Recursion

(Section 3)

Recursive Definitions and Processes
Writing recursive program in C
Some more recursive examples (out of course)
Efficiency of Recursions

By:

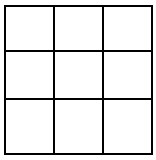
Pramod Parajuli,
Department of Computer Science,
St. Xavier's College,
Nepal.

Recursive Definitions & Processes

Recursion

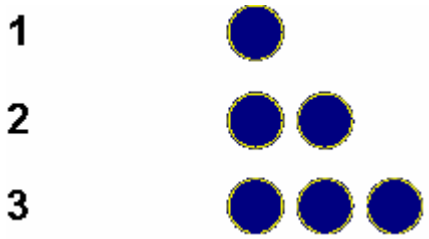
Recursion is defined as a process that defines a function or a process in terms of itself. Seems quite confusing?

During your study, you have encountered lots of iterative processes or models that after each and every iteration you encounter with the model of same kind.



Formal examples

Example 1: Natural Numbers



1, 2, 3, 4, ...

Every element in given format has a property that it is one increment of the preceding value.

Mathematically, element at location 'n' is one increment over element at location 'n-1'.

Example 2: Factorial

Factorial value of a number 'n' is defined as;

factorial value of n = n * factorial value of n-1

0! = 1
1! = 1
2! = 2 * 1! = 2 * 1 = 2
3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 = 6
4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 = 24 and so on.

or, f(n) = n * f(n-1)

Example 3: Fibonacci numbers

Fibonacci numbers also hold recursive property.

0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.

Could you think of the general formulation of the Fibonacci numbers?

0 = 0
1 = 1, these two steps are the basic/preliminary steps for the generation of Fibonacci numbers.

Now, for 2nd '1' ,

1 = 1 + 0
2 = 1 + 1
3 = 2 + 1
5 = 3 + 2
8 = 5 + 3
13 = 8 + 5
21 = 13 + 8 and so on.

Therefore, element at position 'n' = element at position 'n-1' + element at position 'n-2'

or, f(n) = f(n-1) + f(n-2)

Example 4: Mathematical expressions

Infix expression has a format of: **<operand> <operator> <operand>**. The operands on the both sides of the operator are themselves valid infix expressions.

a + b - c + d

This equation can be represented as,

(a + b) - (c + d)

here, **operand1** = (a + b)
operand2 = (c + d)
operator = '-'

again, the **operand1** is also a valid infix expression.

operand11 = a
operand12 = b
operator1 = +

further, the **operand11** itself is also a valid infix with no more intermediate expressions.

Similarly, postfix and prefix expressions, linked list, etc. also hold recursive nature.

What you can conclude here is; **every model/process that contains same kind of iterative process (doing again and again) and after each and every process, the property/behavior of the data also remains same.**

Recursive Procedure

A typical recursive function is;

```
void recursion(void){
      recursion();
}
```

The C language provides recursion facility to be implemented by calling itself. The code given here calls itself. This type of recursion is called **direct recursion**. Let's look at another example;

```
void sxc(void){
      recursion();
}

void recursion(void){
      sxc();
}
```

Here, the function calls are nested. Whenever one of the function is called, it will eventually call another function, and the another function again calls the parent function, and this process goes again and again. This type of indirect way of calling functions each and other iteratively is called as **indirect recursion**.

Let's look at a particular example:

```
void recursion(void){
      recursion();
}

void main(){
      recursion();
}
```

Whenever the program runs, it will fall into recursion function. Since the recursion function calls itself, the program will never exit. This is called as a **trap**. While writing recursive programs/functions we do not expect such things to happen. So, there must be some statements/criteria so that the execution of recursive function stops at the fulfillment of the criteria. This criteria is called as a **base criteria**.

We can modify the given program as;

```
void recursion(int n){

      if(n == 0)
            return 0;

      recursion(n - 1);
}

void main(){
      recursion(5);
}
```

Some new additions to the program!! We changed the definition of recursion to take a number 'n'. If the number 'n' is 0, then the recursion procedure returns 0, i.e. it will not make any further call to itself. Therefore, the base criteria here is, if n = 0, then return 0.

Now, look at code,

```
recursion(n - 1);
```

Why do we need to decrease the value of n? As the definition of recursion says, after each and every iteration, the value of variables must be changed but the property must be the same. Now, let's discuss a good logical behavior of the code. As the function 'recursion (n-1)' is called, the value of n is decreased by 1. In this particular example, value of n provided in the main program is '5'.

