RECURSION

Introduction and Summary

Recursion is a powerful algorithmic technique in which a function calls itself (either directly or indirectly) on a smaller problem of the same type in order to simplify the problem to a solvable state.

Every recursive function must have at least two cases: the recursive case and the base case. The base case is a small problem that we know how to solve and is the case that causes the recursion to end. The recursive case is the more general case of the problem we're trying to solve. As an example, with the factorial function $n!$, the recursive case is $n! = n*(n - 1)!$ and the base case is$n = 1$ when $n = =$ 0 or $n = = 1$.

Recursive techniques can often present simple and elegant solutions to problems. However, they are not always the most efficient. Recursive functions often use a good deal of memory and stack space during their operation. The stack space is the memory set aside for a program to use to keep track of all of the functions and their local states currently in the middle of execution. Because they are easy to implement but relatively inefficient, recursive solutions are often best used in cases where development time is a significant concern.

There are many different kinds of recursion, such as linear, tail, binary, nested, and mutual. All of these will be examined.

Terms

Algorithm - A series of steps for accomplishing a set goal.

Factorial - A mathematical function where f(n) = n * f(n-1), f(0) = 1.

Function -
General Case - The condition in a recursion function

Implementation - How an algorithm is actually done, programmed, coded, etc. For any algorithm, there are many ways to actually code it up, to implement it.

Iteration - A programming construct where looping is used to complete an action multiple times. The for() and while() constructs are prime examples of iterative constructs.

Linear Recursion - Recursion where only one call is made to the function from within the function (thus if we were to draw out the recursive calls, we would see a straight, or linear, path).

Exponential Recursion - Recursion where more than one call is made to the function from within itself. This leads to exponential growth in the number of recursive calls

Circularity - In terms a recursion, circularity refers to a recursive function being called with the same arguments as a previous call, leading to an endless cycle of recursion.

Memory - Space in the computer where information is stored.

Recursive Definition - A definition defined in terms of itself, either directly (explicitly using itself) or indirectly (using a function which then calls itself either directly or indirectly).

Recursion - A method of programming whereby a function directly or indirectly calls itself. Recursion is often presented as an alternative to iteration.

Termination Condition - The condition upon which a recursive solution stops recurring. This terminating condition, known as the base case, is the problem in a recursive that we know how to solve explicitly, the "small" problem to which we know the answer.

Recursion Defined

What is recursion? Sometimes a problem is too difficult or too complex to solve because it is too big. If the problem can be broken down into smaller versions of itself, we may be able to find a way to solve one of these smaller versions and then be able to build up to a solution to the entire problem. This is the idea behind recursion; recursive algorithms break down a problem into smaller pieces which you either already know the answer to, or can solve by applying the same algorithm to each piece, and then combining the results.

Stated more concisely, a recursive definition is defined in terms of itself. Recursion is a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step having a termination condition so that successive repetitions are processed up to the critical step where the condition is met at which time the rest of each repetition is processed from the last one called to the first.

Don't worry about the details of that definition. The main point of it is that it is defined in terms of itself: "Recursion: ... for more information, see Recursion."

Recursion turns out to be a wonderful technique for dealing with many interesting problems. Solutions written recursively are often simple. Recursive solutions are also often much easier to conceive of and code than their iterative counterparts.

<u>Our First Real Example: Factorial</u>

What kinds of problems are well solved with recursion? In general, problems that are defined in terms of themselves are good candidates for recursive techniques. The standard example used by many computer science textbooks is the factorial function.

The factorial function, often denoted as *n*! , describes the operation of multiplying a number by all the positive integers smaller than it. For example,5! = 5*4*3*2*1 . And 9! = 9*8*7*6*5*4*3*2*1 .
Take a good close look at the above, and you may notice something interesting. 5! can be written much more concisely as 5! = 5*4! .

