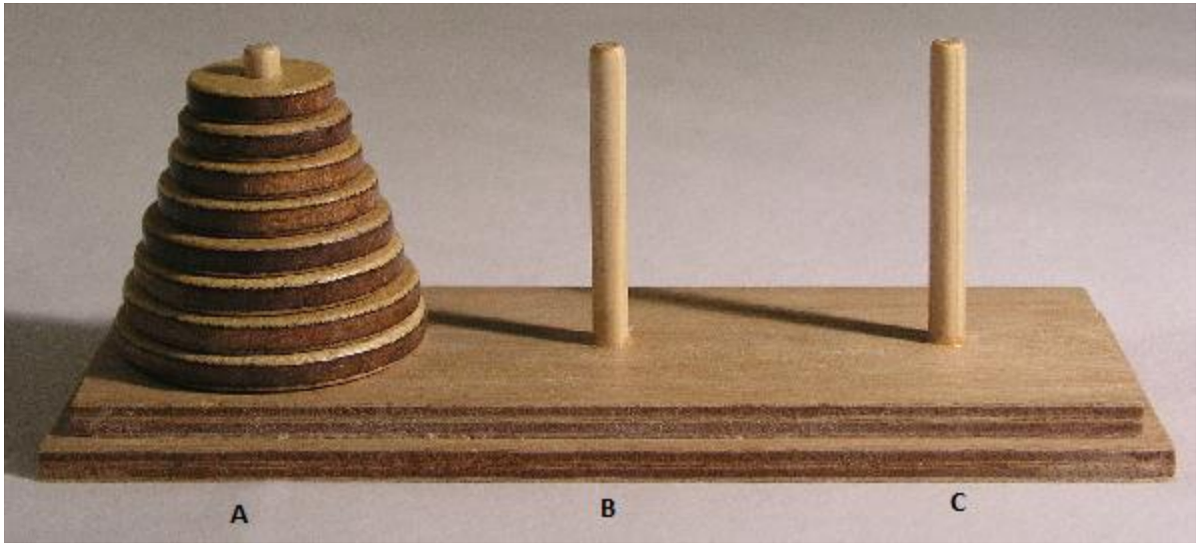
```
int gcd(int p, int q)
{
        if(p % q == 0)
                return q;
        else
                return gcd(q, p % q);
}
```

Example: Tower of Hanoi

- Three vertical poles (A, B, C) are mounted on a platform.
- A number of differently-sized disks are threaded on to pole A, forming a tower with the largest disk at the bottom and the smallest disk at the top.
- We may move **one** disk at a time, from any pole to any other pole, but we must never place a larger disk on top of a smaller disk.
- Problem: Move the tower of disks from pole A to pole B.

Example: Tower of Hanoi

- **Algorithm:** To move n disks from A to C, using B as auxiliary
  1. If n = 1, move the single disk from A to C and stop.
  2. Move the top n - 1 disks from A to B using C as auxiliary.
  3. Move the remaining disk from A to C.
  4. Move then n - 1 disks from B to C using A as auxiliary.



Example: Tower of Hanoi

- **C-Function:**

```
void towers(int n, char frompeg, char topeg, char auxpeg)
{
        if(n == 1)
        {
                printf("\nMove disk 1 from peg %c to peg %c", frompeg, topeg);
                return;
        }
        towers(n - 1, frompeg, auxpeg, topeg);
        printf("\nMove disk %d from peg %c to peg %c", n, frompeg, topeg);
        towers(n - 1, auxpeg, topeg, frompeg);
}
```

Recursion vs. Iteration

- **Recursion:**
  - Recursion uses selection structure.
  - Infinite recursion occurs if the recursive step does not reduce the problem in a manner that converges on some condition (base case) and Infinite recursion can crash the system.
  - Recursion terminates when a base case is recognized.
  - Recursion is usually slower than iteration due to the overhead of maintaining the stack.
  - Recursion uses more memory than iteration.
  - Recursion makes the code smaller.

Recursion vs. Iteration

- **Iteration:**
  - Iteration uses repetition structure.
  - An infinite loop occurs with iteration if the loop condition test never becomes false and Infinite looping uses CPU cycles repeatedly.
  - An iteration terminates when the loop condition fails.
  - An iteration does not use the stack so it's faster than recursion.
  - Iteration consumes less memory.
  - Iteration makes the code longer.

Efficiency of Recursion

- Nonrecursive version of a program executes more efficiently in terms of time and space than a recursive version because extra calls in recursive programs consume time and space
- Overhead involved in entering and exiting a block is avoided in the nonrecursive version
- In a nonrecursive program stacking activity can be eliminated
- Sometimes a recursive solution is the most natural way of solving a problem because recursive solution flows directly from the recursive definitions

Tail Recursion

- A recursive function is tail recursive if there is nothing to do after the function returns except return its value
- The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by the compiler
- As there is no task left after the recursive call, it will be easier for the compiler to optimize the code; So, storing addresses into stack is not needed
- Since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use

Tail Recursion

- **C-Program:**

```
#include<stdio.h>
int facto(int n, int a)
{
        if(n == 0)
                return a;
        else
                return facto(n - 1, n * a);
}
```

```
int main()
{
        int n;
        printf("n = ");
        scanf("%d",&n);
        printf("%d",facto(n, 1));
}
```