Recursion

- רקורסיה היא תהליך בו הפונקציה (שיטה) קוראת לעצמה
- ישנן בעיות שהפיתרון הטבעי שלהן הוא רקורסיבי
- רקורסיה היא שיטה שבה הפתרון לבעיה מסתמך על פתרון בעיות קטנות ודומות לבעיה הגדולה יותר.
- בהינתן בעיה שרוצים לפתור:

✓נמצא כודם תת-בעיה דומה (קטנה יותר) אשר אנו יודעים מהו הפתרון שלה הפתרון של תת-הבעיה יהיה תנאי העצירה של הפונקציה

✓נניח שהפונקציה יודעת לפתור את הבעיה המוקטנת

✓נשתמש בפתרון של הבעיה המוקטנת כדי לפתור את הבעיה הגדולה יותר וזה יהיה הצעד של הרקורסיה.

מימוש לולאה בעזרת
שימוש ברקורסיה

- ניתן להביע כל לולאה באמצעות רקורסיה ולהפך(בפרקטיקה זה לא תמיד פשוט)
  - ✓את הבלוק של הלולאה מעברים לתוך הפונקציה
  - ✓מבצעים קריאות רקורסיביות לפונקציה
  - ✓מוסיפים תנאי עצירה

```
public void whileMethod{
    if (!cond) {
        return;
    }
   statement;
   whileMethod();
}
```

שקולים

```
while (cond) {
  statement;
}
```

הערה: לכל קריאה רקורסיבית לפונקציה יש עותק משלה למשתנים המקומיים - דבר שאין בלולאה

write a function that receives a positive number ‘n’ and prints all numbers between 1…n in descending order

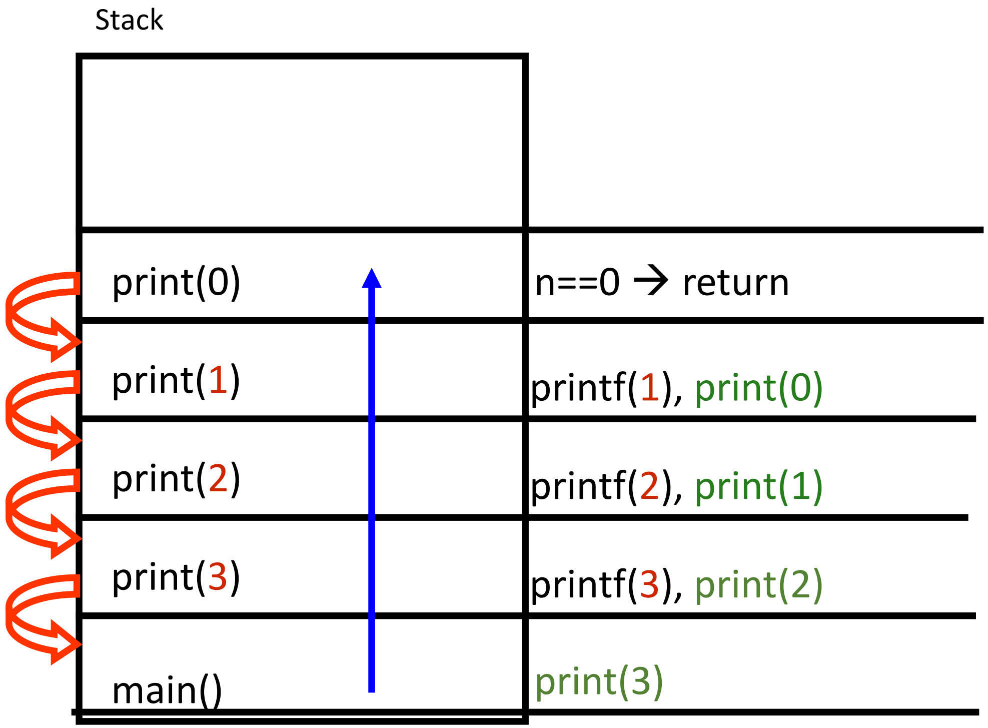
Recursion

```
void print(int n)
{
   if(n<=0)
      return;

   printf("%d\n", n);

   print(n-1);
}
```

```
print(3):
3
2
1
```