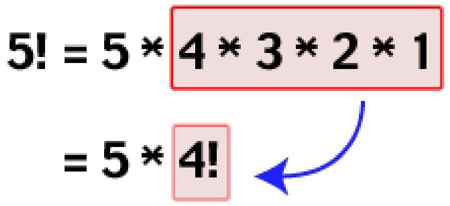


Fig: 5! = 5*4*3*2*1 = 5*4!

And 4! is actually 4*3! .

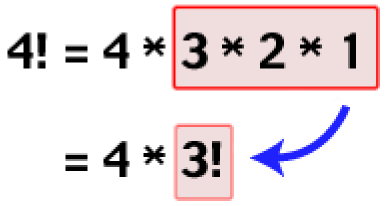


Fig: 4! = 4*3*2*1 = 4*3!

We now see why factorial is often the introductory example for recursion: the factorial function is recursive, it is defined in terms of itself. Taking the factorial of *n* , *n*! = *n**(*n* - 1)! where *n* > 0 .

<u>Coding Factorial</u>

Let's try writing our factorial function int factorial(int n). We want to code in the*n*! = *n**(*n* - 1)! functionality. Easy enough:

```
int factorial(int n)
{
        return n * factorial(n-1);
}
```

Wasn't that easy? Lets test it to make sure it works. We call factorial on a value of 3, factorial(3):

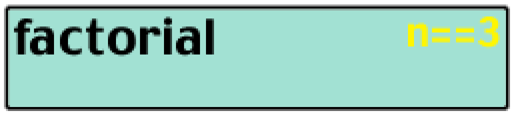


Fig: 3! = 3 * 2!

factorial(3) returns 3 * factorial(2). But what is factorial(2)?

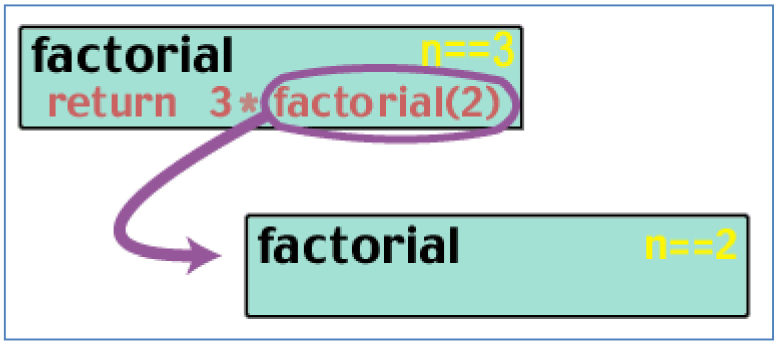


Fig: 2! = 2 * 1! factorial(2) returns 2 * factorial(1). And what is factorial(1)?

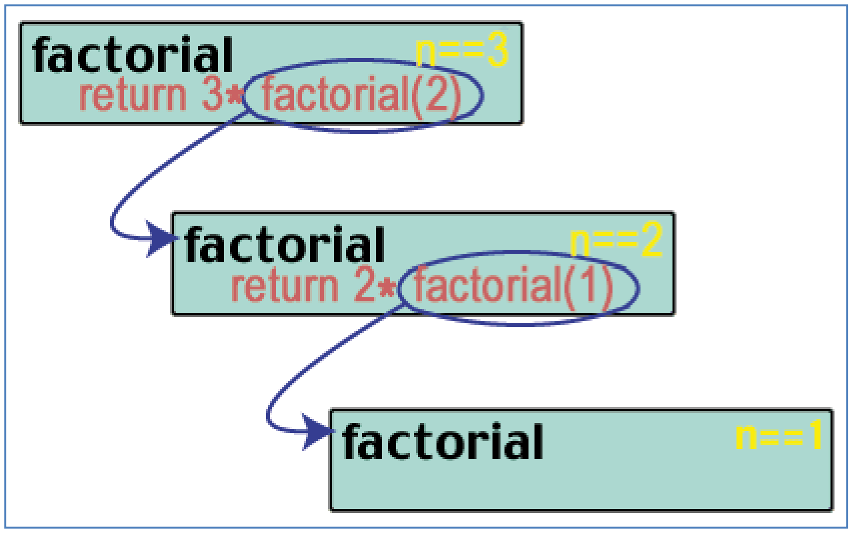


Fig: 1! = 1 * 0!