Step	recursion(n)	value of n in recursive function	Remarks
1	recursion(5)	5	Call from main function
2	recursion(4)	4	First call from recursion function
3	recursion(3)	3	Second call from the recursion function
4	recursion(2)	2	Third call from the recursion function
5	recursion(1)	1	Fourth call from the recursion function
6	recursion(0)	0	No call, since we encountered the base criteria.

Above discussion results into:

1. A recursive function should not run indefinitely. To prevent indefinite run, there must be a **base criteria** for which the procedure does not call itself.
2. Every time the procedure does call itself (directly or indirectly), it must be closer to the base criteria. If it goes far from base criteria, then it will never stop.

For the above example, if the recursion function is called as (n+1), then the value of n will never be 0, and hence the function is called indefinite times.

Writing recursive program in C

C language provides lots of features to implement recursion. The compilers are designed in such a manner that suites full-fledged implementation of recursion.

1. C language allows function call from/within the same function. As we already saw that the function 'recursion' calls itself.
2. C compilers has a very good '**internal stack**' management system to hold previous data.

Internal stack:

Let's have a look at a recursive procedure.

```
int display(int number){
      if (number == 0)
            return number;

      display(number - 1);
      printf("%d", number);
      return 0;
}
```

Can you think how the program remembers the value of 'number' in each and every recursive call? C compiler, has a mechanism that while compiling and linking the programs, it sets instructions to create an '**internal stack**.' The program uses this internal stack to store previous data. This stack is not accessible to the programmer directly i.e. the push and pop operations in the stack are maintained internally, so called 'internal stack.'

Let's check how the values of 'number' is stored in the given recursive function.

Iteration	Value of number	Value in internal stack
1	5	5
2	4	5, 4
3	3	5, 4, 3
4	2	5, 4, 3, 2
5	1	5, 4, 3, 2, 1
6	0	return

The value of variables are stored in the stack whenever recursive function is called. The values of 'number' are accessed by using **printf("%d", number);,** there are no recursive calls, so the program pops all of the stack elements one by one.

Try: Implementation of recursion for decimal - binary conversion using internal stack of C. (Hint: Just calculate the remainder each and every time. After finishing the recursive call, print .)

Sum of natural numbers

Sum of natural numbers from 0 to number 'n' is defined as:

$$\sum n = n + (n-1) + (n-2) + (n-3) + ... + 1$$

This equation can also be represented as:

$$\sum n = n + \sum (n-1)$$

Here, you can see that the sum function of natural number exhibits recursive nature. Sum up to 'n' number is the summation of the 'n' number itself and sum up to 'n-1'. Here, we sum up to 1. If we encounter 0, then we just discard it and stop. So, our base criteria is n = 0.

For algorithmic representation we can write as;

```
sum(n) = 0,  if n = 0
sum(n) = n + sum(n - 1), if n > 0
```

Sample code: sum of natural numbers (PROGRAM - 06)

```
int sum(int n){
      if( n == 0)
            return 0;

      return n + sum(n-1);
}

void main(){

      int number;
      number = 10;
      printf("Sum upto %d is: %d.", number, sum(number));
      getch();
}
```

Factorial function

Factorial value of a given number is the multiplication of all the natural numbers up to that number. Here, we start from 1 to n.

Therefore, we can write as:

$$F(n) = n.(n-1).(n-2). \ldots . 1$$

Which can further be modularized as;

$$F(n) = n \,.\, F(n-1)$$

Sample code: factorial function (PROGRAM - 07)

```
int factorial(int n){

      if(n == 0)return 1;
      return n * factorial(n - 1);
}

void main(){
      int num;
      printf("\nPlease enter a number : ");
      scanf("%d", &num);
```

```
	printf("\nFactorial Value : %.d", factorial(num));
	getch();
}
```

Fibonacci sequence

Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, ...

Here, we must know that the first two elements 0 and 1, are the fixed one. After that, we have formulation that, every element at the given position is the sum of the two elements previous to it. So now, our base criteria is that, if the element value is either 0 or 1, we just quit and return 0 and 1 respectively. Otherwise, we apply the recursive procedure.

It can be formulated as:

fib(0) = 0
fib(1) = 1
fib(n+1) = fib(n) + fib(n-1) or, fib(n) = fib(n-1) + fib(n-2)

Sample code: fibonacci function (PROGRAM - 08)

```
int fibonacci(int n){
	if(n <= 1)
		return n;

	return fibonacci(n - 1) + fibonacci(n - 2);
}

void main(){

	int num, counter;
	clrscr();
	printf("\nEnter number of elements in fibonacci series : ");
	scanf("%d", &num);

	for(counter = 0; counter <= num; counter++)
		printf("%4d", fibonacci(counter));
	printf("\n\n-----------------------------------------------");

	getch();
}
```

Multiplication of natural numbers

Let 'm' and 'n' be two natural numbers to be multiplied. Then, the multiplication result can be written as;

m * n = m + (m * (n-1))

So, again, the multiplication also holds the recursive nature.

e.g.