Non-Recursion

```
void print(int n)
{
   while(n>0)
   {
      printf("%d\n", n);
      n--;
   }
}
```



write a function that receives a positive number ‘n’ and prints all numbers between 1…n in ascending order

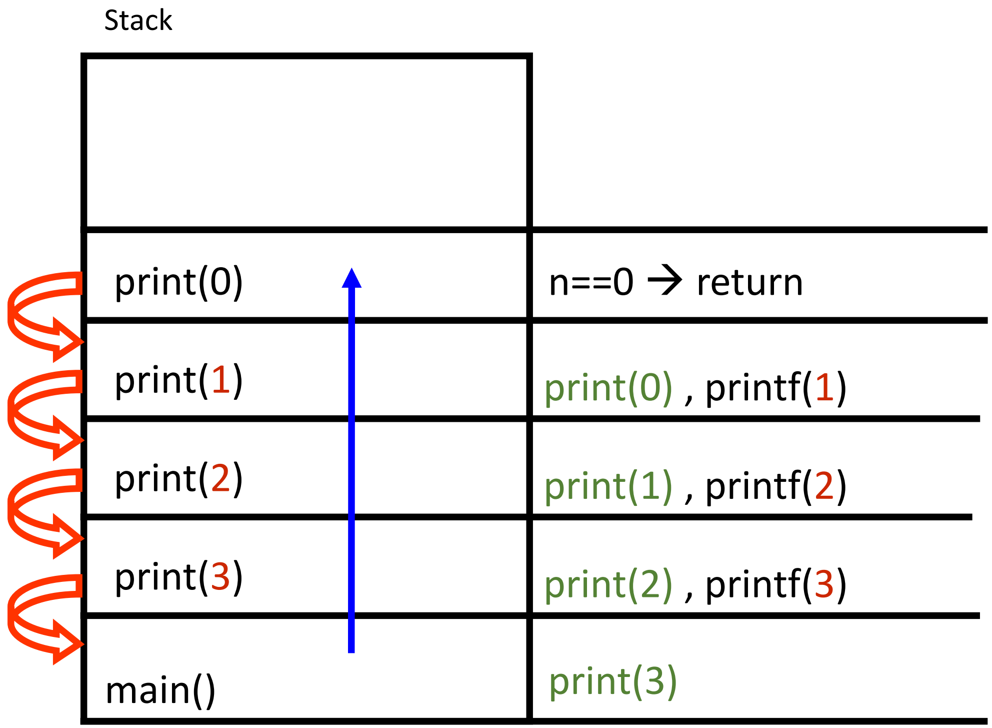
Recursion

```
void print(int n)
{
   if(n<=0)
      return;

   print(n-1);

   printf("%d\n", n);
}
```

```
print(3):
1
2
3
```

Non-Recursion

```
void print(int n)
{
   int i = 1;
   while(i <= n)
   {
      printf("%d\n", i++);
   }
}
```



Stack

print(0) | n==0 → return

print(1) | print(0) , printf(1)

print(2) | print(1) , printf(2)

print(3) | print(2) , printf(3)

main() | print(3)

write a function that receives a positive number ‘n’ and returns the sum of all numbers between 0 …n

‘else‘ can be discarded →

```
int sum (int n)
{
   if(n==0)  → base case
      return 0;
   else
      return sum(n-1) + n;
}
```

‘n’ is used as an index to the recursion

enhancement of the previous program to include negative numbers as well

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

   if(n<0)
      return -sum(-n);

   return sum(n-1) + n;
}
```

Tail and non Tail recursion

Recursive methods are either:

1. Tail recursive: the recursive call is the last statement in the method.
    - are easy to convert to iterative.
    - smart compilers can optimize code by easily detecting tail recursion and convert it to iterative.
    - used to implement loops in programming languages that do not support loop structures e.g. Lisp , Prolog…
2. non tail recursive: method that are not tail recursive are called non-tail recursive