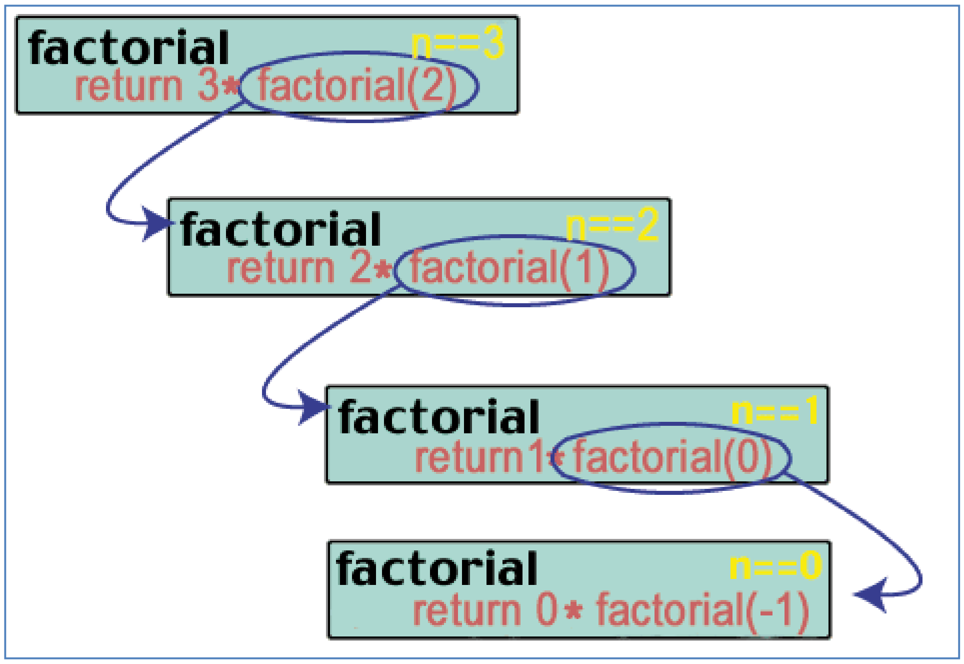


Fig: 0! = … oh!

Uh oh! We messed up. Thus far

```
factorial(3)  = 3 * factorial(2)
              = 3 * 2 * factorial(1)
              = 3 * 2 * 1 * factorial(0)
```

By our function definition, the factorial(0) should be 0! = 0 * factorial(-1). Wrong. This is a good time to talk about how one should write a recursive function, and what two cases must be considered when using recursive techniques.

<u>Two cases: Base Case and General Case</u>

There are four important criteria to think about when writing a recursive function.

1. **What is the base case, and can it be solved?**
2. **What is the general case?**
3. **Does the recursive call make the problem smaller and approach the base case?**

<u>Base Case</u>

The base case, or halting case, of a function is the problem that we know the answer to, that can be solved without any more recursive calls. The base case is what stops the recursion from continuing on forever. Every recursive function *must* have at least one base case (many functions have more than one). If it doesn't, your function will not work correctly most of the time, and will most likely cause your program to crash in many situations, definitely not a desired effect.
Let's return to our factorial example from above. Remember the problem was that we never stopped the recursion process; we didn't have a base case. Luckily, the factorial function in math defines a base case for us. $n! = n*(n - 1)!$as long as $n > 1$. If $n = = 1$ or $n = = 0$, then $n! = 1$. The factorial function is undefined for values less than 0, so in our implementation, we'll return some error value. Using this updated definition, let's rewrite our factorial function.

```
int factorial(int n)
{
       if (n<0) return 0;   /* error value for inappropriate input */
       else if (n<=1) return 1;          /* if n==1 or n==0, n! = 1 */
       else return n * factorial(n-1);    /* n! = n * (n-1)! */
}
```