5 * 3 = 5 + (5 * 2)
5 * 3 = 5 + 5 + (5 * 1)
5 * 3 = 5 + 5 + 5

or, we can do like;

5 * 3 = 3 + (3 * 4)
5 * 3 = 3 + 3 + (3 * 3)
5 * 3 = 3 + 3 + 3 + (3 * 2)
5 * 3 = 3 + 3 + 3 + 3 + (3 * 1)
5 * 3 = 3 + 3 + 3 + 3 + 3

Now, if you look at the two possible ways to implement the algorithm, the first one looks better, it's because the first one takes only 3 steps to compute the multiplication value whereas the second one takes 5 steps to compute the multiplication value. Therefore, it is desirable that we add greater number less number of times.

Our base criteria here is whenever the decrementing value, in this case 'n', reaches less than 1, we just quit.

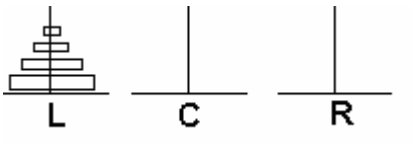
We will see more simplified 'divide and conquer' method to multiply large natural numbers in 'Types of algorithms' section at last.

Sample code: natural number multiplier function (PROGRAM - 09)

```
int multiplier(int m, int n){
	if(n == 1){
		return m;
	}

	return m + multiplier(m, n-1);
}

void main(){

	int m, n;
	clrscr();
	printf("\nEnter two numbers");

	printf("\nFirst number : ");
	scanf("%d", &m);

	printf("\nSecond number : ");
	scanf("%d", &n);

	if(n > m){
		m = m + n;
		n = m - n;
		m = m - n;
	}

	printf("\n%d * %d = %d", m, n, multiplier(m, n));
	getch();
}
```

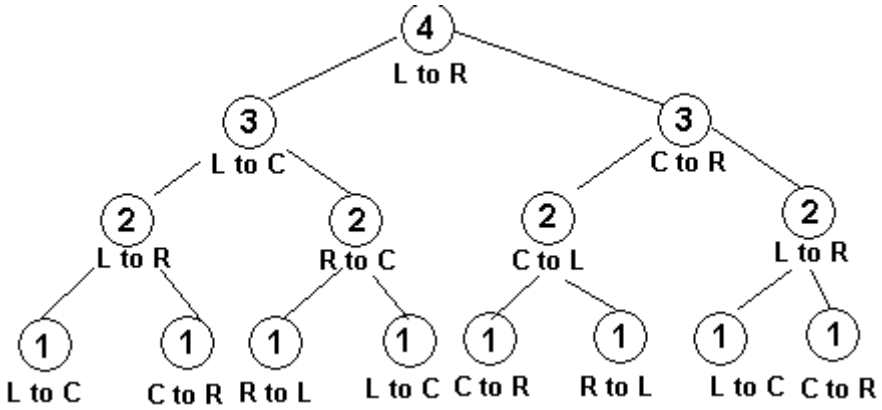
Tower of Hanoi

The problem is to transfer all the disks from left to right by using the center as temporary space and maintaining the same order. Originally came from Hindu myth from Brahma. "If all of the 64 disks are transferred, then that will be the end of Earth."



The general idea is that, if we transfer disks up to 3, then we just transfer disk 4 and then same for 5, 6, 7 etc. Here one must be clear that, the disks are transferred from left to right using the center as temporary space i.e. the disks are transferred in two steps.

So now, we can visualize the working of TOH disk transfer as a binary tree;



Now, to get the final steps, we just traverse the tree in in-order. In in-order traversal, we first go to the left tree, then visit the node and then go to right tree. In short, this can be written as, **LVR** (left-visit-right). You will get more idea about trees and traversals in 'Trees' section.

The traversal will result as:

Move disk 1 from L to C
Move disk 2 from L to R

Move disk 1 from C to R
Move disk 3 from L to C

Move disk 1 from R to L
Move disk 2 from R to C

Move disk 1 from L to C

Move disk 4 from L to R
....

Can you write rest of the steps?

