Recursion

A process by which a function calls itself repeatedly.

- Either directly.
  - $F$ calls $F$.
- Or cyclically in a chain.
  - $F$ calls $G$, $G$ calls $H$, and $H$ calls $F$.

Used for repetitive computations in which each action is stated in terms of a previous result.

$$\text{fact}(n) = n \times \text{fact}(n-1)$$
Basis and Recursion

For a problem to be written in recursive form, two conditions are to be satisfied:

- It should be possible to express the problem in recursive form.
- The problem statement must include a stopping condition

\[
\text{fact}(n) = 1, \quad \text{if} \ n = 0 \quad /* \text{Stopping criteria} */ \\
= n \times \text{fact}(n - 1), \quad \text{if} \ n > 0 \quad /* \text{Recursive form} */
\]
Examples:

• **Factorial:**
  
  \[
  \text{fact}(0) = 1 \\
  \text{fact}(n) = n \times \text{fact}(n - 1), \text{ if } n > 0
  \]

• **GCD:**
  
  \[
  \text{gcd}(m, m) = m \\
  \text{gcd}(m, n) = \text{gcd}(m \mod n, n), \text{ if } m > n \\
  \text{gcd}(m, n) = \text{gcd}(n, n \mod m), \text{ if } m < n
  \]

• **Fibonacci series (1,1,2,3,5,8,13,21,....)**
  
  \[
  \text{fib}(0) = 1 \\
  \text{fib}(1) = 1 \\
  \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2), \text{ if } n > 1
  \]
Example 1 :: Factorial

```c
long int fact ( int n)
{
    if  (n == 1)
        return (1);
    else
        return (n * fact(n - 1));
}
```
Example 1 :: Factorial Execution

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

```
if (4 == 1) return (1);
else return (4 * fact(3));
```

```
if (3 == 1) return (1);
else return (3 * fact(2));
```

```
if (2 == 1) return (1);
else return (2 * fact(1));
```

```
if (1 == 1) return (1);
else return (1 * fact(0));
```
Example 2 :: Fibonacci number

Fibonacci number $f(n)$ can be defined as:

- $f(0) = 0$
- $f(1) = 1$
- $f(n) = f(n − 1) + f(n − 2)$, if $n > 1$

• The successive Fibonacci numbers are:
  $0, 1, 1, 2, 3, 5, 8, 13, 21, .....$

Function definition:

```c
int f (int n) {
    if (n < 2)   return (n);
    else return ( f(n − 1) + f(n − 2) );
}
```
Tracing Execution

```c
int f(int n) {
    if (n < 2) return (n);
    else return (f(n - 1) + f(n - 2));
}
```

How many times is the function called when evaluating f(4) ?

Inefficiency:

- Same thing is computed several times.

Called 9 times
Notable Point

- Every recursive program can also be written without recursion
- Recursion is used for programming convenience, not for performance enhancement
- Sometimes, if the function being computed has a nice recurrence form, then a recursive code may be more readable
Important things to remember

• Think how the whole problem (finding max of n elements in A) can be solved if you can solve the exact same problem on a smaller problem (finding max of first n-1 elements of the array).

• But then, do NOT think how the smaller problem will be solved, just call the function recursively and assume it will be solved.

• When you write a recursive function
  • First write the terminating/base condition
  • Then write the rest of the function
  • Always double-check that you have both
int sumSquares (int m, int n)
{
    int middle;
    if (m == n) return(m*m);
    else
    {
        middle = (m+n)/2;
        return (sumSquares(m,middle) + sumSquares(middle+1,n));
    }
}
int sumSquares (int m, int n)
{
    int middle ;
    if (m == n) return(m*m);
    else {
        middle = (m+n)/2;
        return (sumSquares(m,middle)
            + sumSquares(middle+1,n));
    }
}
Example: Printing the digits of an Integer in Reverse

Print the last digit, then print the remaining number in reverse

- Ex: If integer is 743, then reversed is print 3 first, then print the reverse of 74

```c
void printReversed(int i)
{
    if (i < 10) {
        printf("%d\n", i); return;
    }
    else {
        printf("%d", i%10);
        printReversed(i/10);
    }
}
```
Counting Zeros in a Positive Integer

Check last digit from right
- If it is 0, number of zeros = 1 + number of zeroes in remaining part of the number
- If it is non-0, number of zeros = number of zeroes in remaining part of the number

```c
int zeros(int number)
{
    if(number<10) return 0;
    if (number%10 == 0)
        return(1+zeros(number/10));
    else
        return(zeros(number/10));
}
```
Example: Binary Search