That's it! See how simple that was? Lets visualize what would happen if we were to invoke this function, for example factorial(3):

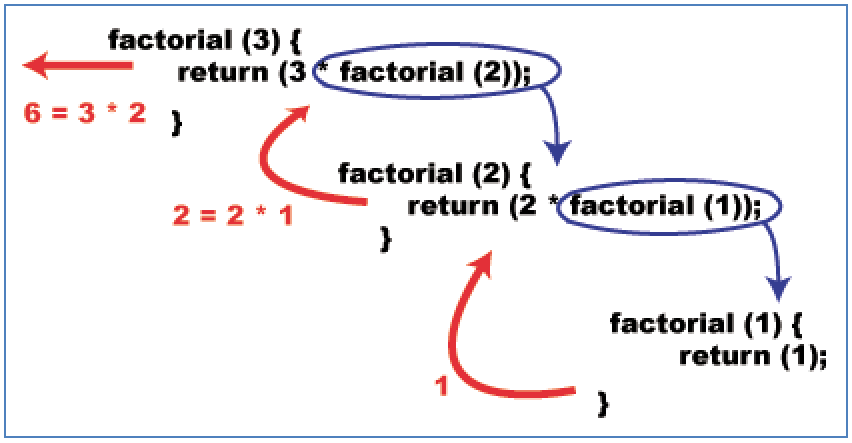


Fig: 3! = 3*2! = 3*2*1

The General Case

The general case is what happens most of the time, and is where the recursive call takes place. In the case of factorial, the general case occurs when $n > 1$, meaning we use the equation and recursive definition $n! = n*(n - 1)!$.

Problem: Find Sum 1 to N

Lets define another function **int sum(int N)**; This function should take an N as input, and give us the sum of 0 through N as output.
Let's start solving the problem:
according to the definition of our function, we get,
sum(N)=0+1+2+3+.....+N
which implies,
sum(N)={0+1+2+3+......+(N-1)} + N; ………… (1).
Now, what is sum(N-1) ?
sum(N-1)=0+1+2+3+......+(N-1); ....(2).
By replacing the portion in curly braces {} of eqn 1 ith eqn 2, we get,
sum(N)=sum(N-1)+N;

```
#include<stdio.h>

int sum(int N)
{
    if(N==0) return 0; // base case, if N=0, we know, answer is 0
    int x=sum(N-1);  // calling the same function to give the the
                     // result of 0+1+2...+(N-1). It is known as
                     // recursive case.
    int ans=x+N;     // if we know the result of N-1, just by
                     // adding N to it, we get our answer for the
    return ans;      // return the answer
}
```

```
int main()
{
    int t1=sum(3);
    int t2=sum(10);
    printf("%d %d\n",t1,t2); // output is 6 55
}
```

Now, how does it work? Let's take sum(3) as example, when we call sum(3), the value of N is 3.

Step	What's going on in the function.	Output
4.	**N=0**, here, if(N==0) is true. So we return our answer as 0.	0, send to Step 3
3.	**N=1**. Consider the body of the function, if(N==0) , this statement is false, so, it will get inside. int x=sum(N-1). This will call the function sum with N=0. Go to STEP 4. so, when this function gives its output, the value of x is 0. int ans=0+1=1. return 1.	1 This output is sent to STEP 2.
2.	**N=2**. Consider the body of the function, if(N==0) , this statement is false, so, it will get inside. int x=sum(N-1). This will call the function sum with N=1. Go to STEP 3. so, when this function gives its output, the value of x is 1. int ans=1+2=3. return 3.	3 This output is sent to STEP 1.
1.	**N=3**. Consider the body of the function, if(N==0) , this statement is false, so, it will get inside. int x=sum(N-1). This will call the function sum with N=2. Go to STEP 2. so, when this function gives its output, the value of x is 3.. int ans=3+3=6. return 6.	6 This output is sent to the main function.