This is how we break the problem. Now, as you can see in the tree, we have two steps for every single movement and then those two steps also have two other movements each. So, the transfer of disks also exhibits recursive nature.

So, now we can formulate a recursive procedure as;

To transfer 'n' disk from L to R using C as temporary space;

1. We first transfer the disk 'n-1' from L to C using R as temporary space.
2. And then, transfer the same disk 'n-1' from C to R using L as temporary space.

Sample code: tower of hanoi (PROGRAM - 10)

```
void transfer(int n, char from, char to, char temp){
      if(n == 0){
            return;
      }

      else{
            transfer(n-1, from, temp, to);
            printf("\nMove disk %d from %c to %c", n, from, to);
            transfer(n - 1, temp, to, from);
      }
}

void main(){
      int number;
      clrscr();
      printf("\tTower of Hanoi");
      printf("\n--------------------------");
      printf("\nEnter number of disks : ");
      scanf("%d", &number);
      transfer(number, 'L', 'R', 'C');
      printf("\n\n----------------------------------------");
      getch();
}
```

Tracing

Have you noticed the different orders of parameters in the two recursive function calls to 'transfer' in the function? Why so? It's because, as already said, we are going to transfer disks from source to destination using a third temporary space and after each and every transfer the positions of disks is changed.

How the program works? Thanks again to the great '**internal stack**.' Let's have a look at how the internal stack is accessed?

n	source	destination	temp
1	L	C	R
2	L	R	C
3	L	C	R
4	L	R	C

Now, from the upper 'transfer' routine,

move disk 1 from L to C
move disk 2 from L to R

Now, lower 'transfer' routine is activated.

n	source	destination	temp
1	C	R	L

move disk 1 from C to R
move disk 3 from L to C

n	source	destination	temp
1	R	L	C
2	R	C	L

move disk 1 from R to L
move disk 2 from R to C

n	source	destination	temp
1	L	C	R

move disk 1 from L to C
move disk 4 from L to R

Homework 2: Draw tree for disks 5, traverse the path and write down all the steps required.
Deadline: February 20, 2004

Binary Search

Let's say, we have a sorted list of numbers or other items. We want to access a particular number or data in the list.

e.g. We have a sorted list in an array 'data' as shown below:

75	151	203	275	318	489	524	591	647	727

Normally, to get number '647', we start from data[0] and then compare the value. We increment the address/array pointer each and every time until we do not find the right value. This way, it takes 9 comparisons in the program. Therefore, if there are 'n' number of elements in the array, then in worst case, it takes 'n' comparisons/access to find a number and in average it takes n/2 comparisons.

There is an efficient way to access given item in a sorted array known as 'binary search.'

Algorithm

```
General procedure: binary_search(k, n, x)

1. [Initialize]

      low ? 1
      high ? N

2. [Perform Search]

      repeat thru step 4 while low <= high

3. [Obtain index of midpoint of interval]

      middle ? (low + high) / 2

4. [Compare]

      if x < k[middle]

            high ? middle – 1
            Call binary_search

      else if x > k[middle]

            low ? middle + 1
            call binary_search

      else

            write ‘successful search’
            return (middle)

5. [Unsuccessful search]

      this condition will be encountered if high < low,
      write ‘unsuccessful search’
      return 0;
```

Tracing with data

75, 151, 203, 275, 318, 489, 524, 591, 647, & 727

Search for x = 275, 721

Search for 275

Iteration	L	H	M
1	1	10	5
2	1	4	2
3	3	4	3
4	4	4	4

Since, key = k[4], search is successful.
Found in 4th location.

Search for 721

Iteration	L	H	M
1	1	10	5
2	6	10	8
3	9	10	9
4	10	10	10
5	10	9	

Now, H < L, which is invalid condition. Therefore the key doesn't exist.

Efficiency:

Let, N = 1000 i.e. 1000 number of nodes
Now, worse case = 11 comparisons (10 for valid keys)

Searching time = log(2)n+1
i.e. 2^10 ˜ 1000 (1024)

Average time = (log(2)n+1) / 2

Best = 1

Sample code: binary search (PROGRAM - 11)

```
int binarySearch(int *k, int x, int low, int high){

      int middle;
      middle = (low + high) / 2;

      if(high < low){

            return -1;
      }

      if(x < k[middle]){

            high = middle - 1;
            binarySearch(k, x, low, high);
      }

      else if(x > k[middle]){
            low = middle + 1;
            binarySearch(k, x, low, high);
      }

      else
            return middle;
}
```

```
void main(){

      int counter, number, position;
      int *data;
      counter = 0;
      clrscr();
      printf("\nPlease enter some data : ");

      do{
            printf("Data : ");
            scanf("%d", &number);
            data[counter++] = number;
      }while(number != 0);

      selectionSort(data, counter -1);
      printElements(data, counter -1);
      printf("\n\n\nPlease enter data to find : ");
      scanf("%d", &number);
      position = binarySearch(data, number, 0, counter -1);

      if(position == -1){
            printf("\n\nUnsuccessful search.");
      }

      else
            printf("\n\nThe data %d is located at location : %d.",
            number, position);

      printf("\n\nPress any key ... ");
      getch();
}
```

Some more recursive examples (out of course)

Infix expression checking

As we have already seen that, infix expression holds recursive property.

