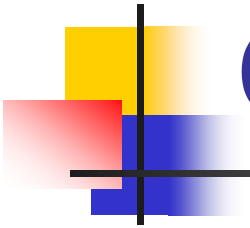
# CS 2213
# Advanced Programming
## Ch 5 – More Recursion examples


Turgay Korkmaz

Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

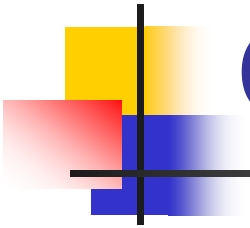


Thanks to Eric S. Roberts, the author of our textbook, for providing some slides/figures/programs.



# Objectives

- To become familiar with several classic examples of recursive programming, but we will just consider the problem of **generating permutations**.
- To recognize that recursion can sometimes offer **concise solutions** to problems that are difficult to solve by any other means.
- To appreciate how recursion can be applied to the problem of generating graphical displays.
  - Tower of Hanoi (Self-Study)
  - **Generating Permutations**
  - Graphical applications (Self-Study)



# Generating Permutations

- All possible arrangements of a particular set
- Write a function **ListPermutations(s)** that displays all permutations of the string **s.**
- For example, **ListPermutations(s);** should display the following six arrangements when **s** is **"ABC"**:

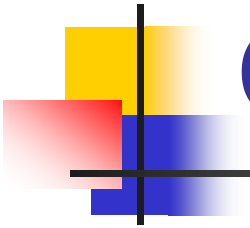
**ABC**
**ACB**
**BAC**
**BCA**
**CBA**
**CAB**

How would you implement the **ListPermutations function?**

❑ Iterative control structures: finding a general solution that works for strings of any length is difficult.

❑ Thinking about the problem recursively, on the other hand, leads to a relatively straightforward solution.

# The recursive insight for Generating Permutations



- The permutations of the five-character string **"ABCDE"** consist of the following strings:
  - The character **'A'** followed by every possible permutation of **"BCDE"**
  - The character **'B'** followed by every possible permutation of **"ACDE"**
  - The character **'C'** followed by every possible permutation of **"ABDE"**
  - The character **'D'** followed by every possible permutation of **"ABCE"**
  - The character **'E'** followed by every possible permutation of **"ABCD"**
- More generally, *take each of* the *n characters in turn and display that character followed by every possible permutation* of the remaining *n – 1 characters.*

# Difficulty in previous strategy

- The recursive sub-problem **does not** have exactly the same form as the original.
  - asymmetry
    - The original problem requires you to **display all permutations** of a string.
    - The sub-problem requires you to **display a character** from a string followed by **all permutations** of the remaining letters.
- Moreover, as the recursion proceeds, the character in front will become two characters, then three, and so forth.
- So *the general sub-problem is to generate all permutations of a string with some characters at the beginning of the string already fixed in their positions*.

# Solving asymmetry between the original problem and its recursive sub-problems

- **RecursivePermute(str, k)** generates all permutations of a string whose first k letters are fixed.
  - When k=0, all letters are free to change
  - As k increases, the problem becomes smaller
  - When k=length of string, there is no more interchange, print the string
- Define **ListPermutations** as a simple wrapper function that calls a subsidiary function to solve the more general case.

```
void ListPermutations(string str) {
   RecursivePermute(str, 0);
}
```

# RecursivePermute: Pseudocode form

```
void RecursivePermute(string str, int k)
{
  if (k is equal to the length of the string) {
    Display the  string.
  } else {
    For each character position i from k to the end of string {
      Exchange the characters in positions i and k
      Use recursion to generate permutations with the first k+1 characters fixed
      Restore the string by exchanging the characters in positions i and k
    }
   }
}
```

# RecursivePermute: Implementation in C

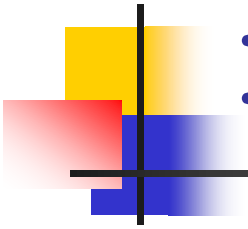
```
void RecursivePermute(string str, int k)
{
  int i;
  if (k == strlen(str)) {
    printf("%s\n", str);
  } else {
    for(i=k; i < strlen(str); i++) {
       ExchangeCharacters(str, k, i);
       RecursivePermute(str, k+1);
       ExchangeCharacters(str, k, i);
    }
  }
}
```

Can you think of any improvement?

What is the complexity?

```
void ExchangeCharacters(string str, int k, int i)
{
   char tmp;
   tmp = str[i];
   str[i] = str[k];
   str[k] = tmp;
}
```

# Exercise: Trace `ListPermutations`



- **`ListPermutations("KLMN")`**

# Exercise: Trace `ListPermutations`

- **`ListPermutations("KLMN")`**
- **`4! =4*3*2*1=24`**

BTW if you call it like this, you will get segmentation fault in C. Why?

How can you fix the problem?

Rest can be skipped

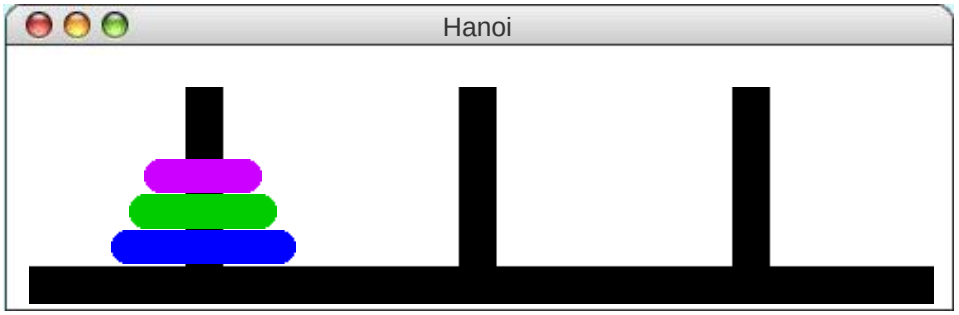
# SELF-STUDY

# The Towers of Hanoi Solution

```
int main() {
```

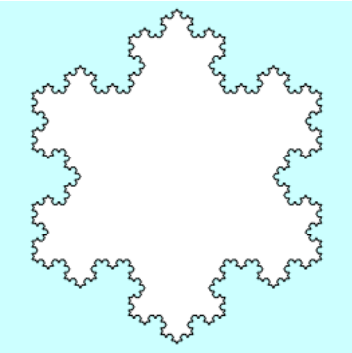
```
void MoveTower(int n, char start, char finish, char temp) {
   if (n == 1) {
      MoveSingleDisk(start, finish);
   } else {
      MoveTower(n - 1, start, temp, finish);
      MoveSingleDisk(start, finish);
      MoveTower(n - 1, temp, finish, start);
   }
}
```

| n | start | finish | temp |
|---|---|---|---|
| 3 | 'A' | 'B' | 'C' |




*skip simulation*

# Graphical Recursion



- Recursion comes up in certain graphical applications, most notably in the creation of fractals, which are mathematical structures that consist of similar figures repeated at various different scales. Fractals were popularized in a 1982 book by Benoit Mandelbrot entitled *The Fractal Geometry of Nature*.
- One of the simplest fractal patterns to draw is the Koch fractal, named after its inventor, the Swedish mathematician Helge von Koch (1870-1924). The Koch fractal is sometimes called a snowflake fractal because of the beautiful, six-sided symmetries it displays as the figure becomes more detailed. as illustrated in the above diagram.