When writing recursion, we should keep in mind that, it has two parts,
(a) Base Case
(b) Recursive Case

Base Case is used to terminate the function at some point. In our previous example, we used,

if(N==0) return 0; as base case. What if we hadn't used it? What would happen?

If we do not use the base case, from the STEP 4, 0 will enter the body of the function and will call sum(N-1), which is sum(-1), -1 will call sum(-2), -2 will call sum(-3) and so on. So, the calling will continue forever. Base case is used to stop the calling of function at some point. Usually, we use the smallest value of our input for which we know the answer, as base case. In our example, it was N=0, for which we knew the answer was 0.

While writing recursion, we must ensure that, our recursive calls will fall into some base case at some moment. Otherwise we will end up having an infinite recursion. If we need, we may introduce more than one base cases.

Recursive Case is used to get the answer of smaller instances of the problem. With the help of the answer we get, we construct the answer we desire. In our example, it was sum(N-1).

How to write good recursion?

First we define our function. In our example, we defined, int sum(int N);

We are expecting that, if we call sum(3), it will give us the result of 0+1+2+3, if we call sum(5) it will give us the result of 0+1+2+3+4+5, if we call sum(100), it will give us 0+1+2+....+99+100.

Now, we write the function:

```
int sum(int N)
{
}
```

This function is empty, right? It is not supposed to give us anything if we call it. But, for now, we will assume that this one is a *magical function*. That is, if we call it with some input, it will give us the correct output. Based on this assumption, we will be writing our recursion.

So, to get the answer for sum(N), we need the answer of sum(N-1). As we assumed this is a magical thing, we will just call this function with N-1, and will hope that it will give us the answer.

```
int sum(int N)
{
	int x=sum(N-1);
}
```

Now that, we have the answer for 0+1+2+...+(N-1), we just need to add the value of N with it and return the answer.

```
int sum(int N)
{
	int x=sum(N-1);
	int ans=x+N;
	return ans;
}
```

The function is almost ready. We just have one problem, the recursive case will go on forever. How to stop this? We use a base case. But, which one? We can see that, the value of N will be decreased by 1 at every step of the recursion. So, it is bound to be 0 at some point. And 0 is our smallest input, and we know the result for 0. So, we add if(N==0) return 0; at the beginning of the function as base case.

```
int sum(int N)
{
        if(N==0) return 0;
        int x=sum(N-1);
        int ans=x+N;
        return ans;
}
```

Fibonacci Problem:

F[0]=1;
F[1]=1;
F[N]=F[N-1]+F[N-2]   //Recursive Relation

```
int fibonacci(int n)
{
        if (n==1 || n==0) return 1;/* base case */
        else return fibonacci(n-1)+ fibonacci(n-2); //relation
}
```

Graphical View:

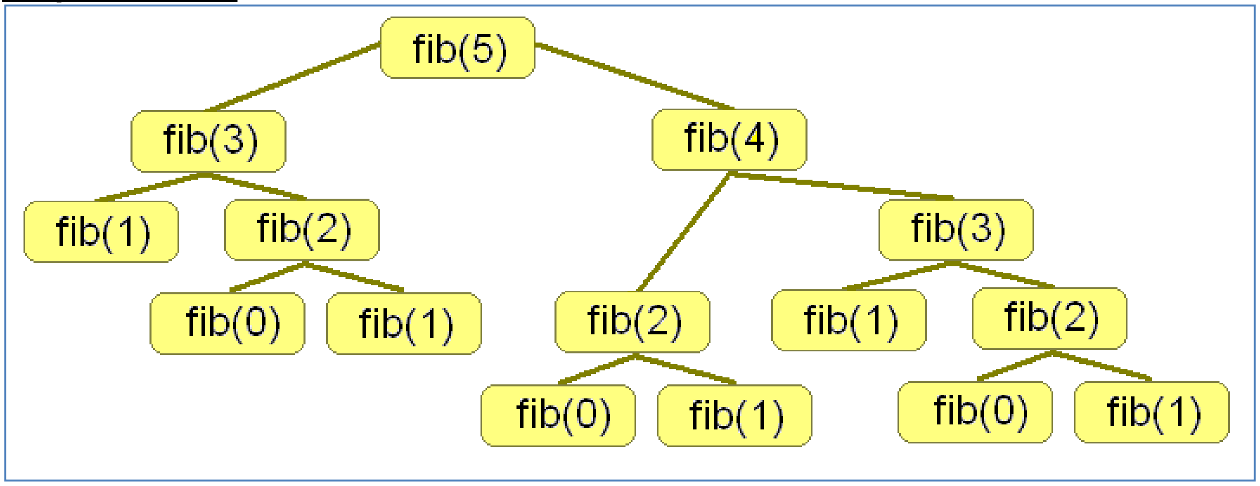
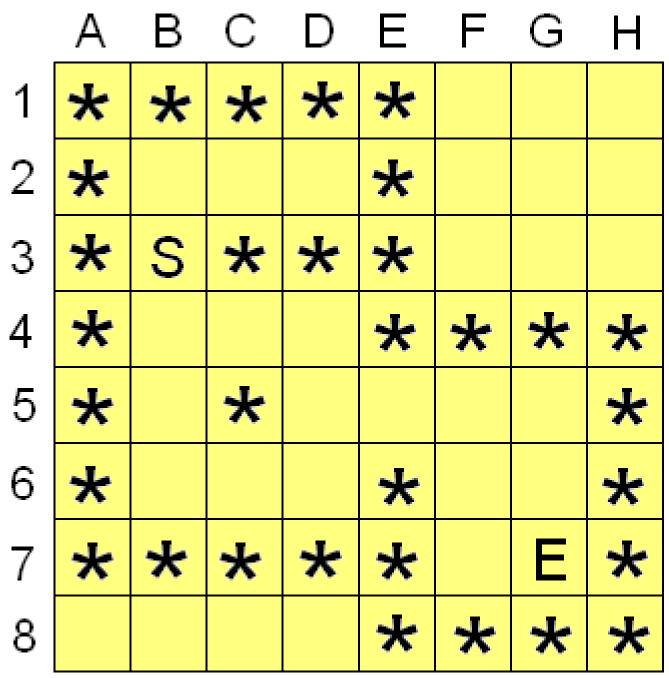


Fig: Calculate 5th Fibonacci Number

Multiple Related Decisions

When our program only has to make one decision, our approach can be fairly simple. We loop through each of the options for our decision, evaluate each one, and pick the best. If we have two decisions, we can have nest one loop inside the other so that we try each possible combination of decisions. However, if we have a lot of decisions to make (possibly we don't even know how many decisions we'll need to make), this approach doesn't hold up.

For example, one very common use of recursion is to solve mazes. In a good maze we have multiple options for which way to go. Each of those options may lead to new choices, which in turn may lead to new choices as the path continues to branch. In the process of getting from start to finish, we may have to make a number of related decisions on which way to turn. Instead of making all of these decisions at once, we can instead make just one decision. For each option we try for the first decision, we then make a recursive call to try each possibility for all of the remaining decisions. Suppose we have a maze like this:



	A	B	C	D	E	F	G	H
1	*	*	*	*	*			
2	*				*			
3	*	S	*	*	*			
4	*				*	*	*	*
5	*		*					*
6	*				*			*
7	*	*	*	*	*		E	*
8					*	*	*	*

For this maze, we want to determine the following: is it possible to get from the 'S' to the 'E' without passing through any '*' characters. The function call we'll be handling is something like this: "isMazeSolveable(maze[ ][ ])". Our maze is represented as a 2 dimensional array of characters, looking something like the grid above. Now naturally we're looking for a recursive solution, and indeed we see our basic "multiple related decision" pattern here. To solve our maze we'll try each possible initial decision (in this case we start at B3, and can go to B2 or B4), and then use recursion to continue exploring each of those initial paths. As we keep recursing we'll explore further and further from the start. If the maze is solveable, at some point we'll reach the 'E' at G7. That's one of our base cases: if we are asked "can we get from G7 to the end", we'll see that we're already at the end and return true without further recursion. Alternatively, if we can't get to the end from either B2 or B4, we'll know that we can't get to the end from B3 (our initial starting point) and thus we'll return false.