- Searching for an element k in a sorted array A with n elements

- Idea:
  - Choose the middle element A[n/2]
  - If k == A[n/2], we are done
  - If k < A[n/2], search for k between A[0] and A[n/2 -1]
  - If k > A[n/2], search for k between A[n/2 + 1] and A[n-1]
  - Repeat until either k is found, or no more elements to search

- Requires less number of comparisons than linear search in the worst case (log₂ n instead of n)
Binary Search

int binsearch(int A[], int low, int high, int k)
{
    int mid;
    printf("low = %d, high = %d\n", low, high);

    if (low < high) return 0;
    mid = (low + high)/2;
    printf("mid = %d, A[%d] = %d\n\n", mid, mid, A[mid]);

    if (A[mid] == k) return 1;
    else {
        if (A[mid] > k)
            return (binsearch(A, low, mid-1, k));
        else
            return(binsearch(A, mid+1, high, k));
    }
}

int main()
{
    int A[25], n, k, i, found;

    scanf("%d", &n);
    for (i=0; i<n; i++) scanf("%d", &A[i]);
    scanf("%d", &k);

    found = binsearch(A, 0, n-1, k);
    if (found == 1)
        printf("%d is present in the array\n", k);
    else
        printf("%d is not present in the array\n", k);
}
Output

```c
int binsearch(int A[], int low, int high, int k)
{
    int mid;
    printf("low = %d, high = %d\n", low, high);

    if (low < high) return 0;
    mid = (low + high)/2;
    printf("mid = %d, A[%d] = %d\n", mid, mid, A[mid]);

    if (A[mid] == k) return 1;
    else {
        if (A[mid] > k)
            return (binsearch(A, low, mid-1, k));
        else
            return(binsearch(A, mid+1, high, k));
    }
}
```

8
9 11 14 17 19 20 23 27
21
low = 0, high = 7
low = 4, high = 7
low = 6, high = 7
low = 6, high = 5
21 is not present in the array

14
8
9 11 14 17 19 20 23 27
14
low = 0, high = 7
low = 0, high = 2
mid = 1, A[1] = 11
low = 2, high = 2
14 is present in the array
static int m1, m2;
int res, temp;
if (i==2) {m1 =1; m2=1;}
if (n == i) res = m1+ m2;
else
    {  temp = m1;
       m1 = m1+m2;
       m2 = temp;
       res = Fib(n, i+1);
    }
return res;
}

Static variables remain in existence rather than coming and going each time a function is activated
Common Errors in Writing Recursive Functions

Non-terminating Recursive Function (Infinite recursion)

- No base case

```c
int badFactorial(int x) {
    return x * badFactorial(x-1);
}
```

- The base case is never reached

```c
int anotherBadFactorial(int x) {
    if (x == 0)
        return 1;
    else
        return x*(x-1)*anotherBadFactorial(x-2);
    // When x is odd, base case never reached!!
}
```

```c
int badSum2(int x) {
    if (x == 1)
        return 1;
    return(badSum2(x--));
}
```
Mixing up loops and recursion

```c
int anotherBadFactorial(int x) {
    int i, fact = 0;
    if (x == 0)
        return 1;
    else {
        for (i=x; i>0; i=i-1) {
            fact = fact + x*anotherBadFactorial(x-1);
        }
        return fact;
    }
}
```

In general, if you have recursive function calls within a loop, think carefully if you need it.

Most recursive functions you will see in this course will not need this
Example :: Towers of Hanoi Problem

The problem statement:

• Initially all the disks are stacked on the LEFT pole.
• Required to transfer all the disks to the RIGHT pole.
  • Only one disk can be moved at a time.
  • A larger disk cannot be placed on a smaller disk.
• CENTER pole is used for temporary storage of disks.
Recursive statement of the general problem of n disks.

• Step 1:
  • Move the top (n-1) disks from LEFT to CENTER.

• Step 2:
  • Move the largest disk from LEFT to RIGHT.

• Step 3:
  • Move the (n-1) disks from CENTER to RIGHT.
Phase-1: Move top n – 1 from LEFT to CENTER
Phase-2: Move the $n^{th}$ disk from LEFT to RIGHT

![Diagram of disk movement from LEFT to CENTER to RIGHT](image)

1. Move the 1st disk from LEFT to CENTER.
2. Move the 2nd disk from LEFT to RIGHT.
3. Move the 3rd disk from CENTER to RIGHT.