```
void print(int arr[], int n)
{
   if(n==0)
   {
      return;
   }

   printf("%d\n", arr[n-1]);

   print(arr,n-1);
}
```

tail recursive

descending order

```
void print(int arr[], int n)
{
   if(n==0)
   {
      return;
   }

   print(arr,n-1);

   printf("%d\n", arr[n-1]);
}
```

non tail recursive

ascending order

given a positive integer ‘n’ write a function that returns n!

tail recursive

```
int factorial2(int n, int fact)
{
   if (n == 0)
      return fact;

   return factorial2(n-1,n*fact);
}

int factorial(int n)
{
   return factorial2(n, 1);
}
```

```
factorial(5, 1)
factorial(4, 5)
factorial(3, 20)
factorial(2, 60)
factorial(1, 120)
factorial(0, 120)
120
```

n!=n*(n-1)!

non tail recursive

```
int factorial(int n)
{
   if (n == 0)
      return 1;

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

```
factorial(5)
5 * factorial(4)
5 * (4 * factorial(3))
5 * (4 * (3 * factorial(2)))
5 * (4 * (3 * (2 * factorial(1))))
5 * (4 * (3 * (2 * (1 * factorial(0)))))
5 * (4 * (3 * (2 * (1 * 1))))
120
```

GCD
Greatest common divisor

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

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

Euclid's Algorithm

$$\mathbf{gcd(a,b)} = \begin{cases} \mathbf{a} & \mathbf{if\ \ b = 0} \\ \mathbf{gcd(b, a\%b)} & \mathbf{otherwise} \end{cases}$$

gcd(18,12)=6

gcd(616,165)=11

Gcd(1071,1029)=21

a%b	a/b	b	a
121	3	165	616
44	1	121	165
33	2	44	121
11	1	33	44
0	3	11	33
-	-	0	11

write a function that returns the sum of the members of the array

```
int sum(int arr[], int n)
{
   if(n == 0)
      return 0;

   return arr[n-1] +   sum(arr, n-1);
}
```

Write a function that returns the number of occurrences of a number in a given array

```
int count (int arr[], int n, int num)
{
   if(n==0)
      return 0;

   if(arr[n - 1] != num)
      return count(arr, n-1, num);
   else
      return 1 + count(arr, n-1, num);
}
```

Write a function that returns the sum of 2 positive number

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

   return sum(a, b-1)+1;
}
```

Enhancement of the previous program to include negative numbers as well

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

    if(b<0)
        return -sum(-a,-b);

    return sum(a,b-1)+1;
}
```

$a + b = -(-a - b)$

Write a function that returns the modulo of 2 positive numbers a%b

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

   if(a<b)
      return a;

   return mod(a-b, b);
}
```

Enhancement of the previous program to include negative numbers as well

(modulo < 0) ⟺ (a< 0)

5 % 3 = 2
5 % -3 = 2
-5 % 3 = -2
-5 % -3 = -2

```
#include <stdlib.h>
int mod (int a, int b)
{
   if(b==0)
      return 0;

   if(abs(a)<abs(b))
      return a;

   if(a<0)
      return -mod(abs(a), b);

   return mod(a - abs(b), abs(b));
}
```

given 2 positive integers ‘a’ and ‘b’– convince your self that below function returns a*b