Our first challenge here is the nature of the input we're dealing with. When we make our recursive call, we're going to want an easy way to specify where to start exploring from - but the only parameter we've been passed is the maze itself. We could try moving the 'S' character around in the maze in order to tell each recursive call where to start. That would work, but would be very slow because in each call we'd have to first look through the entire maze to find where the 'S' is. A better idea would be to find the 'S' once, and then pass around our starting point in separate variables. This happens fairly often when using recursion: we have to use a "starter" function that will initialize any data and get the parameters in a form that will be easy to work with. Once things are ready, the "starter" function calls the recursive function that will do the rest of the work. Our starter function here might look something like this:

```
function isMazeSolveable(maze[][])
{
    declare variables x,y,startX,startY
    startX=-1
    startY=-1
    // Look through grid to find our starting point
    for each x from A to H
    {
        for each y from 1 to 8
        {
            if maze[x][y]=='S' then
            {
                startX=x
                startY=y
            }
        }
    }
    // If we didn't find starting point, maze isn't solveable
    if startX==-1 then return false
    // If we did find starting point, start exploring from that point
    return exploreMaze(maze[][],startX,startY)
}
```

We're now free to write our recursive function exploreMaze. Our mission statement for the function will be "Starting at the position (X,Y), is it possible to reach the 'E' character in the given maze. If the position (x,y) is not traversable, then return false." Here's a first stab at the code:

```
function exploreMaze(maze[][],x,y)
{
    // If the current position is off the grid, then
    // we can't keep going on this path
    if y>8 or y<1 or x<'A' or x>'H' then return false

    // If the current position is a '*', then we
    // can't continue down this path
    if maze[x][y]=='*' then return false

    // If the current position is an 'E', then
    // we're at the end, so the maze is solveable.
    if maze[x][y]=='E' then return true

    // Otherwise, keep exploring by trying each possible
    // next decision from this point.  If any of the options
    // allow us to solve the maze, then return true.  We don't have to
    // worry about going off the grid or through a wall - we can
    //trust our recursive call to handle those possibilitiescorrectly.
    if exploreMaze(maze,x,y-1) then return true  // search up
    if exploreMaze(maze,x,y+1) then return true  // search down
    if exploreMaze(maze,x-1,y) then return true  // search left
    if exploreMaze(maze,x+1,y) then return true  // search right

    // None of the options worked, so we can't solve the maze
    // using this path.
    return false
}
```

Recursion Practice Problems

Problem 1)

Sum of all Subset sum

Example: N=3 Here N is the Number of Element

A[]={1,3,5}

All Subset:

{}   =0
{1}   =1
{3}   =3
{5}   =5
{1,3} =4
{1,5} =6
{3,5} =8
{1,3,5}=9

Total Sum =0+1+3+5+4+6+8+9=36

SampleC++ Code:

```
#include <stdio.h>

int ans,N,A[22];

void GenerateSubSet(int cur,int sum)
{
    if(cur==N)
    {
        ans+=sum;
    }
    else
    {
        GenerateSubSet(cur+1,sum+A[cur]);
        GenerateSubSet(cur+1,sum);
    }
}

int main()
{
    int i;

    while(scanf("%d",&N)==1)
    {
        for(i=0;i<N;i++)
            scanf("%d",&A[i]);
        ans=0;
        GenerateSubSet(0,0);
        printf("Sum All SubSet= %d\n",ans);
    }

    return 0;
}
Input:
3
1 3 5
Output:
Sum All SubSet= 36
```

Problem 2) Find All Permutation Fixed N Length
Example: N=3 Length All Permutation