Let's consider an infix expression is given as:

a + b - c % d

We divide the expression from left hand side.

symbol-1	a	**operand1**
symbol - 2	+	**operator1**
symbol - 3	b	**operand2**

Here, symbol-1 = 'a' is a single variable and therefore a valid infix expression. Immediately after that, symbol-2 is an operator. Since we are dealing with infix expression, we expect operator after every operand, and therefore this is also OK.

Now, after operator1, the format should again be of same kind i.e. we must encounter with an operand and then operator and then again operand and operator and so on.

Instead of the original formulation of the recursive infix nature, here we are defining in terms of the positions of the operands and operators.

Now, what is we encounter with an operand immediately after an operand i.e. what is the expression is like;

a b + c - d % e

Then it would not be a valid infix expression. Similar also holds for operators.

Now, can you think of the any algorithm to check the validity?

Algorithm

```
1. Scan left to right (read a char)
2. Do while till not end of line

      2.1 if it is a letter

            2.1.1 read next

            2.1.2 if it is operator

                        call recursive function

                  else
            // now one tricky step here, if we reached up
            // to the end of the line
            // then also we want to return valid, since
            // all of the previous portion was valid.
```

```
                        if it is carriage return

                              return valid;

                        else

                              return invalid;

      2.2 else

            return invalid;

3. Finish.
```

Trace the algorithm. Can you think of handling parentheses also.

Sample code: infix expression checker (recursive) (PROGRAM - 12)

```
int checkInfix(char *infix){

      printf("\nInfix %s", infix);

      if(isLetter(infix[0])){

            if(isOperator(infix[1])){
                  substr(infix, 2, strlen(infix));
                  checkInfix(infix);
            }

            else if(infix[1] == '\0' || infix[1] == '\n')
                  return VALID;

            else
                  return INVALID;
      }

      else

            return INVALID;
}
```

Euclid's Algorithm to find GCD

The GCD of two +ve integers is the largest integers that divide both of them. e.g. GCD of 18 & 24 = 6.

Recursive representation of GCD function is as follows:

```
GCD(x, y) = x, if y = 0
GCD(x, y) = GCD(y, x % y)
```

(PROGRAM - 13)

Ackermann's function

Ackermann's function is a standard device to determine how well recursion is implemented on a computer. The recursion steps increase rapidly even for small increase in the value provided.

It is defined as:

```
A(0, n) = n + 1 for n>= 0   or we can think as; if m = 0, then return
                                                    (n + 1).
A(m, 0) = A(m – 1, 1) for m>0  or if n = 0, then call (m - 1, 1)
A(m, n) = A(m-1, A(m, n-1) for m > 0 & n > 0.
```

```
(PROGRAM - 14)
```

Efficiency of Recursions

Simulating recursion

The recursive functions can be simulated (mimicked) with non-recursive procedures. These procedures extract the logic of base criteria and the looping feature from the recursive function. Followings are the steps to write non-recursive simulators of the recursive function. **(Available in Tanenbaum's Book)**

1. First write down correct recursive function
2. Now, determine all the internal stack operations that occur in the recursion and write procedures for the stack operations.
3. If there are useless stacks or variables then either remove them or combine with others.
4. To give faster performance, the stack operations should be placed **in-line**. Have a look at '**inline**' keyword in Borland C++ 3.1.

There are some advantages and disadvantages of recursive definition and implementation of some procedures.

\+ Provides easier way to define a real world problem. For certain problems, if a non-recursive simulator is used then the definition would be very much vague. (**Look at page 165 of Tanenbaum's Book for non-recursive definition and implementation of TOH**)

\+ If the number of variables and stack reduces in recursive function than recursive simulator, then recursive functions are faster.

– If a complicated solution for simple problem, then recursion is not good.

– If the number of variables and stack is reduced in non-recursive function, then recursion is not good.

– Recursion is not good for the function that is frequently called by main program.