```
int mult (int a, int b)
{
   if (b == 0)
      return 0;
   if (b % 2 == 0)
      return mult(a + a, b / 2);
   return mult(a + a, b / 2) + a;
}
```

given 2 positive integers ‘a’ and ‘b’– write a function that returns a*b

the result will remain correct if a<0

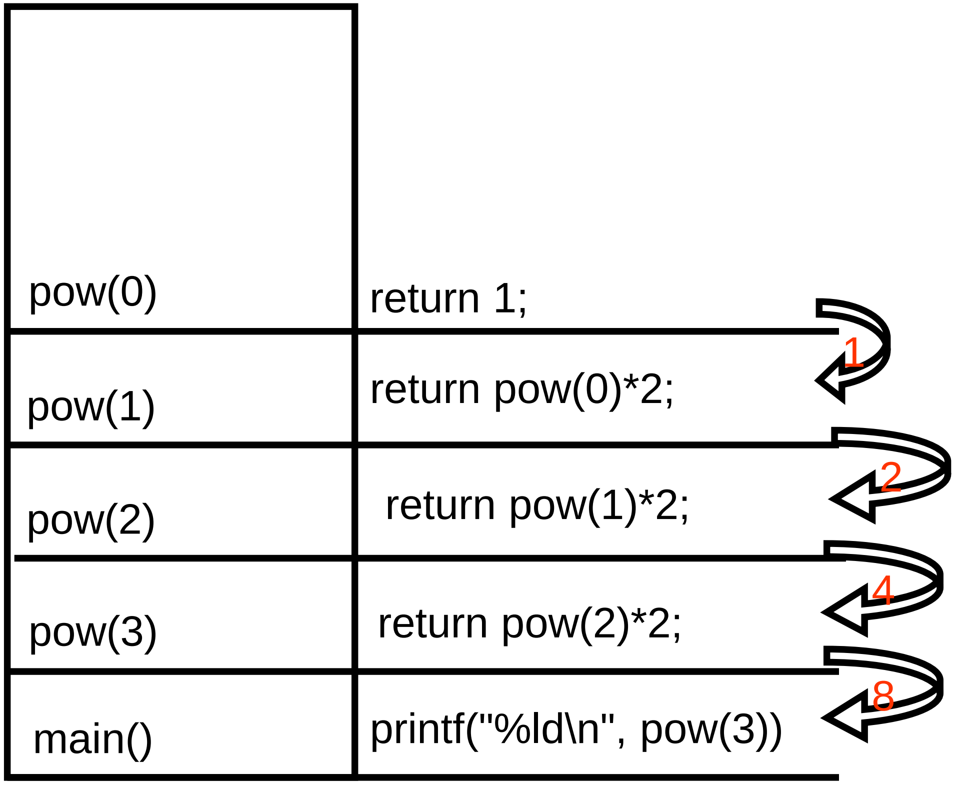
```
int mult (int a, int b)
{
   if(b==0)
      return 0;
   return mult(a, b-1) + a;
}
```

Enhancement of the previous program to include negative numbers as well

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

   if(b<0)
      return - mult(a, -b);

   return a + mult(a, b-1);
}
```

given a positive integer ‘n’ write a function that returns 2^{n}

```
long pow(int n)
{
    if (n == 0)
        return 1;
    return pow(n-1)*2;
}
```



given 2 positive integers ‘a’ and ‘b’– write a function that returns $\mathbf{a^b}$

```
long pow(int a, int b)
{
    if (b == 0)
        return 1;

    return pow(a, b-1) * a;
}
```

$$\mathbf{a^b} = \boldsymbol{a \cdot a^{b-1}}$$

given 2 integers ‘a’ and ‘b’ (could be negatives) – write a function that returns $\mathbf{a^b}$

```
double pow (int a, int b)
{
   if (b == 0)
      return 1;

   if(b<0)
      return 1.0/pow(a, -b);

   return pow(a, b-1) * a;
}
```

$$\mathbf{a^b = a \cdot a^{b-1}, b > 0}$$

$$\mathbf{a^b = \frac{1}{a^{-b}}, b < 0}$$

write a function that returns the maximum in an array

version1

```
int max(int arr[], int size)
{
    int tmp;

    if(size==1)
    {
        return arr[0];
    }

    tmp= max(arr,size-1);

    if(arr[size-1]>tmp)
        tmp=arr[size-1];

    return tmp;
}
```

