DATA STRUCTURES AND ALGORITHMS

Recursion

Summary of the previous lecture

- Complexity of the algorithms
  - Running time
  - Number or basic operations
- Asymptotic analysis of the algorithms
  - Upper asymptotic bound (big-Oh)
  - Lower asymptotic bound (big-Omega)
  - Big-Theta
- Empyrical analysis of the algorithms

Recursion

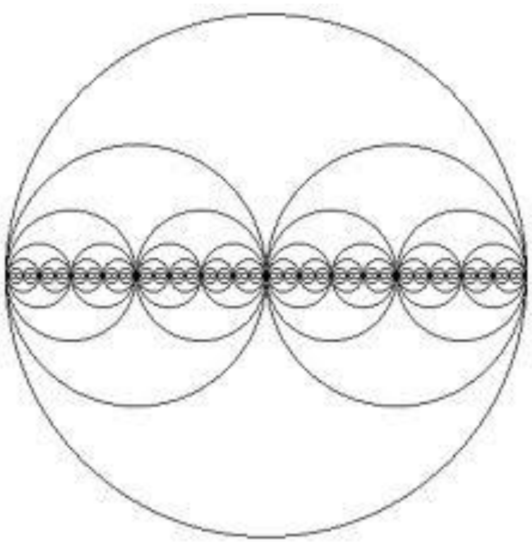
Recursion is the process of repeating items in a self-similar way.

For instance, when the surfaces of two mirrors are exactly parallel with each other the nested images that occur are a form of infinite recursion.

The term has a variety of meanings specific to a variety of disciplines ranging from linguistics to logic. The most common application of recursion is in mathematics and computer science.

Recursion refers to a method of defining functions in which the function being defined is applied within **its own definition**.

Examples of the recursion



Recursion

- The **power of recursion** 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.

Most *high-level computer programming languages* support recursion by **allowing a function to call itself** within the program text.

- Some *functional programming* languages do not define any looping constructs but rely on recursion to repeatedly call code. Computability theory has proven that these recursive-only languages can solve the same kinds of problems even without the typical control structures like "**while**" "**until**" and "**for**".

Recursion

An algorithm is recursive if it calls itself to do part of its work.

A recursive algorithm must have two parts:

- the **base case**, which handles a simple input that can be solved without resorting to a recursive call;
- the **recursive part** which contains one or more recursive calls to the algorithm.

Example - Factorial

A classic example is the recursive definition for the **factorial** function:

$$n! = (n - 1)! * n \qquad \text{for } n > 1; \quad 1! = 0! = 1.$$

```
long Fact(int n)
{
    if (n < 1)
        return 1;            // Base case: returns the base solution
    return n * Fact(n-1);    // Recursive call for n > 1
}
```

Factorial

fact(4)

$n4 = 4 * n3$

$= 4 * 3 * n2$

$= 4 * 3 * 2 * n1$

$= 4 * 3 * 2 * 1 * n0$

$= 4 * 3 * 2 * 1 * 1$

$= 4 * 3 * 2 * 1$

$= 4 * 3 * 2$

$= 4 * 6$

$= 24$

Computational time:

$T(n) = T(n - 1) + 1 = (T(n - 2) + 1) + 1$

$(T(n - 2) + 1) + 1 = T(n - 2) + 2$

$T(n) = T(n - (n - 1)) + (n - 1)$

$= T(1) + n - 1$

$= n - 1$

Iterative code for the factorial

```
int fact_2(n)
{   int factorial = 1;
    if (n <= 1)
        return 1;
    for (int i = 1 ; i <= n ; i++)
        factorial = factorial * i;
    return factorial;
}
```

Recursion features

Depth of recursion is the longest chain of procedure that calculate function F(n) in recursive way.

Recursion doesn’t differ from normal procedure when function are used. It means, that **variables are local** and are valid just inside the single procedure.

Example

```
#include <stdio.h>

int sum (int num)
{
    if (num==0)
        return 0;
  return sum(num-1)+(num);
}
```

```
int main()
{
    int num = 10;
    printf("%d\n", sum(num));
    getchar();

return 0;
}
```

Example

```
#include <stdio.h>
void Triangle (int x)
{
  if (x <= 0)
        return;
  Triangle(x - 1);

  for (int i = 1; i <= x; i++)
     printf("*");
     printf("\n")
}
```

```
int main() {
  Triangle(7);
  return 0;
}
```

Trace of the program

```
Triangle(7)
 Triangle(6)
  Triangle(5)
   Triangle(4)
    Triangle(3)
     Triangle(2)
      Triangle(1)
       Triangle(0) <-- base case
      Triangle(1) <-- prints 1 star & new line
     Triangle(2) <-- prints 2 stars & new line
    Triangle(3) <-- prints 3 stars & new line
   Triangle(4) <-- prints 4 stars & new line
  Triangle(5) <-- prints 5 stars & new line
 Triangle(6) <-- prints 6 stars & new line
Triangle(7) <-- prints 7 stars & new line
```

Fibonacci numbers

In mathematics, things are often defined recursively. For example, the Fibonacci numbers are often defined recursively.

