# Recursive Implementations

# Recursion

- A powerful problem solving strategy
- The body of a function call the function itself.

• What can be the potential problem?

– Let’s say you are calling fun1 from the body of fun1

– Wouldn’t it be like an infinite loop?

– How can it stop?

– So, inside the body of your function you must put something that must prevent it to call the function again (i.e., must not result in a recursive call to prevent to fall in infinite loop)

# Example

```
void rec2(int x)
{
   if (x==0)
      return; //this breaking condition will result in poping the function call stack

   rec2(x-1);

   printf("%d  ", x); //this line (including the value of x)  is kept in the stack for each call of rec(x-1) in the above line
}
```

```
void rec1(int x)
{
   if (x==0)
      return; //breaking condition

   printf("%d  ", x); //prints first then recursion
   rec1(x-1);
}
```

```
int main()
{
   printf("Calling rec1: ");
   rec1(10);

   printf("\nCalling rec2: ");
   rec2(10);
}
```

```
Output:
Calling rec1: 10  9  8  7  6  5  4  3  2  1
Calling rec2: 1  2  3  4  5  6  7  8  9  10
```

# Power calculation

Calculating power $x^n$

• It means you need to multiply x (n times). x*x*x*… n times

```
int Power(int base, int exponent){
        long result = 1;

        while (exponent != 0)
        {
                result *= base;
                --exponent;
        }
        return result;
}
```

• How can you use recursion to do it?

# Power calculation

```
int Power(int base, int exponent) {

    if ( exponent == 0 )
        return 1;
    else
        return (base*Power(base, exponent-1));
}
```

# Recursion

- When we have a problem, we want to break it down into chunks, where one of the chunks is a smaller version of the same problem.
- We break down our original problem enough that our sub-problem is quite easy to solve.
- A general structure of a recursive function has a couple options:

  – Break down the problem further, into a smaller subproblem

  – OR, the problem is small enough on its own, solve it. (base case)

- When we have two options, we often use an if statement. This is typically what is done with recursion.

# Example

- Write a function that takes in one positive integer parameter n, and returns the sum 1+2+...+n.

```
int totalSum(int n) {
        int index, sum = 0;
        for (index=1; index <=n; index++)
            sum = sum + index;
    return sum;
}
```

# Example

- f(n) = 1 + 2 + ... + n = n + (1 + 2 + ... + (n-1)).
- f(n) = 1 + 2 + ... + n = n + (1 + 2 + ... + (n-1)) = n + f(n-1)
- We know that f(0) = 0, so our terminating condition can be n=0

```
int totalSum(int n) {
        if ( n == 0 )
                return 0;
         else
                return (n + totalSum(n-1));
```

# Example

- Write a recursive function that takes a string and the length of the string in parameters, then print the string in reverse order.
- The prototype should look like this:

```
void printReverse(char string[], int n)
```

- If “florida” and 7 is passed, it should print “adirolf”

# Example

- Write a recursive function that takes a string and the length of the string in parameters, then print the string in reverse order.
- The prototype should look like this:

```
void printReverse(char string[], int n)
```

• If “florida” and 7 is passed, it should print “adirolf”

What position should be printed first?

– string[n-1]

• What would be the breaking condition?

– If n==1

# Example

```
void printReverse(string s, int n)
 {

        // Only one character to print, so print it!
        if (n == 1)
                cout<< s[0];

        // Solve the problem recursively: print the last character, then reverse
        // the substring without that last character.
        else
        {
                cout<< s[n-1];
        printReverse(s, n-1);
        }
}
```

# Example

```
//No need to separate a base case (void function)
void printReverse(string s, int n)
{
   if(n>0)
   {
      cout<< s[n-1];
      printReverse(s, n-1);
   }
}
```

# Example

Write a recursive function that calculates the sum $1^1 + 2^2 + 3^3 + ... + n^n$, given an integer value of n in between 1 and 9. You can write a separate power function in this process and call that power function as needed:

```
int crazySum(int n);
```

# Permutations

The permutation problem is as follows:

• Given a list of items, list all the possible orderings of those items.

– The items can be numbers or letters or other object

• For example: all the permutations of 0,1,2:

0, 1, 2  0, 2, 1  1, 0, 2  1, 2, 0  2, 0, 1  2, 1, 0.

• or all the permutations of “CAT”:

CAT  ACT  TAC  CTA  ATC  TCA

How can we write a program to generate these permutations

# Permutations

How to reduce the problem to a smaller problem of same form

- We choose a character for the first position of the permutation
- For example if a permutation starts with C we need to do another permutation for the substring of length 2(starting at the second position), which is smaller and it should be in the same form(CAT, CTA)
- So, we can use recursion for generating the permutations
- We need to do it for each position of our arrangement(permutation) of characters

```
void perm(int * used, char * original, char * current, int len, int pos)
```

`int * used` : includes 0 and 1s for keeping track of which characters have been used in the current arrangement

`char * original:` The input characters

`char * current:` The current arrangement of letters

`int len` : Number of characters

`int pos:` current position

# Permutations

How to reduce the problem to a smaller problem of same form

- We choose a character for the first position of the permutation
- For example if a permutation starts with C we need to do another permutation for the substring of length 2(starting at the second position), which is smaller and it should be in the same form(CAT, CTA)
- So, we can use recursion for generating the permutations
- We need to do it for each position of our arrangement(permutation) of characters

```
void perm(int * used, char * original, char * current, int len, int pos)
```

`int * used` : includes 0 and 1s for keeping track of which characters have been used in the current arrangement

`char * original:` The input characters

`char * current:` the current arrangement of letters (initialized with empty string)

`int len` : Number of characters

`int pos:` current position

What would be the terminating condition for your recursion? If pos = len