משתנה עזר

version2

```
int max2( int a, int b )
{
    return a>b ? a:b;
}

int max( int a[], int n )
{
    if(n==1)
        return a[0];

    return max2( a[n-1], max(a,n-1) );
}
```

שיטת עזר

write a function that returns the row with maximum sum in an array

```
int sumRow(int arr[][2], int m, int n) {
   if ( n == 1)
      return arr[m-1][n-1];
   return arr[m-1][n-1] + sumRow(arr,m,n-1);
}

int max2(int arr[][2], int m, int n,int maxRow, int maxSum) {
   int s;
   if(m == 0)
      return maxRow;
   s = sumRow(arr, m, n);
   if(s > maxSum)  {
      maxSum=s;
      maxRow=m;
   }
   return max2(arr, m-1, n, maxRow, maxSum);
}

int max(int arr[][2], int m, int n) {
   return max2(arr, m, n, 1, sumRow(arr,1,n));
}
```

```
int arr[][2] = {{1,12}, {12,-1}, {9,1}};
printf("%d\n",max(arr,3,2));
```

Print the Binary System of a Decimal number using Recursion

```
void decToBinaryPrint(unsigned char dec)
{
    if (dec == 0)
    {
        putchar('0');
    }
    else if(dec==1)
    {
        putchar('1');
    }
    else
    {
        decToBinaryPrint(dec / 2);
        putchar((dec % 2) + '0');
    }
}
```

Convert a Number Decimal System to Binary System using Recursion

```
unsigned decToBinary(unsigned char dec)
{
   if (dec < 2)
      return dec;

   return (dec % 2 + 10 * decToBinary(dec / 2));
}
```

C program for palindrome check using recursion

```c
int palindrome2(char str[],int left, int right) {
   if(right==0)       return 0;

   if(right==1)       return 1;

   if(str[left] != str[right-1])       return 0;

   return palindrome2(str,left+1,right-1);
}
int palindrome(char str[], int n) {
   return palindrome2(str,0, n);
}
```

version1

```c
int palindrome2(char str[],int left, int right) {
   if(left>=right)       return 1;

   if(str[left] != str[right-1])       return 0;

   return palindrome2(str,left+1,right-1);
}
int palindrome(char str[], int n) {
   if(n==0)       return 0;

   return palindrome2(str,0, n);
}
```

version2

```c
int palindrome2(char* a, char*b) {
   if(a>b)      return 1;

  if(*a != *b)     return 0;

  return palindrome2(a+1,b-1);
}
int palindrome(char str[], int n) {
  return palindrome2(str,str+n-1);
}
```

version3

Find the place of a character in a string (array of char)

```
int find(char str[], int n, char c)
{
    if(n == 0)
        return -1;

    if(str[n-1] == c)
        return n;

    return find(str, n-1, c);
}
```

Recursive function to reverse an array

```
void reverse(char arr[], char rev[], int n)
{
   if(n==0)
      return;

   rev[0]=arr[n-1];

   reverse(arr, rev+1, n-1);
}

int main()
{
   char str[3] = {'a','b','c'};
   char rev[3];
   reverse(str,rev,3);
   return 0;
}
```

Recursive function to print 2D array

```
#include <stdio.h>
void printMat2(int a[][3],int i,int j)
{
   if(i==3)
      return;
   if(j==3) {
      printMat2(a,i+1,0);
      return;
   }
   printf("%d",a[i][j]);
   printMat2(a,i,j+1);
}
void printMat(int a[][3])
{
   printMat2(a,0,0);
}
int main() {
   int a[][3] ={{1,2,3},{4,5,6},{7,8,9}};
   printMat(a);
   return 0;
}
```