LEFT | CENTER | RIGHT
--- | --- | ---
3 | 1 |
1 | 2 |
2 | 1 | 3
Phase-3: Move top n – 1 from CENTER to RIGHT
#include <stdio.h>

void transfer (int n, char from, char to, char temp);

main()
{
    int n; /* Number of disks */
    scanf ("%d", &n);
    transfer (n, ‘L’, ‘R’, ‘C’);
}

void transfer (int n, char from, char to, char temp)
{
    if (n > 0) {
        transfer (n-1, from, temp, to);
        printf ("Move disk %d from %c to %c \n", n, from, to);
        transfer (n-1, temp, to, from);
    }
    return;
}
With 3 discs

With 4 discs
Recursion versus Iteration

Repetition

- **Iteration**: explicit loop
- **Recursion**: repeated nested function calls

Termination

- **Iteration**: loop condition fails
- **Recursion**: base case recognized

Both can have infinite loops

Balance

- **Choice between performance (iteration) and good software engineering (recursion).**
How are recursive calls implemented?

What we have seen ....

- Activation record gets pushed into the stack when a function call is made.
- Activation record is popped off the stack when the function returns.

In recursion, a function calls itself.

- Several function calls going on, with none of the function calls returning back.
  - Activation records are pushed onto the stack continuously.
  - Large stack space required.
• Activation records keep popping off, when the termination condition of recursion is reached.

We shall illustrate the process by an example of computing factorial.

• Activation record looks like:

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>Return Value</th>
<th>Return Addr</th>
</tr>
</thead>
</table>


Example:: main( ) calls fact(3)

```c
main()
{
    int n;
    n = 3;
    printf("%d \n", fact(n));
}

int fact (n)
int n;
{
    if  (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}
```
### TRACE OF THE STACK DURING EXECUTION

**main()** calls **fact()**

<table>
<thead>
<tr>
<th>n = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA .. fact</td>
</tr>
<tr>
<td>n = 1</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>RA .. fact</td>
</tr>
<tr>
<td>n = 2</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>RA .. fact</td>
</tr>
<tr>
<td>n = 3</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>RA .. main</td>
</tr>
</tbody>
</table>

**fact()** returns to **main()**

<table>
<thead>
<tr>
<th>n = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA .. fact</td>
</tr>
<tr>
<td>n = 3</td>
</tr>
<tr>
<td>RA .. fact</td>
</tr>
<tr>
<td>n = 3</td>
</tr>
<tr>
<td>RA .. main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1*1 = 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2*1 = 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3*2 = 6</td>
</tr>
</tbody>
</table>
Do Yourself

Trace the activation records for the following version of Fibonacci sequence.

```c
#include <stdio.h>
int f (int n)
{
    int a, b;
    if (n < 2) return (n);
    else {
        a = f(n-1);
        b = f(n-2);
        return (a+b); }
}

main( ) {
    printf("Fib(4) is: %d \n", f(4));
}
```

Table:

<table>
<thead>
<tr>
<th>Return Addr (either main, or X, or Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Variables (n, a, b)</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>
Examples
What do the following programs print?

```c
void foo( int n )
{
    int data;
    if ( n == 0 ) return;
    scanf("%d", &data);
    foo ( n – 1 );
    printf("%d\n", data);
}
main ( )
{
    int k = 5;
    foo ( k );
}
```

```c
void foo( int n )
{
    int data;
    if ( n == 0 ) return;
    foo ( n – 1 );
    scanf("%d", &data);
    printf("%d\n", data);
}
main ( )
{
    int k = 5;
    foo ( k );
}
```

```c
void foo( int n )
{
    int data;
    if ( n == 0 ) return;
    scanf("%d", &data);
    printf("%d\n", data);
    foo ( n – 1 );
}
main ( )
{
    int k = 5;
    foo ( k );
}
```
Printing cumulative sum -- will this work?

```c
int foo(int n)
{
    int data, sum;
    if (n == 0) return 0;
    scanf("%d", &data);
    sum = data + foo(n - 1);
    printf("%d\n", sum);
    return sum;
}

main()
{
    int k = 5;
    foo(k);
}
```

Input: 1 2 3 4 5

Output: 5 9 12 14 15

How to rewrite this so that the output is: 1 3 6 10 15?
int foo( int n )
{
    int data, sum ;
    if ( n == 0 ) return 0;
    sum = foo ( n – 1 );
    scanf("%d", &data);
    sum = sum + data;
    printf("%d
", sum);
    return sum;
}

main ( ) {
    int k = 5;
    foo ( k );
}

void foo( int n, int sum )
{
    int data ;
    if ( n == 0 ) return 0;
    scanf("%d", &data);
    sum = sum + data;
    printf("%d
", sum);
    foo( k – 1, sum ) ;
}

main ( ) {
    int k = 5;
    foo ( k, 0 );
}
#include <stdio.h>

int factorial (int n)
{
    static int count=0;
    count++;
    printf("n=%d, count=%d \n", n, count);
    if (n == 0) return 1;
    else return (n * factorial(n-1));
}

main()
{
    int i=6;
    printf("Value is: %d \n", factorial(i));
}
What does this program print?