The **Fibonacci** numbers are defined as the sequence beginning with two 1's, and where each succeeding number in the sequence is the sum of the two preceeding numbers.

1 1 2 3 5 8 13 21 34 55 89 ...

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{if } n >= 2 \end{cases}$$

Recursive program

```
#include <stdio.h>

int fib(int);
int N = 10;

void main()
{  int Fnumber;
   for (int i = 0; i < N; i++)
   {   Fnumber = fib(i);
       printf("%d\n", Fnumber);
   }
}
```

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

statement	time $n<2$	$n \geq 2$
3	$O(1)$	$O(1)$
4	$O(1)$	--
6	--	$T(n\text{-}1)+T(n\text{-}2)+O(1)$
TOTAL	$O(1)$	$T(n\text{-}1)+T(n\text{-}2)+O(1)$

Iterative code

```
int fib_2(n)
{
    if (n == 0 || n == 1)
        return 1;
    int fibprev = 1;
    int fib = 1;
    for (int i = 2 ; i < n ; i++)
    {
        int  temp = fib;
        fib += fibprev;
        fibprev = temp;
    }
    return fib;
}
```

Computation of running time

statement	time
5	$O(1)$
6	$O(1)$
7a	$O(1)$
7b	$O(1) \times (n+2)$ iterations
7c	$O(1) \times (n+1)$ iterations
9	$O(1) \times (n+1)$ iterations
10	$O(1) \times (n+1)$ iterations
11	$O(1) \times (n+1)$ iterations
13	$O(1)$
TOTAL	$O(n)$

Notice

Recursive function always has:

- **recursive part** (contains one or more recursive calls)
- **base case** (handles a simple input that can be solved without resorting to a recursive call)

For Fibonacci numbers, the *base case* is when n == 0 and n==1, then the program returns 1 without any further recursive calls.

Recursive programs must always have a base case!

Detail of recursive calculation

The computer will go through the following process to compute **fib(3):**

3 exceeds 1, so I need to compute and return **fib(3 - 1) + fib(3 - 2).**

To compute this, I first need to compute fib(2).
2 exceeds 1, so I need to compute and return **fib(2 - 1) + fib(2 - 2).**

To compute this, I first need to compute fib(1).
1 is less than or equal to 1, so I need to return **1**.

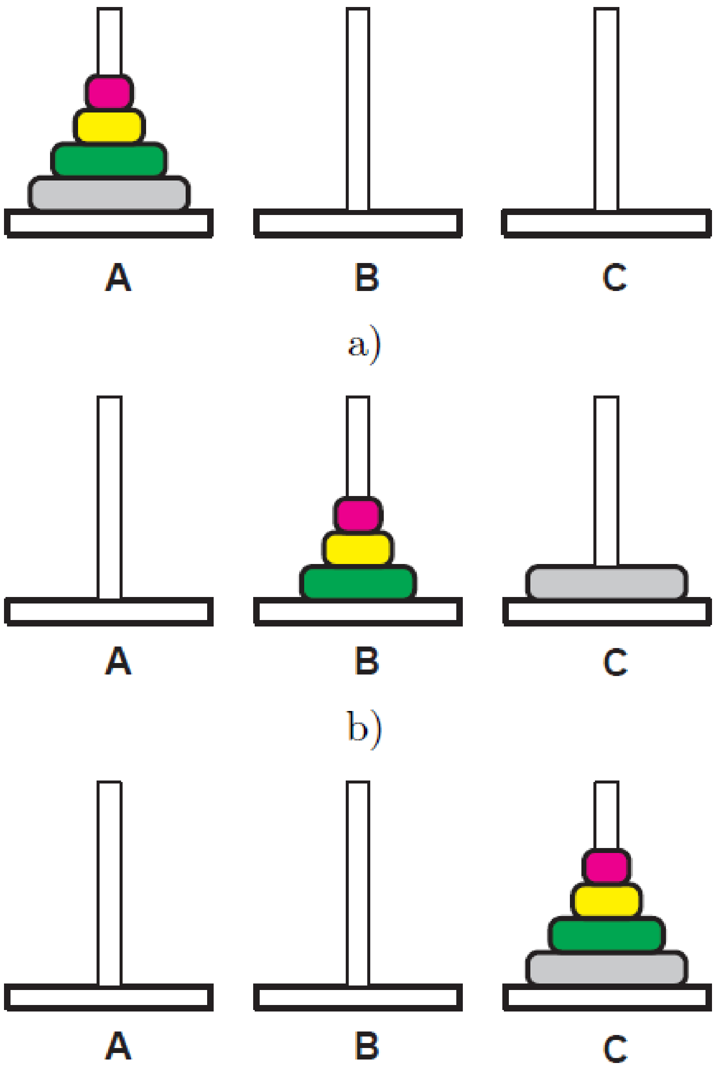
Now that I know fib(1), I need to compute fib(0). 0 is less than or equal to **1**, so I need to return **1**. Now I know fib(2 - 1) + fib(2 - 2) = 1 + 1 = **2**. I return this.
Now that I know fib(2) is **2**, I need to compute fib(1). 1 is less than or equal to 1, so I need to return **1**. I now know fib(3 - 1) + fib(3 - 2) = 2 + 1 = **3**. I return this.