***כתוב שיטה ריקורסיבית equals אשר מקבלת שתי מחרוזות תוים s1 ו- s2. השיטה תחזיר
true אם במחרוזת s2 מופיעים כל התווים של s1 , לפי הסדר , אבל לפעמים תו מסוים מ s2 יכול
להופיע ברצף מספר לא ידוע של פעמים יותר מאשר במחרוזת s1.

```
typedef enum
{
   false,
   true
} Boolean;
Boolean equals (char* s1, char* s2)
{
   if(s1==NULL || s2==NULL)
      return false;

   if(*s1==0 && *s2==0)
      return true;

   if(*s1==0 || *s2==0)
      return false;

   if(*s1 != *s2)
      return false;

   return equals (s1+1, s2+1)  || equals(s1, s2+1);
}
```

s1= “abbcd" , s2=("abbcd" , "aaaabbcd" , "abbcddddddd" , "aabbccdd" , "abbbccd") ➔ true
s1= “abbcd“ , s2=("a" , "abcd" , "aaccbbdd") ➔ false

***כתוב שיטה ריקורסיבית equals אשר מקבלת שתי מחרוזות תוים s1 ו- s2 המורכבות
מאותיות באנגלית ורווחים בלבד. השיטה צריכה להשוות בין המחרוזות ולקבוע אם הן זהות תוך
כדי התעלמות מגודל תווים (case insensitive) וכמו כן תחשיב רצפי רווחים כאילו היו רווח יחיד.

```
typedef enum
{
   false,
   true
} Boolean;
Boolean equals(char* s1, char* s2)
{
   if(s1 == NULL || s2 ==NULL)
     return false;

   if(*s1==0 && *s2==0)
     return true;

   if(*s1==0 || *s2==0)
     return false;

   if(!(*s1 == *s2 || *s1-*s2 == 'a'-'A' || *s2-*s1=='a'-'A'))
     return false;

   return
      equals(s1+1,s2+1) ||
      (*s1==' ' && equals(s1+1,s2)) ||
      (*s2==' ' && equals(s1,s2+1));
}
```

Below strings are equals:

"cat" = "cat"
"i am student" != "i   AM   stuDenT"
"i   am student" != "i am student"

But below are not:

"cat"," cat"
"cat   ","cat"
"cat   ","   cat"
"cat","catt"
"catt","cat"
"cat   t","catt"
"catt","cat   t"
"ca t","cat"
"c a t","cat"

```
#include <stdio.h>
#define M 5
#define N 5
int count(char a[M][N],int i, int j, char* s);
int main()
{
    char a[M][N] =
    {
        {'h','e','l','l','o'},
        {'a','b','h','e','l'},
        {'f','o','e','o','l'},
        {'e','l','l','l','o'},
        {'h','l','o','s','j'}
    };

    printf("%d\n",count(a,1,2,"hello"));
    return 0;
}
```

***write a recursive function ‘count’ that receives a matrix 0f ‘char a[M][N]’ ,int i,int j and a string ‘char *s’ and returns the number of times ‘s’ appears in ‘a’ starting from place (i,j).

count(char a[M][N],int i, int j, char* s)
Note: if s is null or empty string then count returns 0.

Output:
5

```
int count(char a[M][N],int i, int j, char* s)
{
    char x;
    int c;

    if(a==NULL || s==NULL || i<0 || i>=M || j<0 || j>=N || a[i][j]=='#' || a[i][j] != *s)
        return 0;

    if(*(s+1)==0 && a[i][j] == *s)
        return 1;

    x = a[i][j];
    a[i][j] = '#';

    c =
        count(a,i-1, j,s+1)+
        count(a,i+1, j,s+1)+
        count(a,i, j+1,s+1)+
        count(a,i, j-1,s+1);

    a[i][j] = x;
    return c;
}
```

summary

While solving a backtracking problem a one should identify:

- ✓ What are the "choices" in the problem.
- ✓ What is the base case.
- ✓ How to make a choice.
- ✓ Should we create additional variables to remember a previous choice. if yes then is their a need to modify the values of existing variables. And How do we make the next/rest of the choices.
- ✓ Should we remove the made choice from the list of choices once we are done with exploring all the choices and if yes then how to remove a done choice.