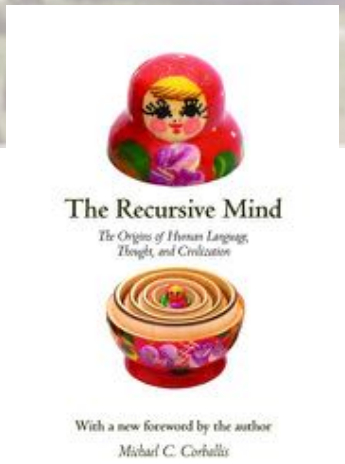
Lecture 7 - Recursion

Meng-Hsun Tsai
CSIE, NCKU



Recursion

- A function is ***recursive*** if it calls itself.
- This lecture gives five examples which can be solved by recursion:
  - Factorial
  - Power n of x
  - Fibonacci sequence
  - Hanoi Tower
  - Sudoku Solver

Factorial

- The following function computes *n*! recursively, using the formula *n*! = *n* × (*n* – 1)!:

```
int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

```
int fact(int n)
{
   int result = 1;
   for(int i = 2; i<=n; i++)
     result *= i;
   return result;
}
```

Non-recursive version

$$f(n) = \begin{cases} 1 & \text{if } n = 1, \\ n \times f(n-1) & \text{otherwise.} \end{cases}$$

Factorial (cont.)

- To see how recursion works, let's trace the execution of the statement

```
i = fact(3);
```

`fact(3)` finds that 3 is not less than or equal to 1, so it calls

`fact(2)`, which finds that 2 is not less than or equal to 1, so it calls

`fact(1)`, which finds that 1 is less than or equal to 1, so it returns 1, causing

`fact(2)` to return $2 \times 1 = 2$, causing

`fact(3)` to return $3 \times 2 = 6$.

Power n of x

- The following recursive function computes x^n, using the formula $x^n = x \times x^{n-1}$.

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

```
int power(int x, int n)
{
   int result = 1;
   for(int i = 0; i < n; i++)
     result *= x;
   return result;
}
```

Non-recursive version

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

Power `n` of `x` (cont.)

- We can condense the `power` function by putting a conditional expression in the `return` statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

- Both `fact` and `power` are careful to test a "termination condition" as soon as they're called.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

Fibonacci Sequence

- In **Fibonacci sequence**, each number is the sum of the two preceding ones.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

- Fibonacci numbers can be defined as the following equation:

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise.} \end{cases}$$

Fibonacci Sequence (cont.)

- The following recursive function computes the Fibonacci number F_x:

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

Try to write a non-recursive version on your own!!

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise.} \end{cases}$$

Hanoi Tower

- Hanoi Tower is a mathematical puzzle consisting of three rods and a number of disks of various diameters.
- The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules:
  - Only one disk may be moved at a time.
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
  - No disk can be placed on top of a disk that is smaller than it.

Hanoi Tower (cont.)

- The key to solving a problem recursively is to recognize that it can be broken down into a collection of smaller sub-problems.
- Each of these created sub-problems being "smaller" guarantees that the base case(s) will eventually be reached.
- Basic settings for the Towers of Hanoi:
  - label the pegs A, B, C,
  - let *n* be the total number of disks,
  - number the disks from 1 (smallest, topmost) to *n* (largest, bottom-most).

Hanoi Tower (cont.)

- To move *m* disks from a source peg to a *target* peg using a *spare* peg, without violating the rules:
  - Move *m* – 1 disks from the **source** to the **spare** peg. This leaves the disk *m* as a top disk on the source peg.
  - Move the disk *m* from the **source** to the **target** peg.
  - Move the *m* – 1 disks that we have just placed on the spare, from the **spare** to the **target** peg.

Hanoi Tower (cont.)

```
#include <stdio.h>
void hanoi_tower(int n,char a,char b,char c)
{
    if(n == 1){
        printf("[Disk %d] %c -> %c\n", n, a, c);
    }
    else{
        hanoi_tower(n-1,a,c,b);
        printf("[Disk %d] %c -> %c\n", n, a, c);
        hanoi_tower(n-1,b,a,c);
    }
}
int main()
{
    int n;
    printf("Input number of disks: ");
    scanf("%d",&n);
    hanoi_tower(n,'A','B','C');
    return 0;
}
```

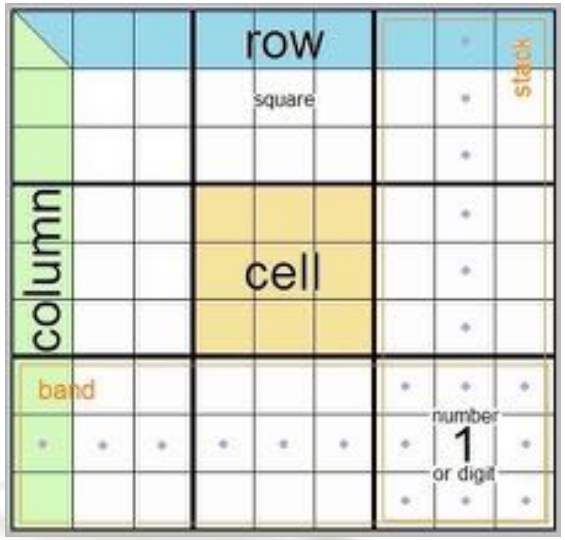
```
Input number of disks: 1
[Disk 1] A -> C