#include <stdio.h>

int factorial (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n == 0) return 1;
    else return (n * factorial(n-1));
}

main()
{
    int i=6;
    printf ("Value is: %d \n", factorial(i));
}

Program output:

n=6, count=1
n=5, count=2
n=4, count=3
n=3, count=4
n=2, count=5
n=1, count=6
n=0, count=7
Value is: 720
What does this program print?

```c
#include <stdio.h>

int fib (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n < 2) return n;
        else return (fib(n-1) + fib(n-2));
}

main()
{
    int i=4;
    printf ("Value is: %d \n", fib(i));
}
```
#include <stdio.h>

int fib (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n < 2) return n;
    else return (fib(n-1) + fib(n-2));
}

main()
{
    int i=4;
    printf ("Value is: %d \n", fib(i));
}

Value is: 3 [0,1,1,2,3,5,8,...]
void mergesort ( int a[ ], int lo, int hi )
{
    int m;
    if (lo<hi) {
        m=(lo+hi)/2;
        mergesort(a, lo, m);
        mergesort(a, m+1, hi);
        merge(a, lo, m, hi);
    }
}

Function Merge

```c
void merge ( int a[ ], int lo, int m, int hi )
{
    int i, j, k;

    // copy both halves of a to auxiliary array b
    for (i=lo; i<=hi; i++) b[i]=a[i];

    i=lo; j=m+1; k=lo;
    // copy back next-greatest element at each time
    while (i<=m && j<=hi)
        if (b[i]<=b[j]) a[k++]=b[i++];
        else a[k++]=b[j++];

    // copy back remaining elements of first half (if any)
    while (i<=m) a[k++]=b[i++];
}
```
Recursive Permutation Generator

```c
#define SWAP(x, y, t) { (t) == (x); (x) = (y); (y) = (t) }
void perm (char list[ ], int i, int n)
{
    int j, tmp;
    if (i == n) {
        for (j=0; j<=n; j++) printf("%c", list[ j ]);  
        printf("\n");
    } 
    else {
        for (j=i; j <= n; j++) {
            SWAP(list[ i ], list[ j ], tmp);
            perm(list, i+1, n);
            SWAP(list[ i ], list[ j ], tmp);
        }
    }
}
```
Transitive Closure

Transclosure ( int adjmat[ ][max], int path[ ][max] )
{
    for (i = 0; i < max; i++)
        for (j = 0; j < max; j++)
            path[i][j] = adjmat[i][j];

    for (k = 0; k < max; k++)
        for (i = 0; i < max; i++)
            for (j = 0; j < max; j++)
                if ((path[i][k] == 1) && (path[k][j] == 1)) path[i][j] = 1;
}
Paying with fewest coins

• A country has coins of denomination 3, 5 and 10 respectively.

• We are to write a function canchange( k ) that returns –1 if it is not possible to pay a value of k using these coins.
  • Otherwise it returns the minimum number of coins needed to make the payment.

• For example, canchange(7) will return –1.

• On the other hand, canchange(14) will return 4 because 14 can be paid as 3+3+3+5 and there is no other way to pay with fewer coins.
Paying with fewest coins

```c
int canchange(int k)
{
    int a = -1;
    if (k == 0) return 0;
    if (________________) return 1;
    if (k < 3) __________________;

    a = canchange(______________); if (a > 0) return ________________;
    a = canchange(k - 5); if (a > 0) return ______________________;
    a = canchange(______________); if (a > 0) return ________________;
    return -1;
}
```
int canchange(int k)
{
    int a = -1;
    if (k==0) return 0;
    if ( (k == 3) || (k == 5) || (k == 10) ) return 1;
    if (k < 3) return -1;

    a = canchange(k - 10); if (a > 0) return a + 1;
    a = canchange(k - 5); if (a > 0) return a + 1;
    a = canchange(k - 3); if (a > 0) return a + 1;
    return -1;
}

Practice Problems

1. Write a recursive function to search for an element in an array
2. Write a recursive function to count the digits of a positive integer (do also for sum of digits)
3. Write a recursive function to reverse a null-terminated string
4. Write a recursive function to convert a decimal number to binary
5. Write a recursive function to check if a string is a palindrome or not
6. Write a recursive function to copy one array to another

Note:

- For each of the above, write the main functions to call the recursive function also
- Practice problems are just for practicing recursion, recursion is not necessarily the most efficient way of doing them