ABC
ACB
BAC
BCA
CAB
CBA

Solution:

```
#include <stdio.h>

int a[100],N;

void Permutation(int cur)
{
    if(cur==N)
    {
        for(int i=0;i<N;i++)
            printf("%c",a[i]+'A');
        printf("\n");
    }
    else
    {
        int i,j;
        for(i=0;i<N;i++)
        {
            for(j=0;j<cur;j++)
            {
                if(a[j]==i)
                    break;
            }
            if(j==cur)  // i element is not used in previous
            {
                a[cur]=i;
                Permutation(cur+1);
            }
        }
    }
}
int main()
{
    while(scanf("%d",&N)==1)
    {
        Permutation(0);
    }
}
Input:
3
Output:
ABC
ACB
BAC
BCA
CAB
CBA
```

Problem 3) Sum of N Elements Using Recursion
Example: N=3
A[]={4,7,8}
Sum=4+7+8=19

```
int rec(int cur)
{
    int (cur==N-1) return A[cur];
    return A[cur]+rec(cur+1);
}
```

Problem 4) Binary Search

```
int BinarySearch(int low,int high,int item)
{
    if(low>=high) return -1; //Item not find

    int mid=(low+high)/2;

    if(A[mid]==item) return mid; //Item find

    if(A[mid]>item) return BinarySearch(low,mid-1,item);
    else return BinarySearch(mid+1,high,item);
}
```

Problem 5) Merge Sort

Problem 6) Finding Maximum Element

Problem 7) Reverse Order Print

```
 void Print(int N)
{
    if(N==0) return ;
    else
    {
        Print(N/2);
        printf("%d",N%2);
    }
}
```

Problem 8) 4 Queen Problem

Problem 9) Divisor Generate (See The Number Theory Sheet)

Problem 10) Power Calculate x^y

```
int rec(int x,int y)
{
    if(y==0) return 1;
    return x*rec(x,y-1);
}
```

Problem 11) BigMod $x^y\%M$

```
int BigMod(int x,int y,int M)
{
    if(y==0) return 1;

    if(y%2==1)
        return (x*BigMod(x,y-1,M))%M;
    else
    {
        val=BigMod(x,y/2,M);
        return (val*val)%M;
    }
}
```

Problem 12) Factorial Factors

```
int Factor(int N,int P)
{
    if(N==0) return 0;

    return Factor(N/P,P)+N/P;
}
```

Problem 13) nCr Calculate

```
int nCr(int n,int r)
{
    if(n==0) return !r;

    int ret=nCr(n-1,r);

    if(r>0)
        ret+=nCr(n-1,r-1);
    return ret;
}
```

Problem 14) Palindrome Check

```
int Palindrome(int i,int j)
{
    if(i>j) return 1;

    if(str[i]==str[j])
        return Palindrome(i+1,j-1);
    else return 0;
}
```

Problem 15) Ternary Search

Problem 16) Quick Sort

Problem 17) Count The How many digits of a number

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

    return digit(n/10)+1;
}
```

Problem 18) Calculate remainder n%m without %

```
int remainder(int n,int m)
{
    if(n<m) return n;

    return remainder(n-m,m);
}
```

Problem 19) GCD Calculate

```
int gcd(int a,int b)
{
    if(b==0) return a;

    return gcd(b,a%b);
}
```

Problem 20) Tower of Hanoi

Problems in Online Judges:

LOJ: 1023,1117
UVa: 195,10696,10776,11753,624
TopCoder: SRM 491 Div 2 1000, TCHS SRM 16 1000, TCHS SRM 55 1000, SRM 273 Div 2 1000, TCHS SRM 2 1000, SRM 449 Div 2 500, TCCC06 Online Round 2A Div 1 250, SRM 275 Div 2 500, TCHS SRM 26 500
TJU: 1306, 2893

Prepared By:

Forhad Ahmed (forhadsustbd@gmail.com)
MD. Mustafijur Rahman (m.mustafijur.rahman@gmail.com)

Links:
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=recursionPt1
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=recursionPt2
http://www.sparknotes.com/cs/recursion/whatisrecursion/problems_1.html