Input number of disks: 2
[Disk 1] A -> B
[Disk 2] A -> C
[Disk 1] B -> C

Input number of disks: 3
[Disk 1] A -> C
[Disk 2] A -> B
[Disk 1] C -> B
[Disk 3] A -> C
[Disk 1] B -> A
[Disk 2] B -> C
[Disk 1] A -> C
```

Sudoku Solver

- A Sudoku solver reads in a Sudoku question, and then solves the question.
- A Sudoku answer is a 9×9 grid filled with digits so that **each column**, **each row**, and **each of the nine 3×3 sub-grids (called cells)** that compose the grid contains all of the digits from 1 to 9.



Solving Sudoku Recursively

- The following program solves a sudoku problem recursively.
- Function `checkUnique(arr[])` checks if parameter `arr[]` contains all the digits from 1 to 9.
- Function `printMap(map[])` prints out `map[]`.
- Function `getFirstZeroIndex(map[])` obtains index of the first empty square (with number 0).
- Function `isCorrect(map[])` checks if `map[]` obeys all sudoku rules (i.e., `true` means `map[]` is the answer).
- Function `solve(question[])` solves the question by recursively trying all possible numbers at empty square.

Solving Sudoku Recursively (cont.)

```
#include <stdio.h>
#include <stdbool.h>
#define SUDOKU_SIZE  81

int ques[SUDOKU_SIZE];
int ans[SUDOKU_SIZE];
int main()
{    // read in map
    for(int i=0; i<SUDOKU_SIZE; ++i)
        scanf("%d", &ques[i]);

    if(solve(ques) == true)
    {
        printf("Solvable!\n");
        printMap(ans);
    }
    else
        printf("Unsolvable!!\n");

    return 0;
}
```

Input

```
8 0 5 3 2 0 4 1 7
2 0 3 1 7 5 8 6 9
1 9 7 6 8 4 5 0 3
3 1 9 0 5 8 6 7 4
4 2 6 0 9 1 3 5 8
5 7 8 4 3 0 1 9 2
7 5 4 9 1 3 2 0 6
6 8 2 5 4 0 9 3 1
9 3 1 8 6 2 7 0 5
```

Output

```
> ./sudoku_solve
Solvable!
8 6 5 3 2 9 4 1 7
2 4 3 1 7 5 8 6 9
1 9 7 6 8 4 5 2 3
3 1 9 2 5 8 6 7 4
4 2 6 7 9 1 3 5 8
5 7 8 4 3 6 1 9 2
7 5 4 9 1 3 2 8 6
6 8 2 5 4 7 9 3 1
9 3 1 8 6 2 7 4 5
```

```
1 2 3 0 5 6 7 8 9
1 2 3 4 5 6 7 8 0
1 0 3 4 5 6 7 8 9
1 2 3 4 5 0 7 8 9
1 2 3 4 5 6 7 0 9
1 2 0 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 0 7 8 9
0 2 3 4 0 6 7 0 9
```

```
>./sudoku_solve
Unsolvable!!
```

Solving Sudoku Recursively (cont.)

```
bool checkUnique(int arr[])
{
    int arr_unity[9];  // counters

    for(int i=0; i<9; ++i)
        arr_unity[i] = 0;  // initialize

    for(int i=0; i<9; ++i)
        ++arr_unity[arr[i]-1];  // count
    for(int i=0; i<9; ++i)
        if(arr_unity[i] != 1) // all element
            return false;     // must be 1
    return true;
}
```

```
void printMap(int map[])
{
   for(int j=0; j<SUDOKU_SIZE; j++)
   {
       printf("%d ", map[j]);
       if(j%9 == 8)
           printf("\n");
   }
}
```

```
int getFirstZeroIndex(int map[])
{
   for(int i=0; i<SUDOKU_SIZE; ++i)
      if(map[i] == 0)
         return i;
   return -1;
}
```

Solving Sudoku Recursively (cont.)

```
bool isCorrect(int map[])
{
   bool check_result;
   int check_arr[9];
   int location;
   for(int i=0; i<81; i+=9){
      // check rows
      for(int j=0; j<9; ++j)
         check_arr[j] = map[i+j];
      check_result =
         checkUnique(check_arr);
      if(check_result == false)
         return false;
    }
    for(int i=0; i<9; ++i){
      // check columns
      for(int j=0; j<9; ++j)
         check_arr[j] = map[i+9*j];
      check_result =
         checkUnique(check_arr);
      if(check_result == false)
            return false;
   }
   for(int i=0; i<9; ++i){
      // check cells
      for(int j=0; j<9; ++j) {
         location = 27*(i/3) +
           3*(i%3) +9*(j/3) + (j%3);
         check_arr[j] = map[location];
      }
      check_result ==
         checkUnique(check_arr);
      if(check_result == false)
         return false;
   }
    return true;
}
```

Solving Sudoku Recursively (cont.)

```
bool solve(int question[])
{
    int firstZero;
    int map[SUDOKU_SIZE];

    firstZero = getFirstZeroIndex(question);
    if(firstZero == -1) {
        // end condition
        if(isCorrect(question))
        {
          //answer = question;
          for(int i=0; i<SUDOKU_SIZE; i++)
              ans[i] = question[i];
          return true;
        }
        else
          return false;
     }
else {
    // copy question[] to map[]
    for(int i=0; i<SUDOKU_SIZE; i++)
      map[i] = question[i];


    for(int num=1; num<=9; ++num)
    {
        map[firstZero] = num;
        if(solve(map))
            return true;
    }
    return false;
  }
}
```

A Quick Review to This Lecture (cont.)

- A function is ***recursive*** if it calls itself.
- Recursion arises ***divide-and-conquer*** technique: a large problem is divided into smaller pieces that are tackled by the same algorithm.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.