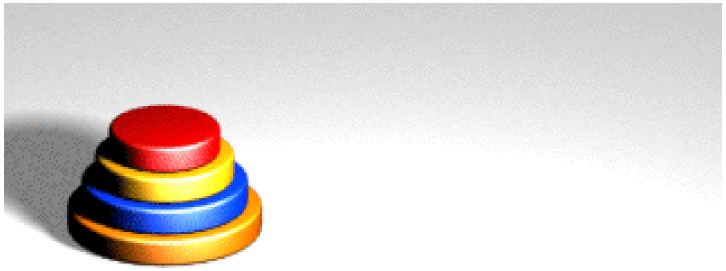
Towers of Hanoi

The Towers of Hanoi puzzle begins with **three poles** and **n** rings, where all rings start on the leftmost pole (Pole A). The rings each have a different size, and are stacked in order of decreasing size with the largest ring at the bottom. **The problem is to move the rings from the leftmost pole to the rightmost pole (Pole C) in a series of steps.** At each step the top ring on some pole is moved to another pole. There is one limitation on where rings may be moved: a ring can never be moved on top of a smaller ring.

“Towers of Hanoi” natural algorithm to solve this problem has multiple recursive calls. It cannot be rewritten easily using **while** loops.

Towers of Hanoi





The minimum number of moves required to solve a Tower of Hanoi puzzle is 2^n - 1, where *n* is the number of disks.

Towers of Hanoi

$$\text{hanoi}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \cdot \text{hanoi}(n-1) + 1 & \text{if } n > 1 \end{cases}$$

hanoi(4)

= 2*hanoi(3) + 1

= 2*(2*hanoi(2) + 1) + 1

= 2*(2*(2*hanoi(1) + 1) + 1) + 1

= 2*(2*(2*1 + 1) + 1) + 1

= 2*(2*(3) + 1) + 1

= 2*(7) + 1

= 15

Recursive code

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

Recursion in linked lists

```
struct node
{ int n;                    // some data struct
  node *next;               // pointer to another struct node
};
typedef struct node *LIST;

…..

void printList(LIST lst)

{ if ( ! isEmpty(lst) )           // base case

    { printf ("%d ", lst->n );    // print integer followed by a space

      printList ( lst->next );    // recursive call

    }

}
```

Recursion in binary tree

```
struct node
{   int n;                          // some data
    struct node *left;              // pointer to the left subtree
    struct node *right;             // point to the right subtree
};
typedef struct node *TREE;

// Inorder printout of the binary tree :
void printTree(TREE t)
{ if (!isEmpty(t)) {                 // base case
        printTree(t->left);          // go to the left
        printf("%d ", t->n);         // print the integer followed by a space
        printTree(t->right);         // go to the right
  }
}
```

Recursion or iteration?

Advantages of the recursion

- Convenient way to control sequence of tasks
- Simple code

Disadvantages of the recursion

- Stack overflow may appear
- Recursion can lead to not efficient way of the algorithm

Recommendation

Avoid to use recursion if you are not sure about problem size, use iterative procedure instead.

Homework

No.1

Write recursive function to determine if an input is prime number.

No.2

Write recursive functions to assign the particular values to the array, and printout array in reverse and in normal order.

No.3

Write recursive function to printout digits of the given number in reverse order i.e. 2015 – 5 1 0 2

Example

```
int isPrime (int p, int i=2)
{
        if (i == p) return 1;     // better  if (i*i > p) return 1;
        if (p%i == 0)
            return 0;
   return isPrime (p, i+1);
}
```